API Design Best Practices

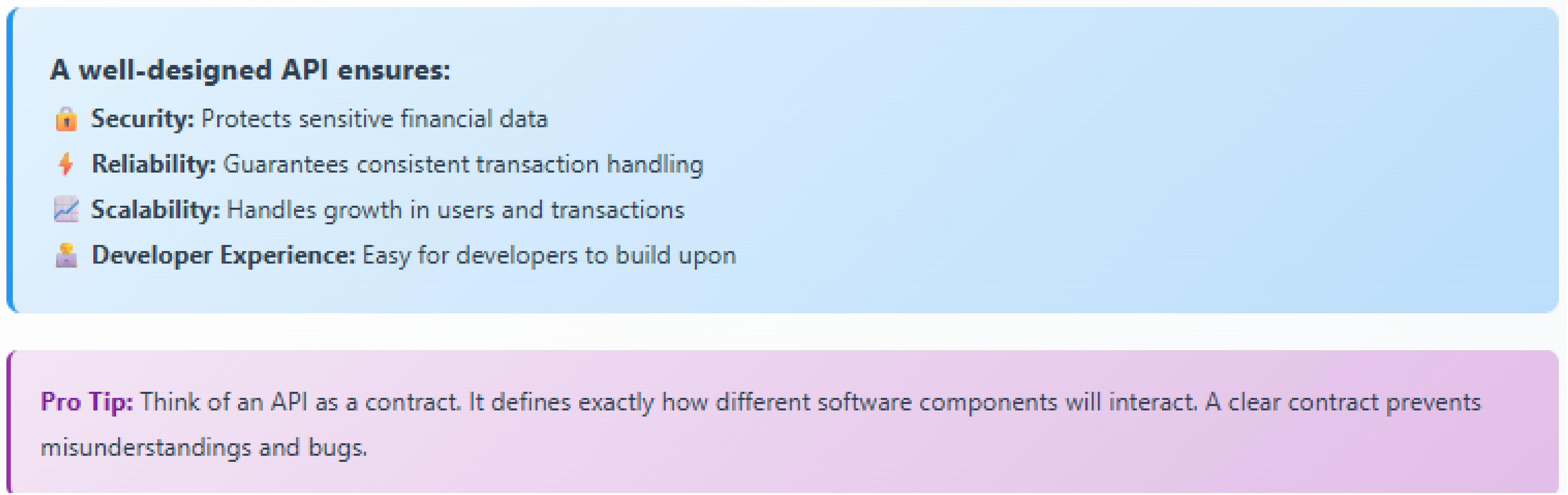
Agenda

1. Introduction: Why API Design Matters in Banking
2. Core Principles: REST Conventions & Versioning
3. Handling Data: Pagination, Filtering & Sorting
4. Communication: HTTP Status Codes & Error Handling
5. Conclusion: Key Takeaways & Best Practices

api_with_filter_query.zip

Why API Design Matters

In a personal banking system, APIs are the backbone of every digital interaction. They connect mobile apps, web portals, and third-party services to sensitive customer data.



A well-designed API ensures:

🔒 **Security:** Protects sensitive financial data

⚡ **Reliability:** Guarantees consistent transaction handling

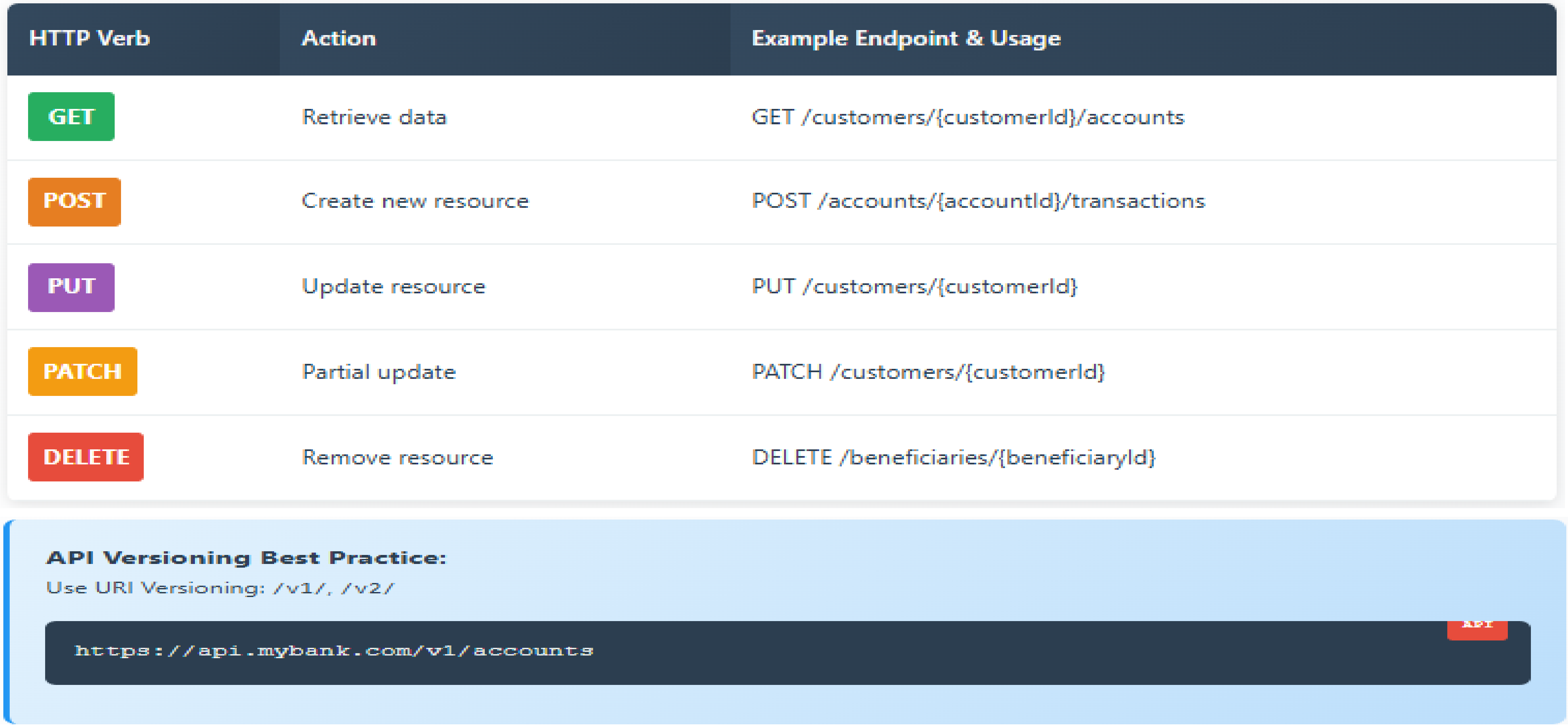
📈 **Scalability:** Handles growth in users and transactions

👨‍💻 **Developer Experience:** Easy for developers to build upon

Pro Tip: Think of an API as a contract. It defines exactly how different software components will interact. A clear contract prevents misunderstandings and bugs.

REST Conventions & Versioning

REST uses standard HTTP methods to work with resources.



HTTP Verb	Action	Example Endpoint & Usage
GET	Retrieve data	GET /customers/{customerId}/accounts
POST	Create new resource	POST /accounts/{accountId}/transactions
PUT	Update resource	PUT /customers/{customerId}
PATCH	Partial update	PATCH /customers/{customerId}
DELETE	Remove resource	DELETE /beneficiaries/{beneficiaryId}

API Versioning Best Practice:
Use URI Versioning: /v1/, /v2/

```
https://api.mybank.com/v1/accounts
```

The Language of the API: HTTP Methods

HTTP Methods are commands that tell the server what action you want to perform on a resource.

A "resource" is data like a customer, account, or transaction.

The Five Most Common Commands:

- **GET** - Read data
- **POST** - Create new data
- **PUT** - Replace entire resource
- **PATCH** - Modify part of the resource
- **DELETE** - Remove data

GET - "Can I see the menu?" 📋

Action: Retrieve/Read Data

Analogy: Asking the waiter for the menu. You're not changing anything, just requesting information.

✅ Safe Operation

Making a GET request won't change or delete any data. You can call it a million times without side effects.

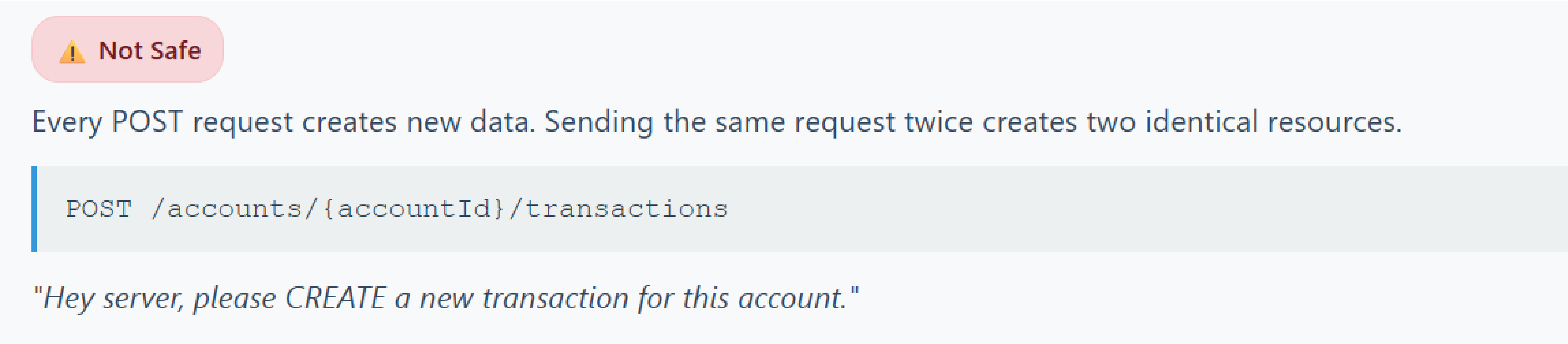
```
GET /customers/{customerId}/accounts
```

"Hey server, please GET me all accounts for this customer."

POST - "I'd like to place a new order" ➕

Action: Create New Resource

Analogy: Giving the waiter a brand-new order to take to the kitchen.



⚠️ Not Safe

Every POST request creates new data. Sending the same request twice creates two identical resources.

```
POST /accounts/{accountId}/transactions
```

"Hey server, please CREATE a new transaction for this account."

PUT vs. PATCH - "I need to change my order" ✏️

PUT (Replace)

🔄

Action: Update entire resource

Analogy: "Instead of my burger with lettuce and tomato, give me a burger with just cheese and pickles."

Complete replacement of the old resource.

PATCH (Modify)

✂️

Action: Partially update resource

Analogy: "Please remove the onions from my burger."

Change only specific fields.

Key Takeaway: Use PUT to replace the whole thing. Use PATCH to change just a few fields.

DELETE - "Please cancel my order" 🗑️

Action: Remove Resource

Analogy: Telling the waiter to cancel an order. The order is removed from the kitchen's queue.

🚨 Destructive Operation

This is a destructive action. Once data is deleted, it's usually gone for good.

```
DELETE /beneficiaries/{beneficiaryId}
```

"Hey server, please find this beneficiary and DELETE them from the system."

The App is Evolving... What About the API? 🤔

So, we have our application and API working perfectly...

