

command_processor.py

```
1 import os
2 import re
3 import subprocess
4 import logging
5 import webbrowser
6 import nltk
7 from nltk.tokenize import word_tokenize
8 from nltk.corpus import stopwords
9
10 # Download necessary NLTK data
11 try:
12     nltk.data.find('tokenizers/punkt')
13 except LookupError:
14     nltk.download('punkt')
15
16 try:
17     nltk.data.find('corpora/stopwords')
18 except LookupError:
19     nltk.download('stopwords')
20
21 class CommandProcessor:
22     """
23     Class for processing and executing voice commands.
24     """
25
26     def __init__(self):
27         """Initialize the command processor with supported commands."""
28         self.logger = logging.getLogger(__name__)
29
30         # Define command categories and their associated keywords
31         self.command_patterns = {
32             "open_app": [
33                 r"open (.*)",
34                 r"launch (.*)",
35                 r"start (.*)",
36                 r"run (.*)"
37             ],
38             "search_web": [
39                 r"search for (.*)",
40                 r"look up (.*)",
41                 r"google (.*)",
42                 r"find information about (.*)"
43             ],
44             "system_control": [
45                 r"(shut down|power off|turn off) (computer|system|pc)",
46                 r"restart (computer|system|pc)",
47                 r"log (out|off)",
48                 r"(lock|unlock) (computer|system|pc)"
```

```

49         ],
50         "file_operations": [
51             r"create (folder|directory) (.*)",
52             r"delete (file|folder|directory) (.*)",
53             r"rename (.*?) to (.*?)",
54             r"move (.*?) to (.*?)",
55         ],
56         "note_taking": [
57             r"take a note (.*)",
58             r"write down (.*)",
59             r"remember (.*)",
60             r"make a note (.*)"
61         ],
62         "date_time": [
63             r"what (time|date) is it",
64             r"tell me the (time|date)",
65             r"current (time|date)"
66         ],
67         "help": [
68             r"help",
69             r"what can you do",
70             r"list commands",
71             r"show commands"
72         ]
73     }
74
75     # Initialize stop words
76     self.stop_words = set(stopwords.words('english'))
77
78     self.logger.debug("Command Processor initialized")
79
80     def process_command(self, command_text):
81         """
82         Process a command string and determine the action to take.
83
84         Args:
85             command_text: The command text to process
86
87         Returns:
88             dict: The result of the command execution
89         """
90         command_text = command_text.lower().strip()
91         self.logger.debug(f"Processing command: {command_text}")
92
93         # Check for matches against our command patterns
94         for category, patterns in self.command_patterns.items():
95             for pattern in patterns:
96                 match = re.match(pattern, command_text, re.IGNORECASE)
97                 if match:
98                     self.logger.debug(f"Command matched category: {category}")

```

```

99
100     # Call the appropriate method based on the command category
101     if category == "open_app":
102         app_name = match.group(1).strip()
103         return self.open_application(app_name)
104
105     elif category == "search_web":
106         query = match.group(1).strip()
107         return self.search_web(query)
108
109     elif category == "system_control":
110         action = match.group(1).lower()
111         return self.system_control(action)
112
113     elif category == "file_operations":
114         # This is a simplified implementation
115         return {
116             "status": "error",
117             "action": "perform file operations",
118             "message": "File operations are not fully implemented yet."
119         }
120
121     elif category == "note_taking":
122         content = match.group(1).strip()
123         return self.take_note(content)
124
125     elif category == "date_time":
126         query = match.group(1).lower()
127         return self.get_date_time(query)
128
129     elif category == "help":
130         return self.provide_help()
131
132     # If no pattern matches, try to infer the intent based on keywords
133     tokens = word_tokenize(command_text)
134     filtered_tokens = [w for w in tokens if w not in self.stop_words]
135
136     # Check for keywords in the filtered tokens
137     if any(word in filtered_tokens for word in ["open", "launch", "start", "run"]):
138         # Find potential app name after the command word
139         for i, word in enumerate(tokens):
140             if word in ["open", "launch", "start", "run"] and i+1 < len(tokens):
141                 app_name = tokens[i+1]
142                 return self.open_application(app_name)
143
144     elif any(word in filtered_tokens for word in ["search", "google", "look", "find"]):
145         # Extract the search query
146         search_terms = [w for w in tokens if w not in ["search", "for", "google", "look",
147 "up", "find", "information", "about"]]
147         if search_terms:

