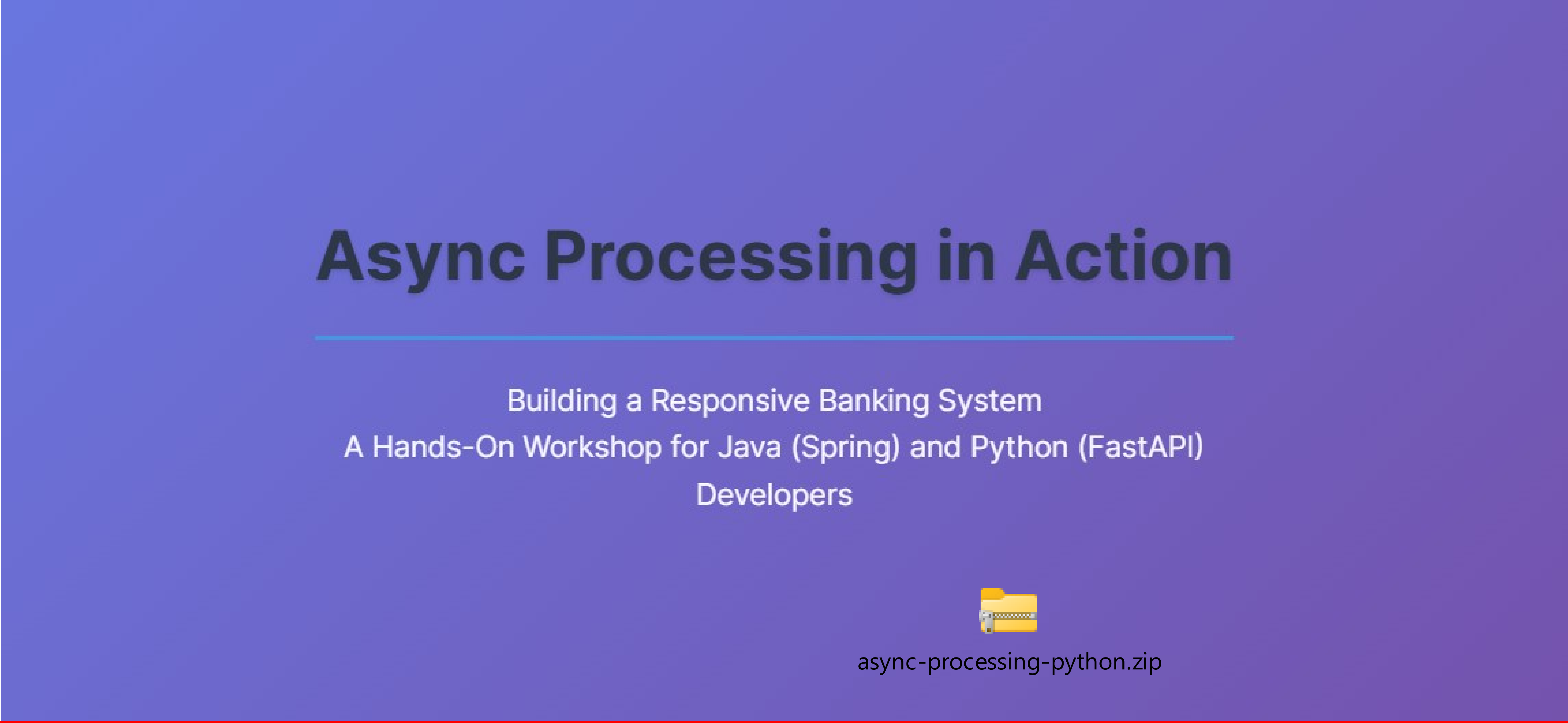


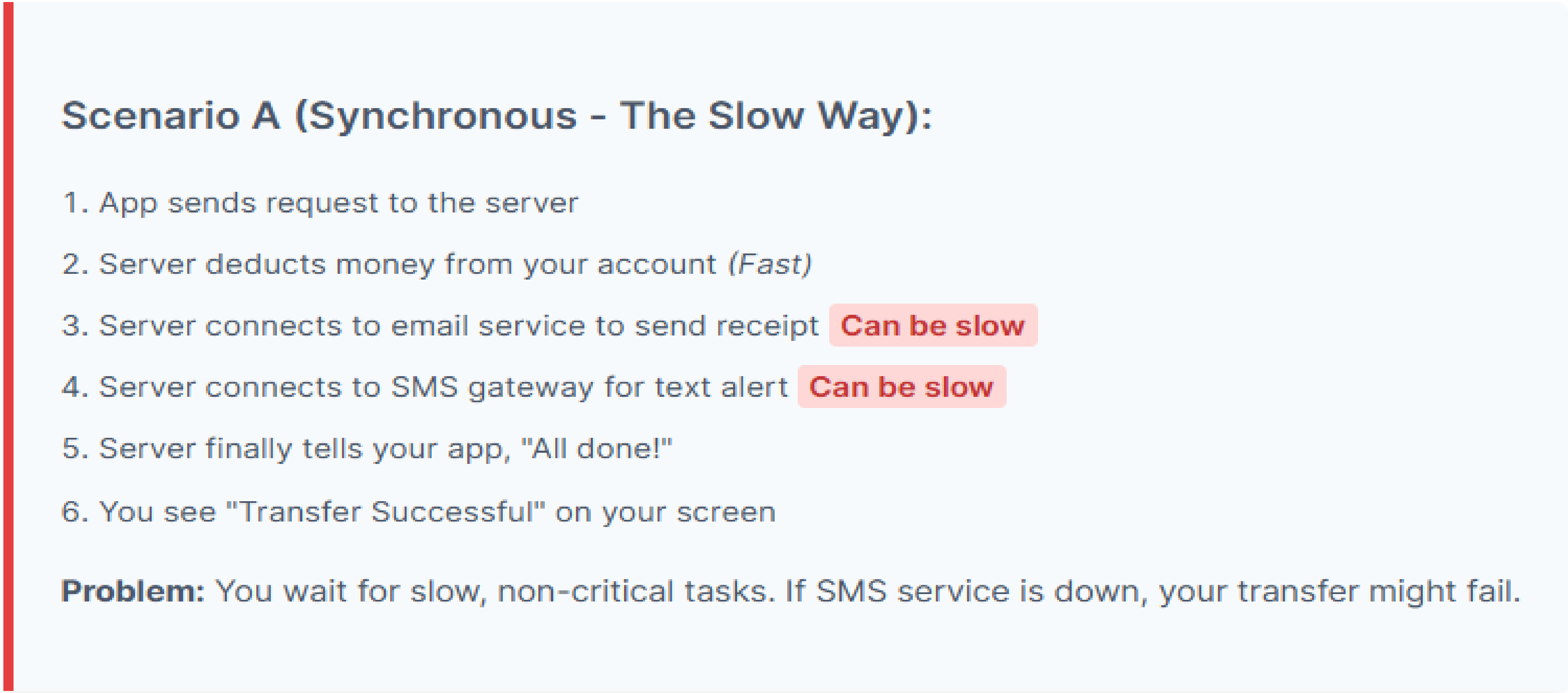
Async Processing in Action

Building a Responsive Banking System
A Hands-On Workshop for Java (Spring) and Python (FastAPI) Developers

async-processing-python.zip

The "Why?" - A Story of a Bank Transfer

Imagine you're transferring money using your banking app. You hit "Send."

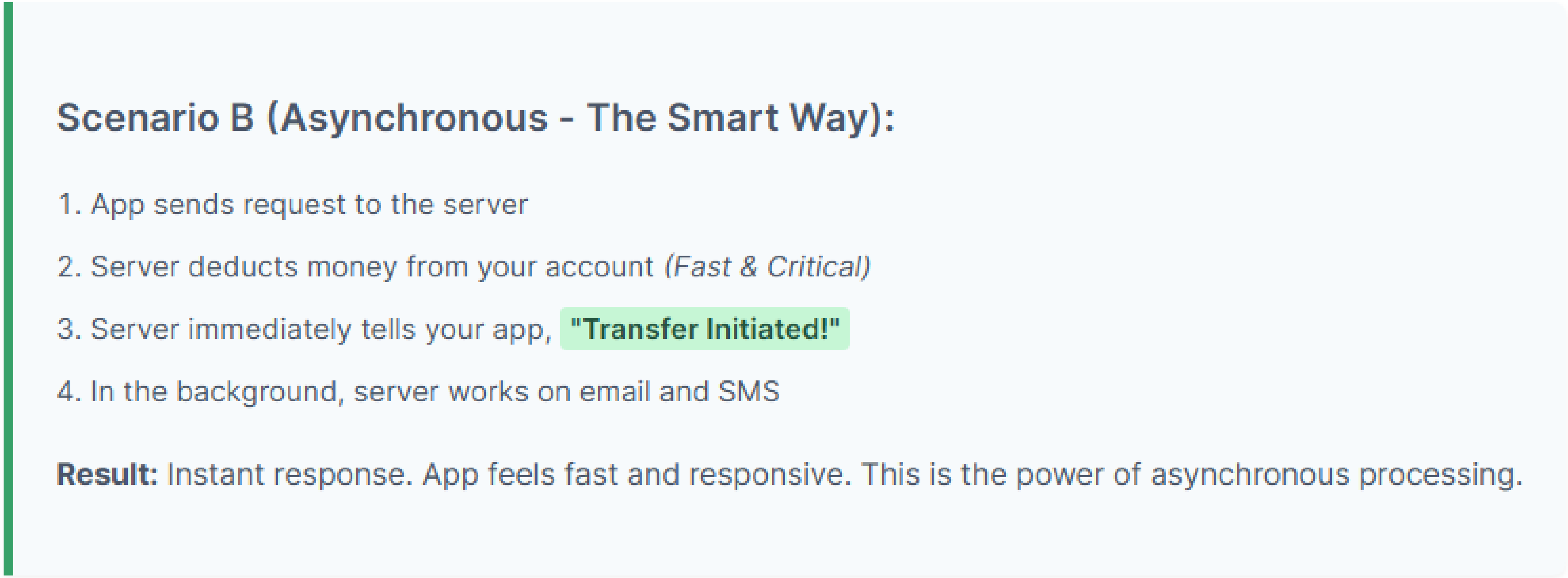


Scenario A (Synchronous - The Slow Way):

1. App sends request to the server
2. Server deducts money from your account *(Fast)*
3. Server connects to email service to send receipt **Can be slow**
4. Server connects to SMS gateway for text alert **Can be slow**