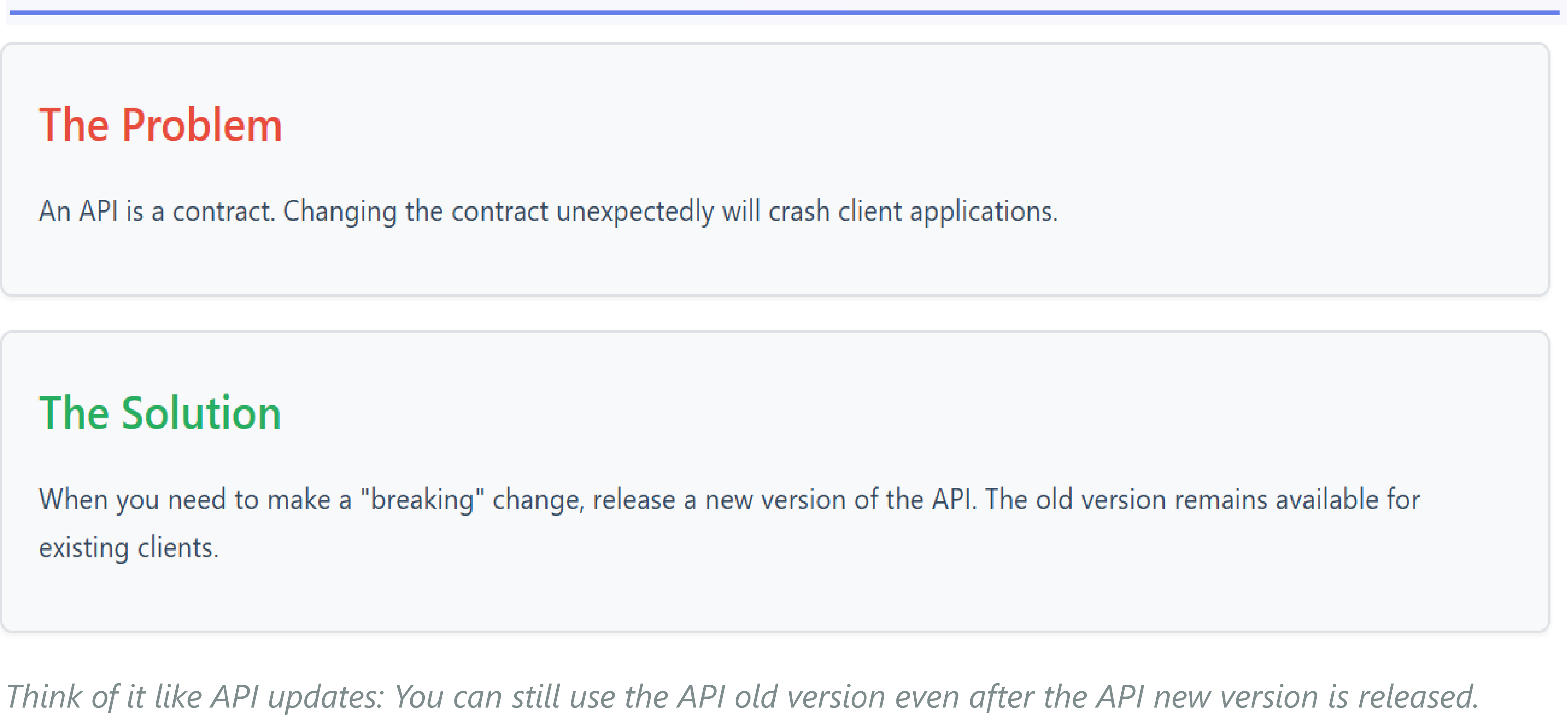
But What Happens Later?

In six months, we need to make a big change to the accounts structure.

If we just change the API, every client application using the old version will break! 😱

This is where API versioning comes to the rescue!

API Versioning - A Promise Not to Break Things 🤝



The Problem

An API is a contract. Changing the contract unexpectedly will crash client applications.

The Solution

When you need to make a "breaking" change, release a new version of the API. The old version remains available for existing clients.

Think of it like API updates: You can still use the API old version even after the API new version is released.

How to Version: The URL Path Method 🛣️

The most common and clearest way is to include the version number in the URL path.

Current Version:

```
https://api.mybank.com/v1/accounts
```

The **/v1/** clearly tells everyone we're using Version 1 of the API.

Future Version:

```
https://api.mybank.com/v2/accounts
```

This allows the old /v1/ endpoint to continue working, ensuring existing apps don't crash.

Project Requirement

Banking API Versioning Implementation Challenge

The Challenge

"Introduce versioning in the **banking_api** project's existing APIs"

Critical Constraints

Minimal Code Changes

Make the smallest possible modifications to existing codebase

Additive Approach Only

Add new code without touching existing functionality

Zero Breaking Changes

Preserve all current API endpoints and behavior

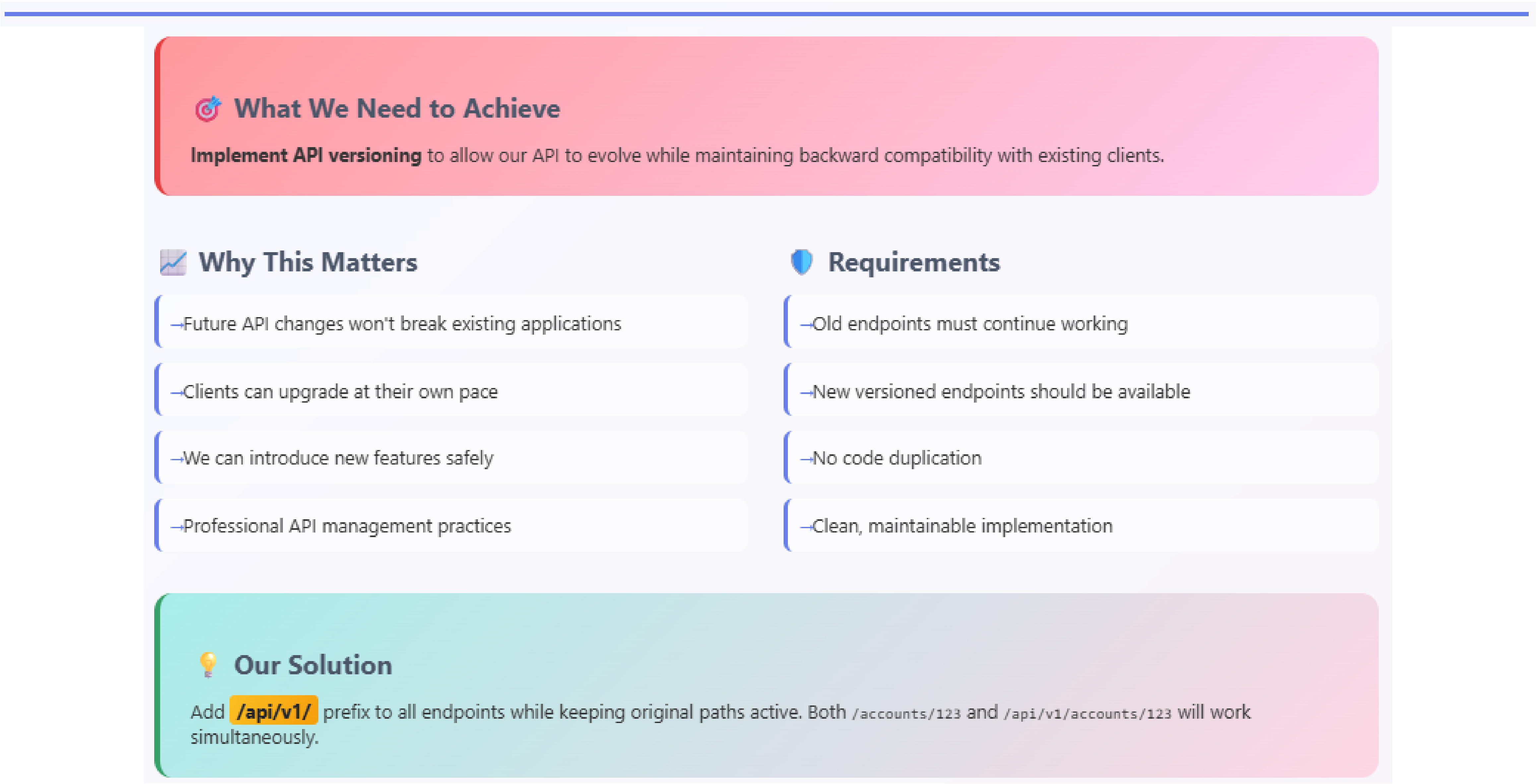
Our Solution Approach

1. **Path-Based Versioning**
   Implement `/api/v1/` prefix routing

2. **Router Aggregation**
   Create new module to aggregate existing routers

3. **Dual Availability**
   Same endpoints on both old and new paths

📋 The Business Requirement



🎯 What We Need to Achieve

Implement API versioning to allow our API to evolve while maintaining backward compatibility with existing clients.

📈 Why This Matters

- →Future API changes won't break existing applications
- →Clients can upgrade at their own pace
- →We can introduce new features safely
- →Professional API management practices

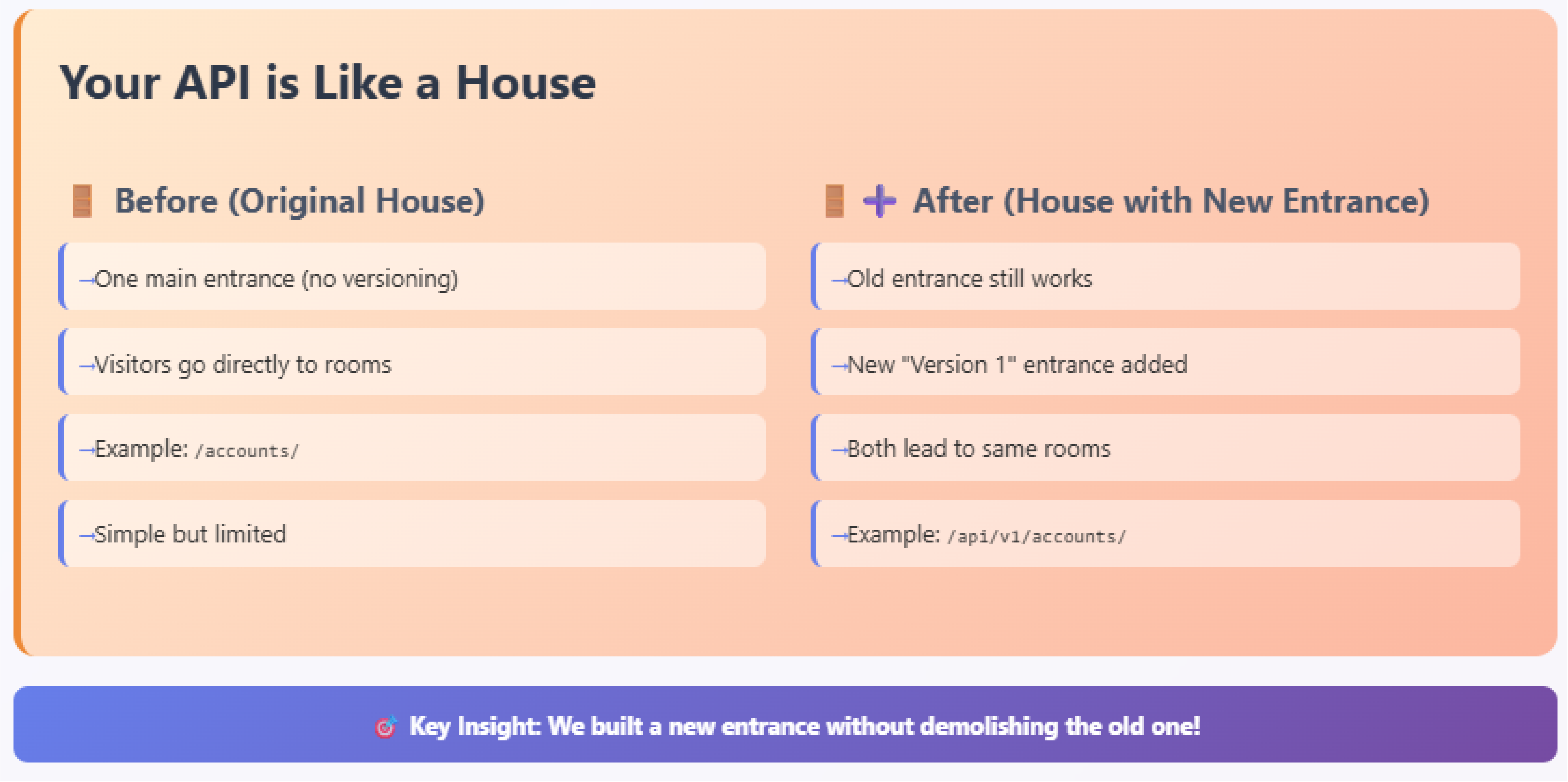
🛡️ Requirements

- →Old endpoints must continue working
- →New versioned endpoints should be available
- →No code duplication
- →Clean, maintainable implementation

💡 Our Solution

Add **/api/v1/** prefix to all endpoints while keeping original paths active. Both `/accounts/123` and `/api/v1/accounts/123` will work simultaneously.

