ChatGPT Emacs Integration Course

Tony Aldon

August 14, 2025

Abstract

Hi there! I'm Tony Aldon a passionate Emacs and AI enthusiast.

If you want to learn **how to build a ChatGPT client for Emacs integrating the OpenAI API**, this course is for you!

`https://tonyaldon.com/chatgpt-emacs-integration-course/`

Through **18 engaging lessons**, **2.5 hours video content** and **a detailed PDF companion** you'll learn how to build **a fully functional Emacs package from scratch** and you'll know how to use the **OpenAI API**.

It can help if you know a little bit of Emacs Lisp, but this is not a requirement as **we'll meticulously write, review, and comment on each line of code**.

In this course, we'll build `chatgpt.el`, a package that lets you send prompts to ChatGPT directly from Emacs using the OpenAI API. Simply call the `chatgpt` command, enter your prompt in the dedicated buffer, press `C-c C-c`, and receive your response in an appended buffer seamlessly.

Beyond its simplicity, `chatgpt.el` offers these key features:

- **Secure API Key Handling:** Automatically retrieves your OpenAI API key from `~/.authinfo.gpg` for secure storage or from the plaintext `~/.authinfo` file.
- **Prompt History:** Keeps track of your previous prompts, allowing you to navigate back and forth through your request history using `M-p` and `M-n`.
- **Request Logging:** Saves all requests and responses in a designated directory, efficiently organizing your interactions with ChatGPT for easy reference and review. Really handy for troubleshooting.

Contents

1 First Request to OpenAI Using the Chat Completion API

Welcome to the first lesson of the **ChatGPT Emacs course**. In this lesson, we will send our first request to **OpenAI** using the **Chat Completion API**, replicating the process of interacting with ChatGPT from the command line.

To illustrate, when we visit chatgpt.com and type "Hello!", we receive a reply saying, "Hey there, how is it going?"

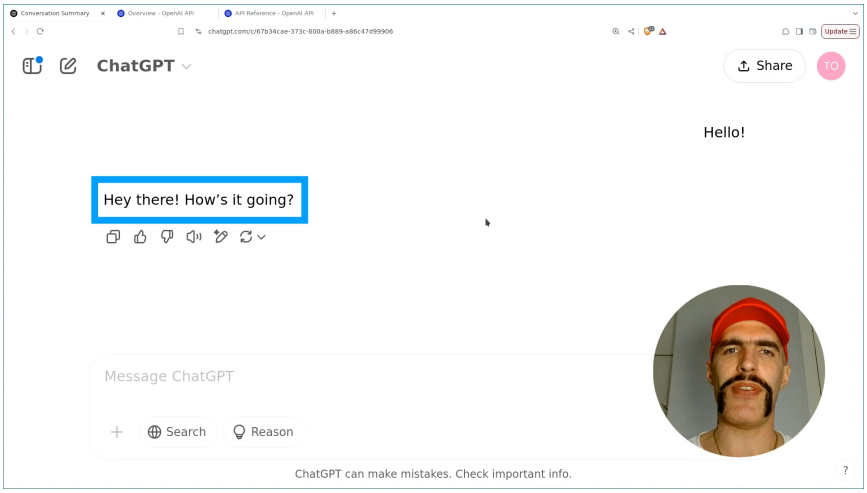


Figure 1: ChatGPT Interaction

Now, we will learn how to achieve this interaction using the `curl` command in the terminal. Let's get started.

Link: `https://chatgpt.com`

1.1 Adding funds to Your credit balance on OpenAI Developer Platform

To add funds to your credit balance on the OpenAI Developer Platform, follow these steps:

1. **Account Access**: If you already have an account on chatgpt.com, you can use the same credentials to log in. If not, create a new account on the platform.

2. **Navigating Projects**: Once logged in, click on your current project in the top left corner.

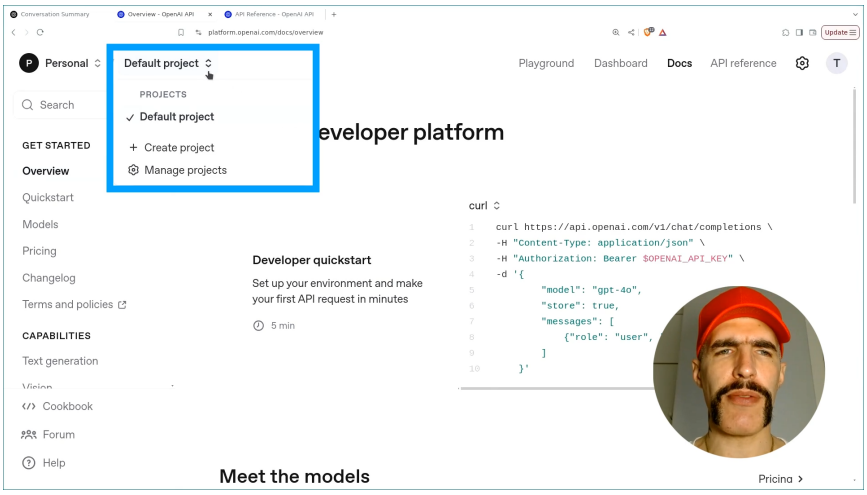


Figure 2: Current Project

3. **Manage Projects**: From the dropdown menu, select `Manage Projects`.

4. **Go to Billing**: In the sidebar on the left, click on `Billing` to access the credit balance management page.

5. **Add to Credit Balance**: Click on `Add to Credit Balance`.

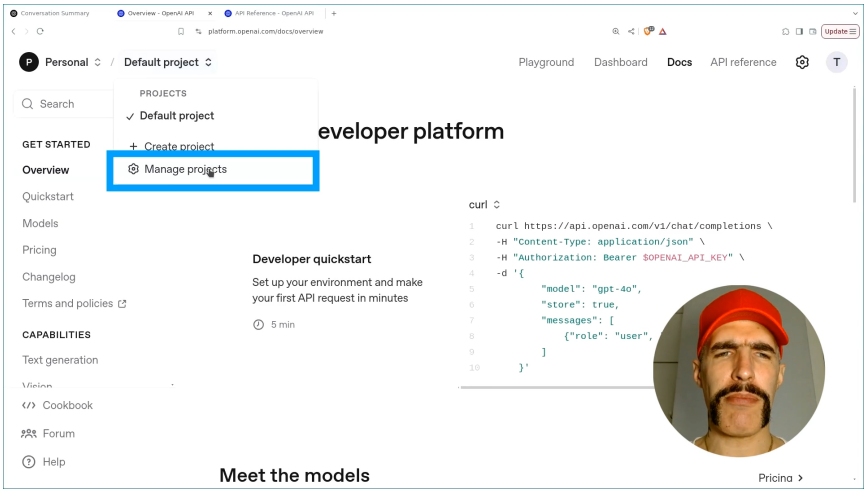


Figure 3: Manage Projects

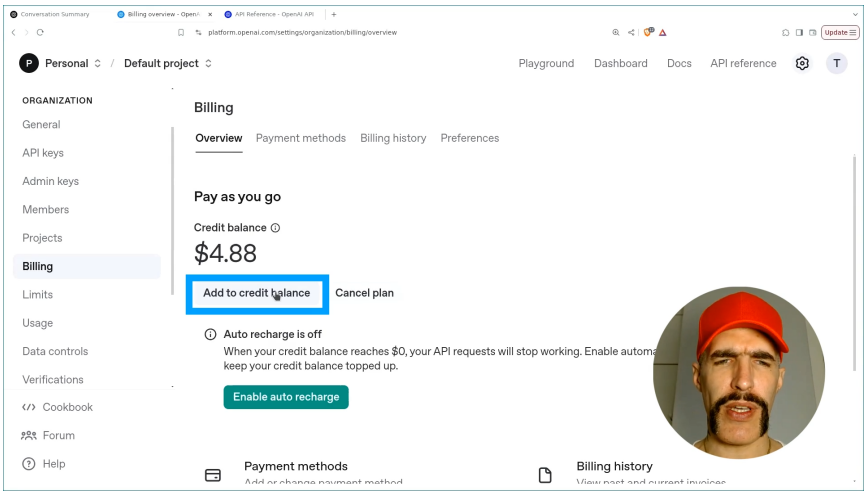


Figure 4: Add to Credit Balance

6. **Input Payment Details**: A recommended starting amount is $5, which is sufficient for the entire course; you will spend less than $0.10 throughout. Enter your payment details and click `Continue`.

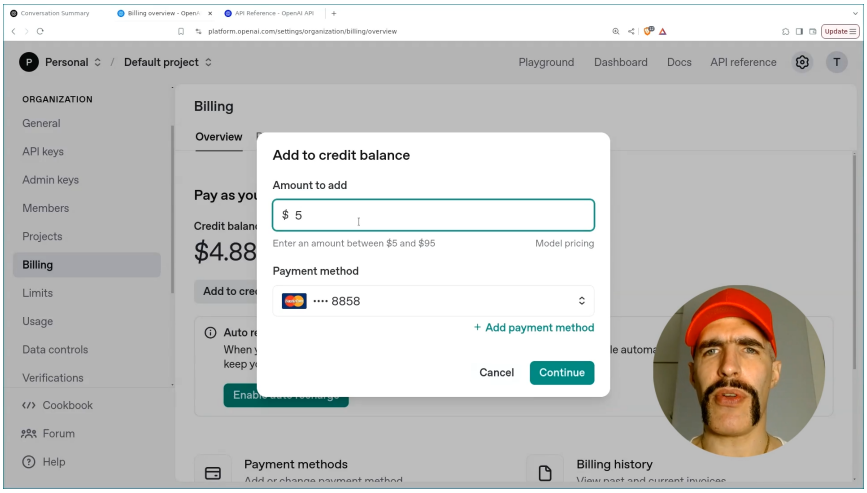


Figure 5: Payment Details

By following these steps, you can easily fund your credit balance for API calls.

Link: `https://platform.openai.com`

1.2 Creating an API Key on OpenAI Developer Platform

With sufficient funds in your credit balance, you can proceed to generate an API key.

1. **Access the Dashboard**: Navigate to the OpenAI Developer Platform and click on the `Dashboard` tab located at the top.

2. **Open API Keys Menu**: On the sidebar, select the `API Keys` option.

3. **Create New API Key**: You will now see the option to create new API keys as well as view existing ones. Click on the option to create a new secret key. Name this key according to your project; for example, use `chatgpt-emacs`.

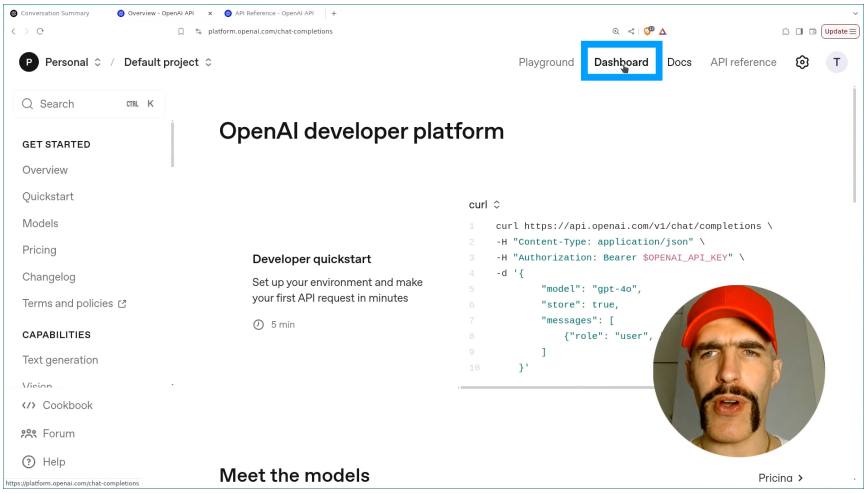


Figure 6: Dashboard

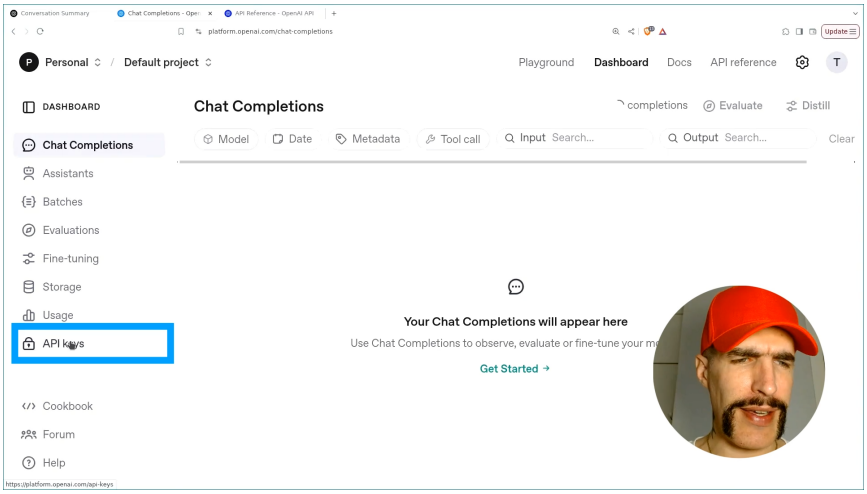
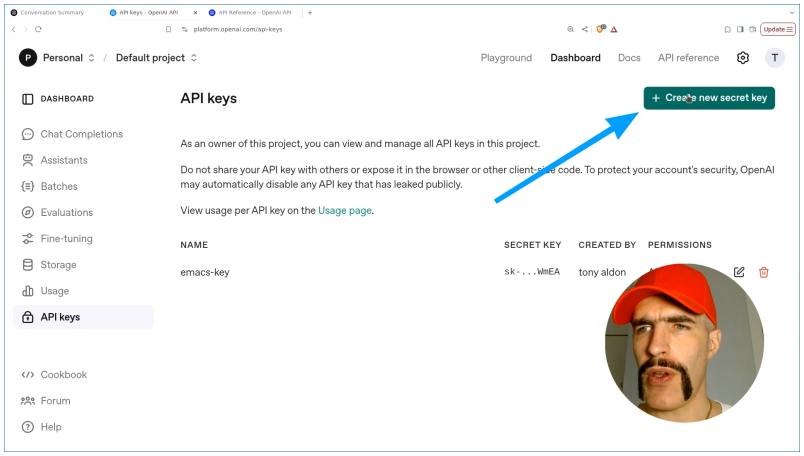
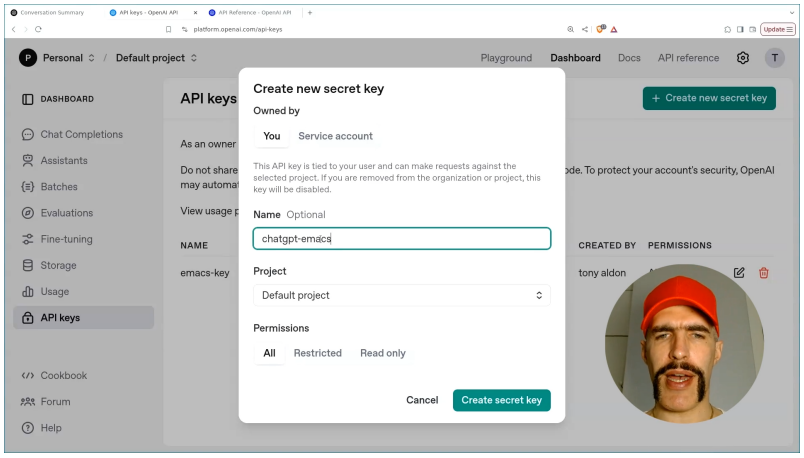
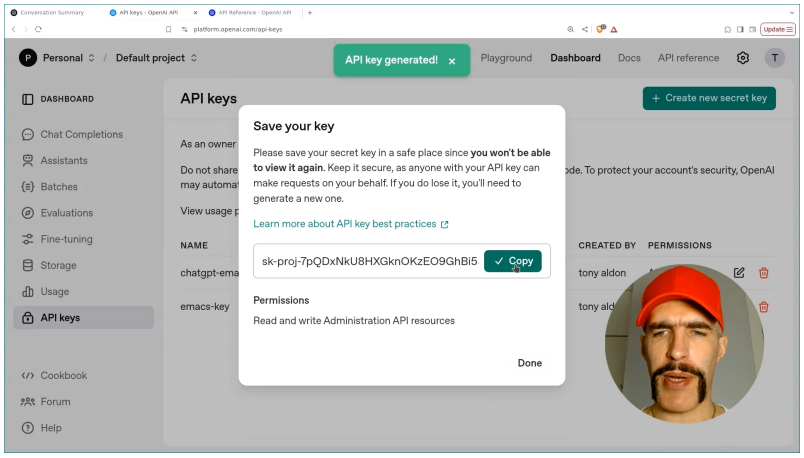


Figure 7: API Keys Menu


Personal / Default project
Playground
Dashboard
Docs
API reference
DASHBOARD
Chat Completions
Assistants
Batches
Evaluations
Fine-tuning
Storage
Usage
API keys
Cookbook
Forum
Help
API keys
+ Create new secret key
As an owner of this project, you can view and manage all API keys in this project.
Do not share your API key with others or expose it in the browser or other client-side code. To protect your account's security, OpenAI may automatically disable any API key that has leaked publicly.
View usage per API key on the Usage page.
NAME
SECRET KEY
CREATED BY
PERMISSIONS
emacs-key
sk-...WmEA
tony aldon


Personal / Default project
Playground
Dashboard
Docs
API reference
API keys
+ Create new secret key
Create new secret key
Owned by
You
Service account
This API key is tied to your user and can make requests against the selected project. If you are removed from the organization or project, this key will be disabled.
Name Optional
chatgpt-emacs
Project
Default project
Permissions
All
Restricted
Read only
Cancel
Create secret key
CREATED BY
PERMISSIONS
tony aldon


Personal / Default project
API key generated!
Playground
Dashboard
Docs
API reference
API keys
+ Create new secret key
Save your key
Please save your secret key in a safe place since you won't be able to view it again. Keep it secure, as anyone with your API key can make requests on your behalf. If you do lose it, you'll need to generate a new one.
Learn more about API key best practices
sk-proj-7pQDxNkU8HXGknOKzEO9GhBi5
Copy
Permissions
Read and write Administration API resources
Done
CREATED BY
PERMISSIONS
tony aldon

4. **Copy the API Key**: After the API key is generated, copy it for use in your project:

   `sk-proj-7pQDxN...w-D40A`

Make sure to store this API key securely and avoid sharing it publicly.