5. Server finally tells your app, "All done!"
6. You see "Transfer Successful" on your screen

Problem: You wait for slow, non-critical tasks. If SMS service is down, your transfer might fail.

The "Why?" - A Story of a Bank Transfer



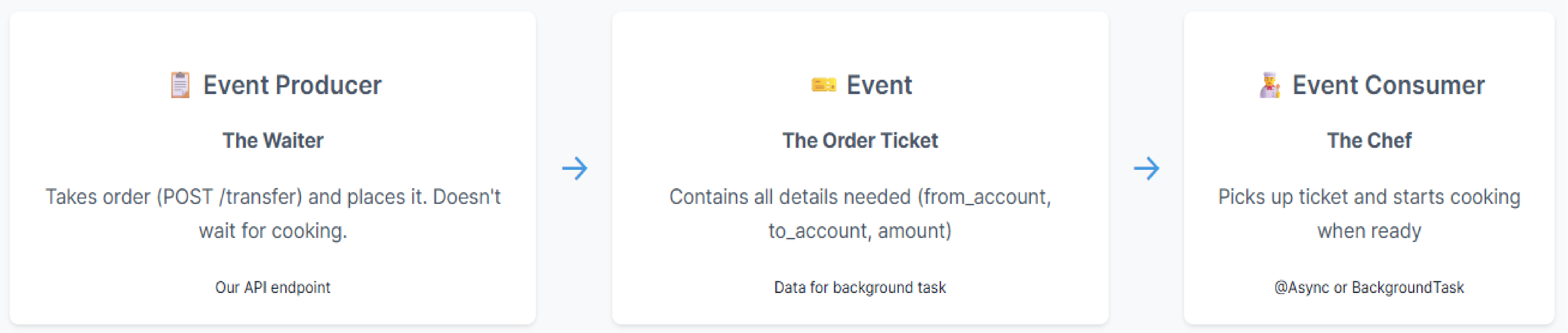
Scenario B (Asynchronous - The Smart Way):

1. App sends request to the server
2. Server deducts money from your account *(Fast & Critical)*
3. Server immediately tells your app, **"Transfer Initiated!"**
4. In the background, server works on email and SMS

Result: Instant response. App feels fast and responsive. This is the power of asynchronous processing.

