# **Asynchronous (Async) vs Synchronous (Sync) Programming: Concepts**

#### **Synchronous Programming (Sync):**

* **Definition:** In synchronous programming, tasks are executed sequentially, meaning each task must be completed before the next one begins.
* **Blocking Execution:** When a function is called, the program "waits" for the function to complete before moving to the next instruction.
* **Flow Control:** The flow of the program is linear. Each task is completed one after the other.

#### **Asynchronous Programming (Async):**

* **Definition:** In asynchronous programming, tasks are initiated and continue executing other tasks without waiting for a specific task to complete.
* **Non-Blocking Execution:** When a function is called, the program does not wait for the function to be completed. Instead, it moves on to other tasks, allowing multiple tasks to progress simultaneously.
* **Concurrency vs. Parallelism:**
  + **Concurrency** means that multiple tasks are making progress at the same time, but they are not necessarily executing at the same exact moment (useful in async programming).
  + **Parallelism** refers to tasks that are executed at the same time on multiple processors/cores (useful for CPU-bound tasks).
* **Event Loop:** Asynchronous programming relies on an event loop (a core part of async). The event loop schedules tasks, executes them, and switches between them as they await completion.

### **Understanding Asynchronous Functionality in FastAPI**

FastAPI is built on top of Starlette for the web parts and Pydantic for the data parts, both of which are fast and capable of handling async I/O operations.

### **Important Asynchronous Features in FastAPI**

* **Async routes**: Declaring asynchronous routes using async def.
* **Awaiting**: Using await for non-blocking calls inside an async def function.
* **Event loops**: The core mechanism to handle async I/O.
* **Concurrency with HTTP requests**: Multiple requests can be handled concurrently, which improves performance.

### **The choice between asyncio.sleep Vs Time.Sleep**

Here’s why you should use await asyncio.sleep instead of time.sleep when working with async functionality:

* **await asyncio.sleep(seconds)**:
  + **Asynchronous Sleep**: asyncio.sleep is a coroutine provided by the asyncio module. When you use await asyncio.sleep, it yields control back to the event loop, allowing other tasks to run concurrently while the current task is waiting.
  + **Non-Blocking**: This means the event loop can continue processing other tasks or handling incoming requests, improving the overall efficiency and responsiveness of your application.
* **time.sleep(seconds)**:
  + **Synchronous Sleep**: time.sleep is a blocking call. When you use time.sleep, the entire thread is blocked for the specified duration, meaning that no other tasks or I/O operations can be processed during this time.
  + **Blocking**: This can lead to reduced concurrency and responsiveness, as the thread waits idly while the sleep operation is in progress.

### **What is an Event Loop?**

**Definition**: An event loop is a core component in asynchronous programming that manages and schedules the execution of asynchronous tasks and events.

**How It Works**:

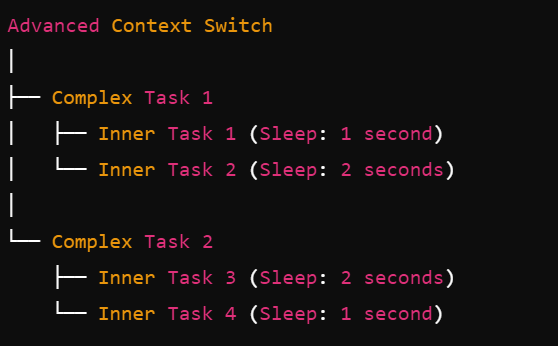
* **Single Threaded**: The event loop runs in a single thread and handles multiple tasks by switching between them, rather than running them simultaneously in separate threads.
* **Task Management**: It manages tasks such as I/O operations, timers, and events. When a task is waiting (e.g., for I/O), the event loop can switch to other tasks that are ready to execute.
* **Non-Blocking**: The event loop facilitates non-blocking operations by allowing other tasks to run while waiting for I/O operations or other asynchronous events.

**Example**: In FastAPI (built on Starlette), the event loop is used to handle multiple concurrent HTTP requests efficiently without blocking.

**Summary**

* **Event Loop**: Manages and schedules asynchronous tasks efficiently within a single thread.
* **Thread**: A unit of execution that runs concurrently with other threads, suitable for tasks that can benefit from parallel execution.

