```python
# utils/gemini_client.py

import os
import google.generativeai as genai
from dotenv import load_dotenv

load_dotenv()

API_KEY = os.getenv("GEMINI_API_KEY")

if not API_KEY:
    raise ValueError("GEMINI_API_KEY not found in environment variables.")

genai.configure(api_key=API_KEY)

model = genai.GenerativeModel("gemini-2.0-flash")

def generate_answer(prompt: str) -> str:
    try:
        response = model.generate_content(
            prompt,
            generation_config={"temperature": 0.7}
        )
        return response.text.strip()
    except Exception as e:
        return f"[ERROR] {str(e)}"
```

```python
# utils/io_utils.py

import json

def load_json(path: str):
    with open(path, "r", encoding="utf-8") as f:
        return json.load(f)

def save_json(data, path: str):
    with open(path, "w", encoding="utf-8") as f:
        json.dump(data, f, indent=2, ensure_ascii=False)
```

```python
# main.py

from utils.io_utils import load_json, save_json
from utils.gemini_client import generate_answer
import re
import os
import argparse
import pandas as pd
import time

def parse_args():
    parser = argparse.ArgumentParser(description="Run prompting with Gemini API.")
    parser.add_argument(
        "--output",
        type=str,
        default="generate_res/dummy.json",
        help="Output file path (default: generate_res/dummy.json)"
    )
    parser.add_argument(
        "--task",
        type=str,
        default="musique",
        help="Dataset type ? (default: musique)"
    )

    return parser.parse_args()

def get_controller(task):
    if task == "musique":
        from musique_controller import MusiqueController
        return MusiqueController()
    elif task == "2wiki":
        from wiki_controller import WikiController
        return WikiController()
    else:
        raise ValueError(f"Unsupported task: {task}")

def main():
    args = parse_args()
    print("Arguments parsed:")
    for arg, value in vars(args).items():
        print(f"{arg}: {value}")
    print("\n")

    controller = get_controller(args.task)

    # get decomposition chains
    controller.solve()

if __name__ == "__main__":
    main()
```

Richard Rafer Guy – 222117056

```python
# wiki_controller.py

from utils.io_utils import load_json, save_json
from utils.gemini_client import generate_answer
import prompts.wiki.template as wiki_template
import re
import os
import time
import json
import ast

class WikiController:
    def __init__(self):
        self.dataset = load_json('dataset/2wiki.json')

    def print_questions(self):
        """Prints all questions from the dataset."""
        for entry in self.dataset:
            print(entry['question'])

    def replace_references(self, s, result_array):
        def replacer(match):
            index = int(match.group(1))-1  # Ambil angka dari #3 misalnya
            if 0 <= index < len(result_array):
                return result_array[index]
            else:
                return f"<Invalid index #{index}>"

        # Ganti semua #angka dengan nilai yang sesuai
        return re.sub(r"#(\d+)", replacer, s)

    def get_generated_answer(self, ans):
        if "Answer:" in ans:
            process_answer = ans.split("Answer: ", 1)[1].strip()
        else:
            process_answer = ans.strip()
        return process_answer

    def clean_text(self, text):
        return re.sub(r'\(.*?\)|\[.*?\]', '', text).strip()

    def get_references(self, qs):
        references = re.findall(r'#(\d+)', qs)
        return [int(ref) for ref in references]

    def get_references_array(self, qs, result_answers):
        references = result_answers[self.get_references(qs)[0]-1]
        array_ver = json.loads(references)
```

```
            return array_ver

        def implic_RAG(self, question, entry):
            formatted_context_list = self.get_formatted_context(entry)
            fulltext_context = ""
            for context in formatted_context_list:
                fulltext_context += context

            prompt = wiki_template.impli_rag_template.replace('{question}',
                    question).replace('{context}', fulltext_context)
            generated_ans = generate_answer(prompt)

            start = generated_ans.find("C1:")
            extracted_output = generated_ans[start:]

            return extracted_output


        def chain_processing(self, qs_lines, entry):
            result_answers = []

            for index, qs in enumerate(qs_lines):
                # Cari yang dalam tanda kurung bulat (command)
                command_match = re.search(r"\[(.*?)\]", qs)
                command = command_match.group(1) if command_match else ''

                process_answer = ""