1.3 First Request to OpenAI Using the Chat Completion API

With our API key generated, we can now send requests to OpenAI. In this section, we will send the prompt "Hello!" and receive a response via the terminal using `curl`.

Referencing the Chat Completion API documentation, we will modify the default `curl` request (non-streaming) by substituting `$OPENAI_API_KEY` with our actual API key:

```
curl https://api.openai.com/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer sk-proj-7pQDxN...w-D40A" \
  -d '{
    "model": "gpt-4o",
    "messages": [
      {
        "role": "developer",
        "content": "You are a helpful assistant."
      },
      {
        "role": "user",
        "content": "Hello!"
      }
    ]
  }'
```

The `-d` option sends a JSON object containing two required parameters: `model` and `messages`. In this case, we use the `gpt-4o` model, and the `messages` array includes our prompt "Hello!".

The `messages` array contains two entries: one with the role `developer` and another with the role `user` with the content "Hello!".

Upon executing the request, we receive the following JSON response from OpenAI:

```
{
  "id": "chatcmpl-B2ENfnqI4JikQY1bE34dNmhPR0OnF",
  "object": "chat.completion",
  "created": 1739871635,
  "model": "gpt-4o-2024-08-06",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "Hello! How can I assist you today?",
        "refusal": null
      },
      "logprobs": null,
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 19,
    "completion_tokens": 10,
    "total_tokens": 29,
    "prompt_tokens_details": {
      "cached_tokens": 0,
      "audio_tokens": 0
    },
    "completion_tokens_details": {
      "reasoning_tokens": 0,
      "audio_tokens": 0,
      "accepted_prediction_tokens": 0,
      "rejected_prediction_tokens": 0
    }
  },
  "service_tier": "default",
  "system_fingerprint": "fp_523b9b6e5f"
}
```


This response contains details such as the completion ID, the model used, and the message generated by the assistant, along with usage statistics regarding token counts.

Specifically, the `choices` field in the response reveals the assistant's reply:

"Hello! How can I assist you today?"

This concludes our first lesson in the ChatGPT Emacs course. In upcoming lessons, we will delve deeper into the Chat Completion API, focusing on the streaming API and various message types that can be included in requests.

Link: `https://platform.openai.com/docs/api-reference/chat`

2 Chat Completion Streaming API

In this lesson, we will explore how to configure OpenAI to return responses as a continuous stream of data instead of a JSON object.

2.1 Curl Request Using a JSON File

To modify the `curl` request from the previous lesson, we will store the JSON payload in a file rather than including it directly in the command line. We will specify the full path to the JSON file as an argument in the `curl` command.

First, we create the JSON file at `/home/tony/chatgpt-emacs/request.json` with the following content:

```
{
  "model": "gpt-4o",
  "messages": [
    {
      "role": "developer",
      "content": "You are a helpful assistant."
    },
    {
      "role": "user",
      "content": "Hello!"
    }
  ]
}
```

To send the request using `curl`, we use the `@` symbol to reference the file:

```
curl https://api.openai.com/v1/chat/completions \
  -H "Content-Type: application/json" \
```

```
  -H "Authorization: Bearer sk-proj-7pQDxN...w-D40A" \
  -d @/home/tony/chatgpt-emacs/request.json
```

Upon sending this request to the OpenAI API, we receive the following response:


```
{
  "id": "chatcmpl-B32ndGQSiexNKDdi5WKvHmYLZOihy",
  "object": "chat.completion",
  "created": 1740065445,
  "model": "gpt-4o-2024-08-06",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "Hello! How can I assist you today?",
        "refusal": null
      },
      "logprobs": null,
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 19,
    "completion_tokens": 10,
    "total_tokens": 29,
    "prompt_tokens_details": {
      "cached_tokens": 0,
      "audio_tokens": 0
    },
    "completion_tokens_details": {
      "reasoning_tokens": 0,
      "audio_tokens": 0,
      "accepted_prediction_tokens": 0,
      "rejected_prediction_tokens": 0
    }
  },
  "service_tier": "default",
  "system_fingerprint": "fp_fee4aaf18f"
}
```

2.2 Chat Completion Streaming API

In previous interactions with the OpenAI Chat Completion API, we received responses formatted as JSON objects. This is standard behavior. To modify this and enable streaming responses, we need to adjust the request payload.

2.2.1 Updating the Request for Streaming

To enable streaming, we include the `stream` parameter set to `true` in our JSON request like this:

```
{
  "model": "gpt-4o",
  "messages": [
    {
      "role": "developer",
      "content": "You are a helpful assistant."
    },
    {
      "role": "user",
      "content": "Hello!"
    }
  ],
  "stream": true
}
```

2.2.2 Observing Streaming Responses

Once the updated request is sent to OpenAI, we start receiving data in chunks. Here's a sample of the streamed output:

```
data: {"id":"chatcmpl-B33LUkTZbglsA7jDBpkMhDuQd6Mxp",
"object":"chat.completion.chunk","created":1740067544
,"model":"gpt-4o-2024-08-06","service_tier":"default"
,"system_fingerprint":"fp_fee4a af18f","choices":[{"i
ndex":0,"delta":{"role":"assistant","content":"","ref
usal":null},"logprobs ":null,"finish_reason":null}]}

data: {"id":"chatcmpl-B33LUkTZbglsA7jDBpkMhDuQd6Mxp",
"object":"chat.completion.chunk","created":1740067544
,"model":"gpt-4o-2024-08-06","service_tier":"default"
```

```
,"system_fingerprint":"fp_fee4aaf18f","choices":[{"in
dex":0,"delta":{"content":"Hello"},"logprobs":null,"f
inish_reason":null}]}

... [additional chunks omitted for brevity] ...

data: {"id":"chatcmpl-B33LUkTZbglsA7jDBpkMhDuQd6Mxp","
object":"chat.completion.chunk","created" :1740067544,
"model":"gpt-4o-2024-08-06","service_tier":"default","
system_fingerprint":"fp_fee4aaf18f","choices":[{"index
":0,"delta":{},"logprobs":null,"finish_reason":"stop"}
]}

data: [DONE]
```

2.2.3 Response Breakdown

In the streamed response, data is sent in chunks. Each chunk may contain parts of the message like so:

```
data: {"id":"chatcmpl-B33LUkTZbglsA7jDBpkMhDuQd6Mxp",
..."choices":[{"...,"delta":{"content":" How"}...}]}

data: {"id":"chatcmpl-B33LUkTZbglsA7jDBpkMhDuQd6Mxp",
..."choices":[{"...,"delta":{"content":" can"}...}]}

data: {"id":"chatcmpl-B33LUkTZbglsA7jDBpkMhDuQd6Mxp",
..."choices":[{"...,"delta":{"content":" I"}...}]}

data: {"id":"chatcmpl-B33LUkTZbglsA7jDBpkMhDuQd6Mxp",
..."choices":[{"...,"delta":{"content":" assist"}...}]}
```

Each `delta` field corresponds to a portion of the generated response, progressively building the output.

3 Developer and System Messages

In this lesson, we will learn how to customize our requests to receive all responses from OpenAI in Spanish.

3.1 Developer and System Messages Overview

In our previous interactions with the OpenAI API, we primarily focused on the last message in the `messages` parameter, specifically the one from the `user` role containing the prompt. This approach overlooked the initial message designated as `developer`, which has been set to "you are a helpful assistant." We will adjust this `developer` message to instruct the model to respond in Spanish.

3.1.1 Understanding Developer and System Messages

According to the API reference, `developer` messages dictate instructions that the model must adhere to irrespective of the user's input. This ensures the model consistently follows specified guidelines.

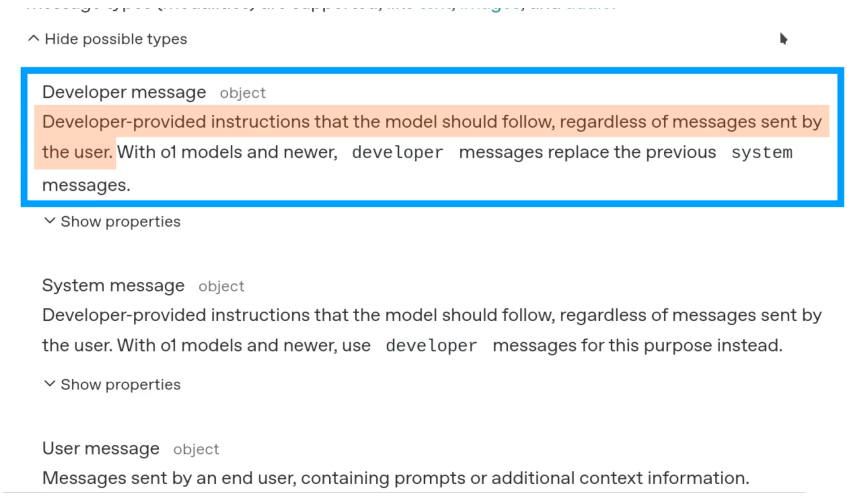


Figure 8: Developer Messages

The `system` messages apply to models such as `gpt-4o` and `gpt-4o-mini`, while models from `o1` onward utilize `developer` messages.

Link: `https://platform.openai.com/docs/api-reference/chat`

3.1.2 Modifying the Developer Instruction

To ensure our model replies in Spanish, we will update the `developer` message in the `/home/tony/chatgpt-emacs/request.json` file to state, "Reply in Spanish":

```
{
  "model": "gpt-4o",
  "messages": [
    {
      "role": "developer",
      "content": "Reply in Spanish."
    },
    {
      "role": "user",
      "content": "Hello!"
    }
  ]
}
```

3.1.3 API Request and Response

Upon submitting this request to the OpenAI API, we receive a response in Spanish:

```
{
  ...
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "¡Hola! ¿Cómo puedo ayudarte hoy?",
        "refusal": null
      },
      ...
    }
  ],
  ...
}
```

This implementation clearly illustrates how modifying the `developer` message influences the model's output, enabling consistent bilingual responses.

3.2 Replying in Spanish with More Constraints

We successfully configured the `gpt-4o` model to respond in Spanish by modifying the optional `developer` message. Although it can be omitted in requests, we will enhance it now by introducing additional constraints without conducting a tutorial on prompting.

Initially, we change the `developer` message to "Reply in Spanish with at least three sentences," resulting in the following JSON request:

```
{
  "model": "gpt-4o",
  "messages": [
    {
      "role": "developer",
      "content": "Reply in Spanish with at least 3 sentences."
    },
    {
      "role": "user",
      "content": "Hello!"
    }
  ]
}
```

The response generated met our criteria, providing three sentences as requested:

```
{
  ...
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "¡Hola! ¿Cómo estás? Espero que estés
teniendo un buen día. Si necesitas ayuda con algo o quie
res hablar sobre un tema en específico, estaré aquí para
ayudarte.",
        "refusal": null
      },
      ...
    }
```

```
  ],
  ...
}
```

Next, we introduce a new constraint specifying that we want the response in a list format. We update the `developer` message to "Reply in Spanish with at least three sentences in a list format" and submit the request:

```
{
  "model": "gpt-4o",
  "messages": [
    {
      "role": "developer",
      "content": "Reply in Spanish with at least 3 sentences
in a list format."
    },
    {
      "role": "user",
      "content": "Hello!"
    }
  ]
}
```

The response generated has been formatted as requested:

```
{
  ...
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "¡Hola!\n\n1. Espero que estés teniendo
un buen día. \n2. Si tienes alguna pregunta o necesitas ay
uda con algo, no dudes en decírmelo.\n3. Estoy aquí para a
yudarte en lo que necesites.",
        "refusal": null
      },
      ...
    }
```

```
  ],
  ...
}
```

In our next discussion, we will explore how to use `assistant` messages in the request data to facilitate multi-turn conversations.

4 Assistant Messages

In this lesson, we will explore how to utilize `assistant` messages for maintaining conversation contexts when interacting with the Chat Completion API, specifically with the `gpt-4o` model.

4.1 Independent Requests

It's crucial to understand that each request to the API is independent. Without context linking, subsequent requests cannot reference earlier ones. So, how can we maintain continuity in a conversation with models like `gpt-4o`? The answer lies in using `assistant` messages, which are the responses generated by the model to user queries. Including these messages in the `messages` array of our requests allows us to create a coherent dialogue.

4.2 Example Conversation

Let's illustrate this with a simple conversation. We start by sending a basic request with the `user` message "Hello!":

```
{
  "model": "gpt-4o",
  "messages": [
    {
      "role": "user",
      "content": "Hello!"
    }
  ]
}
```

In response, we receive an `assistant` message:

```
{
  "id": "chatcmpl-B3Kxois3s7jGJLIVLg6m5xPNQpmWg",
```

```
  "object": "chat.completion",
  "created": 1740135268,
  "model": "gpt-4o-2024-08-06",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "Hello! How can I assist you today?",
        "refusal": null
      },
      ...
    }
  ],
  ...
}
```

4.3 Continuing the Dialogue

Next, we take the assistant's message and include it in the `messages` array for the next request. We also add a new `user` message expressing hunger:

```
{
  "model": "gpt-4o",
  "messages": [
    {
      "role": "user",
      "content": "Hello!"
    },
    {
      "role": "assistant",
      "content": "Hello! How can I assist you today?"
    },
    {
      "role": "user",
      "content": "I'm hungry."
    }
  ]
}
```

When we send this request, the model responds as follows:

```
{
  ...
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "I'm here to help! What type of food are you
in the mood for? Are you thinking of making something at home,
or would you prefer to order out?",
        "refusal": null
      },
      ...
    }
  ],
  ...
}
```

4.4 Building Context

We continue to enhance the conversation by adding the latest `assistant` message before posing another user question, this time focusing on cooking at home:

```
{
  "model": "gpt-4o",
  "messages": [
    {
      "role": "user",
      "content": "Hello!"
    },
    {
      "role": "assistant",
      "content": "Hello! How can I assist you today?"
    },
    {
      "role": "user",
      "content": "I'm hungry."
    },
    {
```

```
      "role": "assistant",
      "content": "I'm here to help! What type of food are you
in the mood for? Are you thinking of making something at home,
or would you prefer to order out?"
    },
    {
      "role": "user",
      "content": "home"
    }
  ]
}
```

4.5 Final Response

Based on the entire conversation context, the model processes the latest input and replies:

```
{
  ...
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "Great! Do you have any specific ingredients
on hand, or any particular type of cuisine you're interested in?
Let me know so I can suggest a recipe!",
        "refusal": null
      },
      ...
    }
  ],
  ...
}
```

4.6 Conclusion

This demonstrates how to leverage assistant messages for interactive conversations with LLMs. In the next lesson, we will transition to writing our Emacs package.

5 The Basics of make-process

In the previous lesson, we explored how to send a request to OpenAI with a simple prompt, "Hello!", using the `curl` command in the terminal. Now, we aim to replicate this functionality in Emacs using an Emacs Lisp program, specifically focusing on executing the `curl` command asynchronously.

To achieve this, we will utilize the `make-process` function, which is designed for creating and managing asynchronous processes within Emacs. Our objective in this lesson is to understand the fundamentals of the `make-process` function and how to leverage it for our `curl` command execution.

Let's delve into the details of using `make-process` for this purpose.

5.1 Executing Commands with `make-process`

To execute the following command `echo foo bar baz` with the function `make-process`

```
$ echo foo bar baz
foo bar baz
```

we can evaluate the following expression

```
(make-process
 :name "foo-name"
 :buffer "foo-buff"
 :command (list "echo" "foo" "bar" "baz"))
```

which starts the program `echo` in a subprocess passing it the arguments `foo`, `bar`, `baz` and appends its standard output to the buffer `foo-buff` creating it if it doesn't exist:

```
foo bar baz

Process foo-name finished
```

Note that `Process foo-name finished` has been added by the default process sentinel function. We'll talk more about this later.

5.2 The Process Object

This section details functions that we can apply to process objects created by the `make-process` function.

First, we define the variable `foo-proc` to hold the process object returned by `make-process`, which executes the command `echo foo bar baz`.

```
(setq foo-proc
      (make-process
       :name "foo-name"
       :buffer "foo-buff"
       :command (list "echo" "foo" "bar" "baz")))
```

Next, we verify that `foo-proc` is indeed of the `process` type and has the name `foo-name`:

```
(type-of foo-proc) ;; process
(process-name foo-proc) ;; "foo-name"
```

We can retrieve the buffer associated with the `foo-proc` process using the `process-buffer` function, and check whether `foo-proc` is still alive with the `process-live-p` function:

```
(process-buffer foo-proc) ;; #<buffer foo-buff>
(process-live-p foo-proc) ;; nil
```

By following these steps, we can effectively manage and query the status of processes in Emacs Lisp.

5.3 Executing Commands with Pipes Using `make-process`

To run a command line that includes a pipe with the `make-process` function, we need to adapt our approach since pipe syntax is interpreted by the command shell, not as arguments to the program.

For example, this command outputs `foo` to `stdout` and returns after 2 seconds:

```
$ sleep 2 | echo foo
foo
```

However, using `make-process`, we cannot directly specify both `sleep` and `echo` as separate programs because the `:command` keyword only accepts a single program file.

Instead, we can leverage the `-c` option of the `sh` command, which allows us to execute the next argument as a command. We can format our command like this:

```
$ sh -c 'sleep 2 | echo foo'
foo
```

This effectively runs the original command by interpreting it as a complete shell command.

Finally, we can use `make-process` with that command like this:

```
(make-process
 :name "foo-name"
 :buffer "foo-buff"
 :command (list "sh" "-c" "sleep 2 | echo foo"))
```

When we evaluate the above expression, it produces the following output in the `foo-buff` buffer after 2 seconds:

```
foo

Process foo-name finished
```

This setup enables us to run complex command lines with pipes using `make-process` effectively.

5.4 Process Sentinel Overview

Finally, in the following 4 sections we will see how to transfer the contents of a process buffer into another buffer once the process finishes. This is achieved using process sentinel functions attached to the initiated process.

A sentinel function is triggered whenever there is a status change in the process. The first argument passed to the sentinel is the `process` itself, while the second argument describes the `event` that caused the status change.