🏠Understanding with an Analogy



Your API is Like a House

🚪 Before (Original House)

- →One main entrance (no versioning)
- →Visitors go directly to rooms
- →Example: `/accounts/`
- →Simple but limited

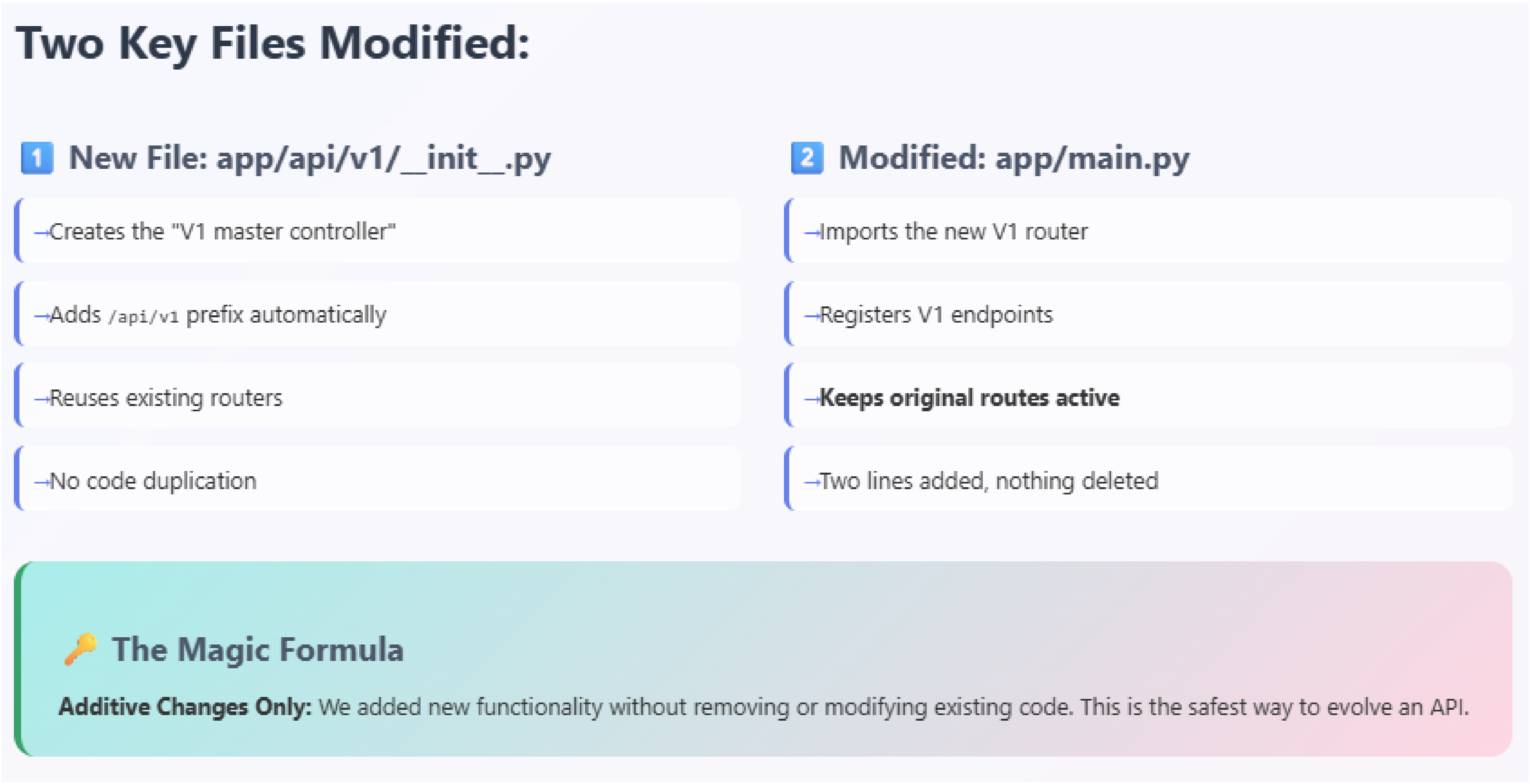
🚪➕ After (House with New Entrance)

- →Old entrance still works
- →New "Version 1" entrance added
- →Both lead to same rooms
- →Example: `/api/v1/accounts/`

🎯 Key Insight: We built a new entrance without demolishing the old one!

SOLUTION

⚙️ What Was Changed and Why



Two Key Files Modified:

1 New File: app/api/v1/__init__.py

- →Creates the "V1 master controller"
- →Adds `/api/v1` prefix automatically
- →Reuses existing routers
- →No code duplication

2 Modified: app/main.py

- →Imports the new V1 router
- →Registers V1 endpoints
- →**Keeps original routes active**
- →Two lines added, nothing deleted

🔑 The Magic Formula

Additive Changes Only: We added new functionality without removing or modifying existing code. This is the safest way to evolve an API.

🔧 The New V1 Router Blueprint

📁 File: app/api/v1/__init__.py

```python
# Import the tool to create a group of routes from fastapi import APIRouter # Import existing account router
(reusing existing logic!) from ...routers.account_router import router as account_router # Create new router
with V1 prefix - THIS IS THE MAGIC! router = APIRouter(prefix="/api/v1") # Include existing router under the
new prefix router.include_router(account_router)
```

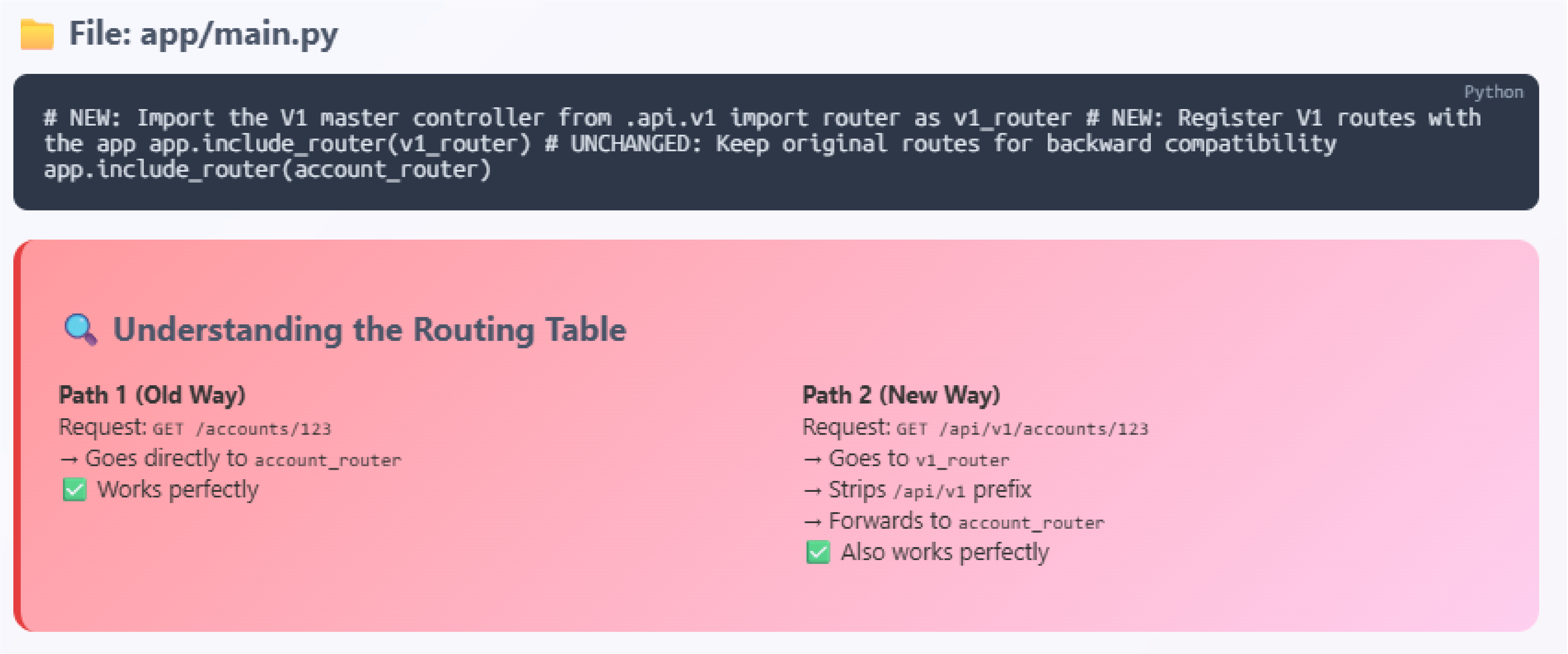
🎯 What This Does

- →Creates a new router group
- →Puts **/api/v1** sign on it
- →Copies all existing account routes
- →Automatically prefixes them

✨ The Result

- →`/accounts/123` becomes
- →`/api/v1/accounts/123`
- →Zero code duplication
- →Automatic transformation

🚀 Connecting to the Main Application



📁 File: app/main.py

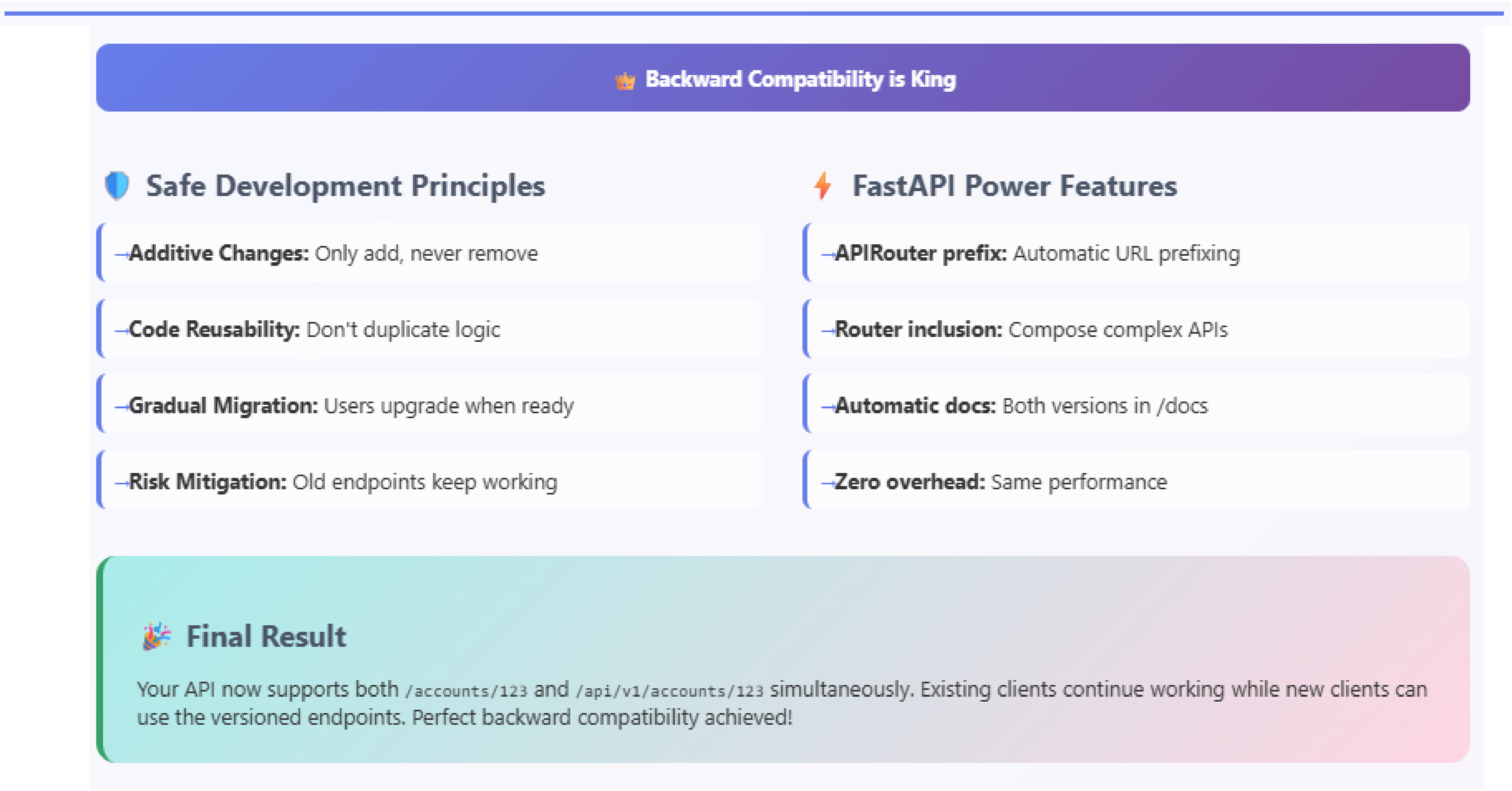
```python
# NEW: Import the V1 master controller from .api.v1 import router as v1_router # NEW: Register V1 routes with
the app app.include_router(v1_router) # UNCHANGED: Keep original routes for backward compatibility
app.include_router(account_router)
```

🔍 Understanding the Routing Table

Path 1 (Old Way)
Request: `GET /accounts/123`
→ Goes directly to `account_router`
✅ Works perfectly

Path 2 (New Way)
Request: `GET /api/v1/accounts/123`
→ Goes to `v1_router`
→ Strips `/api/v1` prefix
→ Forwards to `account_router`
✅ Also works perfectly

🎓 Key Takeaways for Success



👑 **Backward Compatibility is King**

🛡️ Safe Development Principles

- →**Additive Changes:** Only add, never remove
- →**Code Reusability:** Don't duplicate logic
- →**Gradual Migration:** Users upgrade when ready
- →**Risk Mitigation:** Old endpoints keep working

⚡ FastAPI Power Features

- →**APIRouter prefix:** Automatic URL prefixing
- →**Router inclusion:** Compose complex APIs
- →**Automatic docs:** Both versions in /docs
- →**Zero overhead:** Same performance

🎉 Final Result

Your API now supports both `/accounts/123` and `/api/v1/accounts/123` simultaneously. Existing clients continue working while new clients can use the versioned endpoints. Perfect backward compatibility achieved!

