Muzammil Ahmed FA22-BSE-038

I make changes in

dbProject

-views.py file . command to run this program is ‘python manage.py runserver’

Five Major Architectural Problems in Python Systems

1. **Problem: Inefficient Handling of Concurrency**
   - **Description:** Python's Global Interpreter Lock (GIL) limits the execution of multiple threads in CPython, causing inefficiencies in multi-threaded applications.
   - **Solution:** Use **multiprocessing** or frameworks like **asyncio** to handle concurrency.
   - **Example:** Django channels were introduced to handle real-time updates using asynchronous programming.
2. **Problem: Monolithic Django Projects**
   - **Description:** Large Django projects often become monolithic, making it hard to maintain or scale.
   - **Solution:** Split projects into Django apps or use a **microservices approach** with frameworks like **FastAPI** or **Flask**.
   - **Example:** Companies transitioning from a monolithic Django architecture to FastAPI for modularity and speed.
3. **Problem: Dependency Management Issues**
   - **Description:** Managing dependencies across multiple Python projects can lead to version conflicts and "dependency hell."
   - **Solution:** Use tools like **virtualenv** or **Poetry** to isolate environments and manage dependencies effectively.
   - **Example:** A project migrated from global installations to Poetry, reducing dependency conflicts.
4. **Problem: Inefficient Data Processing for Big Data**
   - **Description:** Python’s standard data processing tools are not optimized for large datasets, leading to performance bottlenecks.
   - **Solution:** Use distributed processing frameworks like **Apache Spark** (via PySpark) or libraries like **Dask**.
   - **Example:** Processing large datasets moved from vanilla Pandas to Dask for parallel processing.
5. **Problem: Scalability in Web Applications**
   - **Description:** Traditional WSGI frameworks like Django and Flask can struggle with high concurrent loads.

- **Solution:** Switch to **ASGI frameworks** like FastAPI or add scalability layers with tools like **Gunicorn** and **NGINX**.
- **Example:** Flask apps scaled using Gunicorn and load balancers.

Replicating and Solving a Python Problem

We will replicate **Problem 1: Inefficient Handling of Concurrency** and solve it using Python's `asyncio`.

```
import time


def fetch_data(task_id):

   print(f"Fetching data for task {task_id}...")

   time.sleep(2)  # Simulating a blocking I/O operation

   print(f"Task {task_id} completed.")


# Sequential execution

start = time.time()

for i in range(5):

   fetch_data(i)

print(f"Total time taken: {time.time() - start:.2f} seconds")
```

Solving the Problem with `asyncio`

We can use `asyncio` for concurrent execution of I/O-bound tasks.

```
python
Copy code
import asyncio

async def fetch_data(task_id):
    print(f"Fetching data for task {task_id}...")
    await asyncio.sleep(2)  # Non-blocking delay
    print(f"Task {task_id} completed.")

async def main():
    tasks = [fetch_data(i) for i in range(5)]
    await asyncio.gather(*tasks)

# Concurrent execution
start = time.time()
asyncio.run(main())
print(f"Total time taken: {time.time() - start:.2f} seconds")
```

Here are **five major architectural problems** specific to Python-based systems and their solutions. I'll also include a Python-related example for the second part to replicate and solve a problem.

Part 1: Five Major Architectural Problems in Python Systems

1. **Problem: Inefficient Handling of Concurrency**
   - **Description:** Python's Global Interpreter Lock (GIL) limits the execution of multiple threads in CPython, causing inefficiencies in multi-threaded applications.
   - **Solution:** Use **multiprocessing** or frameworks like **asyncio** to handle concurrency.
   - **Example:** Django channels were introduced to handle real-time updates using asynchronous programming.
2. **Problem: Monolithic Django Projects**
   - **Description:** Large Django projects often become monolithic, making it hard to maintain or scale.
   - **Solution:** Split projects into Django apps or use a **microservices approach** with frameworks like **FastAPI** or **Flask**.
   - **Example:** Companies transitioning from a monolithic Django architecture to FastAPI for modularity and speed.
3. **Problem: Dependency Management Issues**
   - **Description:** Managing dependencies across multiple Python projects can lead to version conflicts and "dependency hell."
   - **Solution:** Use tools like **virtualenv** or **Poetry** to isolate environments and manage dependencies effectively.
   - **Example:** A project migrated from global installations to Poetry, reducing dependency conflicts.
4. **Problem: Inefficient Data Processing for Big Data**
   - **Description:** Python's standard data processing tools are not optimized for large datasets, leading to performance bottlenecks.
   - **Solution:** Use distributed processing frameworks like **Apache Spark** (via PySpark) or libraries like **Dask**.
   - **Example:** Processing large datasets moved from vanilla Pandas to Dask for parallel processing.
5. **Problem: Scalability in Web Applications**

- **Description:** Traditional WSGI frameworks like Django and Flask can struggle with high concurrent loads.
- **Solution:** Switch to **ASGI frameworks** like FastAPI or add scalability layers with tools like **Gunicorn** and **NGINX**.
- **Example:** Flask apps scaled using Gunicorn and load balancers.

Part 2: Replicating and Solving a Python Problem

We will replicate **Problem 1: Inefficient Handling of Concurrency** and solve it using Python's `asyncio`.

Replicating the Problem

In a blocking approach, tasks run sequentially, wasting time waiting for I/O operations.

```
python
Copy code
import time

def fetch_data(task_id):
    print(f"Fetching data for task {task_id}...")
    time.sleep(2)  # Simulating a blocking I/O operation
    print(f"Task {task_id} completed.")

# Sequential execution
start = time.time()
for i in range(5):
    fetch_data(i)
print(f"Total time taken: {time.time() - start:.2f} seconds")
```

Output:

```
text
Copy code
Fetching data for task 0...
Task 0 completed.
Fetching data for task 1...
Task 1 completed.
...
Total time taken: 10.00 seconds
```

Solving the Problem with `asyncio`

We can use `asyncio` for concurrent execution of I/O-bound tasks.

```
python
```

```
Copy code
import asyncio

async def fetch_data(task_id):
    print(f"Fetching data for task {task_id}...")
    await asyncio.sleep(2)  # Non-blocking delay
    print(f"Task {task_id} completed.")

async def main():
    tasks = [fetch_data(i) for i in range(5)]
    await asyncio.gather(*tasks)

# Concurrent execution
start = time.time()
asyncio.run(main())
print(f"Total time taken: {time.time() - start:.2f} seconds")
```

Output:

```
text
Copy code
Fetching data for task 0...
Fetching data for task 1...
Fetching data for task 2...
Fetching data for task 3...
Fetching data for task 4...
Task 0 completed.
Task 1 completed.
...
Total time taken: 2.00 seconds
```

Outcome

- **Original Problem:** The blocking approach caused tasks to execute sequentially, leading to inefficiencies.
- **Solution:** By using `asyncio`, tasks are executed concurrently, significantly reducing the total execution time.