For instance, consider the following code snippet:

```
(make-process
 :name "foo-name"
```

```
 :buffer "foo-buff"
 :command (list "sh" "-c" "sleep 1 | echo foo")
 :sentinel (lambda (process event)
             (message "%S - %S" process event)))
```

When this expression is evaluated, it will display the following message in the echo area after 1 second:

```
#<process foo-name> - "finished\n"
```

To observe a different event type, we can utilize the `kill-process` function, which terminates the process initiated by `make-process`. Here's how we can apply the same sentinel function in this context:

```
(kill-process
 (make-process
  :name "foo-name"
  :buffer "foo-buff"
  :command (list "sh" "-c" "sleep 1 | echo foo")
  :sentinel (lambda (process event)
              (message "%S - %S" process event))))
```

After evaluating this expression, we see the following output in the echo area after 1 second:

```
#<process foo-name> - "killed\n"
```

5.5 Branching on the Event Types in the Process Sentinel

We will now enhance the sentinel function to differentiate between successful process completion and unexpected terminations. If the process status changes to `"finished\n"` we print `OK` in the echo area. Conversely, for any other status, we display `Error`:

```
(make-process
 :name "foo-name"
 :buffer "foo-buff"
 :command (list "sh" "-c" "sleep 1 | echo foo")
 :sentinel (lambda (process event)
             (if (string= event "finished\n")
                 (message "OK")
               (message "Error"))))
```

5.6 Printing the Process Buffer Content in the Echo Area

Now, we can modify the sentinel function to capture the process buffer's content and display it in the echo area:

```
(make-process
 :name "foo-name"
 :buffer "foo-buff"
 :command (list "sh" "-c" "sleep 1 | echo foo")
 :sentinel
 (lambda (process event)
   (if (not (string= event "finished\n"))
       (message "Error")
     (let ((stdout (with-current-buffer (process-buffer process)
                     (buffer-string))))
       (message "%s" stdout)))))
```

Within the `let` binding, we use `with-current-buffer` to switch to the process's output buffer. This allows us to read its contents. `buffer-string` extracts the complete string output, which contains the string `foo`. The final `message` function outputs the captured string to the echo area, indicating the process's output.

5.7 Redirecting Process Buffer Content to Another Buffer

Finally we modify the previous code snippet to redirect the content of the process buffer into a new buffer named `bar`, rather than displaying it in the echo area. To achieve this, we utilize the `get-buffer-create` function to create the `bar` buffer. We then use the `insert` function to place the output from the process buffer into the `bar` buffer:

```
(make-process
 :name "foo-name"
 :buffer "foo-buff"
 :command (list "sh" "-c" "sleep 1 | echo foo")
 :sentinel
 (lambda (process event)
   (if (not (string= event "finished\n"))
       (message "Error")
     (let ((stdout (with-current-buffer (process-buffer process)
                     (buffer-string))))
```

```
      (with-current-buffer (get-buffer-create "bar")
        (insert stdout))))))
```

6 First Request To OpenAI From Emacs Lisp

Until now, we have explored the use of the OpenAI Chat Completion API with `curl` and how to execute command line operations in Emacs Lisp using the `make-process` function. Leveraging this knowledge, we can now create our first Emacs command `chatgtp-send` which sends the prompt "Hello!" to OpenAI and appends the JSON response to a buffer.

6.1 Review of The Last Lesson

Specifically, in the previous lesson, we implemented an expression that invokes the `make-process` function. This function starts an asynchronous process to execute a command that outputs `foo` to the standard output and completes after 1 second.

We also defined the associated sentinel function such that if the process completes successfully, its standard output is appended to the buffer `bar`. Conversely, if the process is terminated prematurely, such as being killed, the message `Error` will be displayed in the echo area.

```
(make-process
 :name "foo-name"
 :buffer "foo-buff"
 :command (list "sh" "-c" "sleep 1 | echo foo")
 :sentinel
 (lambda (process event)
   (if (not (string= event "finished\n"))
       (message "Error")
     (let ((stdout (with-current-buffer (process-buffer process)
                     (buffer-string))))
       (with-current-buffer (get-buffer-create "bar")
         (insert stdout))))))
```

6.2 Renaming The Process Name and Process Buffers

To rename the process from `foo-name` to `chatgpt`, the name of our package, we use the following code:

```
(make-process
 :name "chatgpt"
 ...)
```

Note that if a process named `chatgpt` already exists, the `make-process` function will generate a unique identifier for the new process to avoid naming collisions.

Currently, we are using the same process buffer `foo-buff` for all requests. This single buffer approach poses challenges for sending concurrent requests and managing JSON responses from OpenAI. To address this, we will generate unique buffer names for each request using `generate-new-buffer-name`:

```
(make-process
 ...
 :buffer (generate-new-buffer-name "chatgpt")
 ...)
```

Next, instead of redirecting OpenAI JSON responses to the buffer `bar` (currently, responses are the output `foo` of the command `sleep 1 | echo foo`) we will direct them to the buffer `*chatgpt[requests]*`. Additionally, rather than inserting the response at the current point, we will append it by moving the point to the end of the buffer using `(goto-char (point-max))` before any insertion:

```
(make-process
 :name "chatgpt"
 :buffer (generate-new-buffer-name "chatgpt")
 :command (list "sh" "-c" "sleep 1 | echo foo")
 :sentinel
 (lambda (process event)
   (if (not (string= event "finished\n"))
       (message "Error")
     (let ((stdout (with-current-buffer (process-buffer process)
                     (buffer-string))))
       (with-current-buffer (get-buffer-create "*chatgpt[requests]*")
         (goto-char (point-max))
         (insert stdout))))))
```

This implementation allows us to effectively manage multiple requests and handle the responses in an orderly manner.

6.3 Killing The Process Buffers

In our implementation, we utilize a unique process buffer for each invocation of `make-process`, which can lead to the accumulation of numerous inactive buffers if not properly managed. To prevent this, we ensure that once processing is complete, we kill these buffers using the `kill-buffer` function with the expression `(kill-buffer (process-buffer process))`.

```
(make-process
 :name "chatgpt"
 :buffer (generate-new-buffer-name "chatgpt")
 :command (list "sh" "-c" "sleep 1 | echo foo")
 :sentinel
 (lambda (process event)
   (if (not (string= event "finished\n"))
       (message "Error")
     (let ((stdout (with-current-buffer (process-buffer process)
                     (buffer-string))))
       (with-current-buffer (get-buffer-create "*chatgpt[requests]*")
         (goto-char (point-max))
         (insert stdout)))
     (kill-buffer (process-buffer process)))))
```

6.4 Sending our First OpenAI Request from Emacs Lisp

In this section, we'll send our first request to the OpenAI API using Emacs Lisp. To do so, we modify the previous `make-process` calls by replacing the original command `sleep 1 | echo foo` with the `curl` command we wrote in the lesson 2 that sends the JSON request found in the file `/home/tony/chatgpt-emacs/request.json`:

```
{
  "model": "gpt-4o",
  "messages": [
    {
      "role": "user",
      "content": "Hello!"
    }
  ]
}
```

Here's the updated Emacs Lisp code:

```
(let ((command (concat "curl https://api.openai.com/v1/chat/completions "
                       "-H 'Content-Type: application/json' "
                       "-H 'Authorization: Bearer sk-proj-7pQDxN...w-D40A "
                       "-d @/home/tony/chatgpt-emacs/request.json")))
  (make-process
   :name "chatgpt"
   :buffer (generate-new-buffer-name "chatgpt")
   :command (list "sh" "-c" command)
   :sentinel
   (lambda (process event)
     (if (not (string= event "finished\n"))
         (message "Error")
       (let ((stdout (with-current-buffer (process-buffer process)
                       (buffer-string))))
         (with-current-buffer (get-buffer-create "*chatgpt[requests]*")
           (goto-char (point-max))
           (insert stdout)))
       (kill-buffer (process-buffer process))))))
```

Evaluating the previous expression appends the following JSON response from OpenAI in `*chatgpt[requests]*` buffer:


```
{
  "id": "chatcmpl-B59xiKIxwA1xwxTytSsrcTXBO7CkH",
  "object": "chat.completion",
  "created": 1740569634,
  "model": "gpt-4o-2024-08-06",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "Hello! How can I assist you today?",
        "refusal": null
      },
      "logprobs": null,
      "finish_reason": "stop"
    }
  ],
  "usage": {
```

```
    "prompt_tokens": 9,
    "completion_tokens": 10,
    "total_tokens": 19,
    "prompt_tokens_details": {
      "cached_tokens": 0,
      "audio_tokens": 0
    },
    "completion_tokens_details": {
      "reasoning_tokens": 0,
      "audio_tokens": 0,
      "accepted_prediction_tokens": 0,
      "rejected_prediction_tokens": 0
    }
  },
  "service_tier": "default",
  "system_fingerprint": "fp_eb9dce56a8"
}
```

6.5 Defining `chatgpt-send` Command in chatgpt.el File

Finally, we write the first command for our `chatgpt` package in `chatgpt.el` file, naming it `chatgpt-send`. This command sends a request to OpenAI with the prompt "Hello!" fetched from the file `/home/tony/chatgpt-emacs/request.json`.

```
;;; chatgpt.el --- Simple ChatGPT integration -*- lexical-binding: t; -*-

(defun chatgpt-send ()
  "Send the request \"Hello!\" to OpenAI."
  (interactive)
  (let ((command (concat "curl https://api.openai.com/v1/chat/completions "
                         "-H 'Content-Type: application/json' "
                         "-H 'Authorization: Bearer sk-proj-7pQDxN...w-D40A "
                         "-d @/home/tony/chatgpt-emacs/request.json")))
    (make-process
     :name "chatgpt"
     :buffer (generate-new-buffer-name "chatgpt")
     :command (list "sh" "-c" command)
     :sentinel
     (lambda (process event)
```

```
      (if (not (string= event "finished\n"))
          (message "Error")
        (let ((stdout (with-current-buffer (process-buffer process)
                        (buffer-string))))
          (with-current-buffer (get-buffer-create "*chatgpt[requests]*")
            (goto-char (point-max))
            (insert stdout)))
        (kill-buffer (process-buffer process)))))))

(provide 'chatgpt)
```

Next, we will refactor the `curl` command into a separate function for better organization.

7 Refactoring chatgpt-send and introducing chatgpt-api-key

In this section, we will refactor the `chatgpt-send` function. We will create a new function named `chatgpt-command` to encapsulate the `curl` command logic. Additionally, we will introduce a variable called `chatgpt-api-key` to store our OpenAI API key securely.

7.1 Refactoring `chatgpt-send` with `chatgpt-command`

We introduce the `chatgpt-command` function, which generates the `curl` command string. This allows us to streamline `chatgpt-send` by replacing the existing command code with a call to `chatgpt-command`.

```
(defun chatgpt-command ()
  "Return the curl command."
  (concat "curl https://api.openai.com/v1/chat/completions "
          "-H 'Content-Type: application/json' "
          "-H 'Authorization: Bearer sk-proj-7pQDxN...w-D40A "
          "-d @/home/tony/chatgpt-emacs/request.json"))

(defun chatgpt-send ()
  "Send the request \"Hello!\" to OpenAI."
  (interactive)
  (let ((command (chatgpt-command)))
    (make-process
```

```
    :name "chatgpt"
    :buffer (generate-new-buffer-name "chatgpt")
    :command (list "sh" "-c" command)
    :sentinel (lambda (process event) ...))))
```

After implementing these changes and evaluating these expressions, calling `chatgpt-send` command should successfully populate the buffer `*chatgpt[requests]*` with a new response from OpenAI, confirming that the functionality remains intact.

7.2 Introducing `chatgpt-api-key` to hold OpenAI API Key

In this update, we refactor the hardcoded API key previously embedded within the `chatgpt-command` function. Instead, we store it in a new variable, `chatgpt-api-key`. This approach allows for easier future modifications to the API key. And in upcoming lessons, we will explore a more secure method for storing the API key, rather than embedding it directly in the source code.

7.2.1 Updated Code

```
(defvar chatgpt-api-key "sk-proj-7pQDxN...w-D40A"
  "Variable to hold the OpenAI API key.")

(defun chatgpt-command ()
  "Construct and return the curl command for OpenAI API."
  (format
   (concat "curl https://api.openai.com/v1/chat/completions "
           "-H 'Content-Type: application/json' "
           "-H 'Authorization: Bearer %s' "
           "-d @/home/tony/chatgpt-emacs/request.json")
   chatgpt-api-key))
```

7.2.2 Testing with an Incorrect API Key

For testing purposes, we can temporarily set the `chatgpt-api-key` variable to an invalid key, for example, `foo-api-key`. When invoking the `chatgpt-send` command, the OpenAI API responded with an error message displayed in the `*chatgpt[requests]*` buffer:

```
{
    "error": {
```

```
        "message": "Incorrect API key provided: foo-api-key.
You can find your API key at https://platform.openai.com/acc
ount/api-keys.",
        "type": "invalid_request_error",
        "param": null,
        "code": "invalid_api_key"
    }
}
```

This demonstrates that the API key is validated properly by OpenAI.

7.3 Updating `chatgpt-command` Function Signature

Finaly, we modify the `chatgpt-command` function to accept the absolute path to the `request.json` file as an argument. This change allows for easier adjustments to the file path in the future.

```
(defun chatgpt-command (req-path)
  "Return the curl command."
  (format
   (concat "curl https://api.openai.com/v1/chat/completions "
           "-H 'Content-Type: application/json' "
           "-H 'Authorization: Bearer %s' "
           "-d @%s")
   chatgpt-api-key req-path))

(defun chatgpt-send ()
  "Send the request \"Hello!\" to OpenAI."
  (interactive)
  (let ((command (chatgpt-command "/home/tony/chatgpt-emacs/request.json")))
    (make-process
     :name "chatgpt"
     :buffer (generate-new-buffer-name "chatgpt")
     :command (list "sh" "-c" command)
     :sentinel (lambda (process event) ...))))
```

For testing, we can pass a non-existent file path, such as `"/home/tony/chatgpt-emacs/request"`, to the `chatgpt-command` function in `chatgpt-send`. When invoking the `chatgpt-send` command, the `curl` command produced warnings, and the OpenAI API returned an error message, both of which were displayed in the `*chatgpt[requests]*` buffer:

```
Warning: Couldn't read data from file "/home/tony/chatgpt-emacs/request",
Warning: this makes an empty POST.
{
    "error": {
        "message": "We could not parse the JSON body of your request.
(HINT: This likely means you aren't using your HTTP library correctly. The
OpenAI API expects a JSON payload, but what was sent was not valid JSON.
If you have trouble figuring out how to fix this, please contact us through
our help center at help.openai.com.)",
        "type": "invalid_request_error",
        "param": null,
        "code": null
    }
}
```

8 Making the Prompt Dynamic in Requests

In this lesson, we will enhance the `chatgpt-send` function to facilitate sending requests with prompts entered in a buffer.

8.1 Writing a JSON Object to a File

In the following sections, we will learn how to create a `request.json` file containing the JSON request sent to OpenAI.

First, we ensure to require the built-in `json` package:

```
(require 'json)
```

Next, we use the `json-encode` function to encode an Emacs Lisp object into a JSON string:

```
(json-encode '(:foo "bar")) ;; "{\"foo\":\"bar\"}"
```

We Utilize the `write-region` function to write the JSON string to a file:

```
(write-region (json-encode '(:foo "bar"))
              nil "/home/tony/chatgpt-emacs/request-1.json")
```

This will create the file `/home/tony/chatgpt-emacs/request-1.json` containing:

```
{"foo":"bar"}
```

8.2 Writing the OpenAI Request to a File

We generate the following JSON request

```
{
  "model": "gpt-4o",
  "messages": [
    {
      "role": "user",
      "content": "Hello!"
    }
  ]
}
```

from Emacs Lisp. We bind it to a variable `req`, and write it to disk using `json-encode` and `write-region`:

```
(let ((req `(:model "gpt-4o"
             :messages ,(vector `(:role "user" :content "Hello!")))))
  (write-region (json-encode req)
                nil "/home/tony/chatgpt-emacs/request-1.json"))
```

Evaluating this expression updates the file `/home/tony/chatgpt-emacs/request-1.json` with:

```
{"model":"gpt-4o","messages":[{"role":"user","content":"Hello!"}]}
```

To pretty-print the JSON request, we set `json-encoding-pretty-print` to `t`:

```
(let ((json-encoding-pretty-print t)
      (req `(:model "gpt-4o"
             :messages ,(vector `(:role "user" :content "Hello!")))))
  (write-region (json-encode req)
                nil "/home/tony/chatgpt-emacs/request-1.json"))
```

Evaluating this expression yields the contents:

```
{
  "model": "gpt-4o",
  "messages": [
    {
```

```
      "role": "user",
      "content": "Hello!"
    }
  ]
}
```

We can further enhance our code's flexibility by allowing prompts to change. Let's set `prompt` to `"foo bar baz"`:

```
(let* ((json-encoding-pretty-print t)
       (prompt "foo bar baz")
       (req `(:model "gpt-4o"
              :messages ,(vector `(:role "user" :content ,prompt)))))
  (write-region (json-encode req)
                nil "/home/tony/chatgpt-emacs/request-1.json"))
```

This writes the following to `/home/tony/chatgpt-emacs/request-1.json`:

```
{
  "model": "gpt-4o",
  "messages": [
    {
      "role": "user",
      "content": "foo bar baz"
    }
  ]
}
```

Lastly, we introduce `req-path` to replace the hard-coded file path:

```
(let* ((json-encoding-pretty-print t)
       (prompt "Hello!")
       (req `(:model "gpt-4o"
              :messages ,(vector `(:role "user" :content ,prompt))))
       (req-path "/home/tony/chatgpt-emacs/request-1.json"))
  (write-region (json-encode req) nil req-path))
```

Evaluating this will again result in the file being updated to:

```
{
  "model": "gpt-4o",
```

```
  "messages": [
    {
      "role": "user",
      "content": "Hello!"
    }
  ]
}
```

8.3 Updating `chatgpt-send`

Now we can integrate the above functionality into `chatgpt-send`:

```
(defun chatgpt-send ()
  "Send a request to OpenAI."
  (interactive)
  (let* ((json-encoding-pretty-print t)
         (prompt "Hello!")
         (req `(:model "gpt-4o"
                :messages ,(vector `(:role "user" :content ,prompt))))
         (req-path "/home/tony/chatgpt-emacs/request.json")
         (command (chatgpt-command req-path)))
    (write-region (json-encode req) nil req-path)
    (make-process
     :name "chatgpt"
     :buffer (generate-new-buffer-name "chatgpt")
     :command (list "sh" "-c" command)
     :sentinel (lambda (process event) ...))))
```

After redefining `chatgpt-send`, calling it sends the JSON request in `/home/tony/chatgpt-emacs/request.json` to OpenAI, resulting in the following response in the `*chatgpt[requests]*` buffer:

```
{
  "id": "chatcmpl-B5v9nIsqd7sclUBOXJjrsXugrTWs1",
  "object": "chat.completion",
  "created": 1740751051,
  "model": "gpt-4o-2024-08-06",
  "choices": [
    {
      "index": 0,
      "message": {
```

```
        "role": "assistant",
        "content": "Hi there! How can I assist you today?",
        "refusal": null
      },
      ...
    }
  ],
  ...
}
```

8.4 Entering the Prompt from a Buffer

Finally, we modify `chatgpt-send` to use the current buffer content as the prompt. We replace the hardcoded prompt `"Hello!"` with a call to `buffer-string`:

```
(defun chatgpt-send ()
  "Send a request to OpenAI."
  (interactive)
  (let* ((json-encoding-pretty-print t)
         (prompt (buffer-string))
         (req `(:model "gpt-4o"
                :messages ,(vector `(:role "user" :content ,prompt))))
         (req-path "/home/tony/chatgpt-emacs/request.json")
         (command (chatgpt-command req-path)))
    (write-region (json-encode req) nil req-path)
    (make-process
     :name "chatgpt"
     :buffer (generate-new-buffer-name "chatgpt")
     :command (list "sh" "-c" command)
     :sentinel ...)))
```

Now, in a new buffer named `*chatgpt*`, if we enter the prompt `I'm hungry` and call `chatgpt-send`, we received the following response from OpenAI:

```
{
  "id": "chatcmpl-B5vaCM7XPe9sCdfeE0iNDjU0RiMuf",
  "object": "chat.completion",
  "created": 1740752688,
  "model": "gpt-4o-2024-08-06",
  "choices": [
```

```
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "What are you in the mood for? I can suggest
recipes,  snacks, or nearby restaurants if you let me know your
preferences!",
        "refusal": null
      },
      ...
    }
  ],
  ...
}
```

By following these steps, we have successfully made the prompt dynamic, enhancing the functionality of `chatgpt-send`.