Core Concept: Event-Driven Architecture

This design pattern enables “Scenario B” to be possible. It's like a kitchen in a busy restaurant.



📋 Event Producer

The Waiter

Takes order (POST /transfer) and places it. Doesn't wait for cooking.

Our API endpoint

→

🎫 Event

The Order Ticket

Contains all details needed (from_account, to_account, amount)

Data for background task

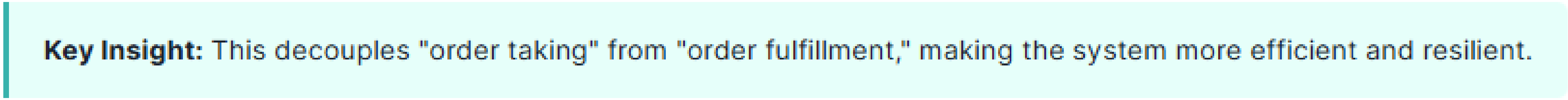
→

🧑‍🍳 Event Consumer

The Chef

Picks up ticket and starts cooking when ready

@Async or BackgroundTask



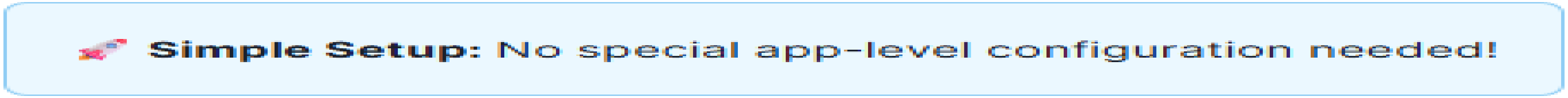
Key Insight: This decouples "order taking" from "order fulfillment," making the system more efficient and resilient.

The Python Path 🐍 - FastAPI BackgroundTasks

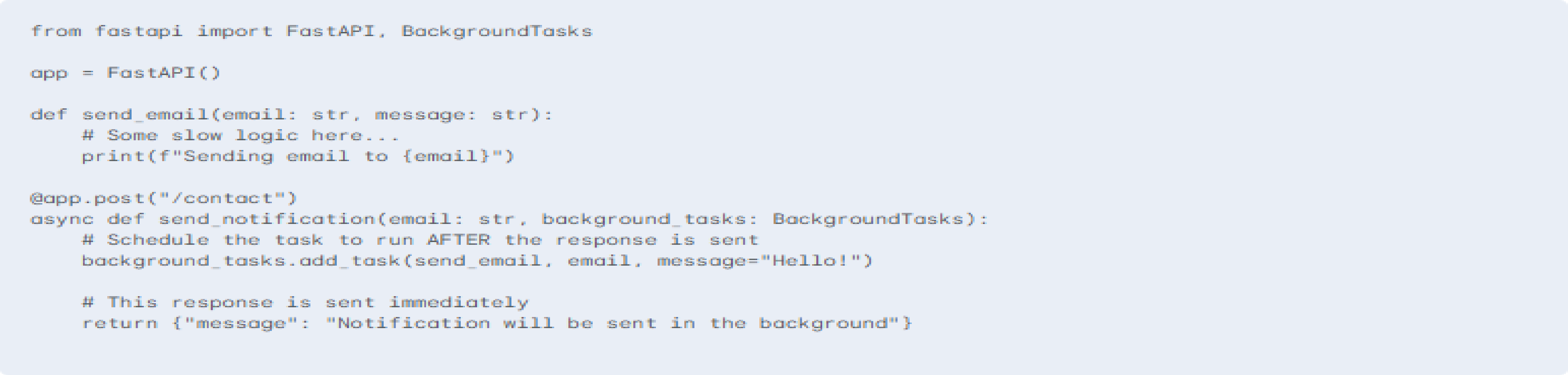
FastAPI provides a clean way to run background tasks using dependency injection.

How it Works:

- Add parameter to endpoint: background_tasks: BackgroundTasks
- FastAPI automatically provides the BackgroundTasks instance
- Use background_tasks.add_task() with function and arguments
- FastAPI sends an HTTP response first, then runs the tasks



🚀 **Simple Setup:** No special app-level configuration needed!