**Understanding of asynchronous programming by creating a more complex scenario with multiple functions and tasks. Here’s an example that demonstrates a more intricate setup using async functions, nested tasks, and varying sleep durations to simulate more realistic and complex operations:**



Code

from fastapi import FastAPI

import asyncio

from datetime import datetime

app = FastAPI()

def get\_current\_time():

return datetime.now().strftime("%H:%M:%S.%f")[:-3] # Format to HH:MM:SS.mmm

async def inner\_task\_1():

sleep\_time = 1

start\_time = get\_current\_time()

print(f"Inner Task 1 started at {start\_time} (Sleep time: {sleep\_time} seconds)")

await asyncio.sleep(sleep\_time) # Simulate a task that takes 1 second

end\_time = get\_current\_time()

print(f"Inner Task 1 completed at {end\_time} (Sleep time: {sleep\_time} seconds)")

return "Inner Task 1 result"

async def inner\_task\_2():

sleep\_time = 2

start\_time = get\_current\_time()

print(f"Inner Task 2 started at {start\_time} (Sleep time: {sleep\_time} seconds)")

await asyncio.sleep(sleep\_time) # Simulate a task that takes 2 seconds

end\_time = get\_current\_time()

print(f"Inner Task 2 completed at {end\_time} (Sleep time: {sleep\_time} seconds)")

return "Inner Task 2 result"

async def complex\_task\_1():

start\_time = get\_current\_time()

print(f"Complex Task 1 started at {start\_time}")

result1 = await inner\_task\_1()

result2 = await inner\_task\_2()

end\_time = get\_current\_time()

print(f"Complex Task 1 completed at {end\_time}")

return {"result1": result1, "result2": result2}

async def inner\_task\_3():

sleep\_time = 2

start\_time = get\_current\_time()

print(f"Inner Task 3 started at {start\_time} (Sleep time: {sleep\_time} seconds)")

await asyncio.sleep(sleep\_time) # Simulate a task that takes 2 seconds

end\_time = get\_current\_time()

print(f"Inner Task 3 completed at {end\_time} (Sleep time: {sleep\_time} seconds)")

return "Inner Task 3 result"

async def inner\_task\_4():

sleep\_time = 1

start\_time = get\_current\_time()

print(f"Inner Task 4 started at {start\_time} (Sleep time: {sleep\_time} seconds)")

await asyncio.sleep(sleep\_time) # Simulate a task that takes 1 second

end\_time = get\_current\_time()

print(f"Inner Task 4 completed at {end\_time} (Sleep time: {sleep\_time} seconds)")

return "Inner Task 4 result"

async def complex\_task\_2():

start\_time = get\_current\_time()

print(f"Complex Task 2 started at {start\_time}")

result3 = await inner\_task\_3()

result4 = await inner\_task\_4()

end\_time = get\_current\_time()

print(f"Complex Task 2 completed at {end\_time}")

return {"result3": result3, "result4": result4}

@app.get("/advanced-context-switch")

async def advanced\_context\_switch():

# Start all complex tasks concurrently

complex\_task\_1\_result, complex\_task\_2\_result = await asyncio.gather(

complex\_task\_1(),

complex\_task\_2()

)