9 Formatting Requests and Responses in Markdown

This lesson discusses how to format requests and responses using Markdown.

9.1 Parsing and Returning a JSON Object with `json-read`

To format responses in Markdown, we first need to parse the JSON received from OpenAI using the `json-read` function. This function parses and returns the JSON object at the current point in the buffer.

For example, if the point is at the start of a buffer containing the following partial JSON response from OpenAI

```
{
  "id": "chatcmpl-B5v9nIsqd7sclUBOXJjrsXugrTWs1",
  "model": "gpt-4o-2024-08-06",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "Hi there! How can I assist you today?",
        "refusal": null
```

```
      },
      "logprobs": null,
      "finish_reason": "stop"
    }
  ]
}
```

evaluating `(json-read)` in the minibuffer (using `pp-eval-expression`) yields the following output in the `*Pp Eval Output*` buffer:

```
((id . "chatcmpl-B5v9nIsqd7sclUBOXJjrsXugrTWs1")
 (model . "gpt-4o-2024-08-06")
 (choices .
          [((index . 0)
            (message
             (role . "assistant")
             (content . "Hi there! How can I assist you today?")
             (refusal))
            (logprobs)
            (finish_reason . "stop"))]))
```

By adjusting the variable bindings, we can alter the object type returned by `json-read`. For instance, using the following expression

```
(let ((json-key-type 'keyword)
      (json-object-type 'plist)
      (json-array-type 'vector))
  (json-read))
```

produces:

```
(:id "chatcmpl-B5v9nIsqd7sclUBOXJjrsXugrTWs1"
 :model "gpt-4o-2024-08-06"
 :choices
 [(:index 0
   :message (:role "assistant"
             :content "Hi there! How can I assist you today?"
             :refusal nil)
   :logprobs nil
   :finish_reason "stop")])
```

We can standardize this parsing with a utility function, `chatgpt-json-read`:

```
(defun chatgpt-json-read ()
  (let ((json-key-type 'keyword)
        (json-object-type 'plist)
        (json-array-type 'vector))
    (json-read)))
```

9.2 Accessing Elements in a Nested Structure with `map-nested-elt`

To retrieve the prompt from the JSON response, we can use the `map-nested-elt` function as shown below:

```
(let ((resp '(:id "chatcmpl-B5v9nIsqd7sclUBOXJjrsXugrTWs1"
              :model "gpt-4o-2024-08-06"
              :choices
              [(:index 0
                :message (:role "assistant"
                          :content "Hi there! How can I assist you today?"
                          :refusal nil)
                :logprobs nil
                :finish_reason "stop")])))
  (map-nested-elt resp [:choices 0 :message :content]))
;; "Hi there! How can I assist you today?"
```

9.3 Inserting the Assistant Response instead of the JSON Response

We will modify the `chatgpt-send` function so that it only appends the content string from the JSON response to the `*chatgpt[requests]*` buffer, rather than the entire JSON structure.

Previously, the sentinel function would append the JSON response as a raw string:

```
(lambda (process event)
  (if (not (string= event "finished\n"))
      (message "Error")
    (let ((stdout (with-current-buffer (process-buffer process)
                    (buffer-string))))
      (with-current-buffer (get-buffer-create "*chatgpt[requests]*")
```

```
        (goto-char (point-max))
        (insert stdout)))
    (kill-buffer (process-buffer process))))
```

Now, instead of treating the JSON response as a string, we will parse it using `chatgpt-json-read` and bind the parsed result to the `resp` variable:

```
(lambda (process event)
  (if (not (string= event "finished\n"))
      (message "Error")
    (let ((resp (with-current-buffer (process-buffer process)
                  (goto-char (point-min))
                  (chatgpt-json-read))))
      (with-current-buffer (get-buffer-create "*chatgpt[requests]*")
        (goto-char (point-max))
        (insert (format "%s\n" resp))))
    (kill-buffer (process-buffer process))))
```

This modification allows us to see the following response in the `*chatgpt[requests]*` buffer after sending a request to OpenAI:

```
(:id chatcmpl-B7IV75YO91840ovOTyUsug9pny371 :object chat.comple
tion :created 1741079113 :model gpt-4o-2024-08-06 :choices [(:i
ndex 0 :message (:role assistant :content Hi there! How can I a
ssist you today? :refusal nil) :logprobs nil :finish_reason sto
p)] :usage (:prompt_tokens 9 :completion_tokens 11 :total_token
s 20 :prompt_tokens_details (:cached_tokens 0 :audio_tokens 0)
:completion_tokens_details (:reasoning_tokens 0 :audio_tokens 0
:accepted_prediction_tokens 0 :rejected_prediction_tokens 0)) :
service_tier default :system_fingerprint fp_eb9dce56a8)
```

We then extract the assistant's response using `map-nested-elt`:

```
(lambda (process event)
  (if (not (string= event "finished\n"))
      (message "Error")
    (let* ((resp (with-current-buffer (process-buffer process)
                   (goto-char (point-min))
                   (chatgpt-json-read)))
           (response (map-nested-elt resp [:choices 0 :message :content])))
      (with-current-buffer (get-buffer-create "*chatgpt[requests]*")
```

```
      (goto-char (point-max))
      (insert response)))
  (kill-buffer (process-buffer process))))
```

Now, after sending a request to OpenAI, the `*chatgpt[requests]*` buffer shows the following:

```
Hello! How can I assist you today?
```

Note that in the previous code snippet the `let` expression has been changed to `let*` to have `resp` bound when we use it to bind `response` variable.

9.4 Formatting with `markdown-mode`

Next, we insert both the prompt and the response in the `*chatgpt[requests]*` buffer and format it using Markdown.

First, we ensure that we require markdown-mode:

```
(require 'markdown-mode)
```

Since `markdown-mode` is not built-in, install it using your preferred method.

Before appending content to the `*chatgpt[requests]*` buffer, we activate `markdown-mode`. Below is the updated sentinel function:

```
(lambda (process event)
  (if (not (string= event "finished\n"))
      (message "Error")
    (let* ((resp (with-current-buffer (process-buffer process)
                   (goto-char (point-min))
                   (chatgpt-json-read)))
           (response (map-nested-elt resp [:choices 0 :message :content])))
      (with-current-buffer (get-buffer-create "*chatgpt[requests]*")
        (markdown-mode)
        (goto-char (point-max))
        (insert "# Request\n\n"
                "## Prompt\n\n" prompt "\n\n"
                "## Response\n\n" response "\n\n")))
    (kill-buffer (process-buffer process))))
```

After sending a request with the prompt "Hello!", we get the following output appended to the `*chatgpt[requests]*` buffer in Markdown format:

```
# Request

## Prompt

Hello!

## Response

Hello! How can I assist you today?
```

Note that in previous code snippet the `prompt` variable is bound in an outer `let` in the `chatgpt-send` function.

Here is the current definition of the `chatgpt-send` function:

```
(defun chatgpt-send ()
  "Send a request to OpenAI."
  (interactive)
  (let* ((json-encoding-pretty-print t)
         (prompt (buffer-string))
         (req `(:model "gpt-4o"
                :messages ,(vector `(:role "user" :content ,prompt))))
         (req-path "/home/tony/chatgpt-emacs/request.json")
         (command (chatgpt-command req-path)))
    (write-region (json-encode req) nil req-path)
    (make-process
     :name "chatgpt"
     :buffer (generate-new-buffer-name "chatgpt")
     :command (list "sh" "-c" command)
     :sentinel
     (lambda (process event)
       (if (not (string= event "finished\n"))
           (message "Error")
         (let* ((resp (with-current-buffer (process-buffer process)
                        (goto-char (point-min))
                        (chatgpt-json-read)))
                (response (map-nested-elt resp [:choices 0 :message :content])))
           (with-current-buffer (get-buffer-create "*chatgpt[requests]*")
             (markdown-mode)
             (goto-char (point-max))
             (insert "# Request\n\n"
```

```
                "## Prompt\n\n" prompt "\n\n"
                "## Response\n\n" response "\n\n")))
        (kill-buffer (process-buffer process)))))))
```

Link: `https://github.com/jrblevin/markdown-mode`

10 Saving Requests to Disk

In this lesson, we will modify our code to save all requests and their corresponding responses to a specified directory instead of only overriding the requests sent to OpenAI.

10.1 Refactoring `chatgpt-send` with `chatgpt-request`

To improve clarity and organization, we refactor our code to separate the request generation process into a dedicated function, `chatgpt-request`.

```
(defun chatgpt-request (prompt)
  "Return an OpenAI request with PROMPT."
  `(:model "gpt-4o"
    :messages ,(vector `(:role "user" :content ,prompt))))

(defun chatgpt-send ()
  "Send a request to OpenAI."
  (interactive)
  (let* ((json-encoding-pretty-print t)
         (prompt (buffer-string))
         (req (chatgpt-request prompt))
         (req-path "/home/tony/chatgpt-emacs/request.json")
         (command (chatgpt-command req-path)))
    (write-region (json-encode req) nil req-path)
    (make-process ...)))
```

10.2 Refactoring `chatgpt-send` with `chatgpt-callback`

Next, we enhance our `chatgpt-send` function by creating another function, `chatgpt-callback`, to manage appending prompts and responses to the `*chatgpt[requests]*` buffer.

```
(defun chatgpt-callback (prompt response)
  "Append PROMPT and RESPONSE to the prompt buffer."
  (with-current-buffer (get-buffer-create "*chatgpt[requests]*")
    (markdown-mode)
    (goto-char (point-max))
    (insert "# Request\n\n"
            "## Prompt\n\n" prompt "\n\n"
            "## Response\n\n" response "\n\n")))

(defun chatgpt-send ()
  "Send a request to OpenAI."
  (interactive)
  (let* (...)
    (write-region (json-encode req) nil req-path)
    (make-process
     ...
     :sentinel
     (lambda (process event)
       (if (not (string= event "finished\n"))
           (message "Error")
         (let* (...)
           (chatgpt-callback prompt response))
         (kill-buffer (process-buffer process)))))))
```

10.3 Saving Requests

In this section, we enhance the `chatgpt-send` command to save each request sent to OpenAI in a unique subdirectory along with its corresponding JSON response.

Before we modify the `chatgpt-send` function, let's review a couple of useful Emacs Lisp functions we'll employ later.

The `make-temp-file` function allows us to create unique subdirectories under the `temporary-file-directory`. For example:

```
(make-temp-file nil t) ;; "/tmp/5ukiOy"
```

This expression generates a subdirectory named `5ukiOy` within `/tmp/`, the current `temporary-file-directory`.

To ensure the path returned by `make-temp-file` ends with a forward slash, we can use the `file-name-as-directory` function:

```
(file-name-as-directory (make-temp-file nil t)) ;; "/tmp/1I39B7/"
```

Additionally, we can temporarily set the `temporary-file-directory` to a specified existing directory, allowing subdirectories to be relative to that path:

```
(make-directory "/tmp/foo/bar/" t)
(let ((temporary-file-directory "/tmp/foo/bar/"))
  (file-name-as-directory (make-temp-file nil t)))
;; "/tmp/foo/bar/LcMrzD/"
```

In this snippet, we used `make-directory` to create `/tmp/foo/bar/`, with the `t` parameter enabling the creation of any missing parent directories.

Now, let's integrate this into our package. We introduce the `chatgpt-dir` variable to specify the requests directory. Within the `chatgpt-send` function, we bind `temporary-file-directory` to `chatgpt-dir`. We then create a new subdirectory, assigning its path to the `req-dir` variable and defining `req-path` as the `request.json` file within `req-dir`. We also ensure the existence of the `chatgpt-dir`. Below is the updated `chatgpt-send` function:

```
(defvar chatgpt-dir "/home/tony/chatgpt-emacs/requests/"
  "Request directory.")

(defun chatgpt-send ()
  "Send a request to OpenAI."
  (interactive)
  (make-directory chatgpt-dir t)
  (let* (...
         (req (chatgpt-request prompt))
         (temporary-file-directory chatgpt-dir)
         (req-dir (file-name-as-directory (make-temp-file nil t)))
         (req-path (concat req-dir "request.json"))
         ...)
    (message "chatgpt: %s" req-dir)
    (write-region (json-encode req) nil req-path)
    (make-process ...)))
```

When invoking `chatgpt-send` with the prompt "Hello!", the request is saved in the file `/home/tony/chatgpt-emacs/requests/U4v02L/request.json` as follows:

```
{
  "model": "gpt-4o",
  "messages": [
    {
      "role": "user",
      "content": "Hello!"
    }
  ]
}
```

Additionally, the following prompt and response are appended to the `*chatgpt[requests]*` buffer:

```
# Request

## Prompt

Hello!

## Response

Hello! How can I assist you today?
```

10.4 Saving Responses

Next, we implement functionality to write the JSON responses to `response.json` files within the same directories as their corresponding requests.

To accomplish this, we adjust the sentinel function binding `resp-path` to point to the `response.json` file located in `req-dir`. The `resp` response is encoded and written similarly to the requests, using `json-encoding-pretty-print` set to `t` for improved readability.

```
(lambda (process event)
  (if (not (string= event "finished\n"))
      (message "Error")
    (let* ((resp (with-current-buffer (process-buffer process)
                   (goto-char (point-min))
                   (chatgpt-json-read)))
           (response (map-nested-elt resp [:choices 0 :message :content]))
           (resp-path (concat req-dir "response.json"))
           (json-encoding-pretty-print t))
```

```
      (write-region (json-encode resp) nil resp-path)
      (chatgpt-callback prompt response))
    (kill-buffer (process-buffer process))))
```

When executing `chatgpt-send` with the "Hello!" prompt, it stores the corresponding response in `/home/tony/chatgpt-emacs/requests/0xI4Yw/response.json`.

10.5 Refactoring `chatgpt-send` with `chatgpt-json-encode`

To streamline our encoding process, we define a new function, `chatgpt-json-encode`, to handle JSON encoding with pretty printing.

```
(defun chatgpt-json-encode (object)
  (let ((json-encoding-pretty-print t))
    (json-encode object)))

(defun chatgpt-send ()
  "Send a request to OpenAI."
  (interactive)
  (make-directory chatgpt-dir t)
  (let* (...)
    (message "chatgpt: %s" req-dir)
    (write-region (chatgpt-json-encode req) nil req-path)
    (make-process
     ...
     :sentinel
     (lambda (process event)
       (if (not (string= event "finished\n"))
           (message "Error")
         (let* (...)
           (write-region (chatgpt-json-encode resp) nil resp-path)
           (chatgpt-callback prompt response))
         (kill-buffer (process-buffer process)))))))
```

10.6 Adding Links to Request Directories

Finally, we update the `chatgpt-callback` function to include links to the request directories when appending prompts and responses to the buffer.

```
(defun chatgpt-callback (prompt response req-dir)
  "Append PROMPT and RESPONSE to the prompt buffer with a link to REQ-DIR."
```

```
  (with-current-buffer (get-buffer-create "*chatgpt[requests]*")
    (markdown-mode)
    (goto-char (point-max))
    (insert "# Request\n\n"
            "<!-- [](" req-dir ") -->\n\n"
            "## Prompt\n\n" prompt "\n\n"
            "## Response\n\n" response "\n\n")))
```

We then pass the `req-dir` argument correctly to `chatgpt-callback` function in the sentinel function:

```
(defun chatgpt-send ()
  ...
  (let* (...
         (temporary-file-directory chatgpt-dir)
         (req-dir (file-name-as-directory (make-temp-file nil t)))
         ...)
    ...
    (make-process
     ...
     :sentinel
     (lambda (process event)
       (if (not (string= event "finished\n"))
           (message "Error")
         (let* (...)
           (write-region (chatgpt-json-encode resp) nil resp-path)
           (chatgpt-callback prompt response req-dir))
         (kill-buffer (process-buffer process)))))))
```

When executing `chatgpt-send` with the prompt "Hello!", the buffer `*chatgpt[requests]*` includes a link to the request directory:

```
# Request

<!-- [](/home/tony/chatgpt-emacs/requests/Xb7cAy/) -->

## Prompt

Hello!
```

```
## Response

Hi there! How can I assist you today?
```

In this lesson, we have learned to save all requests and their corresponding responses to disk. In the next lesson, we will implement a command to display the prompt buffer at the bottom of the frame.

