IT Management & Audits

Practical Lab Manual

API Integration Testing Suite

Practical P10

Learning Domain

API Security & Integration Testing

Course Learning Outcomes

CLO10: Test and validate RESTful API integrations for payment systems

Unit

Unit III: Application Security & Testing

Time Allocation: 3 hours

Learning Mode: Hands-on (80%) + Theory (20%)

Difficulty Level: Intermediate

API Integration Testing Suite

Practical P10

Quick Reference

Practical Code	P10
Practical Name	API Integration Testing Suite
Slot	T/P-10
Duration	3 hours
CLO Mapping	CLO10
Unit	Unit III: Application Security & Testing
Delivery Mode	Hands-on Lab
Target Audience	Intermediate Level
India Integration	HIGH
Screenshot Count	5 Required

Prerequisites

- Basic HTTP concepts (GET, POST, PUT, DELETE methods)
- Understanding of JSON data format and structure
- Python programming basics (variables, functions, imports)
- Familiarity with command-line interface and terminal usage
- Basic understanding of client-server architecture

Tools Required

Tool	**Version**	**Free Tier**	**Notes**
Python	3.8+	✓	Required runtime
Flask	Latest	✓	Mock API framework
requests	Latest	✓	HTTP client library
pytest	Latest	✓	Test framework
pip	Latest	✓	Package manager
Postman	Latest	✓	Optional, GUI API testing
Web Browser	Latest	✓	Chrome/Firefox recommended

Learning Objectives

- ✓ Set up and run a mock RESTful payment API using Flask
- ✓ Write and execute automated API tests using pytest
- ✓ Validate API authentication mechanisms (API keys, OAuth2 flows)
- ✓ Test payment processing endpoints for correctness and edge cases
- ✓ Verify rate limiting enforcement and error handling
- ✓ Use Postman collections for manual and automated API testing
- ✓ Generate and interpret API test reports
- ✓ Apply India-specific API security standards (UPI, RBI guidelines)

What You Will Learn

By the end of this practical, you will:

1. Understand RESTful API design principles and HTTP protocol fundamentals
2. Deploy and interact with a mock payment processing API
3. Write comprehensive test suites covering functional, security, and edge cases
4. Test API authentication and authorization mechanisms
5. Validate rate limiting, input validation, and error handling
6. Use industry-standard tools (pytest, Postman) for API quality assurance
7. Interpret test results and identify security vulnerabilities in APIs

Real-World Application

API integration testing is critical in India's digital payment ecosystem. Companies like **Razorpay**, **Cashfree**, and **PayU** expose RESTful APIs that process millions of transactions daily. The **Unified Payments Interface (UPI)** architecture, governed by NPCI, relies on well-tested API integrations between payment service providers, banks, and merchants. A single untested edge case in a payment API can result in financial loss, duplicate transactions, or security breaches. This practical teaches the systematic testing methodology used by FinTech companies to ensure API reliability and compliance with RBI regulations.

Hands-On Procedure

Part A: Environment Setup

Step 1: Clone the API Testing Suite Repository

Objective: Download the project repository containing the mock payment API and test suite.

Instructions:

1. Open a terminal or command prompt
2. Navigate to your preferred working directory
3. Clone the repository and explore the project structure

```
# Clone the API testing suite repository
git clone https://github.com/itma-practicals/api-testing-suite.git
cd api-testing-suite

# Project structure:
# api/
#   app.py            - Main Flask application
#   auth.py           - Authentication module
#   models.py         - Data models for payments
#   validators.py     - Input validation logic
#   rate_limiter.py   - Rate limiting middleware
#   webhooks.py       - Webhook registration/dispatch
# tests/
#   test_auth.py      - Authentication tests
#   test_payments.py  - Payment endpoint tests
#   test_rate_limit.py - Rate limiting tests
#   conftest.py       - Shared test fixtures
# postman/
#   payment_api_collection.json - Postman collection
# docs/
#   api_spec.yaml     - OpenAPI specification
# requirements.txt    - Python dependencies
```

Clone and Explore Repository

Expected Output

Repository cloned successfully.
Project directory contains: `api/`, `tests/`, `postman/`, `docs/`, `requirements.txt`

If git is not installed, download the repository as a ZIP file and extract it to your working directory.