# All complex tasks are awaited and completed here

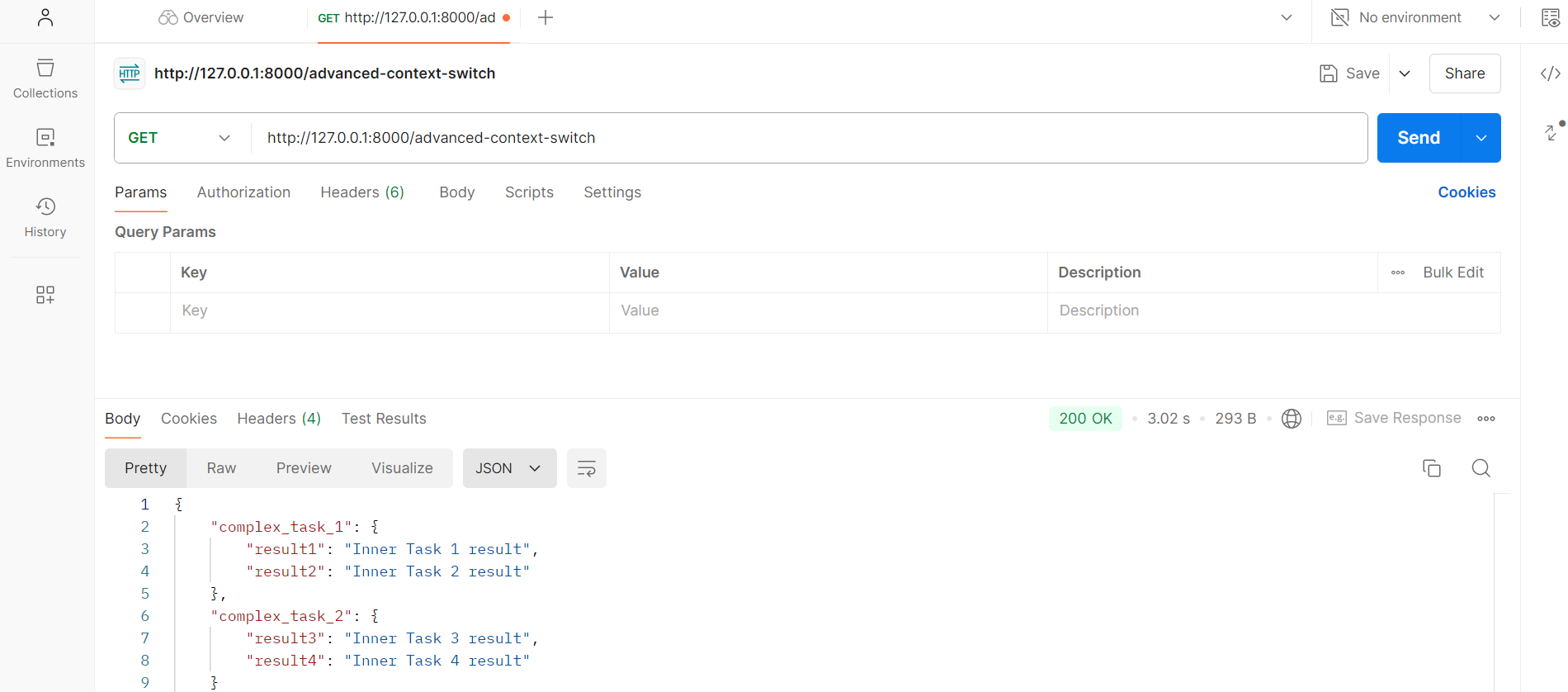
return {

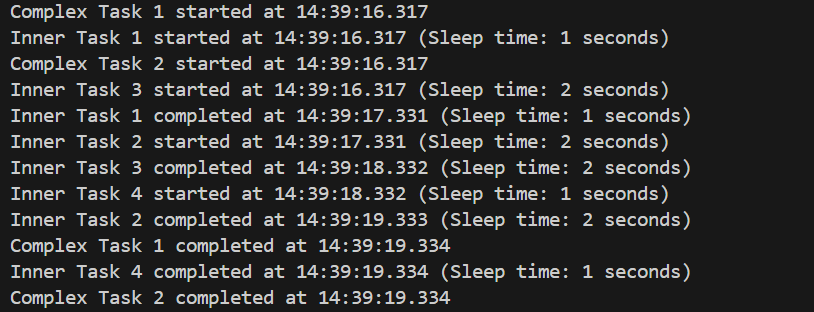
"complex\_task\_1": complex\_task\_1\_result,

"complex\_task\_2": complex\_task\_2\_result

}

API Call Completion



Output of the above code (Async functionality in Context Switching):

### **Interestingly, this below code output explains well what if code was in depicting sync functionality:**

### 

### **Additional Definitions:**

### **Context Switching**

**Context Switching** generally refers to the process by which a CPU switches from one process or thread to another. This involves saving the state of the current process and loading the state of the next one. The goal is to efficiently manage multiple tasks in a system where resources are shared.

**Asynchronous Functionality**

**Asynchronous Programming** allows for non-blocking operations, enabling a program to initiate a task and then continue executing other tasks while waiting for the initiated task to complete. This is particularly useful in I/O-bound operations, like network requests or file operations, where waiting for the operation to complete would otherwise block the execution of the rest of the program.