11 The Prompt Buffer

In this lesson, we will implement the `chatgpt` command to display and select the prompt buffer at the bottom of the frame. We will also define `chatgpt-mode` and apply it within the prompt buffer.

11.1 Displaying the Prompt Buffer with `chatgpt`

Previously, we utilized the buffer `*chatgpt*` to enter prompts for OpenAI. We will continue using this buffer but will replace the `switch-to-buffer` command with our new `chatgpt` command, which displays the buffer at the bottom of the frame.

```
(defun chatgpt ()
  "Display and select the prompt buffer."
  (interactive)
  (let ((buff (get-buffer-create "*chatgpt*")))
    (select-window
     (display-buffer-at-bottom
      buff '(display-buffer-below-selected (window-height . 6))))))
```

The `display-buffer-at-bottom` function opens the specified buffer at the bottom of the frame and returns the associated window. We then use `select-window` to select that window.

11.2 Defining `chatgpt-mode`

We derive `chatgpt-mode` from `markdown-mode` for the prompt buffer:

```
(define-derived-mode chatgpt-mode markdown-mode "ChatGPT"
  "ChatGPT mode.")
```

To activate `chatgpt-mode` in the prompt buffer, we invoke it within the `chatgpt` command after selecting the window:

```
(defun chatgpt ()
  "Display and select the prompt buffer."
  (interactive)
  (let ((buff (get-buffer-create "*chatgpt*")))
    (select-window
     (display-buffer-at-bottom
      buff '(display-buffer-below-selected (window-height . 6))))
    (chatgpt-mode)))
```

Upon calling `chatgpt`, the `*chatgpt*` buffer will appear at the bottom of the frame. By evaluating `major-mode` in the minibuffer (using the `eval-expression` command) we can verify that the active major mode is `chatgpt-mode`.

11.3 Introducing `chatgpt-model` Variable

Next, we introduce the `chatgpt-model` variable to use in the `chatgpt-request` function, replacing the previous hardcoded `gpt-4o` model string:

```
(defvar chatgpt-model "gpt-4o"
  "The OpenAI model to use.")

(defun chatgpt-request (prompt)
  "Return an OpenAI request with PROMPT."
  `(:model ,chatgpt-model
    :messages ,(vector `(:role "user" :content ,prompt)))
```

For example:

```
(let ((chatgpt-model "gpt-4o"))
  (chatgpt-request "foo"))
;; (:model "gpt-4o" :messages [(:role "user" :content "foo")])
(let ((chatgpt-model "gpt-4o-mini"))
  (chatgpt-request "foo"))
;; (:model "gpt-4o-mini" :messages [(:role "user" :content "foo")])
```

11.4 Mode Line of the Prompt Buffer

We enhance the mode line in the prompt buffer to display the currently selected `chatgpt-model`. To do so, we modify `chatgpt-mode` to set `mode-line-format` accordingly:

```
(define-derived-mode chatgpt-mode markdown-mode "ChatGPT"
  "Mode for interacting with ChatGPT."
  (setq mode-line-format
        '(" "
          mode-line-buffer-identification
          " "
          chatgpt-model
          " "
          mode-line-misc-info)))
```

Once `chatgpt-mode` is activated in the prompt buffer, the mode line will display:

```
*chatgpt*    gpt-4o
```

Note that the `mode-line-misc-info` variable allows other commands or minor modes to append additional information to the mode line via `global-mode-string`.

11.5 Executing `chatgpt-mode` Once

Let's enhance the existing `chatgpt` command to ensure that `chatgpt-mode` is invoked only when the `*chatgpt*` buffer is created for the first time. Currently, `chatgpt-mode` is executed every time we call the `chatgpt` function, which is unnecessary.

We modify the function to check for the buffer's existence using `get-buffer`. If the `*chatgpt*` buffer does not exist at the time `chatgpt` is called, we enable `chatgpt-mode`.

Here's the revised code:

```
(defun chatgpt ()
  "Display and Select the prompt buffer."
  (interactive)
  (let* ((buff-name "*chatgpt*")
```

```
        (buff-p (get-buffer buff-name))
        (buff (get-buffer-create buff-name)))
    (select-window
     (display-buffer-at-bottom
      buff '(display-buffer-below-selected (window-height . 6))))
    (when (not buff-p) (chatgpt-mode))))
```

This modification ensures that `chatgpt-mode` is only activated when creating the buffer for the first time.

11.6 Creating `chatgpt-dir` in `chatgpt-mode`

We can further improve our implementation by creating the `chatgpt-dir` in the `chatgpt-mode` definition instead of in the `chatgpt-send` function. This ensures the directory is created only once if it doesn't exist yet. So we remove the call to `make-directory` in `chatgpt-send` and add it to `chatgpt-mode`:

```
(define-derived-mode chatgpt-mode markdown-mode "ChatGPT"
  "ChatGPT mode."
  (setq mode-line-format
        '(" "
          mode-line-buffer-identification
          " "
          chatgpt-model
          " "
          mode-line-misc-info))
  (make-directory chatgpt-dir t))
```

11.7 Defining `chatgpt-mode-map` keymap

Lastly, we bind the `C-c C-c` key combination to the `chatgpt-send` command within the `chatgpt-mode-map` keymap:

```
(defvar chatgpt-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map (kbd "C-c C-c") 'chatgpt-send)
    map)
  "Keymap for `chatgpt-mode'.")
```

To apply these changes in our current Emacs session, we call the `chatgpt` command to access the prompt buffer and activate `chatgpt-mode`.

11.8 chatgpt.el

The current implementation of the `chatgpt.el` package is as follows:

```
;;; chatgpt.el --- Simple ChatGPT integration -*- lexical-binding: t; -*-

(require 'json)
(require 'markdown-mode)

(defvar chatgpt-api-key
  "sk-proj-7pQDxN...w-D40A"
  "OpenAI API key.")

(defvar chatgpt-dir "/home/tony/chatgpt-emacs/requests/"
  "Request directory.")

(defun chatgpt-json-read ()
  (let ((json-key-type 'keyword)
        (json-object-type 'plist)
        (json-array-type 'vector))
    (json-read)))

(defun chatgpt-json-encode (object)
  (let ((json-encoding-pretty-print t))
    (json-encode object)))

(defun chatgpt-command (req-path)
  "Return the curl command."
  (format
   (concat "curl https://api.openai.com/v1/chat/completions "
           "-H 'Content-Type: application/json' "
           "-H 'Authorization: Bearer %s' "
           "-d @%s")
   chatgpt-api-key req-path))

(defvar chatgpt-model "gpt-4o"
  "OpenAI model.")

(defun chatgpt-request (prompt)
  "Return an OpenAI request with PROMPT."
  `(:model ,chatgpt-model
```

```
    :messages ,(vector `(:role "user" :content ,prompt))))

(defun chatgpt-callback (prompt response req-dir)
  "Append PROMPT and RESPONSE to the prompt buffer with a link to REQ-DIR."
  (let ((buff (get-buffer-create "*chatgpt[requests]*")))
    (with-current-buffer buff
      (markdown-mode)
      (goto-char (point-max))
      (insert "# Request\n\n"
              "<!-- [](" req-dir ") -->\n\n"
              "## Prompt\n\n" prompt "\n\n"
              "## Response\n\n" response "\n\n"))))

(defun chatgpt-send ()
  "Send a request to OpenAI."
  (interactive)
  (let* ((prompt (buffer-string))
         (req (chatgpt-request prompt))
         (temporary-file-directory chatgpt-dir)
         (req-dir (file-name-as-directory (make-temp-file nil t)))
         (req-path (concat req-dir "request.json"))
         (command (chatgpt-command req-path)))
    (message "chatgpt: %s" req-dir)
    (write-region (chatgpt-json-encode req) nil req-path)
    (make-process
     :name "chatgpt"
     :buffer (generate-new-buffer-name "chatgpt")
     :command (list "sh" "-c" command)
     :sentinel
     (lambda (process event)
       (if (not (string= event "finished\n"))
           (message "Error")
         (let* ((resp (with-current-buffer (process-buffer process)
                        (goto-char (point-min))
                        (chatgpt-json-read)))
                (response (map-nested-elt resp [:choices 0 :message :content]))
                (resp-path (concat req-dir "response.json")))
           (write-region (chatgpt-json-encode resp) nil resp-path)
           (chatgpt-callback prompt response req-dir))
         (kill-buffer (process-buffer process)))))))
```

```
(defvar chatgpt-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map (kbd "C-c C-c") 'chatgpt-send)
    map)
  "Keymap of `chatgpt-mode'.")

(define-derived-mode chatgpt-mode markdown-mode "ChatGPT"
  "ChatGPT mode."
  (setq mode-line-format
        '(" "
          mode-line-buffer-identification
          " "
          chatgpt-model
          " "
          mode-line-misc-info))
  (make-directory chatgpt-dir t))

(defun chatgpt ()
  "Display and Select the prompt buffer."
  (interactive)
  (let* ((buff-name "*chatgpt*")
         (buff-p (get-buffer buff-name))
         (buff (get-buffer-create buff-name)))
    (select-window
     (display-buffer-at-bottom
      buff '(display-buffer-below-selected (window-height . 6))))
    (when (not buff-p) (chatgpt-mode))))

(provide 'chatgpt)
```

12 Making the response buffer pop up upon receipt

In this lesson, we will adjust the `chatgpt-send` and `chatgpt-callback` functions. The goal is to modify `chatgpt-send` so that it clears the prompt buffer upon sending a request, and to ensure that the `*chatgpt[requests]*` buffer displaying prompts and responses opens once a response is received from OpenAI.

12.1 Refactoring `chatgpt-send` into `chatgpt-send-request`

To implement these changes, we first refactor the `chatgpt-send` function into two distinct functions: `chatgpt-send` and `chatgpt-send-request`. The `chatgpt-send-request` function directly accepts the prompt as an argument, while the modified `chatgpt-send` function retrieves the prompt text from the current buffer and pass it to `chatgpt-send-request`.

```
(defun chatgpt-send-request (prompt)
  "Send the request with PROMPT to OpenAI."
  (let* ((req (chatgpt-request prompt))
         (temporary-file-directory chatgpt-dir)
         (req-dir (file-name-as-directory (make-temp-file nil t)))
         (req-path (concat req-dir "request.json"))
         (command (chatgpt-command req-path)))
    (message "chatgpt: %s" req-dir)
    (write-region (chatgpt-json-encode req) nil req-path)
    (make-process
     :name "chatgpt"
     :buffer (generate-new-buffer-name "chatgpt")
     :command (list "sh" "-c" command)
     :sentinel
     (lambda (process event)
       (if (not (string= event "finished\n"))
           (message "Error")
         (let* ((resp (with-current-buffer (process-buffer process)
                        (goto-char (point-min))
                        (chatgpt-json-read)))
                (response (map-nested-elt resp [:choices 0 :message :content]))
                (resp-path (concat req-dir "response.json")))
           (write-region (chatgpt-json-encode resp) nil resp-path)
           (chatgpt-callback prompt response req-dir))
         (kill-buffer (process-buffer process)))))))

(defun chatgpt-send ()
  "Send the current prompt to OpenAI."
  (interactive)
  (chatgpt-send-request (buffer-string)))
```

12.2 Deleting The Prompt Buffer window

Next, we enhance the `chatgpt-send` function to delete the prompt buffer's contents and delete its window if more than one window is open in the Emacs frame.

We utilize `erase-buffer` to clear the current buffer and `window-list` to check the count of visible windows.

```
(defun chatgpt-send ()
  "Send the current prompt to OpenAI."
  (interactive)
  (chatgpt-send-request (buffer-string))
  (erase-buffer)
  (when (> (length (window-list)) 1)
    (delete-window)))
```

Here are two examples of `window-list` output when 1 and 2 windows are open:

```
(window-list)
;; (#<window 1471 on chatgpt.el>)
(window-list)
;; (#<window 1471 on chatgpt.el> #<window 1503 on *chatgpt*>)
```

12.3 Ensuring the Response Buffer is Displayed

Currently, we need to manually switch to the `*chatgpt[requests]*` buffer to view responses. We will modify the `chatgpt-callback` function to automatically display this buffer and position the latest response at the top.

By using `display-buffer`, we create a new window for the response buffer and, within the body of `with-selected-window` macro, we scroll to the last `## Response` heading, ensuring it appears at the top of the view.

```
(defun chatgpt-callback (prompt response req-dir)
  "Append PROMPT and RESPONSE to the prompt buffer with a link to REQ-
DIR."
  (let ((buff (get-buffer-create "*chatgpt[requests]*")))
    (with-current-buffer buff
      (markdown-mode)
```

```
    (goto-char (point-max))
    (insert "# Request\n\n"
            "<!-- [](" req-dir ") -->\n\n"
            "## Prompt\n\n" prompt "\n\n"
            "## Response\n\n" response "\n\n"))
  (with-selected-window (display-buffer buff nil)
    (goto-char (point-max))
    (re-search-backward "^## Response")
    (recenter-top-bottom 0))))
```

12.4 Adding Notifications

Finally, we implement notifications in the echo area for when requests are sent and responses received from OpenAI.

```
(defun chatgpt-callback (prompt response req-dir)
  "Append PROMPT and RESPONSE to the prompt buffer with a link to REQ-DIR."
  (let ((buff (get-buffer-create "*chatgpt[requests]*")))
    ...
    (message "Response received from OpenAI.")))

(defun chatgpt-send ()
  "Send the current prompt to OpenAI."
  (interactive)
  ...
  (message "Request sent to OpenAI."))
```

This streamlined approach enhances the functionality we are implementing within our Emacs chat interface with OpenAI, facilitating improved user experience and interaction.

13 Handling API Errors

In this lesson, we will address how to manage API errors returned by OpenAI.

13.1 Signaling API Errors

An incorrect model specified in a request to OpenAI will trigger an API error. Let's set the `chatgpt-model` variable to a nonexistent model:

```
(setq chatgpt-model "foo")
```

Now, when we send the request "Hello!" using the `chatgpt-send` command, the following error is displayed in the echo area:

```
error in process sentinel: Wrong type argument: char-or-string-p, nil
```

This error arises from the sentinel function within the `chatgpt-send-request` function. To gather more details, we activate the debugger with the `toggle-debug-on-error` command.

Upon resending the "Hello!" request, we enter the debugger with the following stack trace:

```
Debugger entered--Lisp error: (wrong-type-argument char-or-string-
p nil)
  insert("# Request\n\n" ... "## Response\n\n" nil "\n\n")
  ...
  chatgpt-callback(#("Hello!" 0 6 (fontified t)) nil "/home/tony/chatgpt-
emacs/requests/J2jFKb/")
  ...
```

Here, we notice that a `nil` value is being inserted into a buffer in the `chatgpt-callback` function. Specifically, the `response` parameter is `nil`. In the `chatgpt-callback` function body, this is evident:

```
(defun chatgpt-callback (prompt response req-dir)
  "..."
  (let ((buff (get-buffer-create "*chatgpt[requests]*")))
    (with-current-buffer buff
      ...
      (insert "# Request\n\n"
              ...
              "## Response\n\n" response "\n\n"))
    ...))
```

Examining the sentinel function in `chatgpt-send-request`, we see the `response` is derived from the expression `(map-nested-elt resp [:choices 0 :message :content])`, where `resp` represents the JSON response converted to an Emacs Lisp object. The issue stems from the JSON response received from OpenAI.

```
(lambda (process event)
  (if (not (string= event "finished\n"))
```

```
      (message "Error")
    (let* ((resp (with-current-buffer (process-buffer process)
                   (goto-char (point-min))
                   (chatgpt-json-read)))
           (response (map-nested-elt resp [:choices 0 :message :content]))
           (resp-path (concat req-dir "response.json")))
      (write-region (chatgpt-json-encode resp) nil resp-path)
      (chatgpt-callback prompt response req-dir))
    (kill-buffer (process-buffer process))))
```

Upon inspecting the response file located at `/home/tony/chatgpt-emacs/requests/J2jFKb/`, we discover that it contains an `error` field rather than a `choices` array, indicating that there was an API error. This explains why `response` was `nil`: the response from OpenAI did not include the expected path `[:choices 0 :message :content]`.

```
{
    "error": {
        "message": "The model `foo` does not exist or you do
not have acces to it.",
        "type": "invalid_request_error",
        "param": null,
        "code": "model_not_found"
    }
}
```

To handle this scenario, we modify the sentinel function to check for the presence of an `:error` key in `resp`. If found, we bind `api-error` to its value and signal an `api-error` with `:error` set to `api-error`. Otherwise, we will proceed as usual by writing the response to disk and invoking `chatgpt-callback`.

```
(lambda (process event)
  (if (not (string= event "finished\n"))
      (message "Error")
    (let* ((resp (with-current-buffer (process-buffer process)
                   (goto-char (point-min))
                   (chatgpt-json-read))))
      (if-let ((api-error (plist-get resp :error)))
          (error "%S" `(:type "api-error" :error ,api-error))
```

```
      (let ((response (map-nested-elt resp [:choices 0 :message :content]))
            (resp-path (concat req-dir "response.json")))
        (write-region (chatgpt-json-encode resp) nil resp-path)
        (chatgpt-callback prompt response req-dir))))
  (kill-buffer (process-buffer process))))
```

We can test this by issuing `chatgpt-send` while the model remains incorrect. The error message displayed in the echo area will now provide detailed information rather than the generic error:

```
if: (:type "api-error" :error (:message "The model `foo`
does not exist or you do not have access to it." :type
"invalid_request_error" :param nil :code "model_not_found"))
```

Next, we can set the `chatgpt-model` to a valid model:

```
(setq chatgpt-model "gpt-4o")
```