```

```

148         query = " ".join(search_terms)
149         return self.search_web(query)
150
151     # If we can't determine the intent, return an error
152     return {
153         "status": "error",
154         "action": "understand your command",
155         "message": "Sorry, I didn't understand that command. Try asking for 'help' to see
what I can do."
156     }
157
158     def open_application(self, app_name):
159         """
160         Open an application based on its name.
161
162         Args:
163             app_name: The name of the application to open
164
165         Returns:
166             dict: The result of the operation
167         """
168         self.logger.debug(f"Attempting to open application: {app_name}")
169
170         # Common application mapping
171         app_mapping = {
172             "browser": "google-chrome",
173             "chrome": "google-chrome",
174             "firefox": "firefox",
175             "text editor": "notepad",
176             "notepad": "notepad",
177             "calculator": "calc",
178             "file explorer": "explorer",
179             "explorer": "explorer",
180             "terminal": "cmd"
181         }
182
183         # Map common names to actual executable names
184         executable = app_mapping.get(app_name.lower(), app_name.lower())
185
186         try:
187             # Attempt to open the application
188             # This is a simplified version and might not work for all applications
189             if os.name == 'nt': # Windows
190                 os.startfile(executable)
191             else: # Linux/Mac
192                 subprocess.Popen([executable], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
193
194         return {
195             "status": "success",

```

```

196         "action": f"opened {app_name}",
197         "details": {"app_name": app_name}
198     }
199 except Exception as e:
200     self.logger.error(f"Error opening application {app_name}: {e}")
201     return {
202         "status": "error",
203         "action": f"open {app_name}",
204         "message": f"Could not open {app_name}. {str(e)}"
205     }
206
207 def search_web(self, query):
208     """
209     Search the web for a query.
210
211     Args:
212         query: The search query
213
214     Returns:
215         dict: The result of the operation
216     """
217     self.logger.debug(f"Searching web for: {query}")
218
219     try:
220         # Format the query and open a browser with the search
221         formatted_query = query.replace(' ', '+')
222         webbrowser.open(f"https://www.google.com/search?q={formatted_query}")
223
224         return {
225             "status": "success",
226             "action": f"searched the web for '{query}'",
227             "details": {"query": query}
228         }
229     except Exception as e:
230         self.logger.error(f"Error searching web for {query}: {e}")
231         return {
232             "status": "error",
233             "action": f"search for {query}",
234             "message": f"Could not search the web. {str(e)}"
235         }
236
237 def system_control(self, action):
238     """
239     Control system functions like shutdown, restart, etc.
240
241     Args:
242         action: The system control action to perform
243
244     Returns:
245         dict: The result of the operation

```

```

246 """
247 self.logger.debug(f"System control action: {action}")
248
249 # Note: These operations may require administrative privileges
250 try:
251     if "shut down" in action or "power off" in action or "turn off" in action:
252         # This is just a simulation for safety
253         return {
254             "status": "success",
255             "action": "prepared to shut down the computer",
256             "message": "This is a simulated shutdown. In a real application, the
system would shut down.",
257             "details": {"action": "shutdown"}
258         }
259     elif "restart" in action:
260         # This is just a simulation for safety
261         return {
262             "status": "success",
263             "action": "prepared to restart the computer",
264             "message": "This is a simulated restart. In a real application, the
system would restart.",
265             "details": {"action": "restart"}
266         }
267     elif "log out" in action or "log off" in action:
268         # This is just a simulation for safety
269         return {
270             "status": "success",
271             "action": "prepared to log out",
272             "message": "This is a simulated log out. In a real application, you would
be logged out.",
273             "details": {"action": "logout"}
274         }
275     elif "lock" in action:
276         # This is just a simulation for safety
277         return {
278             "status": "success",
279             "action": "prepared to lock the computer",
280             "message": "This is a simulated lock. In a real application, the system
would be locked.",
281             "details": {"action": "lock"}
282         }
283     else:
284         return {
285             "status": "error",
286             "action": f"perform system action '{action}'",
287             "message": f"Unknown system action: {action}"
288         }
289 except Exception as e:
290     self.logger.error(f"Error performing system action {action}: {e}")
291     return {
292         "status": "error",