Step 2: Create Virtual Environment and Install Dependencies

Objective: Set up an isolated Python environment and install all required packages.

```bash
# Create virtual environment
python -m venv venv

# Activate (Linux/Mac)
source venv/bin/activate
# Activate (Windows)
# venv\Scripts\activate

# Install dependencies
pip install -r requirements.txt

# Verify key packages
python -c "import flask; print('Flask:', flask.__version__)"
python -c "import requests; print('Requests:', requests.__version__)"
python -c "import pytest; print('Pytest:', pytest.__version__)"
```

Virtual Environment Setup

Expected Output

`Flask: 2.3.x` | `Requests: 2.31.x` | `Pytest: 7.4.x`
All dependencies installed successfully.

Always activate the virtual environment before running any commands. Your terminal prompt should show `(venv)` when the environment is active.

Part B: Mock Payment API

Step 3: Start the Mock Payment API Server

Objective: Launch the Flask-based mock payment API and verify endpoints.

```
# Start the mock payment API server
python api/app.py
# Output: * Running on http://127.0.0.1:5000
```

Start Mock Payment API

Open a **new terminal** (keep the server running) and test:

```
# Health check
curl http://127.0.0.1:5000/health

# Create a payment (POST /payments)
curl -X POST http://127.0.0.1:5000/payments \
  -H "Content-Type: application/json" \
  -H "X-API-Key: test_api_key_12345" \
  -d '{"amount": 1500.00, "currency": "INR",
       "description": "Test payment",
       "merchant_id": "merchant_001",
       "customer_email": "test@example.com"}'

# Check payment status (GET /payments/{id})
curl -H "X-API-Key: test_api_key_12345" \
  http://127.0.0.1:5000/payments/pay_001

# List transactions (GET /transactions)
curl -H "X-API-Key: test_api_key_12345" \
  http://127.0.0.1:5000/transactions

# Register webhook (POST /webhooks/register)
curl -X POST http://127.0.0.1:5000/webhooks/register \
  -H "Content-Type: application/json" \
  -H "X-API-Key: test_api_key_12345" \
  -d '{"url": "https://example.com/webhook",
       "events": ["payment.success", "payment.failed"]}'
```

Test API Endpoints with curl

Available Endpoints:

Method	Endpoint	Description
POST	/payments	Create a new payment
GET	/payments/{id}	Check payment status
POST	/payments/{id}/refund	Initiate payment refund
GET	/transactions	List all transactions
POST	/webhooks/register	Register a webhook URL

Expected Output

Health: `{"status": "healthy", "version": "1.0.0"}`
Payment: `{"payment_id": "pay_xxx", "status": "created", "amount": 1500.00}`

Screenshot 1

What to capture: Terminal showing the mock API server running, plus a second terminal showing the result of a `curl POST /payments` request with a successful JSON response.

Paste your screenshot here

Step 4: Explore the API Documentation

Objective: Review the OpenAPI specification to understand request/response schemas.

1. Access interactive docs at `http://127.0.0.1:5000/docs`
2. Or open `docs/api_spec.yaml` directly
3. Review: authentication requirements, request schemas, response codes, rate limits

```
# Payment Creation Request:
# {
#   "amount": number (required, > 0, <= 1000000),
#   "currency": string (required, "INR" | "USD"),
#   "description": string (optional, max 255 chars),
#   "merchant_id": string (required),
#   "customer_email": string (required, valid email),
#   "metadata": object (optional)
# }
#
# Success Response (201):
# { "payment_id": "pay_xxx", "status": "created",
#   "amount": 1500.00, "currency": "INR",
#   "created_at": "2026-02-21T10:00:00Z" }
#
# Error Response (4xx):
# { "error": "...", "code": "...", "details": {} }
```

Payment Request/Response Schema

Expected Output

API documentation displays all endpoints with request parameters, response codes (200, 201, 400, 401, 404, 429, 500), authentication requirements, and example payloads.

The OpenAPI specification (formerly Swagger) is the industry standard for documenting RESTful APIs. Postman can import this spec to auto-generate test collections.

Screenshot 2

What to capture: The API documentation page (Swagger UI at `/docs`) or the OpenAPI spec YAML file showing endpoint definitions and schemas.