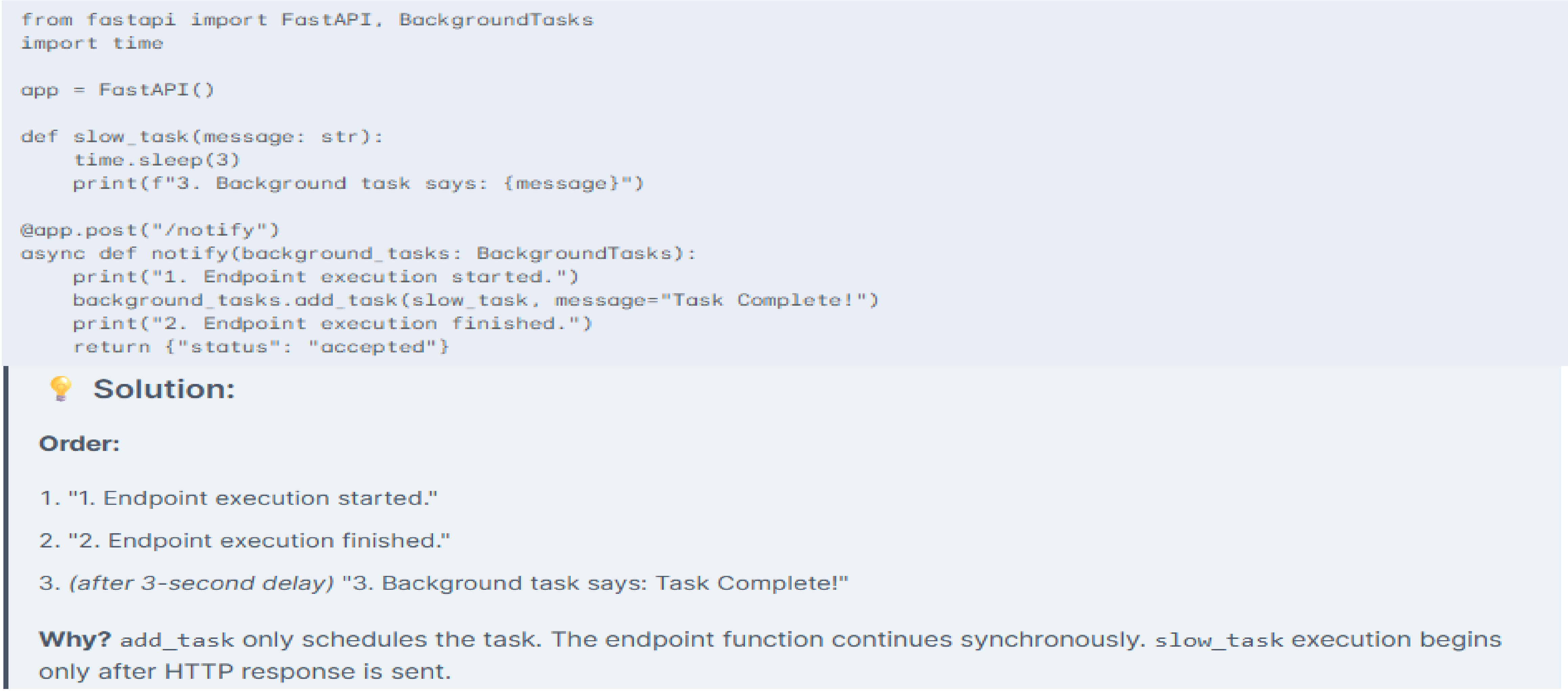


```
from fastapi import FastAPI, BackgroundTasks

app = FastAPI()

def send_email(email: str, message: str):
    # Some slow logic here...
    print(f"Sending email to {email}")

@app.post("/contact")
async def send_notification(email: str, background_tasks: BackgroundTasks):
    # Schedule the task to run AFTER the response is sent
    background_tasks.add_task(send_email, email, message="Hello!")

    # This response is sent immediately
    return {"message": "Notification will be sent in the background"}
```

Short Exercise #2 (Python)

🤔 Question:

A user calls the /notify endpoint below. In what order will the three print statements execute?



```python
from fastapi import FastAPI, BackgroundTasks
import time

app = FastAPI()

def slow_task(message: str):
    time.sleep(3)
    print(f"3. Background task says: {message}")

@app.post("/notify")
async def notify(background_tasks: BackgroundTasks):
    print("1. Endpoint execution started.")
    background_tasks.add_task(slow_task, message="Task Complete!")
    print("2. Endpoint execution finished.")
    return {"status": "accepted"}
```

💡 Solution:

Order:

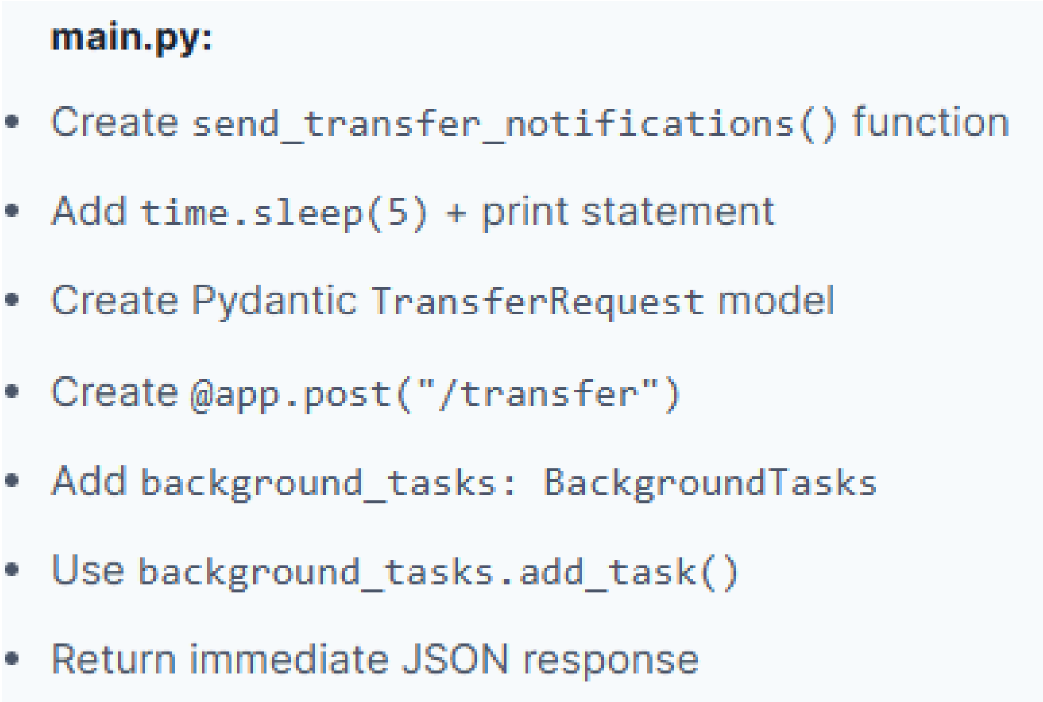
1. "1. Endpoint execution started."
2. "2. Endpoint execution finished."
3. *(after 3-second delay)* "3. Background task says: Task Complete!"

Why? `add_task` only schedules the task. The endpoint function continues synchronously. `slow_task` execution begins only after HTTP response is sent.

Hands-On Lab: Building the Banking Endpoint

Goal: Create a /transfer endpoint that simulates fund transfer and sends notifications asynchronously.