FastAPI Path-Based API Versioning

Minimal, Additive Implementation with Full Backwards Compatibility

What Was Changed

- ✓ **Added** `app/api/v1/__init__.py`
- ✓ Aggregates existing routers under `/api/v1` prefix
- ✓ **Minimal edit** to `app/main.py`
- ✓ Mounted new v1 router
- ✓ Kept all existing endpoints intact

Key Benefits

- ✓ 100% Backward Compatible
- ✓ Zero behavior changes to existing code
- ✓ No modifications to routers/services/schemas
- ✓ Additive-only approach
- ✓ Same endpoints available on both paths

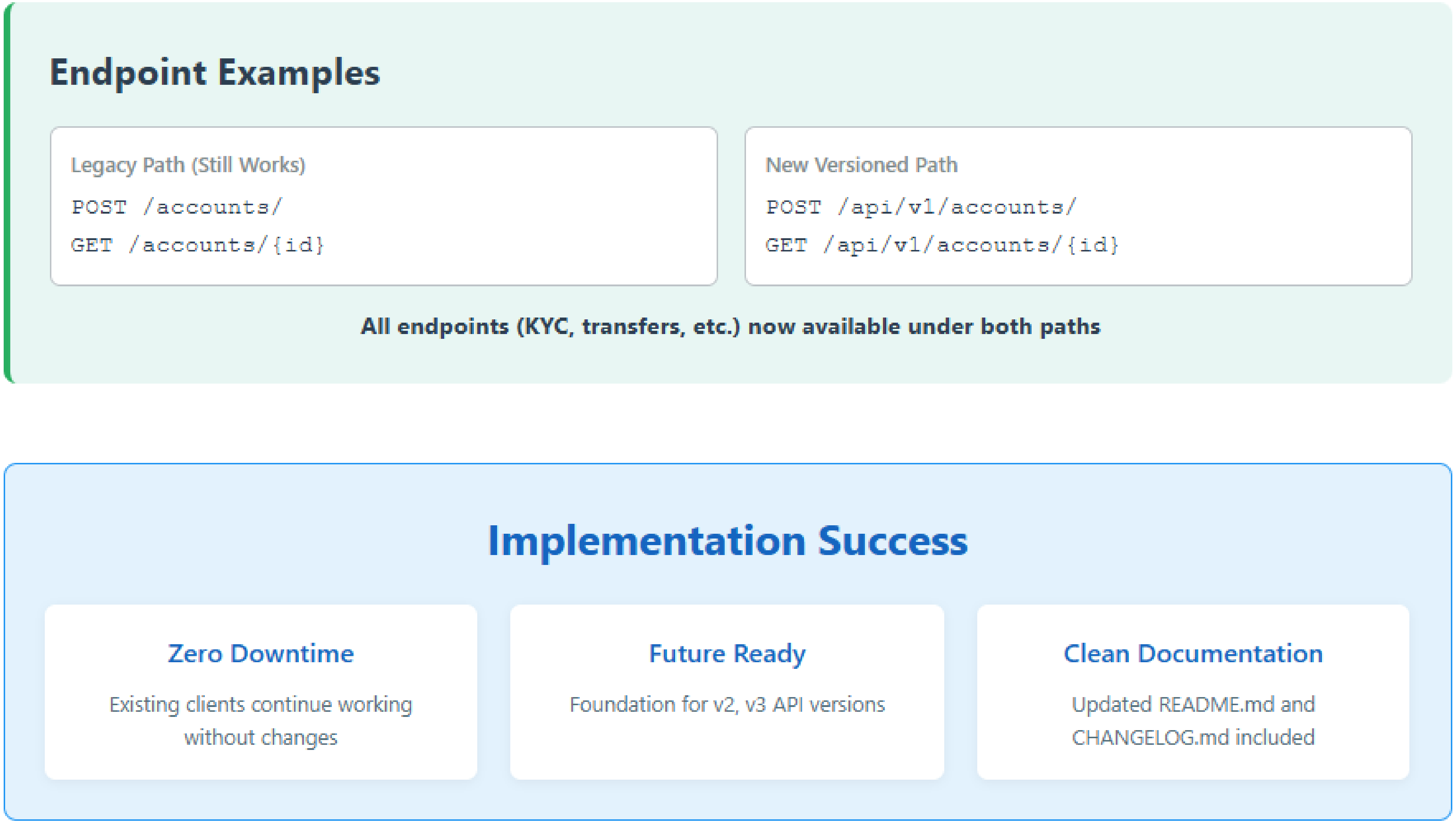
Implementation Overview

```
# main.py - Minimal addition from .api.v1 import router as v1_router app.include_router(v1_router)
```

```
# app/api/v1/__init__.py - New aggregation file from fastapi import APIRouter from
...routers.account_router import router as account_router router = APIRouter(prefix="/api/v1")
router.include_router(account_router)
```

FastAPI Path-Based API Versioning

Minimal, Additive Implementation with Full Backwards Compatibility



Endpoint Examples

Legacy Path (Still Works)

```
POST /accounts/
GET /accounts/{id}
```

New Versioned Path

```
POST /api/v1/accounts/
GET /api/v1/accounts/{id}
```

All endpoints (KYC, transfers, etc.) now available under both paths

Implementation Success

Zero Downtime

Existing clients continue working without changes

Future Ready

Foundation for v2, v3 API versions

Clean Documentation

Updated README.md and CHANGELOG.md included

Summary & Key Takeaways 📚

API: Like a waiter handling requests between client and server

HTTP Methods:

- **GET:** Read data
- **POST:** Create new data
- **PUT:** Replace data
- **PATCH:** Modify part of data
- **DELETE:** Remove data

API Versioning: Essential for evolving APIs without breaking existing applications. It's a promise of stability.

REST Conventions & Versioning(List)

REST uses standard HTTP methods to work with resources.

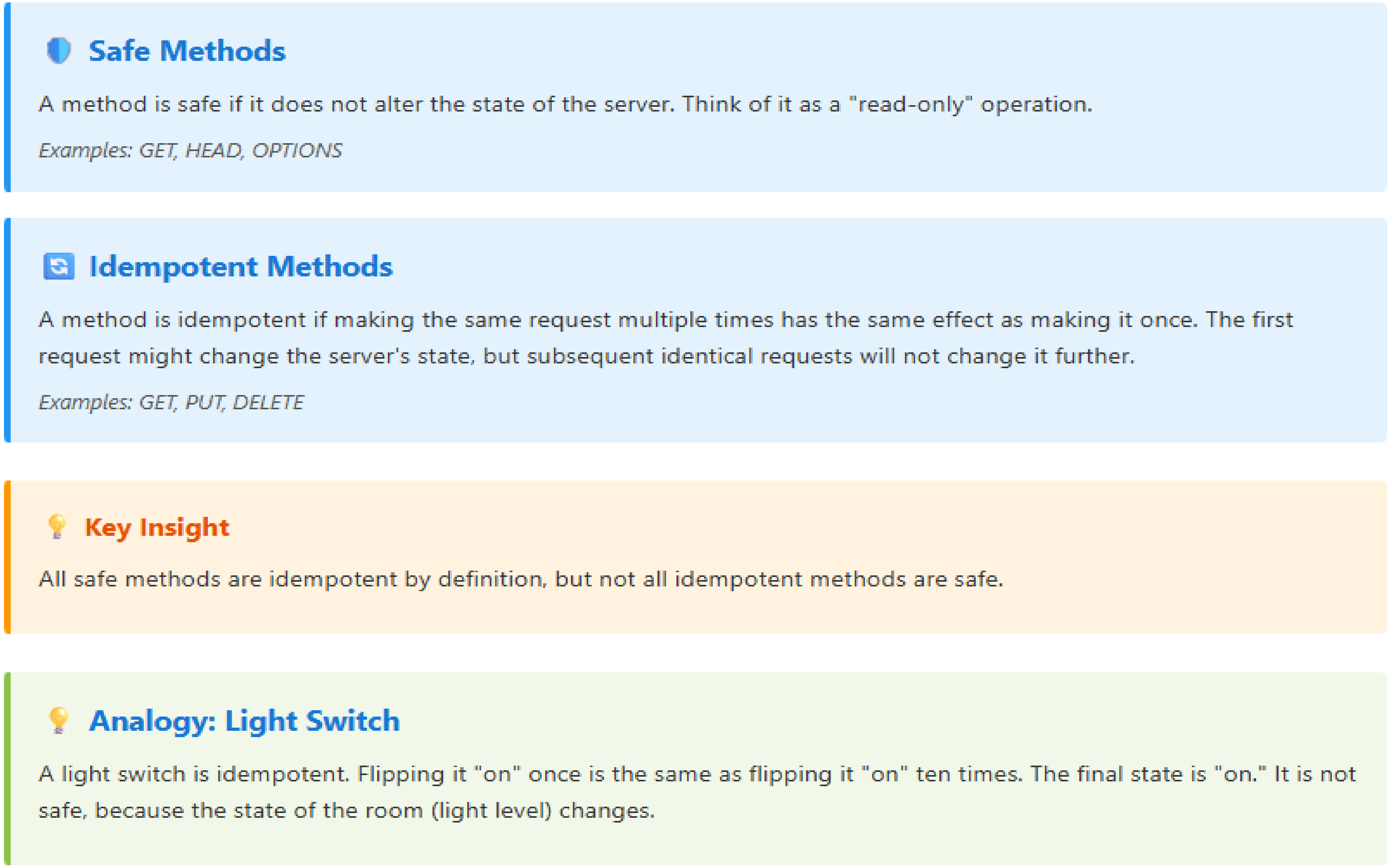
HTTP Verb	Action / Purpose	Safe	Idempotent
GET	Retrieve data, Read-only	✅	✅
POST	Create a new resource	❌	❌
PUT	Update/replace an existing resource	❌	✅
PATCH	Partially update a resource (PUT with partial fields will result in error or reset to default)	❌	❌
DELETE	Remove a resource	❌	✅
HEAD	Retrieve headers only (no body). Useful for testing/metadata.	✅	✅
OPTIONS	Discover supported methods for a resource	✅	✅
TRACE	Debug request path (echoes request). Rarely used in production.	✅	✅
CONNECT	Establish a tunnel (used with HTTPS, proxies).	❌	❌

Safe - request does not modify the server’s state

Idempotent - multiple identical requests have the same effect as a single request

Safe vs. Idempotent Methods

Understanding HTTP Method Properties



Safe Methods

A method is safe if it does not alter the state of the server. Think of it as a "read-only" operation.

Examples: GET, HEAD, OPTIONS

Idempotent Methods

A method is idempotent if making the same request multiple times has the same effect as making it once. The first request might change the server's state, but subsequent identical requests will not change it further.

Examples: GET, PUT, DELETE

Key Insight

All safe methods are idempotent by definition, but not all idempotent methods are safe.

Analogy: Light Switch

A light switch is idempotent. Flipping it "on" once is the same as flipping it "on" ten times. The final state is "on." It is not safe, because the state of the room (light level) changes.

Problem 1: The "Like Button"

Testing a Social Media API

❓ Scenario

When a user clicks a "like" button on a post, the frontend sends the following request:

```
POST /api/v1/posts/123/like
```

This request has an empty body. Each time the server receives this request, it increments a like_count column in the database for post 123.

🤔 Question

Is this POST /.../like operation Safe?
Is it Idempotent?

Discuss your reasoning.

Solution 1: The "Like Button"

Analysis and Answer

✅ Answer

Not Safe and Not Idempotent

❌ Why is it not Safe?

- The definition of a safe method is that it doesn't alter the server's state
- This request directly causes a write operation to the database (like_count = like_count + 1)
- Because the server's state is modified, the operation is not safe

❌ Why is it not Idempotent?

- Idempotency means repeating the exact same request multiple times yields the same result
- Request 1: like_count becomes 11
- Request 2: like_count becomes 12
- Request 3: like_count becomes 13
- Since each identical request produces a different final state, it's not idempotent

💡 **Pro-Tip**

A common way to make this interaction idempotent is to use a different endpoint design, like PUT `/api/v1/posts/123/likes/{userId}`, where adding or removing the like is a state change for that specific user's "like status," which is idempotent.