Paste your screenshot here

Part C: Automated API Testing

Step 5: Run Authentication Tests

Objective: Verify API key validation, missing credentials handling, and OAuth2 mock flow.

```
# Run authentication tests with verbose output
pytest tests/test_auth.py -v

# Test cases:
# test_valid_api_key          - Valid API key returns 200
# test_missing_api_key        - Missing key returns 401
# test_invalid_api_key        - Wrong key returns 401
# test_expired_api_key        - Expired key returns 401
# test_malformed_api_key      - Malformed key returns 401
# test_oauth2_token_request - OAuth2 token endpoint works
# test_oauth2_invalid_grant - Invalid grant returns 400
# test_bearer_token_access  - Bearer token auth works
```

Run Authentication Tests

Expected Output

```
tests/test_auth.py::test_valid_api_key PASSED
tests/test_auth.py::test_missing_api_key PASSED
tests/test_auth.py::test_invalid_api_key PASSED
tests/test_auth.py::test_expired_api_key PASSED
tests/test_auth.py::test_malformed_api_key PASSED
tests/test_auth.py::test_oauth2_token_request PASSED
tests/test_auth.py::test_oauth2_invalid_grant PASSED
tests/test_auth.py::test_bearer_token_access PASSED
========== 8 passed in 1.24s ==========
```

Authentication is the first line of defense for any API. Every payment API must validate API keys or tokens before processing requests. A single bypass can expose the entire payment system to unauthorized access.

Screenshot 3

What to capture: Terminal output showing pytest authentication test results with all test case names, PASSED/FAILED status, and the final summary line.

Paste your screenshot here

Step 6: Run Payment Endpoint Tests

Objective: Execute comprehensive tests against payment creation, status check, and refund endpoints.

```
pytest tests/test_payments.py -v

# 16 test cases covering:
# Happy Path: create_payment_success, create_payment_inr,
#   create_payment_with_metadata, get_payment_status,
#   list_transactions
# Validation: invalid_amount, zero_amount, exceeds_max,
#   missing_fields, invalid_currency, invalid_email
# Business Logic: duplicate_payment_idempotency,
#   refund_completed_payment, refund_already_refunded,
#   get_nonexistent_payment, partial_refund
```

Run Payment Endpoint Tests

Expected Output

```
tests/test_payments.py::test_create_payment_success PASSED
tests/test_payments.py::test_create_payment_invalid_amount PASSED
tests/test_payments.py::test_duplicate_payment_idempotency PASSED
tests/test_payments.py::test_refund_completed_payment PASSED
...
========== 16 passed in 3.45s ==========
```

In production payment APIs like Razorpay, idempotency keys prevent duplicate charges when a client retries a failed request. A duplicate `POST /payments` could charge the customer twice without this safeguard.

Step 7: Run Rate Limiting Tests

Objective: Verify rate limiting enforcement, 429 responses, and retry-after headers.

```
pytest tests/test_rate_limit.py -v

# test_rate_limit_headers_present    - Headers in response
# test_rate_limit_counter_decrements - Counter goes down
# test_rate_limit_exceeded_429       - 429 when exceeded
# test_retry_after_header            - Retry-After present
# test_rate_limit_resets             - Counter resets
# test_rate_limit_per_api_key        - Per-key limiting
```

Run Rate Limiting Tests

Expected Output

```
tests/test_rate_limit.py::test_rate_limit_headers_present PASSED
tests/test_rate_limit.py::test_rate_limit_exceeded_429 PASSED
tests/test_rate_limit.py::test_retry_after_header PASSED
...
========== 6 passed in 2.87s ==========
```

Rate limiting is mandatory for payment APIs. Without it, attackers can perform brute-force attacks, cause denial of service, or exhaust transaction quotas. RBI guidelines mandate rate limiting for all financial APIs.

Step 8: Run Full Test Suite with Coverage

Objective: Execute the entire test suite and review consolidated results.

```
# Run all tests with short traceback
pytest tests/ -v --tb=short

# Run with coverage report
pytest tests/ -v --tb=short --cov=api --cov-report=term-missing
```

Run Full Test Suite

Expected Output

```
----- coverage:  platform linux, python 3.10.x -----
Name                  Stmts    Miss    Cover
api/app.py              85      4       95%
api/auth.py             42      2       95%
api/models.py           38      1       97%
api/validators.py        31      0      100%
api/rate_limiter.py      28      3        89%
api/webhooks.py         22      2       91%
TOTAL                  246     12      95%
========== 30 passed in 7.62s ==========
```

Coverage of 95%+ is excellent for API code. Focus on 100% coverage for critical paths (payment creation, authentication). In FinTech, untested code paths represent financial and security risks.

Screenshot 4

What to capture: Terminal output showing the full test suite results with all test names, PASSED/FAILED status, coverage summary table, and the final pass/fail count.