By sending a "Hello!" request again, we should see the `*chatgpt[requests]*` buffer populated with this response:

```
# Request

<!-- [](/home/tony/chatgpt-emacs/requests/retcdg/) -->

## Prompt

Hello!

## Response

Hello! How can I assist you today?
```

13.2 Saving API Errors

While signaling API errors is valuable, saving these errors for future reference is even more beneficial. Therefore, we adjust the sentinel function to write the error to disk. We store the error in a file named `error.json` within the `req-dir` directory before signaling the error:

```
(lambda (process event)
  (if (not (string= event "finished\n"))
      (message "Error")
    (let* (...)
      (if-let ((api-error (plist-get resp :error)))
          (let ((err `(:type "api-error" :error ,api-error))
                (err-path (concat req-dir "error.json")))
            (write-region (chatgpt-json-encode err) nil err-path)
            (error "%S" err))
        ...))
    (kill-buffer (process-buffer process))))
```

13.3 Signaling and Saving Process Errors

Next, we implement a mechanism to handle and save errors that arise from process status changes. Currently, when an event other than "finished" occurs, we simply print "Error". We now enhance this to save the error within the corresponding request directory and signal the error. This is similar to handling API errors, except the error will be of type `process-error`, with the `:error` key containing the triggering event information.

```
(lambda (process event)
  (if (not (string= event "finished\n"))
      (let ((err `(:type "process-error" :error (:event ,event)))
            (err-path (concat req-dir "error.json")))
        (write-region (chatgpt-json-encode err) nil err-path)
        (error "%S" err))
    ...))
```

To test our changes to `chatgpt-send-request`, we temporarily modify the `chatgpt-send` function to immediately terminate the process using the `kill-process` function. This allows us to evaluate the new execution path:

```
(defun chatgpt-send ()
  "Send the current prompt to OpenAI."
  (interactive)
  (kill-process (chatgpt-send-request (buffer-string)))
  (erase-buffer)
  (when (> (length (window-list)) 1)
    (delete-window))
  (message "Request sent to OpenAI."))
```

After invoking the `chatgpt-send` command with the prompt "Hello!", the following error is signaled in the echo area:

```
(:type "process-error" :error (:event "killed\n"))
```

Additionally, we can confirm the request directory from the `*Messages*` buffer:

```
chatgpt: /home/tony/chatgpt-emacs/requests/Ot8tMl/
Wrote /home/tony/chatgpt-emacs/requests/Ot8tMl/request.json
Request sent to OpenAI.
Wrote /home/tony/chatgpt-emacs/requests/Ot8tMl/error.json
let: (:type "process-error" :error (:event "killed\n"))
```

We can also verify that the error file at `/home/tony/chatgpt-emacs/requests/Ot8tMl/error.json` contains the following JSON object:

```
{
  "type": "process-error",
  "error": {
    "event": "killed\n"
  }
}
```

Finally, we revert the `chatgpt-send` function to its original state:

```
(defun chatgpt-send ()
  "Send the current prompt to OpenAI."
  (interactive)
  (chatgpt-send-request (buffer-string))
  (erase-buffer)
  (when (> (length (window-list)) 1)
    (delete-window))
  (message "Request sent to OpenAI."))
```

13.4 chatgpt.el

The current implementation of the `chatgpt.el` package is as follows:

```
;;; chatgpt.el --- Simple ChatGPT integration -*- lexical-binding: t; -*-

(require 'json)
```

```
(require 'markdown-mode)

(defvar chatgpt-api-key
  "sk-proj-7pQDxN...w-D40A"
  "OpenAI API key.")

(defvar chatgpt-dir "/home/tony/chatgpt-emacs/requests/"
  "Request directory.")

(defun chatgpt-json-read ()
  (let ((json-key-type 'keyword)
        (json-object-type 'plist)
        (json-array-type 'vector))
    (json-read)))

(defun chatgpt-json-encode (object)
  (let ((json-encoding-pretty-print t))
    (json-encode object)))

(defun chatgpt-command (req-path)
  "Return the curl command."
  (format
   (concat "curl https://api.openai.com/v1/chat/completions "
           "-H 'Content-Type: application/json' "
           "-H 'Authorization: Bearer %s' "
           "-d @%s")
   chatgpt-api-key req-path))

(defvar chatgpt-model "gpt-4o"
  "OpenAI model.")

(defun chatgpt-request (prompt)
  "Return an OpenAI request with PROMPT."
  `(:model ,chatgpt-model
    :messages ,(vector `(:role "user" :content ,prompt))))

(defun chatgpt-callback (prompt response req-dir)
  "Append PROMPT and RESPONSE to the prompt buffer with a link to REQ-DIR."
  (let ((buff (get-buffer-create "*chatgpt[requests]*")))
    (with-current-buffer buff
```

```
      (markdown-mode)
      (goto-char (point-max))
      (insert "# Request\n\n"
              "<!-- [](" req-dir ") -->\n\n"
              "## Prompt\n\n" prompt "\n\n"
              "## Response\n\n" response "\n\n"))
    (with-selected-window (display-buffer buff nil)
      (goto-char (point-max))
      (re-search-backward "^## Response")
      (recenter-top-bottom 0))
    (message "Response received from OpenAI.")))

(defun chatgpt-send-request (prompt)
  "Send the request with PROMPT to OpenAI."
  (let* ((req (chatgpt-request prompt))
         (temporary-file-directory chatgpt-dir)
         (req-dir (file-name-as-directory (make-temp-file nil t)))
         (req-path (concat req-dir "request.json"))
         (command (chatgpt-command req-path)))
    (message "chatgpt: %s" req-dir)
    (write-region (chatgpt-json-encode req) nil req-path)
    (make-process
     :name "chatgpt"
     :buffer (generate-new-buffer-name "chatgpt")
     :command (list "sh" "-c" command)
     :sentinel
     (lambda (process event)
       (if (not (string= event "finished\n"))
           (let ((err `(:type "process-error" :error (:event ,event)))
                 (err-path (concat req-dir "error.json")))
             (write-region (chatgpt-json-encode err) nil err-path)
             (error "%S" err))
         (let* ((resp (with-current-buffer (process-buffer process)
                        (goto-char (point-min))
                        (chatgpt-json-read))))
           (if-let ((api-error (plist-get resp :error)))
               (let ((err `(:type "api-error" :error ,api-error))
                     (err-path (concat req-dir "error.json")))
                 (write-region (chatgpt-json-encode err) nil err-path)
                 (error "%S" err))
```

```
            (let ((response (map-nested-elt resp [:choices 0 :message :content]))
                  (resp-path (concat req-dir "response.json")))
              (write-region (chatgpt-json-encode resp) nil resp-path)
              (chatgpt-callback prompt response req-dir))))
        (kill-buffer (process-buffer process)))))))

(defun chatgpt-send ()
  "Send the current prompt to OpenAI."
  (interactive)
  (chatgpt-send-request (buffer-string))
  (erase-buffer)
  (when (> (length (window-list)) 1)
    (delete-window))
  (message "Request sent to OpenAI."))

(defvar chatgpt-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map (kbd "C-c C-c") 'chatgpt-send)
    map)
  "Keymap of `chatgpt-mode'.")

(define-derived-mode chatgpt-mode markdown-mode "ChatGPT"
  "ChatGPT mode."
  (setq mode-line-format
        '(" "
          mode-line-buffer-identification
          " "
          chatgpt-model
          " "
          mode-line-misc-info))
  (make-directory chatgpt-dir t))

(defun chatgpt ()
  "Display and Select the prompt buffer."
  (interactive)
  (let* ((buff-name "*chatgpt*")
         (buff-p (get-buffer buff-name))
         (buff (get-buffer-create buff-name)))
    (select-window
     (display-buffer-at-bottom
```

```
      buff '(display-buffer-below-selected (window-height . 6))))
    (when (not buff-p) (chatgpt-mode))))

(provide 'chatgpt)
```

14 Timestamp Files

In our ongoing development of the package, an essential feature we need is the capability to navigate the prompt history. Specifically, we want to enable users to retrieve previous prompts sent to OpenAI by using keyboard shortcuts like `M-p` for previous prompts and `M-n` for the next. In the upcoming lessons, we will implement this feature, starting with our discussion on using timestamp files.

14.1 Purpose of Timestamp Files

To facilitate prompt history navigation, we must organize previous prompts chronologically. For this, we will utilize timestamp files, which we will elaborate on shortly. First, let's explore the reason for this approach.

Examining the JSON response in the `response.json` file located at `/home/tony/chatgpt-emacs/requests/zsewR8/`, we note that it contains a `created` field with a timestamp:

```
{
  "id": "chatcmpl-B8UAwZapHTpsUexLppYeadC0sRG2v",
  "object": "chat.completion",
  "created": 1741362318,
  "model": "gpt-4o-2024-08-06",
  "choices": [...],
  ...
}
```

While we could theoretically use this timestamp to sort requests, this method would result in reading numerous files if the request directory contains hundreds or thousands of them, leading to considerable inefficiency. Moreover, requests that lead to API or processing errors do not generate a `response.json` file, further complicating sorting.

Instead, we will create a timestamp file for each request sent to OpenAI. This file will have its timestamp embedded in the filename as follows:

```
/home/tony/chatgpt-emacs/requests/zsewR8/timestamp-1741592528.2623844
```

This approach allows us to sort the requests in the `chatgpt-dir` directory efficiently using the `directory-files-recursively` function, without needing to read individual files. We will still need to read each request file to retrieve the prompts.

14.2 Writing Timestamp Files

We modify the `chatgpt-send-request` function to create a timestamp file alongside each request sent to OpenAI. We define the path for the timestamp file using `format` and `time-to-seconds`, which gives the current time as a float representing seconds since the epoch. We then create the file with the `write-region` function:

```
(defun chatgpt-send-request (prompt)
  "Send the request with PROMPT to OpenAI."
  (let* (...
         (timestamp-path
          (format "%stimestamp-%s" req-dir (time-to-seconds)))
         (command (chatgpt-command req-path)))
    (message "chatgpt: %s" req-dir)
    (write-region (chatgpt-json-encode req) nil req-path)
    (write-region "" nil timestamp-path)
    (make-process ...)))
```

By invoking `chatgpt-send-request` in the prompt buffer with the input "Hello!", a timestamp file is created alongside the `request.json` and `response.json` files:

```
/home/tony/chatgpt-emacs/requests/JVjcXV:
drwx------ 2 tony tony 4.0K Mar 10 10:50 .
drwxrwxr-x 5 tony tony 4.0K Mar 10 10:50 ..
-rw-rw-r-- 1 tony tony  101 Mar 10 10:50 request.json
-rw-rw-r-- 1 tony tony  807 Mar 10 10:50 response.json
-rw-rw-r-- 1 tony tony    0 Mar 10 10:50 timestamp-1741600223.2159884
```

Calling `chatgpt-send` in the prompt buffer, we send the request with the prompt "Hello!" to OpenAI and a timestamp file is created alongside the `request.json` and `response.json` files:

```
/home/tony/chatgpt-emacs/requests/JVjcXV:
drwx------ 2 tony tony 4.0K Mar 10 10:50 .
```

```
drwxrwxr-x 5 tony tony 4.0K Mar 10 10:50 ..
-rw-rw-r-- 1 tony tony  101 Mar 10 10:50 request.json
-rw-rw-r-- 1 tony tony  807 Mar 10 10:50 response.json
-rw-rw-r-- 1 tony tony    0 Mar 10 10:50 timestamp-1741600223.2159884
```

14.3 Defining the `chatgpt-timestamp` Function

Next, we implement the `chatgpt-timestamp` function, which retrieves the timestamp number from a specified timestamp file. This will assist in organizing prompts chronologically.

```
(defun chatgpt-timestamp (file)
  "Return the timestamp number from FILE."
  (string-to-number (nth 1 (string-split file "timestamp-"))))
```

For example:

```
(let ((file "/home/tony/chatgpt-emacs/requests/JVjcXV/timestamp-1741600223.2159884"))
  (chatgpt-timestamp file))
;; => 1741600223.2159884
```

For Emacs versions prior to 29.1, utilize `split-string` in place of `string-split`.

14.4 Defining the `chatgpt-requests` Function

Finally, we create the `chatgpt-requests` function, which returns a sorted list of requests in the `chatgpt-dir` directory, prioritizing the most recent entries.

First, we list the timestamp files with `directory-files-recursively`:

```
(directory-files-recursively chatgpt-dir "timestamp.*")
;; ("/home/tony/chatgpt-emacs/requests/JVjcXV/timestamp-1741600223.2159884"
;;  "/home/tony/chatgpt-emacs/requests/wVcThg/timestamp-1741610135.553578"
;;  "/home/tony/chatgpt-emacs/requests/wavggx/timestamp-1741610117.0591946")
```

Next, we sort these files using `seq-sort` in conjunction with our `chatgpt-timestamp` function, ensuring the most recent timestamps come first:

```
(let ((files (directory-files-recursively chatgpt-dir "timestamp.*")))
  (seq-sort
   (lambda (f1 f2) (> (chatgpt-timestamp f1) (chatgpt-timestamp f2)))
```

```
   files))
;; ("/home/tony/chatgpt-emacs/requests/wVcThg/timestamp-1741610135.553578"
;;  "/home/tony/chatgpt-emacs/requests/wavggx/timestamp-1741610117.0591946"
;;  "/home/tony/chatgpt-emacs/requests/JVjcXV/timestamp-1741600223.2159884")
```

After that, we extract the request directories by trimming the file paths:

```
(let ((files (directory-files-recursively chatgpt-dir "timestamp.*")))
  (mapcar (lambda (f) (string-trim-right f "timestamp.*"))
          (seq-sort
           (lambda (f1 f2) (> (chatgpt-timestamp f1) (chatgpt-timestamp f2)))
           files)))
;; ("/home/tony/chatgpt-emacs/requests/wVcThg/"
;;  "/home/tony/chatgpt-emacs/requests/wavggx/"
;;  "/home/tony/chatgpt-emacs/requests/JVjcXV/")
```

We can then consolidate the above logic into the `chatgpt-requests` function:

```
(defun chatgpt-requests ()
  "Return a sorted list of the requests in `chatgpt-dir'.

The most recent requests are listed first."
  (let ((files (directory-files-recursively chatgpt-dir "timestamp.*")))
    (mapcar (lambda (f) (string-trim-right f "timestamp.*"))
            (seq-sort
             (lambda (f1 f2)
               (> (chatgpt-timestamp f1) (chatgpt-timestamp f2)))
             files))))
```

We can test the `chatgpt-requests` function like so:

```
(chatgpt-requests)
;; ("/home/tony/chatgpt-emacs/requests/wVcThg/"
;;  "/home/tony/chatgpt-emacs/requests/wavggx/"
;;  "/home/tony/chatgpt-emacs/requests/JVjcXV/")
```

15 Overview of the Ring Package

In this lesson, we will explore the fundamentals of the built-in `ring` package, which is useful for managing histories. We will leverage this package to implement a the prompt history feature of chatgpt.el package.

With the `ring` package, we can efficiently add or remove elements and, given any element, easily access its previous or next neighbor.

15.1 Creating Rings and Inserting Elements

In this section, we demonstrate two methods to create rings and two methods for inserting elements.

First, we create a ring variable `r` with a size of 3 using the `make-ring` function:

```
(setq r (make-ring 3)) ;; (0 0 . [nil nil nil])
```

We confirm that the ring is empty by checking:

```
(ring-empty-p r) ;;
```

Next, we insert three elements into the ring using the `ring-insert` function:

```
(ring-insert r "foo-1") ;; "foo-1"
(ring-insert r "foo-2") ;; "foo-2"
(ring-insert r "foo-3") ;; "foo-3"
```

To view the elements in the ring, we can use the `ring-elements` function. For our ring `r`, it returns the following list, `"foo-1"` being the oldest element and `"foo-3"` the newest:

```
(ring-elements r) ;; ("foo-3" "foo-2" "foo-1")
```

Next, we insert a fourth element, `"foo-4"`, into this full ring:

```
(ring-insert r "foo-4")
```

Listing the elements again, we can observe that the oldest element `"foo-1"` has been removed, the ring rotated, and `"foo-4"` is now the newest element:

```
(ring-elements r) ;; ("foo-4" "foo-3" "foo-2")
```

Now, let's explore another approach for creating rings and inserting elements that allows for ring enlargement without dropping the oldest element.

We initialize the ring `r` with three elements using `ring-convert-sequence-to-ring`, which converts a sequence into a ring:

```
(setq r (ring-convert-sequence-to-ring '("foo-3" "foo-2" "foo-1")))
```

We can list `r` elements:

```
(ring-elements r) ;; ("foo-3" "foo-2" "foo-1")
```

Now, we use `ring-insert+extend` to insert `"foo-4"` while enlarging the ring instead of discarding the oldest element:

```
(ring-insert+extend r "foo-4" t) ;; "foo-4"
```

The updated elements in ring `r` now show that the oldest element remains the same, and the ring has been enlarged:

```
(ring-elements r) ;; ("foo-4" "foo-3" "foo-2" "foo-1")
```

15.2 Accessing Ring Elements

In this section, we learn how to access elements in a ring.

First, we define the ring `r` again with three elements:

```
(setq r (ring-convert-sequence-to-ring '("foo-3" "foo-2" "foo-1")))
```

We can confirm the elements in ring `r` as follows:

```
(ring-elements r) ;; ("foo-3" "foo-2" "foo-1")
```

Now, we use the `ring-ref` function to access elements by their index:

```
(ring-ref r 0) ;; "foo-3"
(ring-ref r 1) ;; "foo-2"
```

To retrieve the next or previous element for a given value, we use the `ring-next` and `ring-previous` functions:

```
(ring-next r "foo-2") ;; "foo-1"
(ring-previous r "foo-2") ;; "foo-3"
```

If we attempt to get the next element of the oldest entry, `ring-next` wraps back to the newest element:

```
(ring-next r "foo-1") ;; "foo-3"
```

Conversely, attempting to get the previous element of the newest entry wraps back to the oldest:

```
(ring-previous r "foo-3") ;; "foo-1"
```

Lastly, note that trying to access the next or previous element of a non-existent item raises an error:

```
(ring-next r "not-in-r")
 ;; ring-next: Item is not in the ring: ‘not-in-r’
```

In the subsequent lesson, we will apply the concepts from this lesson and the previous one to implement the prompt history feature.