Problem 2: The "Recalculate Report" Endpoint

Analytics Platform API

Scenario

An analytics platform has an API endpoint that triggers a complex calculation for a monthly report. The report is generated based on raw data that was imported earlier in the month.

```
POST /api/v1/reports/456/recalculate
```

When this endpoint is called, the server:

- Deletes the old report file for report 456
- Re-runs the aggregation logic on the unchanged raw data
- Saves a new report file

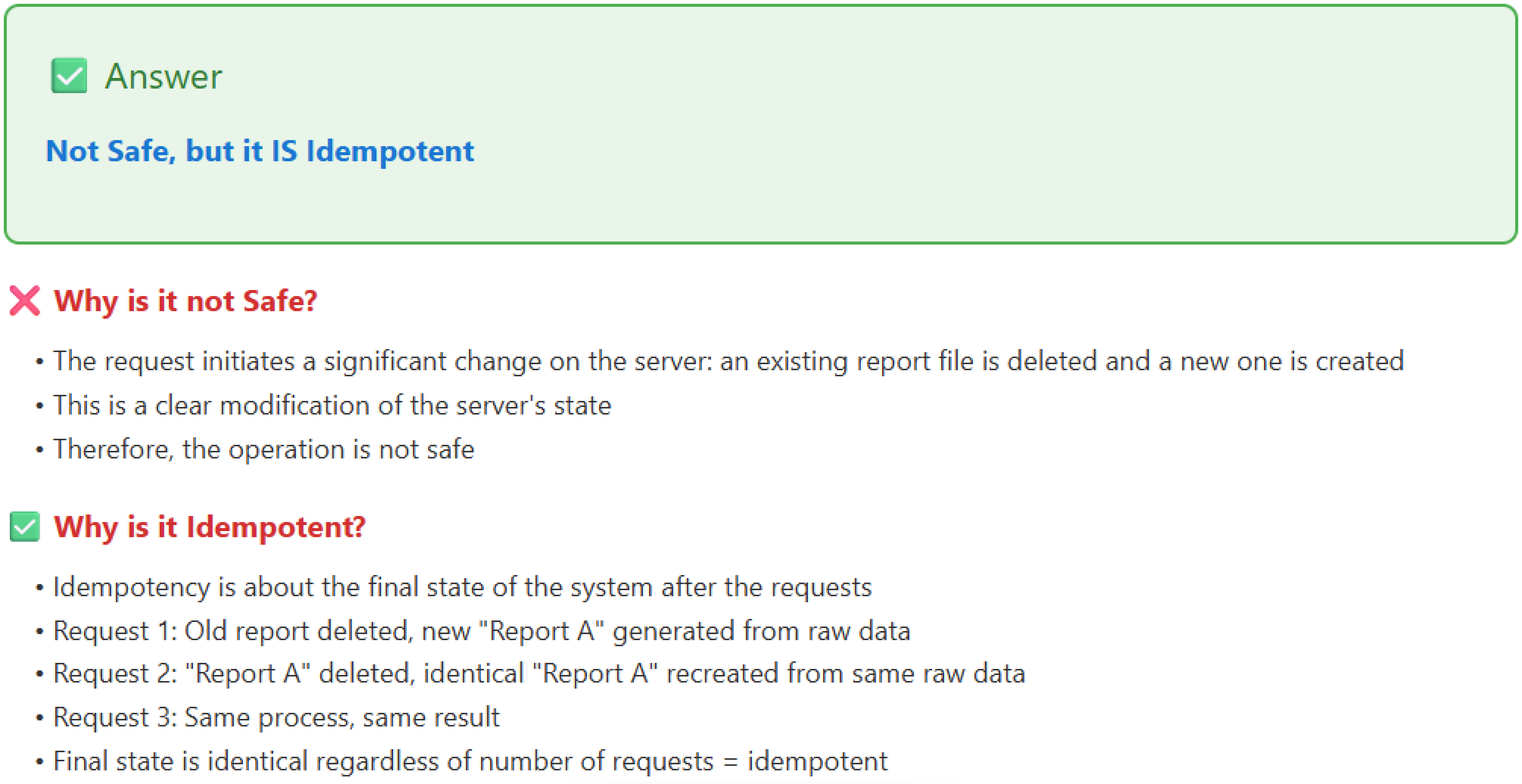
Question

Is this POST /.../recalculate operation Safe?

Is it Idempotent?

Solution 2: The "Recalculate Report" Endpoint

Analysis and Answer



✅ Answer

Not Safe, but it IS Idempotent

❌ Why is it not Safe?

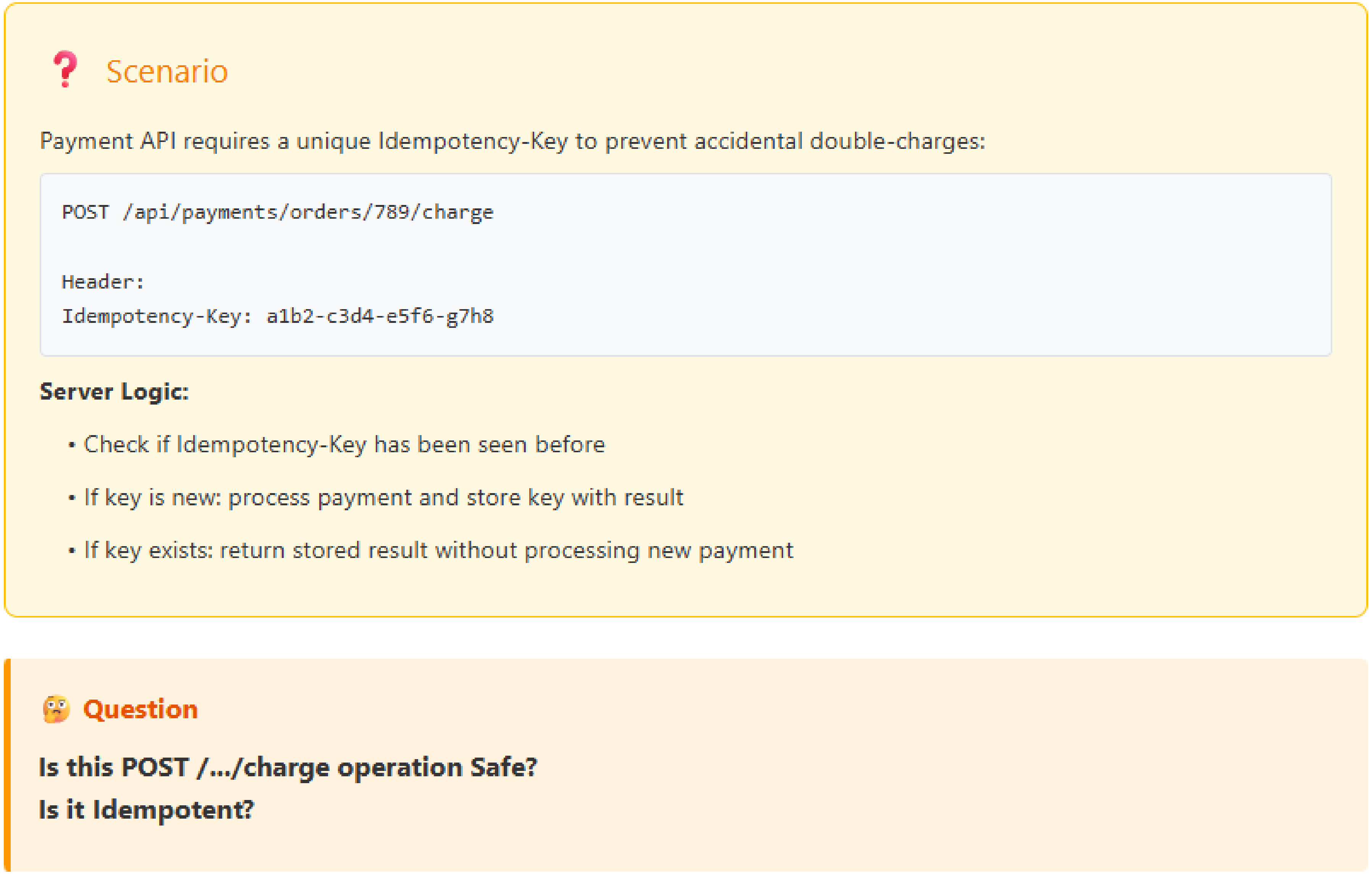
- The request initiates a significant change on the server: an existing report file is deleted and a new one is created
- This is a clear modification of the server's state
- Therefore, the operation is not safe

✅ Why is it Idempotent?

- Idempotency is about the final state of the system after the requests
- Request 1: Old report deleted, new "Report A" generated from raw data
- Request 2: "Report A" deleted, identical "Report A" recreated from same raw data
- Request 3: Same process, same result
- Final state is identical regardless of number of requests = idempotent

Problem 3: The Payment Gateway

Preventing Double-Charges with Idempotency Keys



❓ Scenario

Payment API requires a unique Idempotency-Key to prevent accidental double-charges:

```
POST /api/payments/orders/789/charge

Header:
Idempotency-Key: a1b2-c3d4-e5f6-g7h8
```

Server Logic:

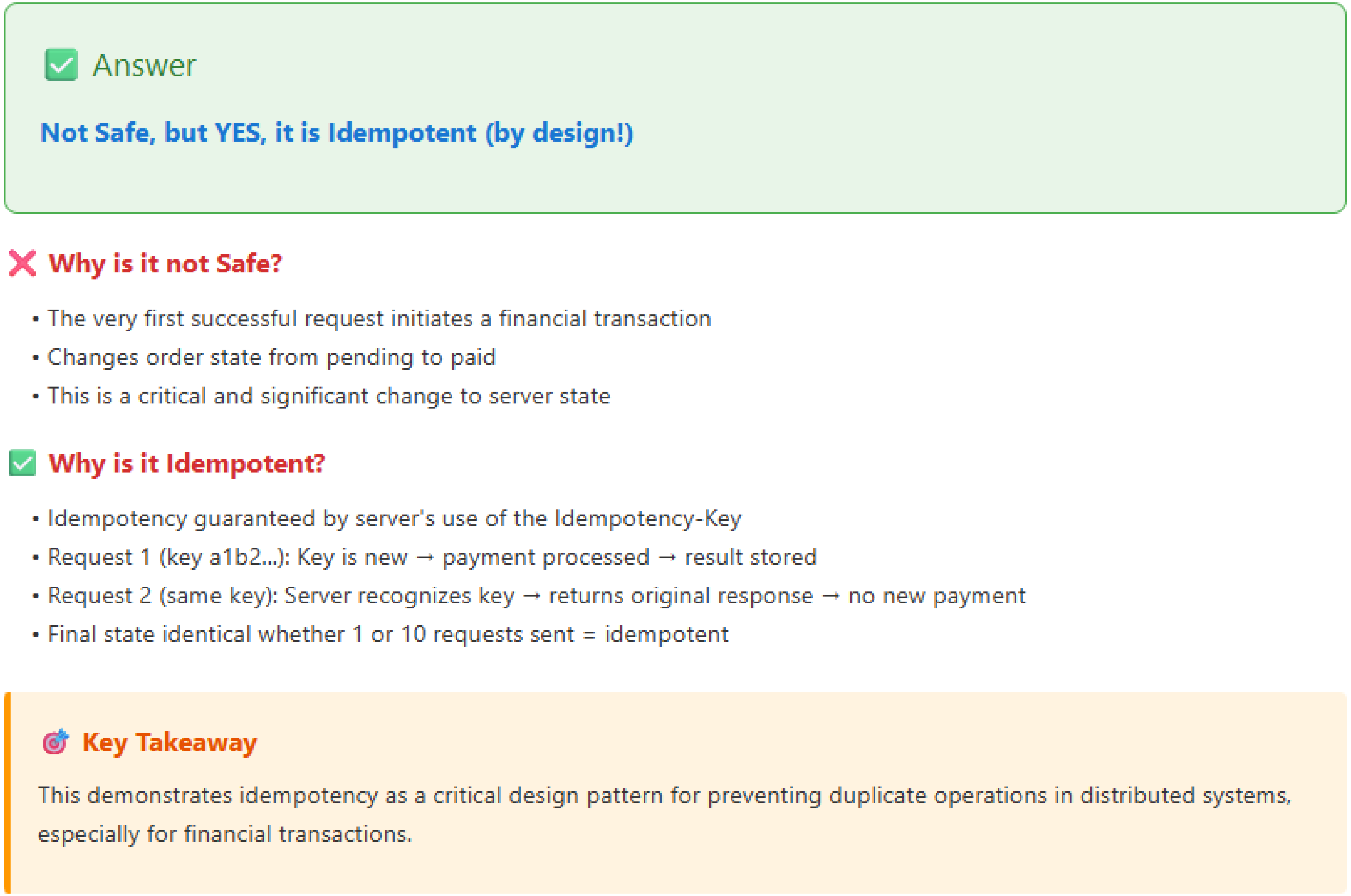
- Check if Idempotency-Key has been seen before
- If key is new: process payment and store key with result
- If key exists: return stored result without processing new payment

🤔 Question

Is this POST /.../charge operation Safe?
Is it Idempotent?

Solution 3: The Payment Gateway

Idempotency by Design



✅ Answer

Not Safe, but YES, it is Idempotent (by design!)

❌ Why is it not Safe?