Paste your screenshot here

Part D: Postman Testing and Reporting

Step 9: Import and Run Postman Collection

Objective: Import the pre-built Postman collection and run API tests through the GUI.

1. Open Postman (download from `https://www.postman.com/downloads/` if needed)
2. Click **Import** and select `postman/payment_api_collection.json`
3. Set environment variables: `base_url` = `http://127.0.0.1:5000`, `api_key` = `test_api_key_12345`
4. Run individual requests or click **Run Collection** for all requests
5. Review test results in the Runner tab

Collection contains: Health Check, Authentication (valid/missing/invalid key), Payments (create INR/USD, get status, list, invalid amount, missing fields), Refunds (refund, double refund), Webhooks (register, list) – 13 requests with 25 test assertions.

Expected Output

Postman Runner: Total Requests: 13 | Passed Tests: 25/25 | Failed: 0 | Avg Response Time: < 50ms

Run Postman collections from the command line using Newman: `npx newman run postman/payment_api_collection.json`

Step 10: Generate and Review Test Report

Objective: Generate a comprehensive test report and review results.

```
# Install pytest-html if needed
pip install pytest-html

# Generate HTML test report
pytest tests/ -v --html=report.html --self-contained-html

# Alternative: JUnit XML for CI/CD
pytest tests/ -v --junitxml=report.xml
```

Generate Test Report

Error Codes Encountered During Testing:

Code	Meaning	When Returned
200	OK	Successful GET requests
201	Created	Payment created successfully
400	Bad Request	Invalid input, missing fields
401	Unauthorized	Missing or invalid API key
404	Not Found	Payment ID does not exist
409	Conflict	Duplicate refund attempt
429	Too Many Requests	Rate limit exceeded
500	Internal Server Error	Unexpected server failure

Expected Output

HTML report at `report.html`: Tests Run: 30 | Passed: 30 (100%) | Failed: 0 | Duration: 7.62s

Screenshot 5

What to capture: Postman collection runner showing test results with request names, status codes, and pass/fail indicators. Alternatively, show the pytest HTML report in a browser.

Paste your screenshot here

Conceptual Background

REST API Fundamentals

REST (Representational State Transfer) is an architectural style for designing networked applications. RESTful APIs use HTTP methods to perform operations on resources identified by URLs.

Method	Action	Example	Idempotent
GET	Retrieve resource	GET /payments/123	Yes
POST	Create resource	POST /payments	No
PUT	Replace resource	PUT /payments/123	Yes
PATCH	Partial update	PATCH /payments/123	No
DELETE	Remove resource	DELETE /payments/123	Yes

Key HTTP Headers: `Content-Type: application/json` (body format), `Authorization: Bearer <token>` (auth), `X-API-Key` (API key), `X-RateLimit-Remaining` (quota), `Idempotency-Key` (dedup).

API Authentication Methods

1. **API Keys:** Simple string tokens in headers. Used by Razorpay, Stripe for server-to-server calls.
2. **OAuth 2.0:** Token-based authorization with multiple grant types. Used for user-delegated access.
3. **JWT (JSON Web Tokens):** Self-contained signed tokens encoding user claims. Stateless auth.
4. **HMAC Signatures:** Request signing using shared secret. Prevents tampering and replay attacks.

API Testing Methodology

1. **Functional Testing:** Verify correct responses for valid inputs, CRUD operations, business rules.
2. **Security Testing:** Validate authentication, authorization, input sanitization, OWASP API Top 10.
3. **Performance Testing:** Measure response times, throughput, rate limiting behavior under load.
4. **Integration Testing:** Validate end-to-end flows across multiple endpoints (payment to refund lifecycle).

HTTP Status Codes Reference

Code	Name	Meaning
2xx – Success		
200	OK	Request succeeded
201	Created	Resource created successfully
204	No Content	Success, no response body
4xx – Client Error		
400	Bad Request	Invalid syntax or missing fields
401	Unauthorized	Authentication required or failed
403	Forbidden	Authenticated but not authorized
404	Not Found	Resource does not exist
409	Conflict	Duplicate operation or state conflict
422	Unprocessable Entity	Valid syntax but semantic errors
429	Too Many Requests	Rate limit exceeded
5xx – Server Error		
500	Internal Server Error	Unexpected server failure
502	Bad Gateway	Upstream service unavailable
503	Service Unavailable	Server temporarily overloaded

API Security Best Practices

1. **Input Validation:** Validate all parameters (type, length, range, format). Use allowlists.
2. **Rate Limiting:** Enforce per-key quotas. Return 429 with Retry-After header.
3. **CORS:** Restrict which domains can make API requests. Configure allowed origins explicitly.
4. **HTTPS:** Encrypt all API traffic with TLS 1.2+. Use HSTS headers.
5. **Error Handling:** Return generic messages. Never expose stack traces or internal paths.