Python/FastAPI Implementation:

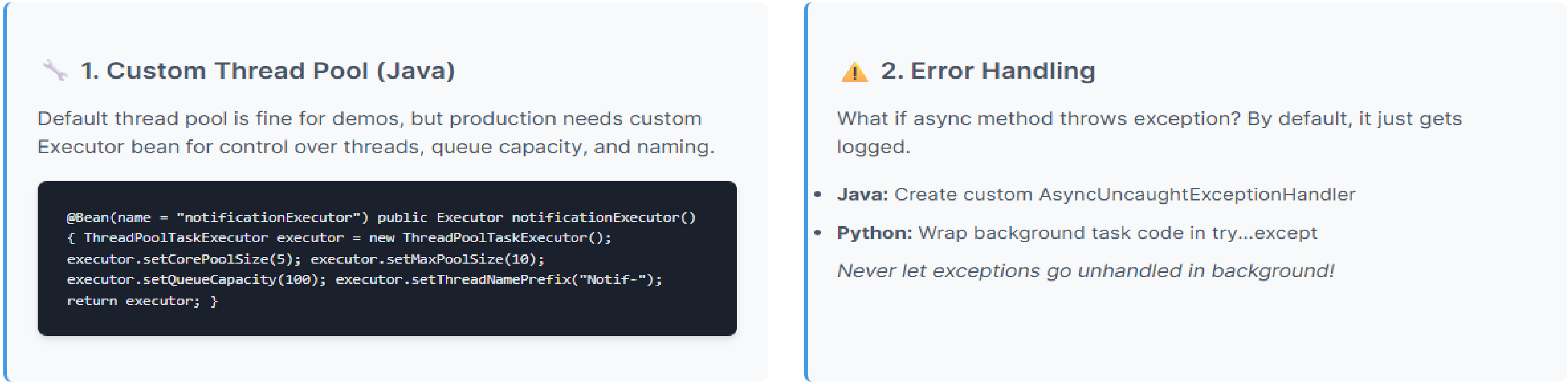


main.py:

- Create `send_transfer_notifications()` function
- Add `time.sleep(5)` + print statement
- Create Pydantic `TransferRequest` model
- Create `@app.post("/transfer")`
- Add `background_tasks: BackgroundTasks`
- Use `background_tasks.add_task()`
- Return immediate JSON response

Pro-Level Tips & Best Practices

Going from a simple demo to a production system requires more thought.



🔧 1. Custom Thread Pool (Java)

Default thread pool is fine for demos, but production needs custom Executor bean for control over threads, queue capacity, and naming.

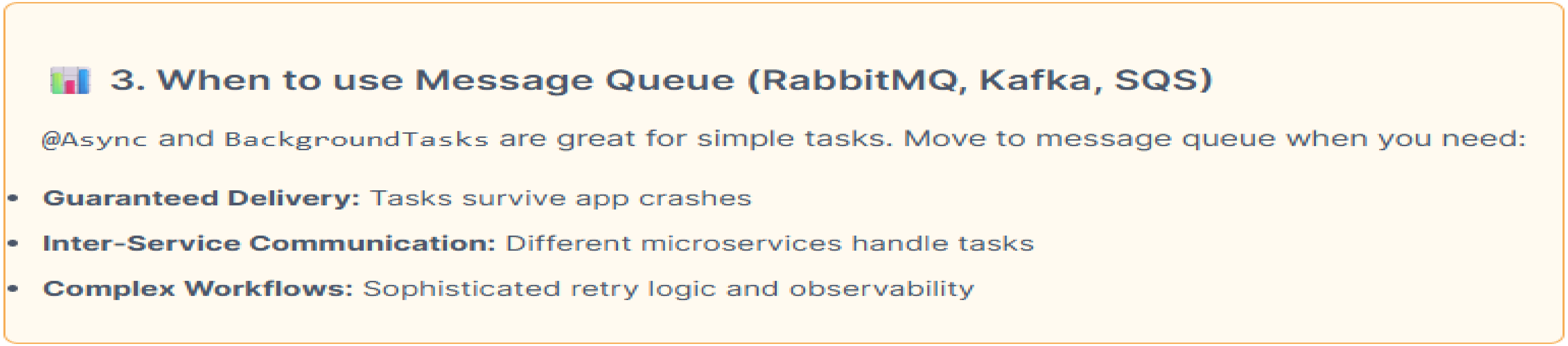
```
@Bean(name = "notificationExecutor") public Executor notificationExecutor()
{ ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
executor.setCorePoolSize(5); executor.setMaxPoolSize(10);
executor.setQueueCapacity(100); executor.setThreadNamePrefix("Notif-");
return executor; }
```

⚠️ 2. Error Handling

What if async method throws exception? By default, it just gets logged.

- **Java:** Create custom AsyncUncaughtExceptionHandler
- **Python:** Wrap background task code in try...except

Never let exceptions go unhandled in background!

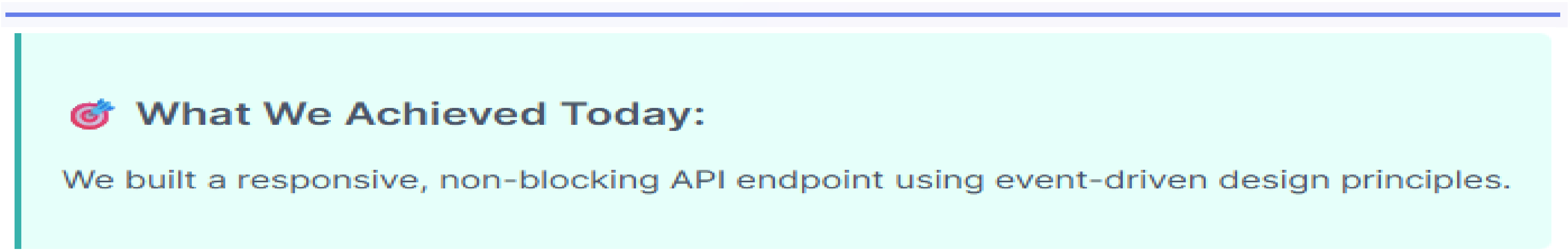


📊 3. When to use Message Queue (RabbitMQ, Kafka, SQS)

@Async and BackgroundTasks are great for simple tasks. Move to message queue when you need:

- **Guaranteed Delivery:** Tasks survive app crashes
- **Inter-Service Communication:** Different microservices handle tasks
- **Complex Workflows:** Sophisticated retry logic and observability