16 Implementing Prompt History Feature

In this session, we'll integrate a prompt history feature, allowing us to navigate between previous prompts using the `M-p` and `M-n` keystrokes.

16.1 Binding `M-p` and `M-n` in `chatgpt-mode-map`

We define the commands `chatgpt-previous` and `chatgpt-next`, binding them to `M-p` and `M-n` respectively in `chatgpt-mode-map`. Initially, these commands will output "prev prompt" and "next prompt" in the echo area.

```
(defun chatgpt-previous ()
  "Replace current buffer content with previous prompt."
  (interactive)
  (message "prev prompt"))

(defun chatgpt-next ()
  "Replace current buffer content with next prompt."
  (interactive)
  (message "next prompt"))

(defvar chatgpt-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map (kbd "M-p") 'chatgpt-previous)
    (define-key map (kbd "M-n") 'chatgpt-next)
    (define-key map (kbd "C-c C-c") 'chatgpt-send)
    map)
  "Keymap of `chatgpt-mode'.")
```

16.2 Defining `chatgpt-history` and `chatgpt-push`

We introduce the `chatgpt-history` variable to store the history of request directories, initializing it as an empty ring:

```
(defvar chatgpt-history (make-ring 0)
  "Ring of request directories.")
```

Whenever a request is sent, its directory will be added to this ring via the `chatgpt-push` function:

```
(defun chatgpt-push (req-dir)
  "Push REQ-DIR into `chatgpt-history' ring."
  (ring-insert+extend chatgpt-history req-dir t))
```

Now, we modify the `chatgpt-send-request` function to incorporate a call to `chatgpt-push`, ensuring each request directory is recorded before sending the request:

```
(defun chatgpt-send-request (prompt)
  "Send the request with PROMPT to OpenAI."
  (let* (...
         (req (chatgpt-request prompt))
         (req-dir (file-name-as-directory (make-temp-file nil t)))
         ...)
    (chatgpt-push req-dir)
    (make-process ...)))
```

We can confirm the functionality by checking that `chatgpt-history` starts empty:

```
(ring-elements chatgpt-history) ;; nil
```

Then we enter `foo-1` into the prompt buffer and press `C-c C-c` to submit a request. After this, evaluating the following expression in the minibuffer confirms that the `chatgpt-history` now includes one request directory:

```
(ring-elements chatgpt-history)
;; ("/home/tony/chatgpt-emacs/requests/w32X0a/")
```

Next, we repeat the process with the prompt `foo-2`. Upon evaluation, checking the `chatgpt-history` again shows it contains two request directories:

```
(ring-elements chatgpt-history)
;; ("/home/tony/chatgpt-emacs/requests/utuN4O/"
;;  "/home/tony/chatgpt-emacs/requests/w32X0a/")
```

16.3 Implementing the `chatgpt-previous` Command

Next, we redefine `chatgpt-previous` to update the prompt buffer with the last used prompt instead of just printing a message. We introduce `chatgpt-request-dir`, a variable to hold the request directory of the current prompt:

```
(defvar chatgpt-request-dir nil
  "Request directory of the current prompt.")
```

Each time a request is sent, we reset this variable to `nil`. We modify `chatgpt-push` accordingly:

```
(defun chatgpt-push (req-dir)
  "Push REQ-DIR into `chatgpt-history' ring."
  (setq chatgpt-request-dir nil)
  (ring-insert+extend chatgpt-history req-dir t))
```

Let's revisit the `chatgpt-previous` function:

```
(defun chatgpt-previous ()
  "Replace current buffer content with previous prompt."
  (interactive)
  (let* ((req-dir (if (null chatgpt-request-dir)
                      (ring-ref chatgpt-history 0)
                    (ring-next chatgpt-history chatgpt-request-dir)))
         (req (with-temp-buffer
                (insert-file-contents (concat req-dir "request.json"))
                (chatgpt-json-read)))
         (prompt (map-nested-elt req [:messages 0 :content])))
    (setq chatgpt-request-dir req-dir)
    (erase-buffer)
    (save-excursion (insert prompt))))
```

We begin by binding `req-dir` to the request directory associated with the most recent request. If `chatgpt-request-dir` is `nil`, we use the `ring-ref` function to access the latest entry from the `chatgpt-history` ring. Otherwise, we obtain the next entry using `ring-next`.

Next, we bind `req` to the object that represents the request stored in `req-dir`. Using the `map-nested-elt` function, we extract the `prompt` from `req`.

Before updating the prompt buffer, we assign `req-dir` to the `chatgpt-request-dir` variable. This ensures that the subsequent invocation of `chatgpt-previous` will retrieve the next prompt in the `chatgpt-history`.

Finally, we clear the current buffer and insert the `prompt` value, which reflects the previous request corresponding to `chatgpt-request-dir`. The `save-excursion` function retains the cursor position at the beginning of the buffer.

16.3.1 Testing the `chatgpt-previous` Command

Before testing it, we verify that the `chatgpt-request-dir` variable is set to `nil` in the minibuffer.

In the prompt buffer, pressing `M-p` invokes the `chatgpt-previous` command, which updates the prompt buffer to display `foo-2`, the prompt of the previous request. In the minibuffer we check that the `chatgpt-request-dir` variable value has been updated:

```
chatgpt-request-dir ;; "/home/tony/chatgpt-emacs/requests/utuN4O/"
```

Pressing `M-p` again updates the prompt buffer to show `foo-1`. The `chatgpt-request-dir` is again updated:

```
chatgpt-request-dir ;; "/home/tony/chatgpt-emacs/requests/w32X0a/"
```

Using the `ring-elements` function, we can confirm that `chatgpt-history` contains two elements:

```
(ring-elements chatgpt-history)
;; ("/home/tony/chatgpt-emacs/requests/utuN4O/"
;;  "/home/tony/chatgpt-emacs/requests/w32X0a/")
```

Now, pressing `M-p` once more in the prompt buffer updates it to show `foo-2` again. This behavior occurs because `foo-1` is the oldest element in the `chatgpt-history` ring, and `ring-next` wraps around to the most recent element after reaching the oldest.

Next, we enter the prompt `foo-3` in the prompt buffer and send it to OpenAI by pressing `C-c C-c`. A subsequent check in the minibuffer confirms that `chatgpt-request-dir` is reset to `nil`. Thus, pressing `M-p` now updates the prompt buffer to display the last prompt, `foo-3`.

16.4 Handling Empty `chatgpt-history`

We successfully implemented the `chatgpt-previous` command, but we need to address the scenario where the `chatgpt-history` ring is empty.

Let's redefine the `chatgpt-history` variable as an empty ring with the following code:

```
(defvar chatgpt-history (make-ring 0)
  "Ring of the request directories.")
```

When we attempt to access a previous prompt using `M-p`, we receive an error message:

```
Accessing an empty ring
```

To improve user experience, we modify the `chatgpt-previous` command to check if the `chatgpt-history` ring is empty. If it is, we display a more informative message in the echo area:

```
(defun chatgpt-previous ()
  "Replace current buffer content with previous prompt."
  (interactive)
  (if (ring-empty-p chatgpt-history)
      (message "`chatgpt-history' empty.  Send a request first.")
    ...))
```

Now, when we press `M-p` in the prompt buffer with an empty `chatgpt-history`, we receive the message:

```
`chatgpt-history' empty.  Send a request first.
```

Following this prompt, we can send a request with the input `foo-4` to OpenAI. After sending the request, when we press `M-p` again in the prompt buffer, it updates with `foo-4`, as expected.

16.5 Initializing `chatgpt-history` from Disk

The `chatgpt-history` ring currently contains request directories from the ongoing Emacs session. However, it does not include requests from previous sessions stored in the `chatgpt-dir` directory. To address this, we will initialize the `chatgpt-history` variable with those previously saved request directories.

To achieve this, we define the `chatgpt-history-set` function. This function ensures that the `chatgpt-dir` directory exists, and it sets the `chatgpt-history` variable to a ring containing the request directories retrieved from `chatgpt-dir`, ordered with the most recent requests listed first. We use the previously defined `chatgpt-requests` function alongside the `ring-convert-sequence-to-ring` function.

Here's the implementation of `chatgpt-history-set`:

```
(defun chatgpt-history-set ()
  "Set `chatgpt-history' with request in `chatgpt-dir'."
  (when (file-exists-p chatgpt-dir)
    (setq chatgpt-history
          (ring-convert-sequence-to-ring (chatgpt-requests)))))
```

To ensure that `chatgpt-history` is initialized only once during an Emacs session—specifically, the first time the `chatgpt` command is invoked and the `*chatgpt*` buffer is created—we will modify the `chatgpt-mode` definition. This modification will include a call to the `chatgpt-history-set` function to initialize the history at that point:

```
(define-derived-mode chatgpt-mode markdown-mode "ChatGPT"
  "Major mode for ChatGPT interaction."
  (setq mode-line-format
        '(" "
          mode-line-buffer-identification
          " "
          chatgpt-model
          " "
          mode-line-misc-info))
  (make-directory chatgpt-dir t)
  (chatgpt-history-set))
```

Now, after killing the `*chatgpt*` prompt buffer and calling the `chatgpt` command again, Emacs recreates the prompt buffer, invoking `chatgpt-mode`, which subsequently calls `chatgpt-history-set`. This action populates `chatgpt-history` with request directories from the last three lessons where we defined prompt history feature, as shown below:

```
chatgpt-history
;; (0 7 . ["/home/tony/chatgpt-emacs/requests/JVjcXV/"
;;         "/home/tony/chatgpt-emacs/requests/wavggx/"
```

```
;;          "/home/tony/chatgpt-emacs/requests/wVcThg/"
;;          "/home/tony/chatgpt-emacs/requests/w32X0a/"
;;          "/home/tony/chatgpt-emacs/requests/utuN4O/"
;;          "/home/tony/chatgpt-emacs/requests/cOKJCK/"
;;          "/home/tony/chatgpt-emacs/requests/vymuEI/"])
```

This structured approach ensures that all requests, regardless of session, are accessible in the `chatgpt-history`.

16.6 Refactoring for Clean Code

We rename the `chatgpt-previous` command to `chatgpt-prompt`, modify its signature to include a `direction` argument, and adjust its implementation to handle updating the prompt buffer with the previous prompt or the next prompt:

```
(defun chatgpt-prompt (direction)
  "Replace current buffer content with DIRECTION prompt."
  (interactive)
  (if (ring-empty-p chatgpt-history)
      (message "`chatgpt-history' empty.  Send a request first.")
    (let* ((req-dir
            (if (null chatgpt-request-dir)
                (ring-ref chatgpt-history 0)
              (if (eq direction 'previous)
                  (ring-next chatgpt-history chatgpt-request-dir)
                (ring-previous chatgpt-history chatgpt-request-dir))))
           (req (with-temp-buffer
                  (insert-file-contents (concat req-dir "request.json"))
                  (chatgpt-json-read)))
           (prompt (map-nested-elt req [:messages 0 :content])))
      (setq chatgpt-request-dir req-dir)
      (erase-buffer)
      (save-excursion (insert prompt)))))
```

Finally we redefine the `chatgpt-previous` function and create the `chatgpt-next` function using the `chatgpt-prompt` function:

```
(defun chatgpt-previous ()
  "Replace current buffer content with next prompt."
  (interactive)
```

```
  (chatgpt-prompt 'previous))

(defun chatgpt-next ()
  "Replace current buffer content with next prompt."
  (interactive)
  (chatgpt-prompt 'next))
```

This concludes the implementation of the prompt history feature.

In our next session, we will implement a waiting widget that will be displayed in the mode line while awaiting a response from OpenAI.

16.7 chatgpt.el

The current implementation of the `chatgpt.el` package is as follows:

```
;;; chatgpt.el --- Simple ChatGPT integration -*- lexical-binding: t; -*-

(require 'json)
(require 'markdown-mode)

(defvar chatgpt-api-key
  "sk-proj-7pQDxN...w-D40A"
  "OpenAI API key.")

(defvar chatgpt-dir "/home/tony/chatgpt-emacs/requests/"
  "Request directory.")

(defun chatgpt-json-read ()
  (let ((json-key-type 'keyword)
        (json-object-type 'plist)
        (json-array-type 'vector))
    (json-read)))

(defun chatgpt-json-encode (object)
  (let ((json-encoding-pretty-print t))
    (json-encode object)))

(defun chatgpt-command (req-path)
  "Return the curl command."
  (format
   (concat "curl https://api.openai.com/v1/chat/completions "
```

```
           "-H 'Content-Type: application/json' "
           "-H 'Authorization: Bearer %s' "
           "-d @%s")
   chatgpt-api-key req-path))

(defvar chatgpt-model "gpt-4o"
  "OpenAI model.")

(defun chatgpt-request (prompt)
  "Return an OpenAI request with PROMPT."
  `(:model ,chatgpt-model
    :messages ,(vector `(:role "user" :content ,prompt))))

(defun chatgpt-callback (prompt response req-dir)
  "Append PROMPT and RESPONSE to the prompt buffer with a link to REQ-DIR."
  (let ((buff (get-buffer-create "*chatgpt[requests]*")))
    (with-current-buffer buff
      (markdown-mode)
      (goto-char (point-max))
      (insert "# Request\n\n"
              "<!-- [](" req-dir ") -->\n\n"
              "## Prompt\n\n" prompt "\n\n"
              "## Response\n\n" response "\n\n"))
    (with-selected-window (display-buffer buff nil)
      (goto-char (point-max))
      (re-search-backward "^## Response")
      (recenter-top-bottom 0))
    (message "Response received from OpenAI.")))

(defun chatgpt-send-request (prompt)
  "Send the request with PROMPT to OpenAI."
  (let* ((req (chatgpt-request prompt))
         (temporary-file-directory chatgpt-dir)
         (req-dir (file-name-as-directory (make-temp-file nil t)))
         (req-path (concat req-dir "request.json"))
         (timestamp-path (format "%stimestamp-%s" req-dir (time-to-seconds)))
         (command (chatgpt-command req-path)))
    (message "chatgpt: %s" req-dir)
    (write-region (chatgpt-json-encode req) nil req-path)
    (write-region "" nil timestamp-path)
```

```
    (chatgpt-push req-dir)
    (make-process
     :name "chatgpt"
     :buffer (generate-new-buffer-name "chatgpt")
     :command (list "sh" "-c" command)
     :sentinel
     (lambda (process event)
       (if (not (string= event "finished\n"))
           (let ((err `(:type "process-error" :error (:event ,event)))
                 (err-path (concat req-dir "error.json")))
             (write-region (chatgpt-json-encode err) nil err-path)
             (error "%S" err))
         (let* ((resp (with-current-buffer (process-buffer process)
                        (goto-char (point-min))
                        (chatgpt-json-read))))
           (if-let ((api-error (plist-get resp :error)))
               (let ((err `(:type "api-error" :error ,api-error))
                     (err-path (concat req-dir "error.json")))
                 (write-region (chatgpt-json-encode err) nil err-path)
                 (error "%S" err))
             (let ((response (map-nested-elt resp [:choices 0 :message :content]))
                   (resp-path (concat req-dir "response.json")))
               (write-region (chatgpt-json-encode resp) nil resp-path)
               (chatgpt-callback prompt response req-dir))))
           (kill-buffer (process-buffer process)))))))

(defun chatgpt-send ()
  "Send the current prompt to OpenAI."
  (interactive)
  (chatgpt-send-request (buffer-string))
  (erase-buffer)
  (when (> (length (window-list)) 1)
    (delete-window))
  (message "Request sent to OpenAI."))

(defun chatgpt-timestamp (file)
  "Return the timestamp number associated with timestamp FILE."
  (string-to-number (nth 1 (string-split file "timestamp-"))))

(defun chatgpt-requests ()
```

```
  "Return a sorted list of the requests in `chatgpt-dir'.

The most recent requests are listed first."
  (let ((files (directory-files-recursively chatgpt-dir "timestamp.*")))
    (mapcar (lambda (f) (string-trim-right f "timestamp.*"))
            (seq-sort
             (lambda (f1 f2) (> (chatgpt-timestamp f1) (chatgpt-timestamp f2)))
             files))))

(defvar chatgpt-request-dir nil
  "Hold request directory of the current prompt.")

(defvar chatgpt-history (make-ring 0)
  "Ring of the request directories.")

(defun chatgpt-history-set ()
  "Set `chatgpt-history' with request in `chatgpt-dir'."
  (when (file-exists-p chatgpt-dir)
    (setq chatgpt-history (ring-convert-sequence-to-ring (chatgpt-requests)))))

(defun chatgpt-push (req-dir)
  "Push REQ-DIR into `chatgpt-history' ring."
  (setq chatgpt-request-dir nil)
  (ring-insert+extend chatgpt-history req-dir t))

(defun chatgpt-prompt (direction)
  "Replace current buffer content with DIRECTION prompt."
  (interactive)
  (if (ring-empty-p chatgpt-history)
      (message "`chatgpt-history' empty.  Send a request first.")
    (let* ((req-dir (if (null chatgpt-request-dir)
                        (ring-ref chatgpt-history 0)
                      (if (eq direction 'previous)
                          (ring-next chatgpt-history chatgpt-request-dir)
                        (ring-previous chatgpt-history chatgpt-request-dir))))
           (req (with-temp-buffer
                  (insert-file-contents (concat req-dir "request.json"))
                  (chatgpt-json-read)))
           (prompt (map-nested-elt req [:messages 0 :content])))
      (setq chatgpt-request-dir req-dir)
```

```
      (erase-buffer)
      (save-excursion (insert prompt)))))

(defun chatgpt-previous ()
  "Replace current buffer content with next prompt."
  (interactive)
  (chatgpt-prompt 'previous))

(defun chatgpt-next ()
  "Replace current buffer content with next prompt."
  (interactive)
  (chatgpt-prompt 'next))

(defvar chatgpt-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map (kbd "M-p") 'chatgpt-previous)
    (define-key map (kbd "M-n") 'chatgpt-next)
    (define-key map (kbd "C-c C-c") 'chatgpt-send)
    map)
  "Keymap of `chatgpt-mode'.")

(define-derived-mode chatgpt-mode markdown-mode "ChatGPT"
  "ChatGPT mode."
  (setq mode-line-format
        '(" "
          mode-line-buffer-identification
          " "
          chatgpt-model
          " "
          mode-line-misc-info))
  (make-directory chatgpt-dir t)
  (chatgpt-history-set))

(defun chatgpt ()
  "Display and Select the prompt buffer."
  (interactive)
  (let* ((buff-name "*chatgpt*")
         (buff-p (get-buffer buff-name))
         (buff (get-buffer-create buff-name)))
    (select-window
```

```
     (display-buffer-at-bottom
      buff '(display-buffer-below-selected (window-height . 6))))
    (when (not buff-p) (chatgpt-mode))))

(provide 'chatgpt)
```

17 The Waiting Widget

In this lesson, we implement a waiting widget that appears in the mode line while awaiting a response from OpenAI.