Payment API Patterns

Idempotency: If a client sends the same `POST /payments` twice with the same idempotency key, the API returns the same response without duplicate charges.

Webhooks: APIs push notifications to registered URLs when events occur (payment completed, refund processed). Payloads are signed with HMAC to prevent spoofing.

Reconciliation: Transaction listing endpoints with filtering and pagination enable merchants to reconcile payments against their records.

India Context: API Ecosystems in Indian FinTech

UPI API Architecture

The Unified Payments Interface (UPI), developed by NPCI, is built on RESTful API principles:

- **PSP APIs:** PhonePe, Google Pay, Paytm integrate with NPCI through standardized APIs
- **Collect/Pay APIs:** Two transaction flows – collect (pull) and pay (push) requests
- **Callback Architecture:** Asynchronous payment confirmation via webhook-style callbacks
- **VPA Resolution:** API calls to resolve Virtual Payment Addresses to bank accounts
- **Transaction Limits:** API-enforced limits (INR 1,00,000 P2P, INR 5,00,000 P2M)

RBI API Security Guidelines

1. All API communication must use TLS 1.2 or higher
2. API keys must be rotated every 90 days
3. Rate limiting mandatory on all public-facing endpoints
4. API access logs retained for 5 years minimum
5. Two-factor authentication for sensitive operations
6. Transaction APIs must support idempotency
7. Webhook signatures must be verified by receiving parties

Account Aggregator Framework APIs

India's Account Aggregator (AA) framework enables consent-based financial data sharing:

- **FIP:** Banks expose APIs to share account data
- **FIU:** Lenders consume APIs to access financial data
- **AA:** Middleware managing consent and data flow via APIs

Real-World: Razorpay and Cashfree API Design

Razorpay: RESTful endpoints (`/v1/payments`), API versioning in URL, dual auth (API key + secret for server, OAuth for dashboard), comprehensive webhooks with 24-hour retry, mandatory idempotency keys, sandbox environment with test keys.

Cashfree: Token-based auth with short-lived sessions, separate production/sandbox endpoints, automated settlement APIs for reconciliation, real-time webhook notifications.

Assessment & Deliverables

Assessment Questions

Q1. Explain the difference between API keys and OAuth 2.0 authentication. When would you use each in a payment system?

Q2. What HTTP status code should a payment API return when rate limits are exceeded? What headers should accompany this response?

Q3. Describe idempotency in payment APIs. Why is it critical, and how is it implemented?

Q4. List five test cases for a `POST /payments` endpoint with expected HTTP status codes.

Q5. Differentiate functional testing from security testing for APIs. Provide two examples of each.

Q6. Explain how webhooks work in payment processing. How does a merchant verify webhook authenticity?

Q7. List four key RBI guidelines for securing financial APIs in India.

Q8. How does UPI API architecture differ from traditional payment gateway APIs? Discuss NPCI's role.

Deliverables Checklist

Item	Description	Type	Status
Screenshot 1	Mock API running + curl test	Paste	☐
Screenshot 2	API documentation (OpenAPI)	Paste	☐
Screenshot 3	pytest authentication test results	Paste	☐
Screenshot 4	Full test suite pass/fail summary	Paste	☐
Screenshot 5	Postman collection test results	Paste	☐
Test Report	HTML or terminal test report	File	☐
Answers	Assessment questions Q1–Q8	Text	☐
Coverage	Code coverage percentage	Text	☐

Verification Checklist

☐ Repository cloned and project structure verified

- ☐ Virtual environment created and all dependencies installed
- ☐ Mock payment API server starts without errors
- ☐ All five API endpoints respond correctly to curl requests
- ☐ API documentation accessible at /docs or via YAML file
- ☐ Authentication tests pass (8/8 tests)
- ☐ Payment endpoint tests pass (16/16 tests)
- ☐ Rate limiting tests pass (6/6 tests)
- ☐ Full test suite runs with 95%+ pass rate
- ☐ Postman collection imported and run successfully
- ☐ Test report generated (HTML or XML format)
- ☐ All 5 screenshots captured and pasted
- ☐ All 8 assessment questions answered