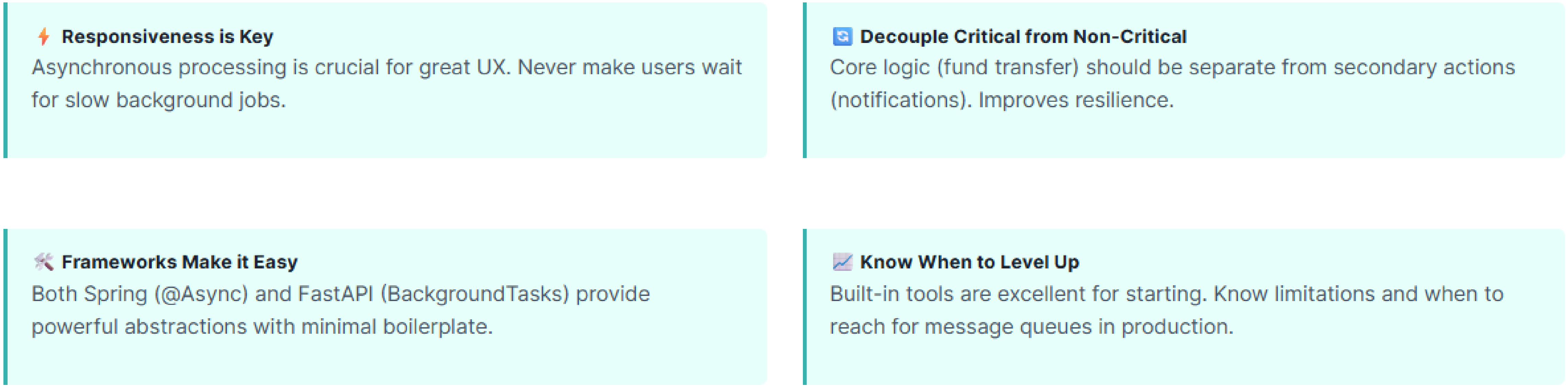
Conclusion & Key Takeaways



🎯 What We Achieved Today:

We built a responsive, non-blocking API endpoint using event-driven design principles.

Key Takeaways:



⚡ Responsiveness is Key
Asynchronous processing is crucial for great UX. Never make users wait for slow background jobs.

🔄 Decouple Critical from Non-Critical
Core logic (fund transfer) should be separate from secondary actions (notifications). Improves resilience.

🛠️ Frameworks Make it Easy
Both Spring (@Async) and FastAPI (BackgroundTasks) provide powerful abstractions with minimal boilerplate.

📈 Know When to Level Up
Built-in tools are excellent for starting. Know limitations and when to reach for message queues in production.

🚀 You now have the foundational knowledge to build faster, more scalable applications!

Async Processing in Action
A Banking Mini-Project 🚀

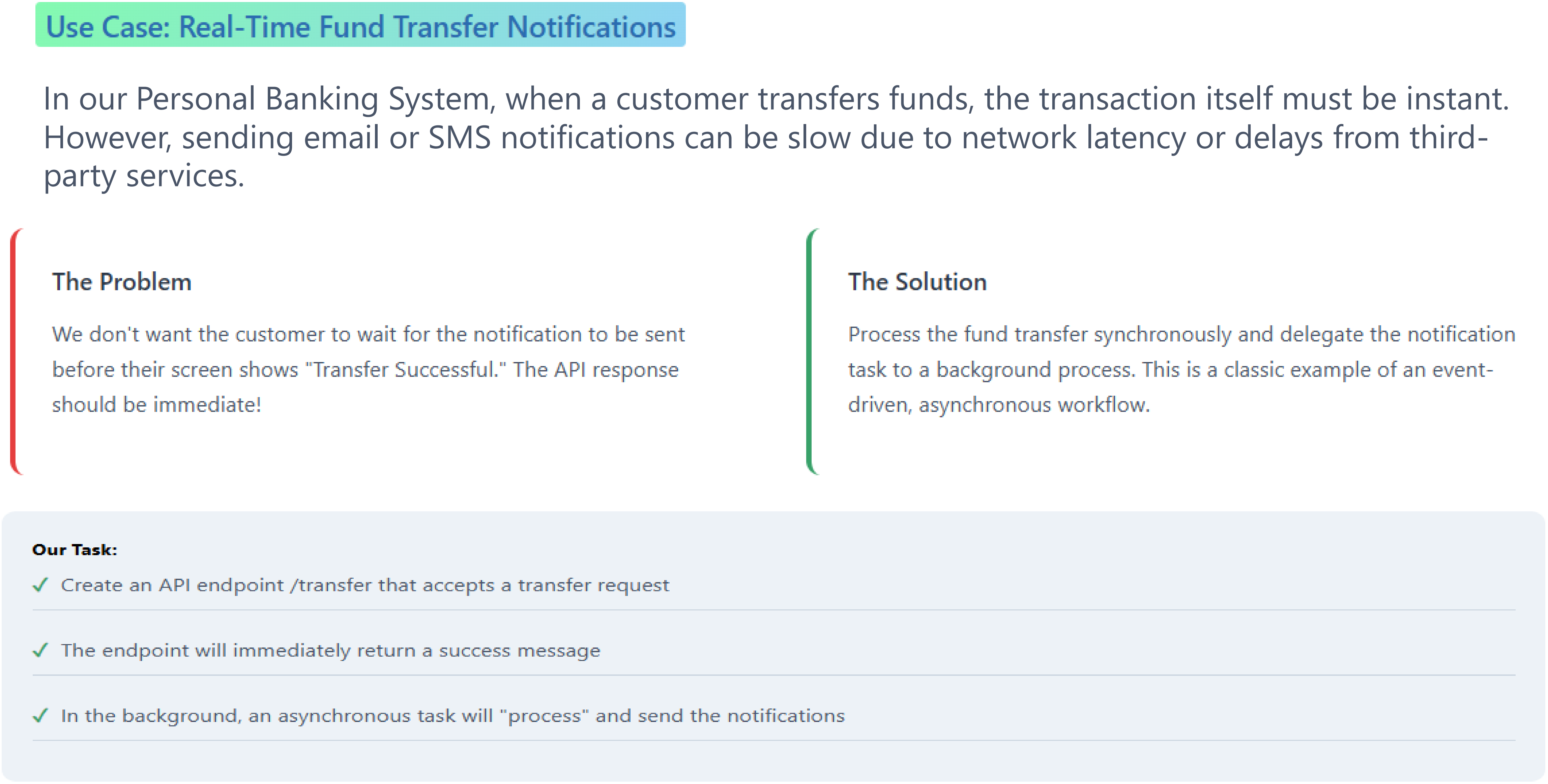
A Hands-On Lab for Java (Spring) and Python (FastAPI) Developers

Goal: Understand and implement a basic event-driven, asynchronous task in a familiar context.

async processing _mini project_java.zip

Async Processing_miniproject_python.zip

The Mini-Project Scenario



Use Case: Real-Time Fund Transfer Notifications

In our Personal Banking System, when a customer transfers funds, the transaction itself must be instant. However, sending email or SMS notifications can be slow due to network latency or delays from third-party services.