- The very first successful request initiates a financial transaction
- Changes order state from pending to paid
- This is a critical and significant change to server state

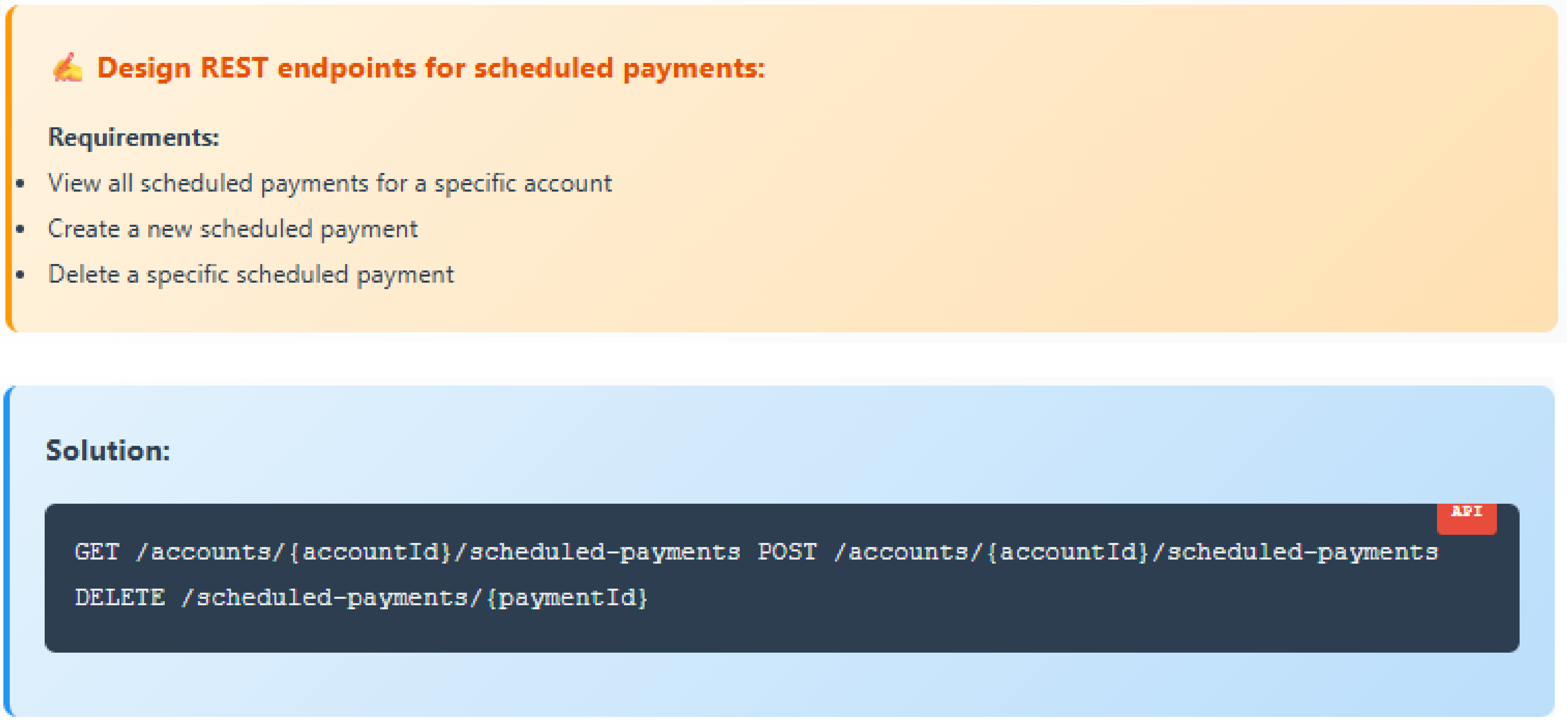
✅ Why is it Idempotent?

- Idempotency guaranteed by server's use of the Idempotency-Key
- Request 1 (key a1b2...): Key is new → payment processed → result stored
- Request 2 (same key): Server recognizes key → returns original response → no new payment
- Final state identical whether 1 or 10 requests sent = idempotent

🎯 Key Takeaway

This demonstrates idempotency as a critical design pattern for preventing duplicate operations in distributed systems, especially for financial transactions.

Exercise 1: REST Endpoints



✍ Design REST endpoints for scheduled payments:

Requirements:

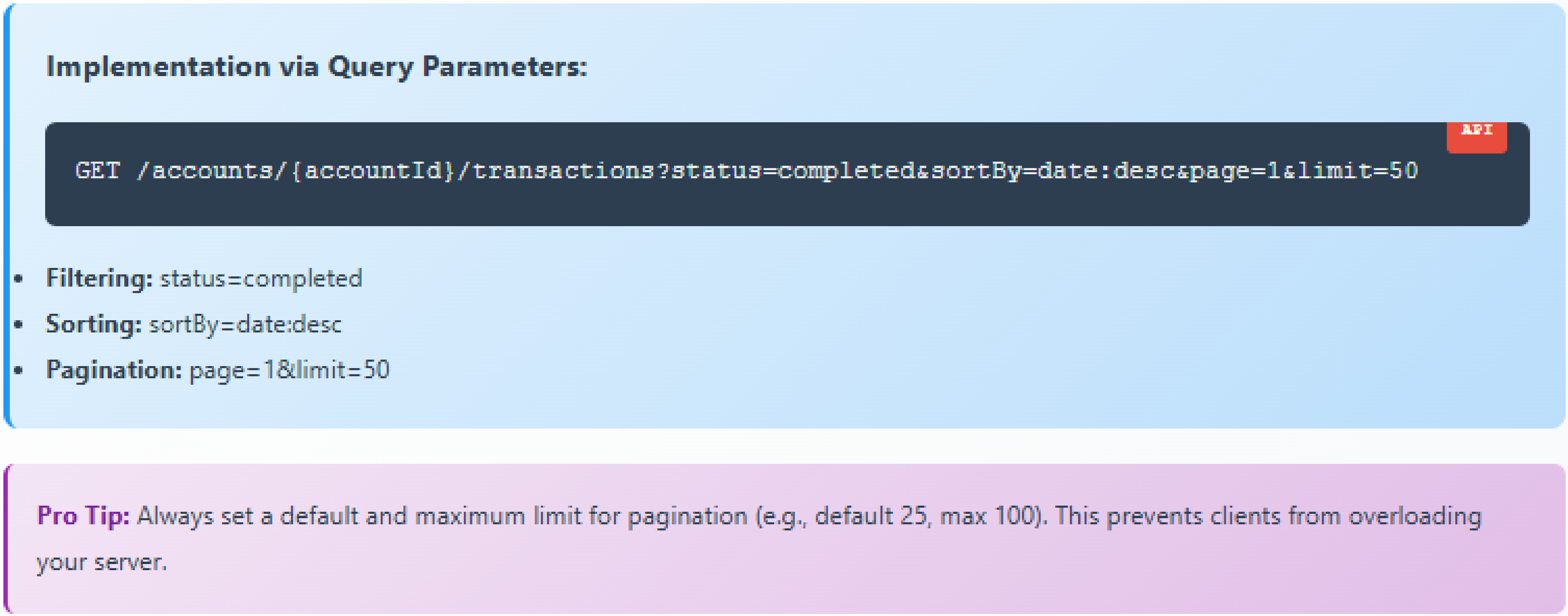
- View all scheduled payments for a specific account
- Create a new scheduled payment
- Delete a specific scheduled payment

Solution:

```
GET /accounts/{accountId}/scheduled-payments POST /accounts/{accountId}/scheduled-payments
DELETE /scheduled-payments/{paymentId}
```

Pagination, Filtering & Sorting

Banking systems deal with large amounts of data. You can't return it all at once.



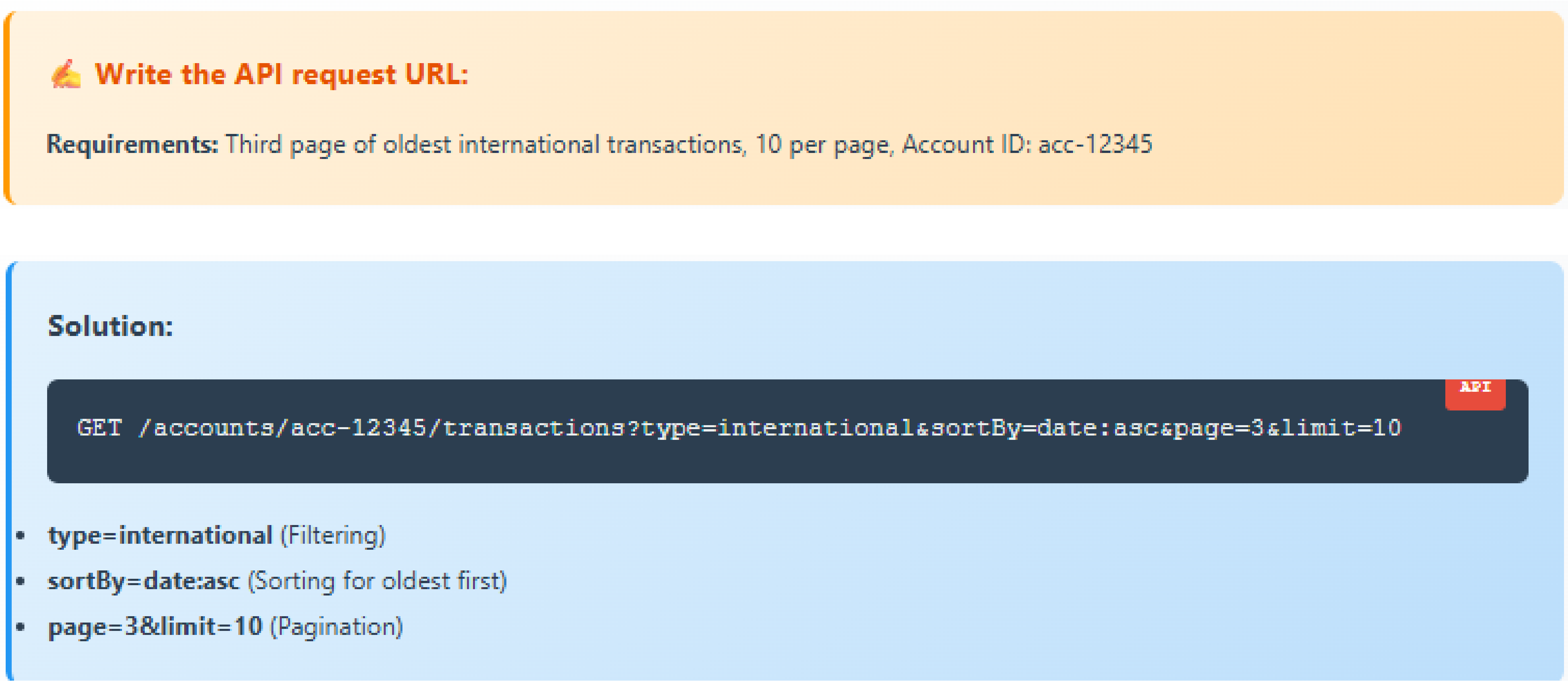
Implementation via Query Parameters:

```
GET /accounts/{accountId}/transactions?status=completed&sortBy=date:desc&page=1&limit=50
```

- **Filtering:** status=completed
- **Sorting:** sortBy=date:desc
- **Pagination:** page=1&limit=50

Pro Tip: Always set a default and maximum limit for pagination (e.g., default 25, max 100). This prevents clients from overloading your server.