                if command == "get_ent_qa":
                    formatted_qs = self.replace_references(qs, result_answers)
                    prompt = wiki_template.get_ent_qa_template.replace('{input}',
                            formatted_qs).replace('{context}', entry['question'])
                    generated_ans = generate_answer(prompt)
                    print(f"Generated answer for get_ent_qa: {generated_ans}")
                    process_answer = self.get_generated_answer(generated_ans)
                    print(f"Processed answer for get_ent_qa: {process_answer}")

                elif command == "get_atr_qa":
                    references = self.get_references_array(qs, result_answers)
                    ans_list = []
                    for ref in references:
                        # format and generate prompt template
                        formatted_qs = re.sub(r"#\d+", ref, qs)
                        cleaned_text = self.clean_text(formatted_qs)
                        print(f"Formatted question: {cleaned_text}")

                        context = self.implic_RAG(cleaned_text, entry)
                        print(f"Context: {context}\n")
                        prompt = wiki_template.get_atr_qa_template
                                 .replace('{question}', cleaned_text)
                                 .replace('{context}', context)

                        generated_ans = generate_answer(prompt)
                        formatted_ans = self.get_generated_answer(generated_ans)
                        ans_list.append(f"{ref} => {formatted_ans}")

                        print(ans_list)

                    process_answer = ans_list
                elif command == "comp_qa":
                    latest_ans = result_answers[-1]
                    full_context = ""
                    for ans in latest_ans:
                        full_context += f"{ans}\n"
                    prompt = wiki_template.comp_qa_template.replace('{question}', qs).replace('{context}', full_context)
                    generated_ans = generate_answer(prompt)
                    process_answer = self.get_generated_answer(generated_ans)

                    print(f"Processed answer for comp_qa: {process_answer}")
                elif command == "EOQ":
                    Break

                print(f"Processing step {index+1}: {command} -> {process_answer}")
                result_answers.append(process_answer)


            return result_answers


        def get_formatted_context(self, entry):
            formatted_data = []
            formatted_entry_context = json.loads(entry['context'])
            for index, context in enumerate(formatted_entry_context):
                fulltext = ""
                title = context[0]
                for text in context[1]:
                    fulltext += text

                formatted_context_text = f"{index+1}. {title} => {fulltext}\n"
                formatted_data.append(formatted_context_text)

            return formatted_data

        def generate_chain(self, question):
            template = wiki_template.decomp_template
```

```python
        prompt = template.replace('{input}', question)
        decomp_chain = generate_answer(prompt)

        qs_lines = [line.replace("QS: ", "").strip() for line in decomp_chain.splitlines() if line.startswith("QS:")]
        print(f"Decomposition chain generated: {qs_lines}")
        return qs_lines

    def solve(self):

        output_log = []
        score = 0

        for entry in self.dataset:

            qs_lines = self.generate_chain(entry['question'])
            output = self.chain_processing(qs_lines, entry)

            def check_answer(output):
                return entry['answer'].lower() == output[-1].lower() if output else False

            output_log.append({
                'question': entry['question'],
                'qs_lines': qs_lines,
                "output": output,
                "is_correct": check_answer(output),
            })

            current_score = 1 if check_answer(output) else 0
            score += current_score

            save_json(output_log, 'generate_res/dummy.json')
            time.sleep(60)  # To avoid hitting rate limits

        print(f"final score: {score}")
        return True
```

```python
# musique_controller.py

from utils.io_utils import load_json, save_json
from utils.gemini_client import generate_answer
import prompts.musique.template as musique_template
import re
import os
import time
import json
import ast

class MusiqueController:
    def __init__(self):
        self.dataset = load_json('dataset/musique.json')

    def print_questions(self):
        """Prints all questions from the dataset."""