```

```

293         "action": f"perform system action '{action}'",
294         "message": f"Could not perform system action. {str(e)}"
295     }
296
297 def take_note(self, content):
298     """
299     Save a note with the given content.
300
301     Args:
302         content: The content of the note
303
304     Returns:
305         dict: The result of the operation
306     """
307     self.logger.debug(f"Taking note: {content}")
308
309     try:
310         # Create notes directory if it doesn't exist
311         notes_dir = os.path.join(os.path.expanduser("~"), "voice_assistant_notes")
312         os.makedirs(notes_dir, exist_ok=True)
313
314         # Create a timestamp for the filename
315         import datetime
316         timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
317
318         # Write the note to a file
319         filename = os.path.join(notes_dir, f"note_{timestamp}.txt")
320         with open(filename, 'w') as f:
321             f.write(content)
322
323         return {
324             "status": "success",
325             "action": "saved your note",
326             "details": {"content": content, "file": filename}
327         }
328     except Exception as e:
329         self.logger.error(f"Error taking note: {e}")
330         return {
331             "status": "error",
332             "action": "save your note",
333             "message": f"Could not save the note. {str(e)}"
334         }
335
336 def get_date_time(self, query):
337     """
338     Get the current date or time.
339
340     Args:
341         query: Whether to get the date or time
342

```

```

343 Returns:
344     dict: The result of the operation
345     """
346 self.logger.debug(f"Getting date/time for query: {query}")
347
348 try:
349     import datetime
350     now = datetime.datetime.now()
351
352     if "time" in query:
353         formatted_time = now.strftime("%I:%M %p")
354         return {
355             "status": "success",
356             "action": "checked the time",
357             "message": f"The current time is {formatted_time}",
358             "details": {"time": formatted_time}
359         }
360     elif "date" in query:
361         formatted_date = now.strftime("%A, %B %d, %Y")
362         return {
363             "status": "success",
364             "action": "checked the date",
365             "message": f"Today is {formatted_date}",
366             "details": {"date": formatted_date}
367         }
368     else:
369         return {
370             "status": "error",
371             "action": "get the date or time",
372             "message": "I'm not sure if you wanted the date or time."
373         }
374 except Exception as e:
375     self.logger.error(f"Error getting date/time: {e}")
376     return {
377         "status": "error",
378         "action": "get the date or time",
379         "message": f"Could not retrieve the date or time. {str(e)}"
380     }
381
382 def provide_help(self):
383     """
384     Provide help information about available commands.
385
386     Returns:
387         dict: The result of the operation
388         """
389     self.logger.debug("Providing help information")
390
391     help_text = """
392     Here are the commands I can understand:

```



```
393
394     1. Open applications: "open [app name]", "launch [app name]"
395     2. Search the web: "search for [query]", "google [query]"
396     3. System control: "shut down computer", "restart computer", "lock computer"
397     4. Take notes: "take a note [content]", "remember [content]"
398     5. Date and time: "what time is it", "what date is it"
399     6. Help: "help", "what can you do"
400
401     You can speak naturally, and I'll try to understand your intent.
402     ""
403
404     return {
405         "status": "success",
406         "action": "provided help information",
407         "message": help_text,
408         "details": {"help_text": help_text}
409     }
410
```