Exercise 2: Query Parameters



✍ Write the API request URL:

Requirements: Third page of oldest international transactions, 10 per page, Account ID: acc-12345

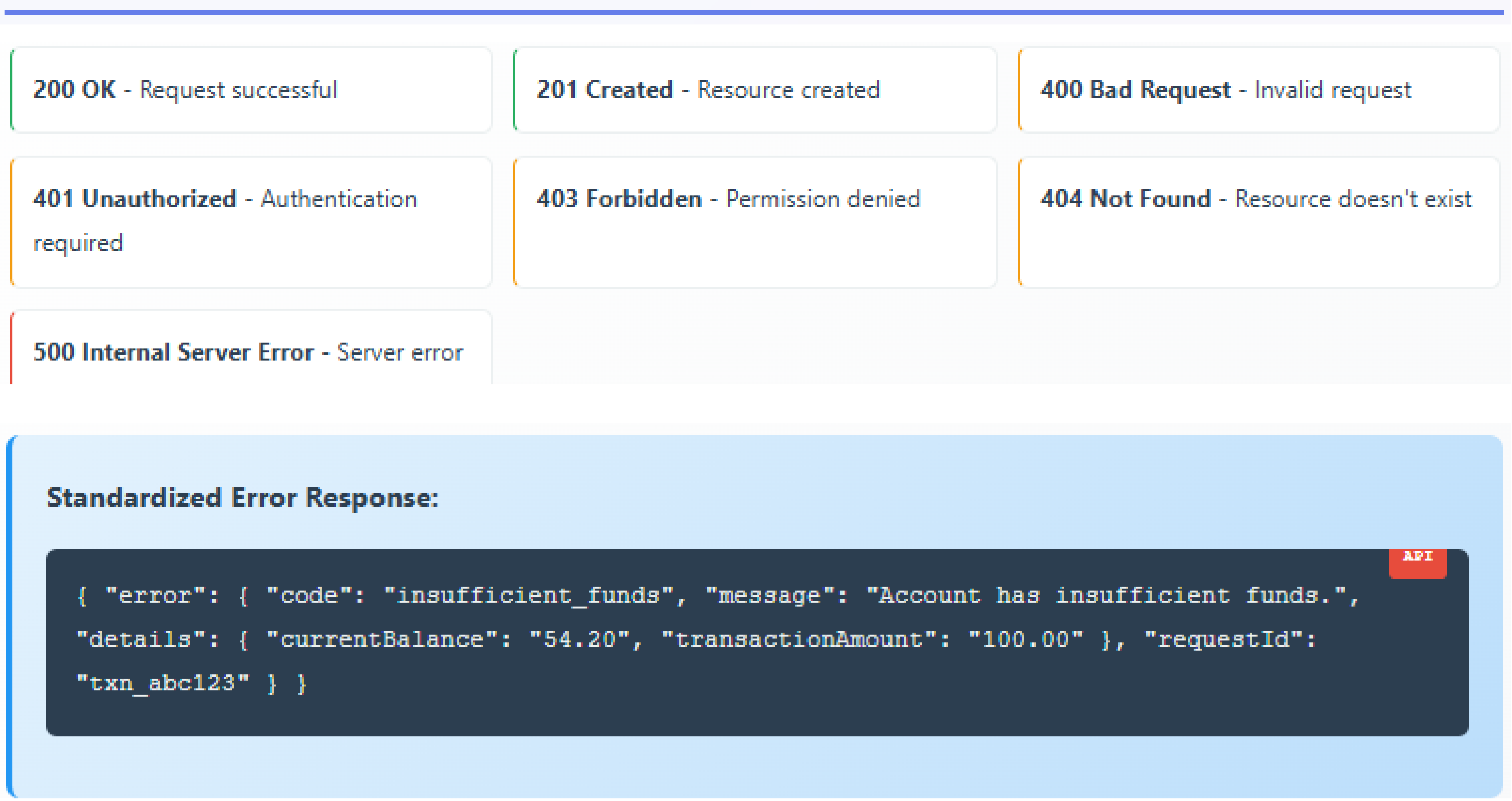
Solution:

API

```
GET /accounts/acc-12345/transactions?type=international&sortBy=date:asc&page=3&limit=10
```

- **type=international** (Filtering)
- **sortBy=date:asc** (Sorting for oldest first)
- **page=3&limit=10** (Pagination)

HTTP Status Codes & Error Handling



200 OK - Request successful

201 Created - Resource created

400 Bad Request - Invalid request

401 Unauthorized - Authentication required

403 Forbidden - Permission denied

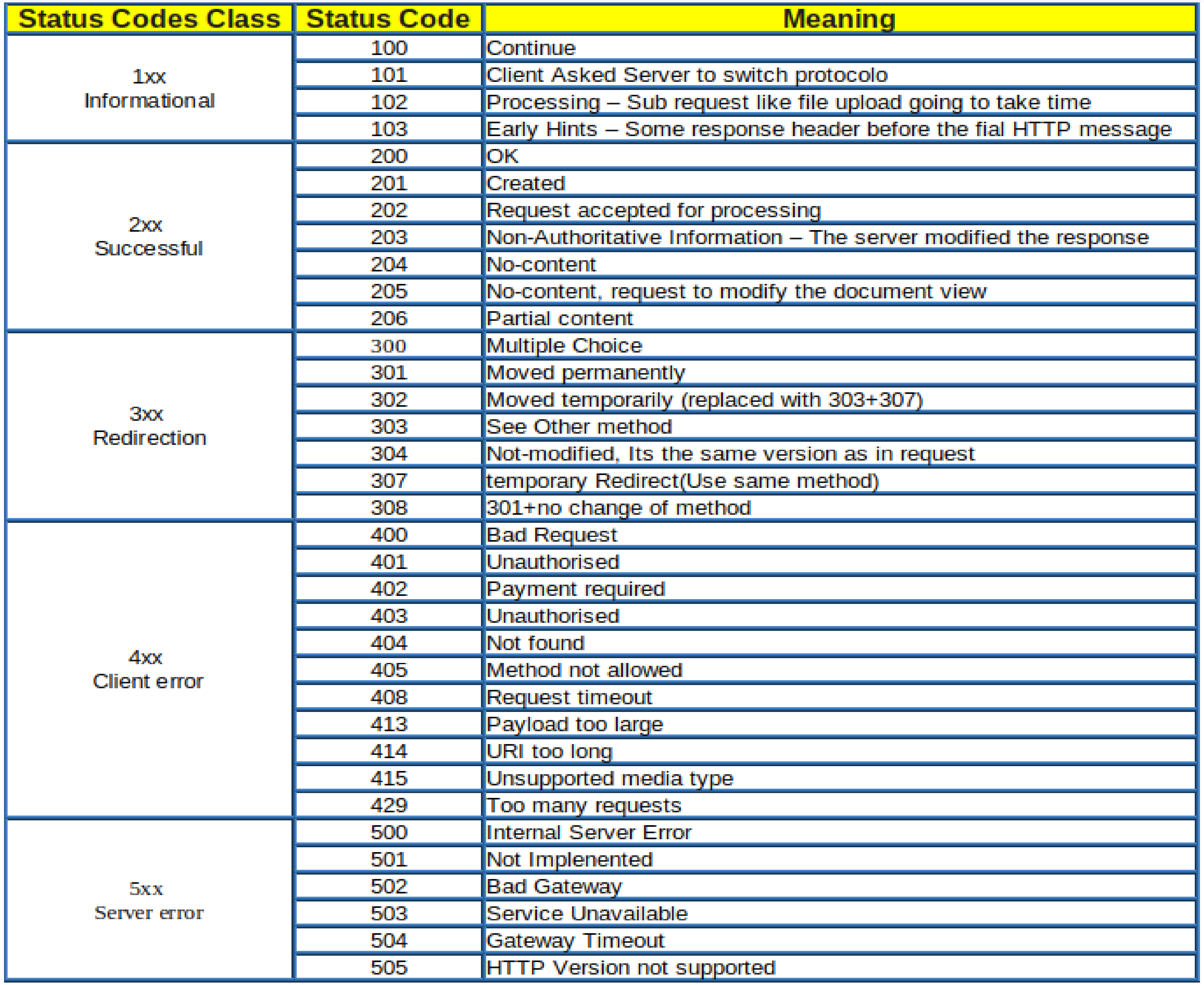
404 Not Found - Resource doesn't exist

500 Internal Server Error - Server error

Standardized Error Response:

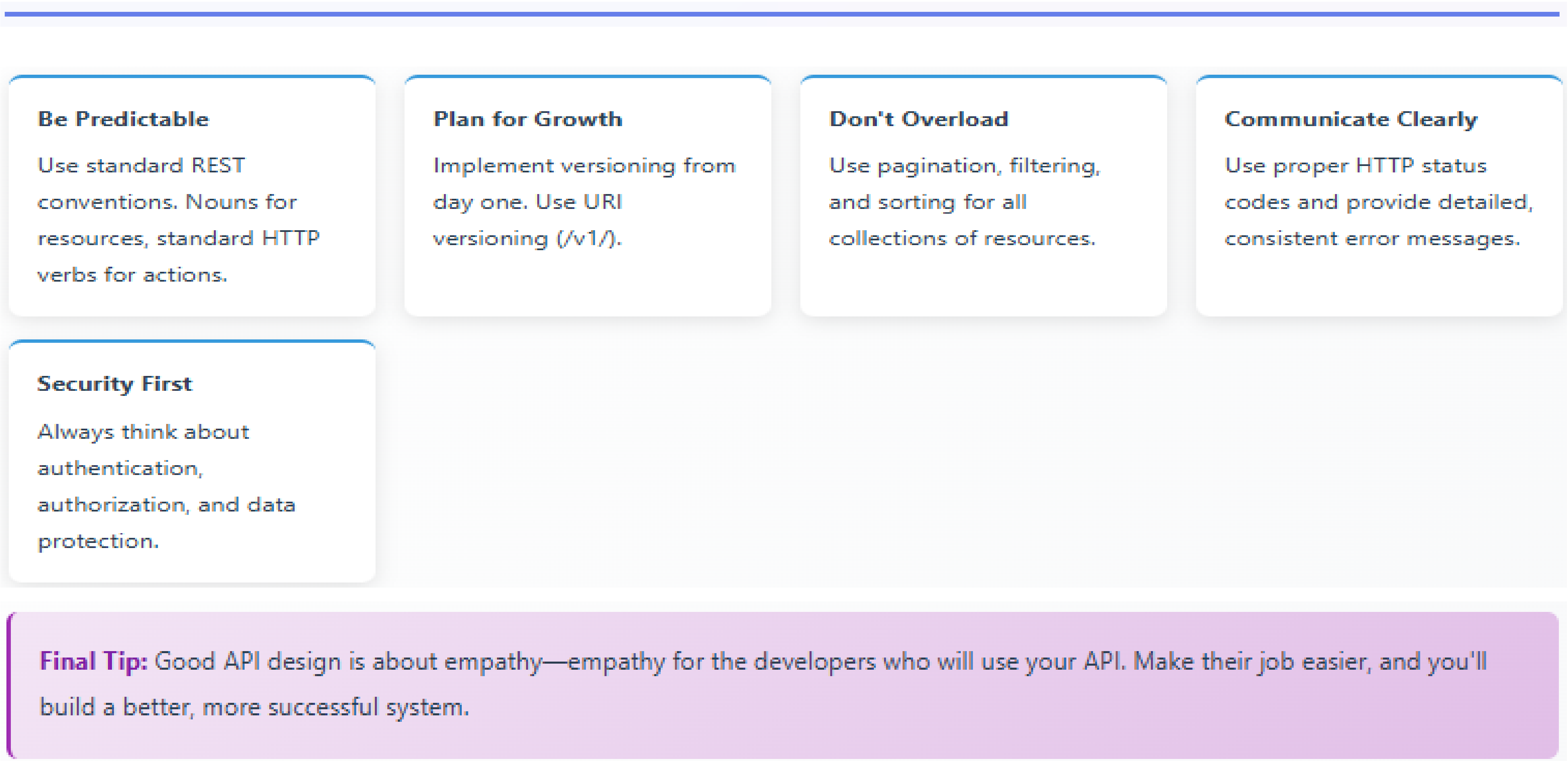
```
{ "error": { "code": "insufficient_funds", "message": "Account has insufficient funds.",
"details": { "currentBalance": "54.20", "transactionAmount": "100.00" }, "requestId":
"txn_abc123" } }
```

HTTP Status Codes



Status Codes Class	Status Code	Meaning
1xx Informational	100	Continue
	101	Client Asked Server to switch protocolo
	102	Processing – Sub request like file upload going to take time
	103	Early Hints – Some response header before the fial HTTP message
2xx Successful	200	OK
	201	Created
	202	Request accepted for processing
	203	Non-Authoritative Information – The server modified the response
	204	No-content
	205	No-content, request to modify the document view
	206	Partial content
3xx Redirection	300	Multiple Choice
	301	Moved permanently
	302	Moved temporarily (replaced with 303+307)
	303	See Other method
	304	Not-modified, Its the same version as in request
	307	temporary Redirect(Use same method)
	308	301+no change of method
4xx Client error	400	Bad Request
	401	Unauthorised
	402	Payment required
	403	Unauthorised
	404	Not found
	405	Method not allowed
	408	Request timeout
	413	Payload too large
	414	URI too long
	415	Unsupported media type
	429	Too many requests
5xx Server error	500	Internal Server Error
	501	Not Implenented
	502	Bad Gateway
	503	Service Unavailable
	504	Gateway Timeout
	505	HTTP Version not supported

Key Takeaways & Best Practices



Be Predictable

Use standard REST conventions. Nouns for resources, standard HTTP verbs for actions.

Plan for Growth

Implement versioning from day one. Use URI versioning (/v1/).

Don't Overload

Use pagination, filtering, and sorting for all collections of resources.

Communicate Clearly

Use proper HTTP status codes and provide detailed, consistent error messages.

Security First

Always think about authentication, authorization, and data protection.

Final Tip: Good API design is about empathy—empathy for the developers who will use your API. Make their job easier, and you'll build a better, more successful system.