        for entry in self.dataset:
            print(entry['question'])

    def get_generated_answer(self, ans):
        if "Answer:" in ans:
            process_answer = ans.split("Answer: ", 1)[1].strip()
        else:
            process_answer = ans.strip()
        return process_answer

    def get_fullcontext (self, paragraphs):
        fulltext_context = ""
        for index, paragraph in enumerate(paragraphs):
            fulltext_context += f"{paragraph['title']} =>
                                {paragraph['paragraph_text']}\n"
        return fulltext_context

    def implic_RAG(self, question, paragraphs):

        fulltext_context = self.get_fullcontext(paragraphs)

        prompt = musique_template.impli_rag_template.replace('{question}',
                question).replace('{context}', fulltext_context)
        generated_ans = generate_answer(prompt)

        start = generated_ans.find("C1:")
        extracted_output = generated_ans[start:]

        output_lines = extracted_output.splitlines()
        output_cleaned = [line.split(":", 1)[-1].strip() for line in output_lines]

        # Gabungkan kembali konteks menjadi string yang bersih
        cleaned_output = "\n".join(output_cleaned)

        return cleaned_output

    def chain_processing(self, qs_lines, paragraphs):
        result_answers = []

        for index, qs in enumerate(qs_lines):
            process_answer = ""

            if index == 0:
                context = self.get_fullcontext(paragraphs)

                prompt = musique_template.starter_qa.replace('{question}', qs['question']).replace('{context}', context)
                generated_ans = generate_answer(prompt)
                process_answer = self.get_generated_answer(generated_ans)

            else:
                prev_ans = result_answers[-1]
                context = self.get_fullcontext(paragraphs)

                formatted_qs = re.sub(r"#\d+", prev_ans, qs['question'])

                prompt = musique_template.finisher_template.replace('{question}', formatted_qs).replace('{context}', context)
                generated_ans = generate_answer(prompt)
                process_answer = self.get_generated_answer(generated_ans)

            result_answers.append(process_answer)
        return result_answers

    def solve(self):

        output_log = []
        score = 0

        for entry in self.dataset:

            output = self.chain_processing(entry['question_decomposition'], entry['paragraphs'])
            print("processing index: ", entry['id'])
            output_log.append({
                'question': entry['question'],
                'qs_lines': entry['question_decomposition'],
                'result': output,
                'expected_answer': entry['answer'],
                "is_correct": entry['answer'].lower() == output[-1].lower()
            })

            # break
            save_json(output_log, 'generate_res/dummy.json')
            time.sleep(20)  # To avoid hitting rate limits

        print(f"final score: {score}")
        return True
```

```
# prompts/wiki/template.py

decomp_template = '''
QC: Which film came out first, The Love Route or Engal Aasan?
QS: [get_ent_qa] Which films are being compared in the question?
QS: (foreach)[get_atr_qa] What is the release date of #1?
QS: [comp_qa] Which film came out first based on the release date?

QC: Are Matraville Sports High School and Wabash High School both located in the same country?
QS: [get_ent_qa] Which schools are being compared in the question?
QS: (foreach)[get_atr_qa] What is the location of #1?
QS: [comp_qa] Are the schools located in the same country?

QC: Are Alison Skipper and Diane Gilliam Fisher from the same country?
QS: [get_ent_qa] Who are the people being compared in the question?
QS: (foreach)[get_atr_qa] What is the nationality of #1?
QS: [comp_qa] Do they have the same nationality?

QC: Do the movies Bloody Birthday and The Beckoning Silence, originate from the same country?
QS: [get_ent_qa] Which movies are being compared in the question?
QS: (foreach)[get_atr_qa] What is the country of origin of #1?
QS: [comp_qa] Do the movies originate from the same country?

QC: Are both businesses, Vakıfbank and Infopro Sdn Bhd, located in the same country?
QS: [get_ent_qa] Which businesses are being compared in the question?
QS: (foreach)[get_atr_qa] What is the location of #1?
