command_processor.py

```python
import os
import re
import subprocess
import logging
import webbrowser
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords

# Download necessary NLTK data
try:
    nltk.data.find('tokenizers/punkt')
except LookupError:
    nltk.download('punkt')

try:
    nltk.data.find('corpora/stopwords')
except LookupError:
    nltk.download('stopwords')

class CommandProcessor:
    """
    Class for processing and executing voice commands.
    """

    def __init__(self):
        """Initialize the command processor with supported commands."""
        self.logger = logging.getLogger(__name__)

        # Define command categories and their associated keywords
        self.command_patterns = {
            "open_app": [
                r"open (.*)",
                r"launch (.*)",
                r"start (.*)",
                r"run (.*)"
            ],
            "search_web": [
                r"search for (.*)",
                r"look up (.*)",
                r"google (.*)",
                r"find information about (.*)"
            ],
            "system_control": [
                r"(shut down|power off|turn off) (computer|system|pc)",
                r"restart (computer|system|pc)",
                r"log (out|off)",
                r"(lock|unlock) (computer|system|pc)"
```

```
            ],
            "file_operations": [
                r"create (folder|directory) (.*)",
                r"delete (file|folder|directory) (.*)",
                r"rename (.*) to (.*)",
                r"move (.*) to (.*)"
            ],
            "note_taking": [
                r"take a note (.*)",
                r"write down (.*)",
                r"remember (.*)",
                r"make a note (.*)"
            ],
            "date_time": [
                r"what (time|date) is it",
                r"tell me the (time|date)",
                r"current (time|date)"
            ],
            "help": [
                r"help",
                r"what can you do",
                r"list commands",
                r"show commands"
            ]
        }

        # Initialize stop words
        self.stop_words = set(stopwords.words('english'))

        self.logger.debug("Command Processor initialized")

    def process_command(self, command_text):
        """
        Process a command string and determine the action to take.

        Args:
            command_text: The command text to process

        Returns:
            dict: The result of the command execution
        """
        command_text = command_text.lower().strip()
        self.logger.debug(f"Processing command: {command_text}")

        # Check for matches against our command patterns
        for category, patterns in self.command_patterns.items():
            for pattern in patterns:
                match = re.match(pattern, command_text, re.IGNORECASE)
                if match:
                    self.logger.debug(f"Command matched category: {category}")
```

```
                    # Call the appropriate method based on the command category
                    if category == "open_app":
                        app_name = match.group(1).strip()
                        return self.open_application(app_name)

                    elif category == "search_web":
                        query = match.group(1).strip()
                        return self.search_web(query)

                    elif category == "system_control":
                        action = match.group(1).lower()
                        return self.system_control(action)

                    elif category == "file_operations":
                        # This is a simplified implementation
                        return {
                            "status": "error",
                            "action": "perform file operations",
                            "message": "File operations are not fully implemented yet."
                        }

                    elif category == "note_taking":
                        content = match.group(1).strip()
                        return self.take_note(content)

                    elif category == "date_time":
                        query = match.group(1).lower()
                        return self.get_date_time(query)

                    elif category == "help":
                        return self.provide_help()

        # If no pattern matches, try to infer the intent based on keywords
        tokens = word_tokenize(command_text)
        filtered_tokens = [w for w in tokens if w not in self.stop_words]

        # Check for keywords in the filtered tokens
        if any(word in filtered_tokens for word in ["open", "launch", "start", "run"]):
            # Find potential app name after the command word
            for i, word in enumerate(tokens):
                if word in ["open", "launch", "start", "run"] and i+1 < len(tokens):
                    app_name = tokens[i+1]
                    return self.open_application(app_name)

        elif any(word in filtered_tokens for word in ["search", "google", "look", "find"]):
            # Extract the search query
            search_terms = [w for w in tokens if w not in ["search", "for", "google", "look",
"up", "find", "information", "about"]]
            if search_terms:
```

```
                query = " ".join(search_terms)
                return self.search_web(query)

        # If we can't determine the intent, return an error
        return {
            "status": "error",
            "action": "understand your command",
            "message": "Sorry, I didn't understand that command. Try asking for 'help' to see what I can do."
        }

    def open_application(self, app_name):
        """
        Open an application based on its name.

        Args:
            app_name: The name of the application to open

        Returns:
            dict: The result of the operation
        """
        self.logger.debug(f"Attempting to open application: {app_name}")

        # Common application mapping
        app_mapping = {
            "browser": "google-chrome",
            "chrome": "google-chrome",
            "firefox": "firefox",
            "text editor": "notepad",
            "notepad": "notepad",
            "calculator": "calc",
            "file explorer": "explorer",
            "explorer": "explorer",
            "terminal": "cmd"
        }

        # Map common names to actual executable names
        executable = app_mapping.get(app_name.lower(), app_name.lower())

        try:
            # Attempt to open the application
            # This is a simplified version and might not work for all applications
            if os.name == 'nt':  # Windows
                os.startfile(executable)
            else:  # Linux/Mac
                subprocess.Popen([executable], stdout=subprocess.PIPE, stderr=subprocess.PIPE)

            return {
                "status": "success",
```

```
                "action": f"opened {app_name}",
                "details": {"app_name": app_name}
            }
        except Exception as e:
            self.logger.error(f"Error opening application {app_name}: {e}")
            return {
                "status": "error",
                "action": f"open {app_name}",
                "message": f"Could not open {app_name}. {str(e)}"
            }

    def search_web(self, query):
        """
        Search the web for a query.