The Problem

We don't want the customer to wait for the notification to be sent before their screen shows "Transfer Successful." The API response should be immediate!

The Solution

Process the fund transfer synchronously and delegate the notification task to a background process. This is a classic example of an event-driven, asynchronous workflow.

Our Task:

- ✓ Create an API endpoint /transfer that accepts a transfer request
- ✓ The endpoint will immediately return a success message
- ✓ In the background, an asynchronous task will "process" and send the notifications

Project Architecture & Flow

Here's the simple, event-driven architecture we'll build today.

1. Client sends a POST request to our /transfer endpoint
2. API Endpoint receives the request
3. It performs the critical, synchronous action (we'll simulate this with a log message)
4. It dispatches a non-critical, asynchronous event/task to send notifications
5. It immediately returns a 202 Accepted response to the client
6. Async Worker picks up the task from the background and processes the notifications

This pattern ensures our application is responsive and resilient.

pip install "fastapi[all]"

Implementation - The Python/FastAPI Path 🐍

Objective: Use Spring's @Async to handle background notification processing.

Step 1: Project Setup

Installation

```
pip install "fastapi[all]"
```

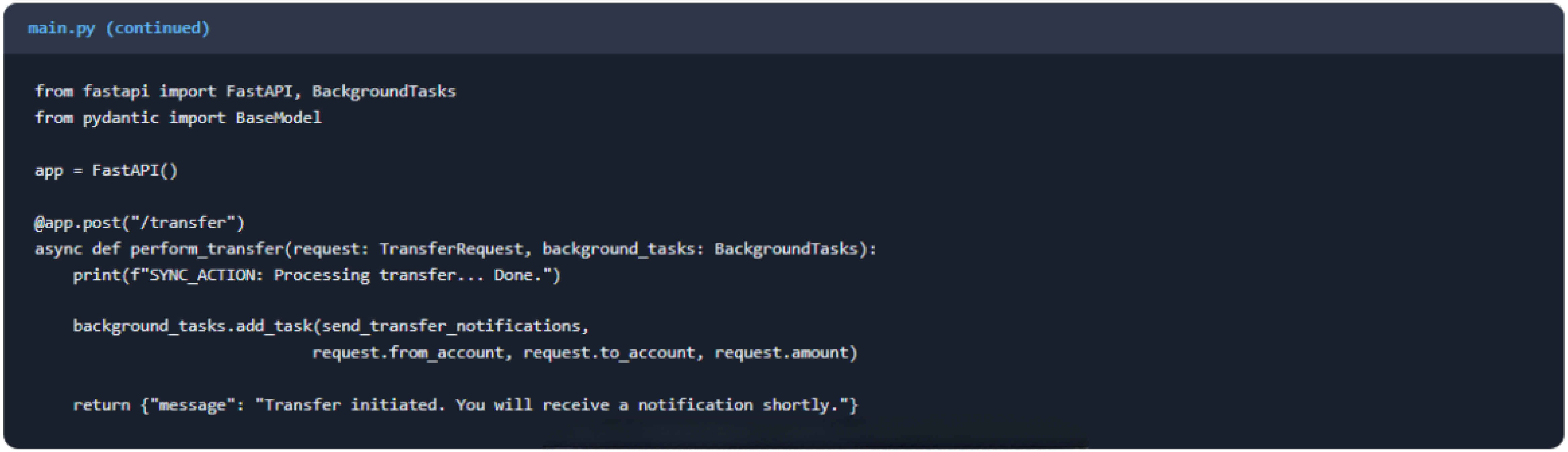
Step 2: Define the Background Task

main.py

```
import time

def send_transfer_notifications(from_account: str, to_account: str, amount: float):
    print("ASYNC_TASK: Preparing to send notifications...")
    time.sleep(5)  # Simulate network delay
    print(f"✅ ASYNC_TASK: Notifications sent for transfer of ${amount:.2f}")
```

Implementation - The Java/Spring Boot Path ☕

Step 3: Create the FastAPI Endpoint



main.py (continued)

```python
from fastapi import FastAPI, BackgroundTasks
from pydantic import BaseModel

app = FastAPI()

@app.post("/transfer")
async def perform_transfer(request: TransferRequest, background_tasks: BackgroundTasks):
    print(f"SYNC_ACTION: Processing transfer... Done.")

    background_tasks.add_task(send_transfer_notifications,
                              request.from_account, request.to_account, request.amount)

    return {"message": "Transfer initiated. You will receive a notification shortly."}
```

Running & Testing Your Application

Now, let's see it in action!

Java/Spring

```
./mvnw spring-boot:run
```

Default port: 8080

Python/FastAPI

```
uvicorn main:app --reload
```

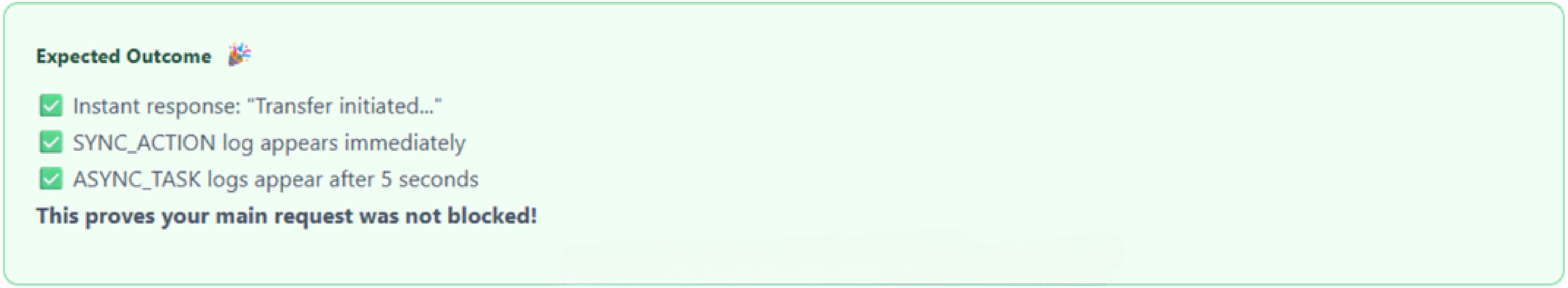
Default port: 8000

Test with cURL

Test Command

```
curl -X POST http://localhost:8080/transfer \
-H "Content-Type: application/json" \
-d '{
    "fromAccount": "user-A",
    "toAccount": "user-B",
    "amount": 150.75
}'
```