Grading Rubric

Criteria	**Description**	**Points**	**Score**
Environment Setup	Repo cloned, venv, deps installed	10	___/10
Mock API	API running, endpoints accessible	10	___/10
API Documentation	OpenAPI spec reviewed	5	___/5
Auth Tests	Authentication suite passes	15	___/15
Payment Tests	Payment endpoint tests pass	15	___/15
Rate Limit Tests	Rate limiting validated	10	___/10
Full Suite	All tests pass with coverage	10	___/10
Postman	Collection imported and run	10	___/10
Screenshots	All 5 clear and complete	10	___/10
Assessment	All 8 questions answered	5	___/5
	TOTAL	**100**	___/100

Appendix A: REST API Design Best Practices

- Use nouns for URLs: `/payments`, `/transactions` (not `/createPayment`)
- Use HTTP methods for actions: GET=read, POST=create, PUT=replace, DELETE=remove
- Version your API: `/v1/payments` in URL path or via headers
- Consistent error format: always include error code, message, and details
- Support pagination (`?page=1&limit=20`) and filtering (`?status=completed`)
- Use plural resource names; document with OpenAPI specifications

Appendix B: pytest Quick Reference

```
pytest tests/                         # Run all tests
pytest tests/test_auth.py             # Run specific file
pytest tests/test_auth.py::test_valid_api_key  # Specific test
pytest -v                             # Verbose output
pytest --tb=short                     # Short traceback
pytest -x                             # Stop on first failure
pytest -k "auth"                      # Match keyword
pytest -s                             # Show print output
pytest --cov=api --cov-report=term-missing  # Coverage
pytest --html=report.html --self-contained-html  # HTML report
pytest --junitxml=results.xml         # JUnit XML for CI/CD
```

pytest Commands

Appendix C: curl and Postman Reference

```
# GET with headers
curl -H "X-API-Key: key" http://localhost:5000/payments
# POST with JSON body
curl -X POST http://localhost:5000/payments \
  -H "Content-Type: application/json" \
  -H "X-API-Key: key" \
  -d '{"amount": 500, "currency": "INR"}'
# DELETE request
curl -X DELETE -H "X-API-Key: key" \
  http://localhost:5000/payments/pay_001
# Show headers (-i) or verbose (-v)
curl -i http://localhost:5000/health
```

curl Commands for API Testing

Postman: Import collection (`.json` file), set environment variables (`base_url`, `api_key`), right-click collection and Run. Test scripts use `pm.test()` and `pm.expect()` for assertions on status codes, response body, and headers.

Appendix D: Troubleshooting Guide

Solutions: (1) Verify venv is activated (`(venv)` in prompt). (2) Reinstall: `pip install -r requirements.txt`. (3) Check Python 3.8+: `python -version`. (4) Check port 5000 not in use: `lsof -i :5000`. (5) Kill existing process and retry.

Solutions: (1) Ensure API server is running in separate terminal. (2) Verify URL matches `conftest.py` config (default: `http://127.0.0.1:5000`). (3) Set `export API_PORT=5001` if using different port. (4) Wait for "Running on" message before running tests.

Solutions: (1) Verify server accessible via curl first. (2) Check `base_url` env variable in Postman. (3) Disable SSL verification if needed. (4) For WSL/Docker, use host IP instead of localhost. (5) Check firewall allows port 5000.

Appendix E: Resources and Tools

Documentation: Flask (`flask.palletsprojects.com`), pytest (`docs.pytest.org`), Requests (`docs.python-requests.org`), OpenAPI (`swagger.io/specification`), Postman (`learning.postman.com`), RBI (`rbi.org.in`), NPCI UPI (`npci.org.in/what-we-do/upi`), Razorpay API (`razorpay.com/docs/api`), Cashfree (`docs.cashfree.com`)

Books: "API Security in Action" (Manning), "Test-Driven Development with Python" (O'Reilly).
Tools: Python 3.8+ (runtime), Flask (mock API), requests (HTTP client), pytest/pytest-cov/pytest-html (testing, coverage, reports), Postman (GUI testing), curl (CLI HTTP client) – all free and open source.

—END OF LAB MANUAL—

Document Version: 1.0

IT Management & Audits – Practical Lab Series