        Args:
            query: The search query

        Returns:
            dict: The result of the operation
        """
        self.logger.debug(f"Searching web for: {query}")

        try:
            # Format the query and open a browser with the search
            formatted_query = query.replace(' ', '+')
            webbrowser.open(f"https://www.google.com/search?q={formatted_query}")

            return {
                "status": "success",
                "action": f"searched the web for '{query}'",
                "details": {"query": query}
            }
        except Exception as e:
            self.logger.error(f"Error searching web for {query}: {e}")
            return {
                "status": "error",
                "action": f"search for {query}",
                "message": f"Could not search the web. {str(e)}"
            }

    def system_control(self, action):
        """
        Control system functions like shutdown, restart, etc.

        Args:
            action: The system control action to perform

        Returns:
            dict: The result of the operation
```

```
        """
        self.logger.debug(f"System control action: {action}")

        # Note: These operations may require administrative privileges
        try:
            if "shut down" in action or "power off" in action or "turn off" in action:
                # This is just a simulation for safety
                return {
                    "status": "success",
                    "action": "prepared to shut down the computer",
                    "message": "This is a simulated shutdown. In a real application, the system would shut down.",
                    "details": {"action": "shutdown"}
                }
            elif "restart" in action:
                # This is just a simulation for safety
                return {
                    "status": "success",
                    "action": "prepared to restart the computer",
                    "message": "This is a simulated restart. In a real application, the system would restart.",
                    "details": {"action": "restart"}
                }
            elif "log out" in action or "log off" in action:
                # This is just a simulation for safety
                return {
                    "status": "success",
                    "action": "prepared to log out",
                    "message": "This is a simulated log out. In a real application, you would be logged out.",
                    "details": {"action": "logout"}
                }
            elif "lock" in action:
                # This is just a simulation for safety
                return {
                    "status": "success",
                    "action": "prepared to lock the computer",
                    "message": "This is a simulated lock. In a real application, the system would be locked.",
                    "details": {"action": "lock"}
                }
            else:
                return {
                    "status": "error",
                    "action": f"perform system action '{action}'",
                    "message": f"Unknown system action: {action}"
                }
        except Exception as e:
            self.logger.error(f"Error performing system action {action}: {e}")
            return {
                "status": "error",
```

```
                "action": f"perform system action '{action}'",
                "message": f"Could not perform system action. {str(e)}"
            }

    def take_note(self, content):
        """
        Save a note with the given content.

        Args:
            content: The content of the note

        Returns:
            dict: The result of the operation
        """
        self.logger.debug(f"Taking note: {content}")

        try:
            # Create notes directory if it doesn't exist
            notes_dir = os.path.join(os.path.expanduser("~"), "voice_assistant_notes")
            os.makedirs(notes_dir, exist_ok=True)

            # Create a timestamp for the filename
            import datetime
            timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")

            # Write the note to a file
            filename = os.path.join(notes_dir, f"note_{timestamp}.txt")
            with open(filename, 'w') as f:
                f.write(content)

            return {
                "status": "success",
                "action": "saved your note",
                "details": {"content": content, "file": filename}
            }
        except Exception as e:
            self.logger.error(f"Error taking note: {e}")
            return {
                "status": "error",
                "action": "save your note",
                "message": f"Could not save the note. {str(e)}"
            }

    def get_date_time(self, query):
        """
        Get the current date or time.

        Args:
            query: Whether to get the date or time

```

```
        Returns:
            dict: The result of the operation
        """
        self.logger.debug(f"Getting date/time for query: {query}")

        try:
            import datetime
            now = datetime.datetime.now()

            if "time" in query:
                formatted_time = now.strftime("%I:%M %p")
                return {
                    "status": "success",
                    "action": "checked the time",
                    "message": f"The current time is {formatted_time}",
                    "details": {"time": formatted_time}
                }
            elif "date" in query:
                formatted_date = now.strftime("%A, %B %d, %Y")
                return {
                    "status": "success",
                    "action": "checked the date",
                    "message": f"Today is {formatted_date}",
                    "details": {"date": formatted_date}
                }
            else:
                return {
                    "status": "error",
                    "action": "get the date or time",
                    "message": "I'm not sure if you wanted the date or time."
                }
        except Exception as e:
            self.logger.error(f"Error getting date/time: {e}")
            return {
                "status": "error",
                "action": "get the date or time",
                "message": f"Could not retrieve the date or time. {str(e)}"
            }

    def provide_help(self):
        """
        Provide help information about available commands.

        Returns:
            dict: The result of the operation
        """
        self.logger.debug("Providing help information")

        help_text = """
        Here are the commands I can understand:
```

```
        1. Open applications: "open [app name]", "launch [app name]"
        2. Search the web: "search for [query]", "google [query]"
        3. System control: "shut down computer", "restart computer", "lock computer"
        4. Take notes: "take a note [content]", "remember [content]"
        5. Date and time: "what time is it", "what date is it"
        6. Help: "help", "what can you do"

        You can speak naturally, and I'll try to understand your intent.
        """

        return {
            "status": "success",
            "action": "provided help information",
            "message": help_text,
            "details": {"help_text": help_text}
        }

```