Running & Testing Your Application



Expected Outcome 🎉

✅ Instant response: "Transfer initiated..."

✅ SYNC_ACTION log appears immediately

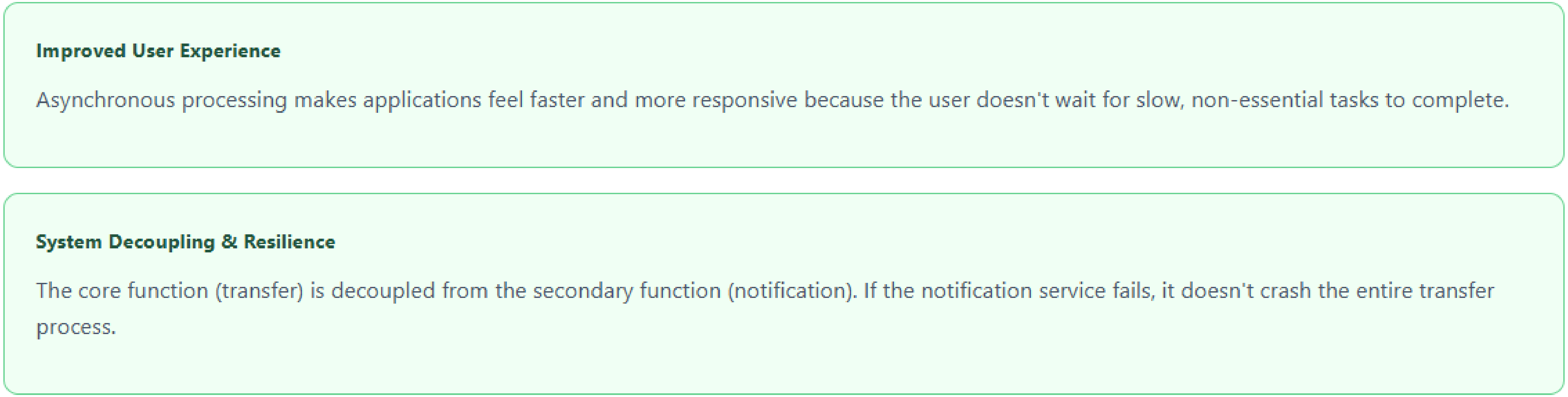
✅ ASYNC_TASK logs appear after 5 seconds

This proves your main request was not blocked!

Conclusion & Key Learnings

Congratulations! You've successfully built a non-blocking, event-driven API endpoint.

What did we learn today?



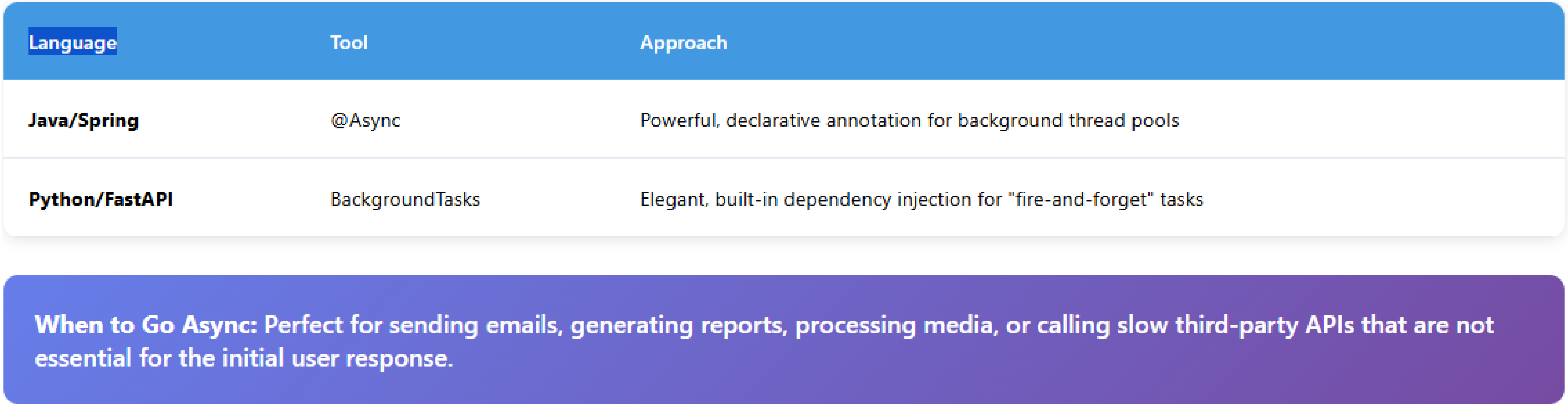
Improved User Experience

Asynchronous processing makes applications feel faster and more responsive because the user doesn't wait for slow, non-essential tasks to complete.

System Decoupling & Resilience

The core function (transfer) is decoupled from the secondary function (notification). If the notification service fails, it doesn't crash the entire transfer process.

Conclusion & Key Learnings



Language	Tool	Approach
Java/Spring	@Async	Powerful, declarative annotation for background thread pools
Python/FastAPI	BackgroundTasks	Elegant, built-in dependency injection for "fire-and-forget" tasks

When to Go Async: Perfect for sending emails, generating reports, processing media, or calling slow third-party APIs that are not essential for the initial user response.

This simple exercise is the foundation for building complex, scalable, and robust modern applications. 🚀