To achieve this, we define the `chatgpt-mode-line-waiting` function. This function either starts a timer that updates the mode line with the string "| ChatGPT" followed by a variable number of dots or stops this timer and remove the waiting widget from the mode line. This is done by updating the `global-mode-string` variable every 0.66 seconds. The timer is stored in the `chatgpt-timer` variable.

```
(defvar chatgpt-timer nil "Timer for waiting widget in mode line")

(defun chatgpt-mode-line-waiting (action)
  "Start or stop a waiting widget in mode line.

Accepted values for ACTION includes `start' and `stop'."
  (pcase action
    ('start
     (setq chatgpt-timer
           (run-with-timer
            0 0.66
            (let ((idx 0))
              (lambda ()
                (progn
                  (setq global-mode-string
                        `(:eval
                          ,(concat "| ChatGPT."
                                   (make-string (mod idx 3) ?.))))
                  (force-mode-line-update 'all)
                  (cl-incf idx)))))))
    ('stop
     (cancel-timer chatgpt-timer)
     (setq chatgpt-timer nil)
```

```
    (setq global-mode-string nil)
    (force-mode-line-update 'all))))
```

Next, we modify the `chatgpt-send-request` function to incorporate calls to `chatgpt-mode-line-waiting`. Before sending the request, we invoke this function to initiate the `chatgpt-timer`. Upon receiving a response from OpenAI, we also call this function to stop the timer in the sentinel function, ensuring it is the first action executed:

```
(defun chatgpt-send-request (prompt)
  "Send the request with PROMPT to OpenAI."
  (let* (...)
    ...
    (chatgpt-mode-line-waiting 'start)
    (make-process
     :name "chatgpt"
     :buffer (generate-new-buffer-name "chatgpt")
     :command (list "sh" "-c" command)
     :sentinel
     (lambda (process event)
       (chatgpt-mode-line-waiting 'stop)
       (if (not (string= event "finished\n"))
           ...
         ...)))))
```

This implementation effectively manages the waiting widget within the mode line, providing us with clear visual feedback while awaiting a response from OpenAI.

18 Managing the API Key

In this lesson, we will focus on securely managing the OpenAI API key within Emacs. Currently, the `chatgpt-api-key` variable is defined in the `chatgpt.el` file. However, storing API keys in code is not advisable. Instead, we can utilize the `~/.authinfo` and `~/.authinfo.gpg` files (the latter being the encrypted version using GPG) to securely store our API keys.

This section outlines the process for retrieving the API key from these files. We will provide an example using the non-encrypted file; however, the code can be applied uniformly across both encrypted and non-encrypted file methods.

18.1 Redefining the API Key Variable

First, we reset the `chatgpt-api-key` variable to `nil`:

```
(defvar chatgpt-api-key nil "OpenAI API key.")
```

18.2 Modifying the `chatgpt-command` Function

Next, we update the `chatgpt-command` function. This function checks if `chatgpt-api-key` is `nil`. If it is, it fetches the API key from either `~/.authinfo` or `~/.authinfo.gpg` using `auth-source-pick-first-password` and then store it for future use:

```
(defun chatgpt-command (req-path)
  "Return the curl command."
  (when (null chatgpt-api-key)
    (setq chatgpt-api-key
          (auth-source-pick-first-password :host "openai")))
  (format
   (concat "curl https://api.openai.com/v1/chat/completions "
           "-H 'Content-Type: application/json' "
           "-H 'Authorization: Bearer %s' "
           "-d @%s")
   chatgpt-api-key req-path))
```

18.3 Adding the API Key to `~/.authinfo` File

We add our OpenAI API key in `~/.authinfo` with the following format:

```
machine openai password sk-proj-7pQDxN...w-D40A
```

18.4 Restarting Emacs for Changes to Take Effect

I don't know why, but if we make changes to `~/.authinfo` or `~/.authinfo.gpg` during our Emacs session, they will not be reflected immediately. To test, we can evaluate the following which returns `nil` instead of our API key:

```
(auth-source-pick-first-password :host "openai") ;; nil
```

So we restart Emacs, open the `chatgpt.el` file, and evaluate it using `M-x eval-buffer`.

18.5 Testing the Setup

Now, we invoke the `chatgpt-command` function. In the prompt buffer, we enter the prompt `foo`, and press `C-c C-c` to send the request to OpenAI. We then receive a response which confirms that the integration is functioning as expected.

Finally, we verify that the `chatgpt-api-key` variable now contains the API key from our `~/.authinfo` file by evaluating it in the minibuffer.

19 chatgpt.el - Simple ChatGPT Emacs Integration

19.1 Overview

`chatgpt.el` is a simple Emacs package that allows you to interact with OpenAI's ChatGPT directly from within Emacs. It leverages the OpenAI API to send prompts and receive responses.

19.2 Key Features

- **Easy API Integration**: Automatically retrieves your OpenAI API key from `~/.authinfo.gpg` (for secure storage) or from the plaintext `~/.authinfo` file.
- **Prompt History**: Keeps track of your previous prompts, allowing you to navigate back and forth through your request history pressing `M-p` and `M-n` in the prompt buffer.
- **Request Logging**: Saves all requests and responses in the `chatgpt-dir` directory (by default, `~/.emacs.d/chatgpt-requests/`).

19.3 Get Started in Minutes

1. Add the directory containing `chatgpt.el` to your `load-path` and require the **chatgpt** package by adding the following lines to your init file, ensuring to replace `/path/to/chatgpt/` with the appropriate directory:

```
(add-to-list 'load-path "/path/to/chatgpt/")
(require 'chatgpt)
```

2. Store your OpenAI API key in either the `~/.authinfo.gpg` file (encrypted with `gpg`) or the `~/.authinfo` file (plaintext):

    - After funding your OpenAI account ($5.00 is enough to get started), create an OpenAI API key visiting `https://platform.openai.com/api-keys`.
    - Add the API key in the selected file as follows:

      ```
      machine openai password <openai-api-key>
      ```

      where `<openai-api-key>` is your API key.
    - Restart Emacs to apply this change.

3. Call the command `chatgpt` to switch to `*chatgpt*` prompt buffer,
4. Enter your prompt,
5. Press `C-c C-c` to send your prompt to OpenAI API,
6. Finally, the response will asynchronously show up in a dedicated buffer upon receipt.

Links:

- `https://platform.openai.com`
- `https://platform.openai.com/api-keys`

19.4 chatgpt.el

```
;;; chatgpt.el --- Simple ChatGPT client -*- lexical-binding: t; -*-
;;
;; Copyright (C) 2025 Tony Aldon
;;
;; Author: Tony Aldon <tony@tonyaldon.com>
;; Version: 1.0
;; Package-Requires: ((emacs "25.1"))
;; Homepage: https://tonyaldon.com
;;
;;; Commentary:
;;
;;;; Overview
```

```
;;
;; chatgpt.el is a simple Emacs package that allows you to interact
;; with OpenAI's ChatGPT directly from within Emacs.  It leverages the
;; OpenAI API to send prompts and receive responses.
;;
;;;; Key Features
;;
;; - Secure API Key Handling: Automatically retrieves your OpenAI API key
;;   from ~/.authinfo.gpg for secure storage or from the plaintext
;;   ~/.authinfo file.
;; - Prompt History: Keeps track of your previous prompts, allowing you
;;   to navigate back and forth through your request history pressing `M-p'
;;   and `M-n' in the prompt buffer.
;; - Request Logging: Saves all requests and responses in the
;;   `chatgpt-dir' directory (by default, ~/.emacs.d/chatgpt-requests/).
;;
;;;; Get started in minutes
;;
;; 1) Add the directory containing chatgpt.el to your `load-path' and
;;    require the chatgpt.el package by adding the following lines to
;;    your init file, ensuring to replace /path/to/chatgpt/ with the
;;    appropriate directory:
;;
;;        (add-to-list 'load-path "/path/to/chatgpt/")
;;        (require 'eden)
;;
;; 2) Store your OpenAI API key in either the ~/.authinfo.gpg file
;;    (encrypted with gpg) or the ~/.authinfo file (plaintext):
;;
;;    - After funding your OpenAI account (https://platform.openai.com)
;;      ($5.00 is enough to get started), create an OpenAI API key
;;      visiting https://platform.openai.com/api-keys.
;;    - Add the API key in the selected file as follows:
;;
;;          machine openai password <openai-api-key>
;;
;;      where <openai-api-key> is your API key.
;;
;;    - Restart Emacs to apply this change.
;;
```

```
;; 3) Call the command `chatgpt' to switch to *chatgpt* prompt buffer,
;; 4) Enter your prompt,
;; 5) Press C-c C-c to send your prompt to OpenAI API,
;; 6) Finally, the response will asynchronously show up in a dedicated
;;    buffer upon receipt.


(require 'json)
(require 'markdown-mode)

;;; Code:

(defvar chatgpt-api-key nil "OpenAI API key.")

(defvar chatgpt-dir
  (expand-file-name (concat user-emacs-directory "chatgpt-requests/"))
  "Request directory.

This directory path must be absolute and end with a forward slash
like this:

    \"/home/tony/chatgpt-emacs/requests/\"")

(defun chatgpt-json-encode (object)
  "Return a JSON representation of OBJECT as a string."
  (let ((json-encoding-pretty-print t))
    (json-encode object)))

(defun chatgpt-json-read ()
  "Parse and return the JSON object following point."
  (let ((json-key-type 'keyword)
        (json-object-type 'plist)
        (json-array-type 'vector))
    (json-read)))

(defun chatgpt-command (req-path)
  "Return the curl command with REQ-PATH request data for OpenAI API call.

Also retrieve OpenAI API key from `~/.authinfo.gpg' (encrypted
with gpg) or `~/.authinfo' files looking for a line like this
```

```
      machine openai password <openai-api-key>"
  (when (null chatgpt-api-key)
    (setq chatgpt-api-key
          (auth-source-pick-first-password :host "openai")))
  (format
   (concat "curl https://api.openai.com/v1/chat/completions "
           "-H 'Content-Type: application/json' "
           "-H 'Authorization: Bearer %s' "
           "-d @%s")
   chatgpt-api-key req-path))

(defvar chatgpt-model "gpt-4o" "OpenAI model.")

(defun chatgpt-request (prompt)
  "Return an OpenAI request with PROMPT."
  `(:model ,chatgpt-model
    :messages ,(vector `(:role "user" :content ,prompt))))

(defun chatgpt-callback (prompt response req-dir)
  "Append PROMPT and RESPONSE to the prompt buffer with a link to REQ-DIR.

Also display the response buffer."
  (let ((buff (get-buffer-create "*chatgpt[requests]*")))
    (with-current-buffer buff
      (markdown-mode)
      (goto-char (point-max))
      (insert "# Request\n\n"
              "<!-- [](" req-dir ") -->\n\n"
              "## Prompt\n\n" prompt "\n\n"
              "## Response\n\n" response "\n\n"))
    (with-selected-window (display-buffer buff nil)
      (goto-char (point-max))
      (re-search-backward "^## Response")
      (recenter-top-bottom 0))
    (message "Response received from OpenAI.")))

(defvar chatgpt-timer nil "Timer for waiting widget in mode line.")

(defun chatgpt-mode-line-waiting (action)
```

```
  "Start or stop a waiting widget in mode line.

Accepted values for ACTION includes `start' and `stop'."
  (pcase action
    ('start
     (setq chatgpt-timer
           (run-with-timer
            0 0.66
            (let ((idx 0))
              (lambda ()
                (progn
                  (setq global-mode-string
                        `(:eval ,(concat "| ChatGPT." (make-string (mod idx 3) ?.))))
                  (force-mode-line-update 'all)
                  (cl-incf idx)))))))
    ('stop
     (cancel-timer chatgpt-timer)
     (setq chatgpt-timer nil)
     (setq global-mode-string nil)
     (force-mode-line-update 'all))))

(defun chatgpt-send-request (prompt)
  "Send the request with PROMPT to OpenAI."
  (let* ((req (chatgpt-request prompt))
         (temporary-file-directory chatgpt-dir)
         (req-dir (file-name-as-directory (make-temp-file nil t)))
         (req-path (concat req-dir "request.json"))
         (timestamp-path (format "%stimestamp-%s" req-dir (time-to-seconds)))
         (command (chatgpt-command req-path)))
    (message "chatgpt: %s" req-dir)
    (write-region (chatgpt-json-encode req) nil req-path)
    (write-region "" nil timestamp-path)
    (chatgpt-push req-dir)
    (chatgpt-mode-line-waiting 'start)
    (make-process
     :name "chatgpt"
     :buffer (generate-new-buffer-name "chatgpt")
     :command (list "sh" "-c" command)
     :sentinel
     (lambda (process event)
```

```
        (chatgpt-mode-line-waiting 'stop)
        (if (not (string= event "finished\n"))
            (let ((err `(:type "process-error" :error (:event ,event)))
                  (err-path (concat req-dir "error.json")))
              (write-region (chatgpt-json-encode err) nil err-path)
              (error "%S" err))
          (let* ((resp (with-current-buffer (process-buffer process)
                         (goto-char (point-min))
                         (chatgpt-json-read))))
            (if-let ((api-error (plist-get resp :error)))
                (let ((err `(:type "api-error" :error ,api-error))
                      (err-path (concat req-dir "error.json")))
                  (write-region (chatgpt-json-encode err) nil err-path)
                  (error "%S" err))
              (let ((response (map-nested-elt resp [:choices 0 :message :content]))
                    (resp-path (concat req-dir "response.json")))
                (write-region (chatgpt-json-encode resp) nil resp-path)
                (chatgpt-callback prompt response req-dir))))
          (kill-buffer (process-buffer process)))))))

(defun chatgpt-send ()
  "Send the current prompt to OpenAI."
  (interactive)
  (chatgpt-send-request (buffer-string))
  (erase-buffer)
  (when (> (length (window-list)) 1)
    (delete-window))
  (message "Request sent to OpenAI."))

(defun chatgpt-timestamp (file)
  "Return the timestamp number associated with timestamp FILE."
  (string-to-number (nth 1 (split-string file "timestamp-"))))

(defun chatgpt-requests ()
  "Return a sorted list of the requests in `chatgpt-dir'.

The most recent requests are listed first."
  (let ((files (directory-files-recursively chatgpt-dir "timestamp.*")))
    (mapcar (lambda (f) (string-trim-right f "timestamp.*"))
            (seq-sort
```

```
            (lambda (f1 f2) (> (chatgpt-timestamp f1) (chatgpt-timestamp f2)))
            files))))

(defvar chatgpt-request-dir nil
  "Hold request directory of the current prompt.")

(defvar chatgpt-history (make-ring 0)
  "Ring of the request directories.")

(defun chatgpt-history-set ()
  "Set `chatgpt-history' with request in `chatgpt-dir'."
  (when (file-exists-p chatgpt-dir)
    (setq chatgpt-history (ring-convert-sequence-to-ring (chatgpt-requests)))))

(defun chatgpt-push (req-dir)
  "Insert REQ-DIR into `chatgpt-history' ring."
  (setq chatgpt-request-dir nil)
  (ring-insert+extend chatgpt-history req-dir t))

(defun chatgpt-prompt (direction)
  "Replace current buffer content with DIRECTION prompt."
  (interactive)
  (if (ring-empty-p chatgpt-history)
      (message "`chatgpt-history' empty.  Send a request first.")
    (let* ((req-dir (if (null chatgpt-request-dir)
                        (ring-ref chatgpt-history 0)
                      (if (eq direction 'previous)
                          (ring-next chatgpt-history chatgpt-request-dir)
                        (ring-previous chatgpt-history chatgpt-request-dir))))
           (req (with-temp-buffer
                  (insert-file-contents (concat req-dir "request.json"))
                  (chatgpt-json-read)))
           (prompt (map-nested-elt req [:messages 0 :content])))
      (setq chatgpt-request-dir req-dir)
      (erase-buffer)
      (save-excursion (insert prompt)))))

(defun chatgpt-previous ()
  "Replace current buffer content with next prompt."
  (interactive)
```

```
  (chatgpt-prompt 'previous))

(defun chatgpt-next ()
  "Replace current buffer content with next prompt."
  (interactive)
  (chatgpt-prompt 'next))

(defvar chatgpt-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map (kbd "M-p") 'chatgpt-previous)
    (define-key map (kbd "M-n") 'chatgpt-next)
    (define-key map (kbd "C-c C-c") 'chatgpt-send)
    map)
  "Keymap of `chatgpt-mode'.")

(define-derived-mode chatgpt-mode markdown-mode "ChatGPT"
  "ChatGPT mode."
  (setq mode-line-format
        '(" "
          mode-line-buffer-identification
          " "
          chatgpt-model
          " "
          mode-line-misc-info))
  (make-directory chatgpt-dir t)
  (chatgpt-history-set))

(defun chatgpt ()
  "Display and Select the prompt buffer.

Once in that buffer you can enter your prompt and send it
to OpenAI with `chatgpt-send' command bound by default to `C-c C-c'.

Your OpenAI API key will be retrieved in from `~/.authinfo.gpg'
\(encrypted with gpg) or `~/.authinfo' files looking for a line
like this

       machine openai password <openai-api-key>"
  (interactive)
  (let* ((buff-name "*chatgpt*")
```

```
          (buff-p (get-buffer buff-name))
          (buff (get-buffer-create buff-name)))
      (select-window
       (display-buffer-at-bottom
        buff '(display-buffer-below-selected (window-height . 6))))
      (when (not buff-p) (chatgpt-mode))))

(provide 'chatgpt)
;;; chatgpt.el ends here
```