QS: [comp_qa] Are the businesses located in the same country?

QC: Does Mukasa Mbidde have the same nationality as Erich Maas?
QS: [get_ent_qa] Who are the people being compared in the question?
QS: (foreach)[get_atr_qa] What is the nationality of #1?
QS: [comp_qa] Do they have the same nationality?

QC: {input}

Output:
QS: <QS-1>
QS: <QS-2>
.....
QS: <QS-N>

'''

```

```
get_ent_qa_template = '''
QC: Are Matraville Sports High School and Wabash High School both located in the same country?
Q: Which schools are being compared in the question?
Answer: ["Matraville Sports High School", "Wabash High School"]

QC: Are Alison Skipper and Diane Gilliam Fisher from the same country?
Q: Which people are being compared in the question?
Answer: ["Alison Skipper", "Diane Gilliam Fisher"]

QC: Do the movies Bloody Birthday and The Beckoning Silence, originate from the same country?
Q: Which movies are being compared in the question?
Answer: ["Bloody Birthday", "The Beckoning Silence"]

QC: Are both businesses, Vakıfbank and Infopro Sdn Bhd, located in the same country?
Q: Which businesses are being compared in the question?
Answer: ["Vakıfbank", "Infopro Sdn Bhd"]

QC: Did the movies Pony Express (Film) and The Da Vinci Code (Film), originate from the same country?
Q: Which people are being compared in the question?
Answer: ["Sam Earle", "Felix Luckeneder"]

QC: Are Sam Earle and Felix Luckeneder from the same country?
Q: Which locations are being compared in the question?
Answer: ["Lesser Slave Lake", "Medeweger See"]

QC: Are both Lesser Slave Lake and Medeweger See located in the same country?
Q: Which films are being compared in the question?
Answer: ["Alsino And The Condor", "1922 (2017 Film)"]

QC: Are Alsino And The Condor and 1922 (2017 Film) both from the same country?
Q: Which bands are being compared in the question?
Answer: ["Cinematic Sunrise", "Kingston Falls"]

QC: Are the movies Carnival Of Souls and Uvanga, from the same country?
Q: Which schools are being compared in the question?
Answer: ["St. Mary High School (Rutherford, New Jersey)", "Mother Teresa High School"]

Input:
QC: {context}
Q: {input}

Output (your answer MUST be in the same format "Answer: <your answer here>", you MUST add "Answer: " before your answer):
```

```
Answer: <your answer here>
'''

get_atr_qa_template = '''
Q: {question}
Context:
{context}

Output:
Answer: <Your answer>
'''

comp_qa_template = '''
QS: {question}
Context:
{context}

Output (your answer MUST be in the same format "Answer: <your answer here>", you MUST add "Answer: " before your answer):
Answer: <your answer here>
'''

impli_rag_template = '''
Given the following context data and a specific question, please provide the top 3 most relevant contexts that help answer the question. The output should be formatted as follows:

Context:
C1: "..."
C2: "..."
C3: "..."

Context Data:
{context}

Question: {question}

Output:
C1: <Best Context>
C2: <Second Best Context>
C3: <Third Best Context>
'''
```

```
# prompts/musique/template.py

starter_qa = '''
Input:
Q: {question}
```

```
Context:
{context}

Output (your answer MUST be in the same format "Answer: <your answer here>", you MUST add "Answer: " before your answer):
Answer: <your answer here>
'''

finisher_template = '''
Input:
Q: {question}
Context:
{context}

Output (your answer MUST be in the same format "Answer: <your answer here>", you MUST add "Answer: " before your answer):
Answer: <your answer here>
'''

impli_rag_template = '''
Given the following context data and a specific question, please provide the top 3 most relevant contexts that help answer the question. The output should be formatted as follows:

Context:
C1: "..."
C2: "..."
C3: "..."

Context Data:
{context}

Question: {question}

Output:
C1: <Best Context>
C2: <Second Best Context>
C3: <Third Best Context>
'''
```