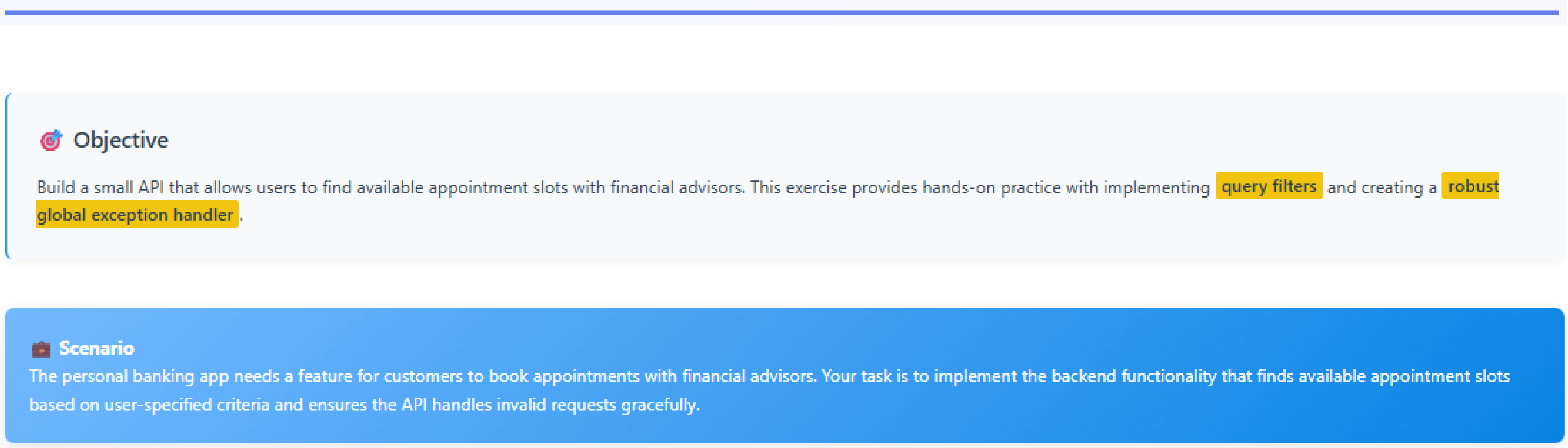
Mini-Project

Financial Advisor Appointment Booker

⏱ 30-45 minutes

Build a robust API with query filters and global exception handling

Objective & Scenario



🎯 Objective

Build a small API that allows users to find available appointment slots with financial advisors. This exercise provides hands-on practice with implementing **query filters** and creating a **robust global exception handler**.

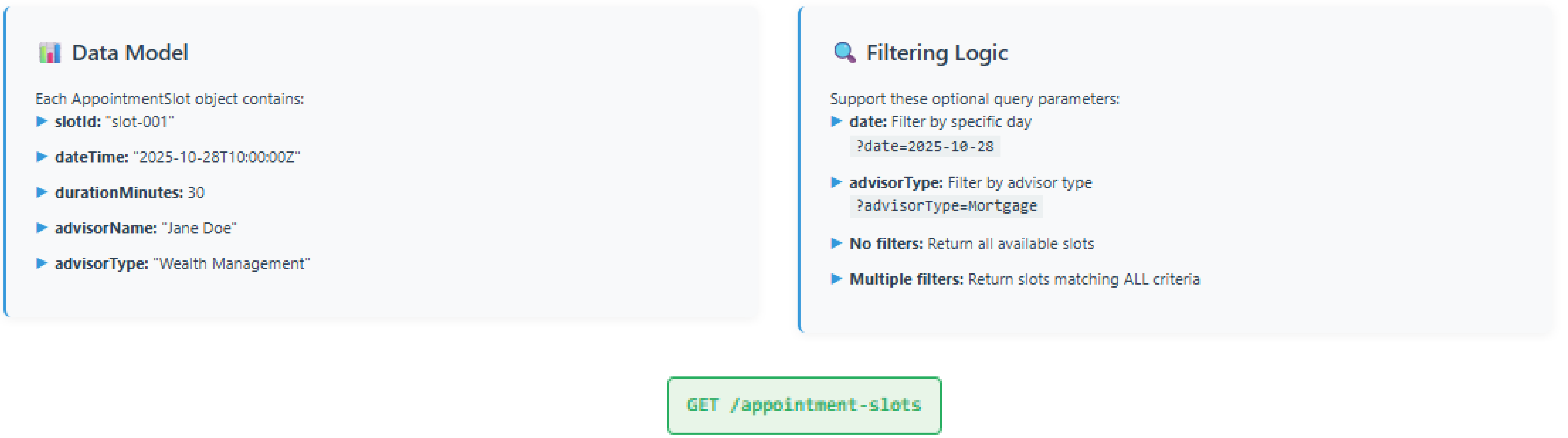
💼 Scenario

The personal banking app needs a feature for customers to book appointments with financial advisors. Your task is to implement the backend functionality that finds available appointment slots based on user-specified criteria and ensures the API handles invalid requests gracefully.

appointment-booker-java.zip

qpi-filter-mini-project.zip

Part 1: GET /appointment-slots Endpoint



📊 Data Model

Each AppointmentSlot object contains:

- ▶ **slotId:** "slot-001"
- ▶ **dateTime:** "2025-10-28T10:00:00Z"
- ▶ **durationMinutes:** 30
- ▶ **advisorName:** "Jane Doe"
- ▶ **advisorType:** "Wealth Management"

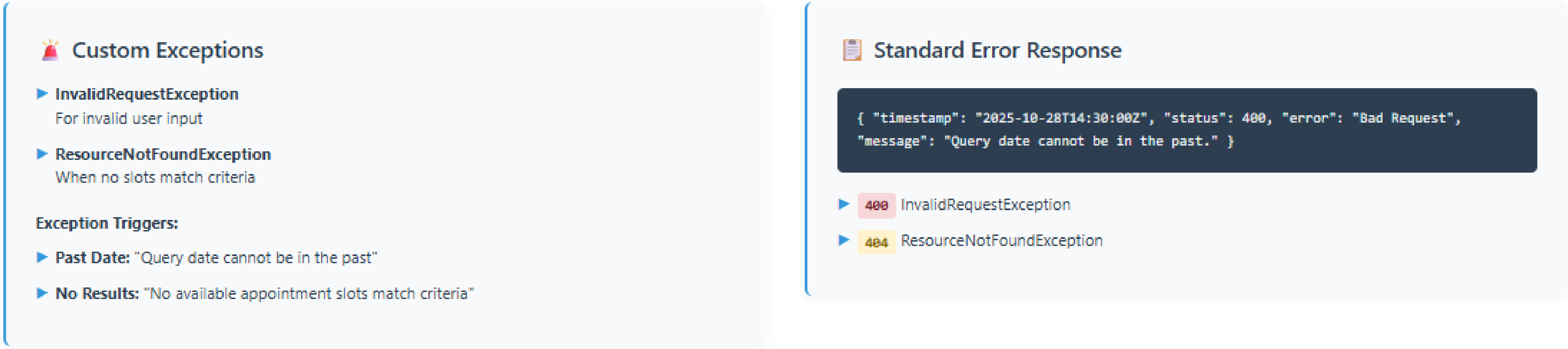
🔍 Filtering Logic

Support these optional query parameters:

- ▶ **date:** Filter by specific day `?date=2025-10-28`
- ▶ **advisorType:** Filter by advisor type `?advisorType=Mortgage`
- ▶ **No filters:** Return all available slots
- ▶ **Multiple filters:** Return slots matching ALL criteria

`GET /appointment-slots`

Part 2: Global Exception Handler



🚨 Custom Exceptions

- **InvalidRequestException**
  For invalid user input
- **ResourceNotFoundException**
  When no slots match criteria

Exception Triggers:

- **Past Date:** "Query date cannot be in the past"
- **No Results:** "No available appointment slots match criteria"

📋 Standard Error Response

```
{ "timestamp": "2025-10-28T14:30:00Z", "status": 400, "error": "Bad Request",
"message": "Query date cannot be in the past." }
```

- `400` InvalidRequestException
- `404` ResourceNotFoundException

Testing Your Implementation

✅ Success Scenarios

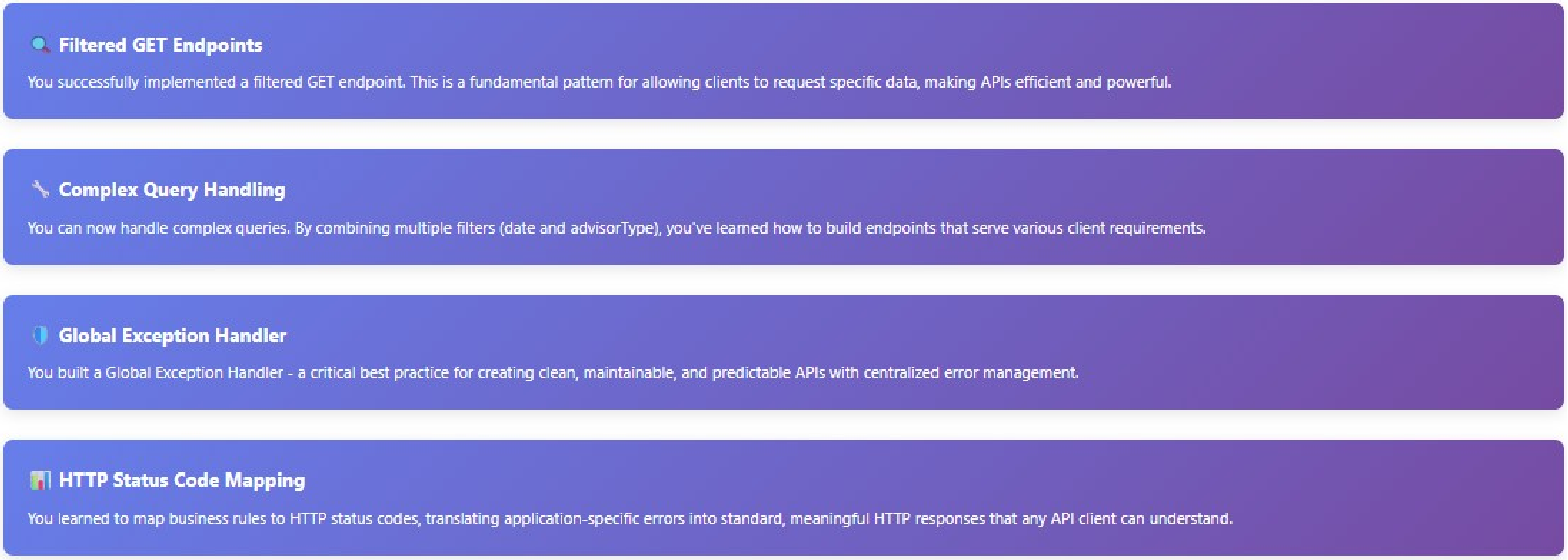
- **No Filters:**
  `GET /appointment-slots`
  `200 OK` Full list of slots
- **With Filters:**
  `GET /appointment-slots?date=2025-10-28&advisorType=Mortgage`
  `200 OK` Filtered list

❌ Error Scenarios

- **Date in Past:**
  `GET /appointment-slots?date=2020-01-01`
  `400 Bad Request` Standard error JSON
- **No Slots Found:**
  `GET /appointment-slots?advisorType=Crypto`
  `404 Not Found` Standard error JSON

💡 **Pro Tip:** Use Postman or curl to test all scenarios systematically

Mini-Project Learnings



🔍 Filtered GET Endpoints

You successfully implemented a filtered GET endpoint. This is a fundamental pattern for allowing clients to request specific data, making APIs efficient and powerful.

🔧 Complex Query Handling

You can now handle complex queries. By combining multiple filters (date and advisorType), you've learned how to build endpoints that serve various client requirements.

🛡️ Global Exception Handler

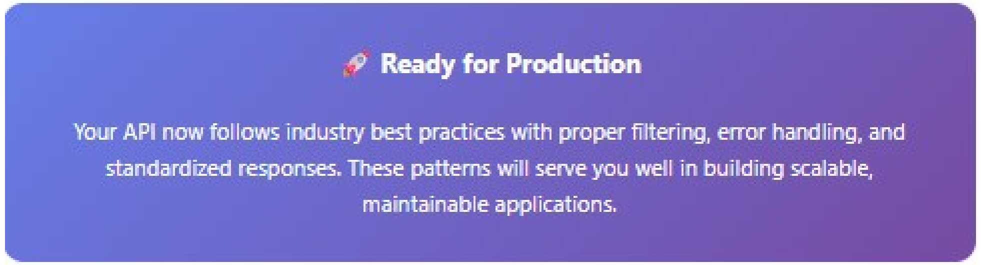
You built a Global Exception Handler - a critical best practice for creating clean, maintainable, and predictable APIs with centralized error management.

📊 HTTP Status Code Mapping

You learned to map business rules to HTTP status codes, translating application-specific errors into standard, meaningful HTTP responses that any API client can understand.

🎉 Congratulations!

You've successfully built a robust API with filtering capabilities and professional error handling.



🚀 Ready for Production

Your API now follows industry best practices with proper filtering, error handling, and standardized responses. These patterns will serve you well in building scalable, maintainable applications.