Designing Efficient Workflows with AsyncIO

Dan Tofan

PhD

@dan_tofan | https://programmingwithdan.com

Managing Event Loops

Best practices

Avoid Blocking Code in the Event Loop

```
async def bad_practice():
    time.sleep(1)
    print("Finished the blocking code")


async def workaround():
    await asyncio.to_thread(time.sleep, 1)
    print("Finished without blocking")
```

Never Create Multiple Event Loops in the Same Thread

Close Resources Properly

Prevent leaks

Clean shutdown

Increase stability

How to Close Resources Properly

```
async def fetch(session, url):
    async with session.get(url) as response:
        return await response.json()


async def fetch_all():
    async with ClientSession() as session:
        tasks = [fetch(session, url) for url in API_URLS]
        results = await asyncio.gather(*tasks)
        print(results)
```

Custom Event Loops

Ensure proper cleanup

Handle exceptions

Log important events

```python
class TimingEventLoop(
        asyncio.SelectorEventLoop):

 def run_until_complete(self, f):
  start_time = time.perf_counter()

  result = super().run_until_complete(f)

  end_time = time.perf_counter()
  print(f"{end_time-start_time}")

  return result



class TimingEventLoopPolicy(
    asyncio.DefaultEventLoopPolicy):

 def new_event_loop(self):
        return TimingEventLoop()
```

◀ **Extend an existing event loop**

◀ **Override to measure execution time**

◀ **Call parent implementation**

◀ **Calculate and print duration**

◀ **Extend existing event loop policy**

◀ **Returns instance of the custom event loop**

Demo: Implementing a Custom Event Loop

Managing Tasks and Coroutines

Scheduling tasks

Composing coroutines

Gathering Multiple Tasks

```
customer, product = await asyncio.gather(
                               get_customer(customer_id),
                               get_product(product_id))
```

```
async def fetch_all():
    async with ClientSession() as session:
         tasks = [fetch(session, url) for url in API_URLS]
         results = await asyncio.gather(*tasks)
         print(results)
```

Grouping Tasks

```
async with asyncio.TaskGroup() as tg:
    customer_task = tg.create_task(get_customer(customer_id))
    product_task = tg.create_task(get_product(product_id))


customer = customer_task.result()
product = product_task.result()
```

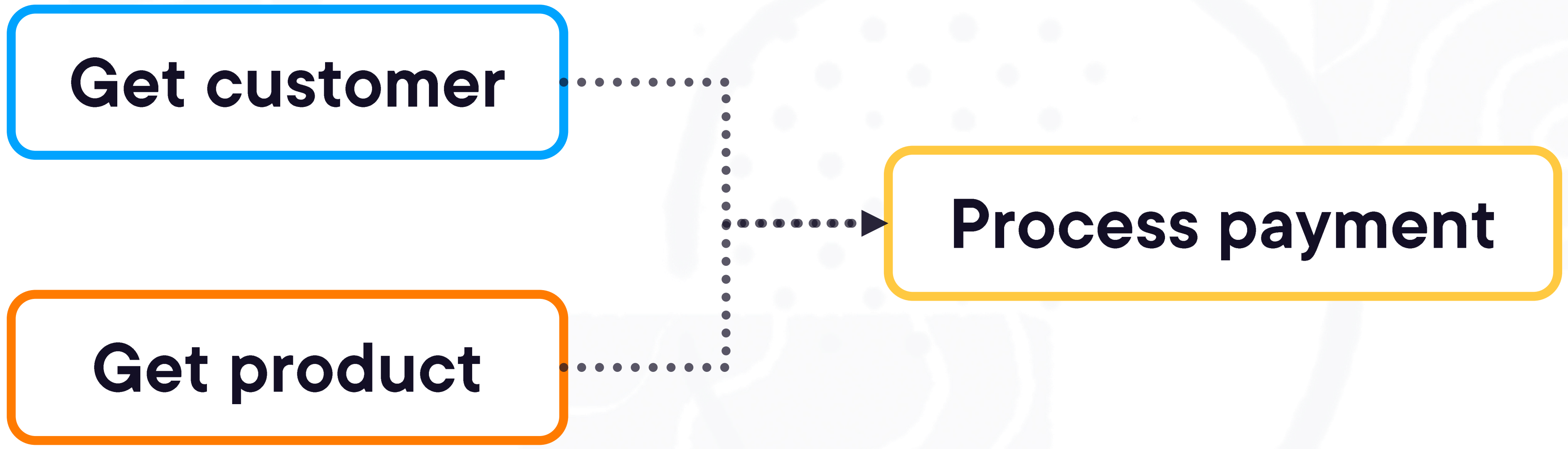
Composing Coroutines

Enable data flow

Write reusable code

Build complex workflows

Demo: Building a Workflow



How to Avoid Blocking Operations

Use async libraries

Offload blocking operations

Chunk large operations

Profile application

Asynchronous Libraries

Use aiohttp for HTTP requests

Use aiofiles for file operations

Use asyncpg for PostgreSQL operations

Why Timeouts Matter

- Prevent indefinite waiting
- Meet responsiveness requirements
- Free resources
- Degrade gracefully
- Prevent cascading failures

How to Manage Timeouts

Set appropriate timeout values

Handle timeout exceptions

Use retry logic

Log timeout events

Test scenarios

Use cancellations

```python
async def long_running():
    try:
        print("Starting task")
        await asyncio.sleep(10)
        print("Task completed")

    except asyncio.CancelledError:
        print("Task is cancelled")
        raise

    finally:
        print("Cleaning up")
```

◀ **Long running coroutine**

◀ **Do the actual work**

◀ **Catch and re-raise to propagate cancellation**

◀ **Clean up resources**

Demo: Optimize Slow Code

Enforcing timeouts