Programming for Automation

TEKsystems Global Services

WORKFORCE DEVELOPMENT

What Is a REST API?

- A REST API is a way for software systems to communicate over HTTP using standardized rules. It exposes data and actions through URLs and HTTP methods rather than user interfaces.

  - Designed for program-to-program communication
  - Uses HTTP as the transport protocol
  - Common in web services and automation tooling

REST Is Resource-Oriented

- REST APIs organize data around resources instead of actions or commands. Each resource is identified by a URL and manipulated using HTTP methods.
  - Resources are nouns like /users or /routes
  - URLs identify what data is being accessed
  - HTTP methods define the action taken

Stateless Communication

- REST APIs are stateless, meaning the server does not remember previous requests. Every request must include all information needed to process it.
  - No server-side session memory by default
  - Improves scalability and reliability
  - Clients are responsible for context

HTTP Methods in Practice

- HTTP methods describe what action a client wants to perform on a resource. Using the correct method is critical for predictable API behavior.
  - GET retrieves existing data
  - POST creates new data
  - PUT updates or replaces data
  - DELETE removes data

Why REST Matters for Automation

- Most modern infrastructure and cloud platforms expose REST APIs. Automation scripts rely on REST to create, modify, and inspect systems programmatically.
  - Used by cloud providers, network devices, and SaaS platforms
  - Enables repeatable and scriptable workflows
  - Forms the foundation of API-driven automation

HTTP Requests and Responses

- All REST communication happens through HTTP requests and responses. The client sends a request and the server returns a structured response.
  - Requests include method, URL, headers, and optional body
  - Responses include status code, headers, and optional body
  - Clients must interpret responses correctly

Request Headers

- Headers provide metadata about an HTTP request. They tell the server how to interpret the request and its contents.
  - Content-Type describes the request body format
  - Accept tells the server what response formats are allowed
  - Headers are key-value pairs

Response Headers

- Response headers provide metadata about the server's reply. They help clients understand how to process the response.
  - Content-Type describes the response body format
  - Headers can control caching and behavior
  - Clients rely on headers for parsing

Status Codes Drive Logic

- Clients should rely on HTTP status codes instead of message text. Status codes allow programs to make decisions automatically.
  - 2xx codes indicate success
  - 4xx codes indicate client errors
  - 5xx codes indicate server errors

Why JSON Is the Default

- JSON is the most common data format used in REST APIs. It balances readability for humans with ease of parsing for machines.
  - Lightweight text-based format
  - Maps cleanly to Python dictionaries
  - Widely supported across platforms

APIs Are Contracts

- An API defines a contract between a client and a server. Breaking that contract causes client applications to fail.
  - Endpoints should remain stable
  - Response formats should be consistent
  - Changes require coordination

Versioning APIs

- API versions allow changes without breaking existing clients. Versioning is a design decision, not an afterthought.
  - Versions may appear in URLs
  - Older versions should remain supported
  - Clients must opt into breaking changes

Introduction to http.server

- Python includes a built-in HTTP server in the standard library. It allows you to build APIs without external dependencies.
  - Part of the Python standard library
  - Useful for learning and internal tools
  - Not designed for production scale

HTTPServer

- HTTPServer is responsible for listening on a network port. It accepts incoming HTTP requests and forwards them to a handler.
  - Binds to an address and port
  - Runs a request loop
  - Delegates request handling

BaseHTTPRequestHandler

- BaseHTTPRequestHandler processes individual HTTP requests. You override methods to define how the server behaves.
  - do_GET handles GET requests
  - do_POST handles POST requests
  - Other methods exist for PUT and DELETE

Manual Routing with Paths

- The http.server module does not provide automatic routing. Your code must inspect the request path and decide how to respond.
  - Use self.path to read the request URL
  - startswith() is commonly used for variable paths
  - Routing logic lives inside each HTTP method

Parsing IDs from URLs

- REST APIs often encode resource identifiers directly in the URL. Handlers must safely extract and validate these identifiers.
  - Split paths using '/'
  - Convert IDs to integers carefully
  - Invalid IDs should return 400 or 404

Reading Request Bodies

- POST and PUT requests usually include a body with data. The server must read the exact number of bytes sent by the client.
  - Read Content-Length from headers
  - Use self.rfile.read(length)
  - Decode bytes before parsing JSON

Parsing JSON Input

- JSON request bodies must be parsed into Python objects. Invalid JSON should always be treated as a client error.
  - Use json.loads()
  - Wrap parsing in try/except
  - Return 400 on malformed input

Returning JSON Output

- Servers return JSON by serializing Python objects. The response must include both data and correct headers.
  - Use json.dumps() to serialize
  - Always set Content-Type
  - Encode strings to bytes before writing

Status Codes in Handlers

- Status codes communicate success or failure to the client. Clients depend on these codes to decide next actions.
  - 200 for successful reads or updates
  - 201 for new resources
  - 204 when no response body is needed

Handling Missing Resources

- Requests may target resources that do not exist. The server must clearly communicate this condition.
  - Return 404 when a resource is missing
  - Avoid returning empty success responses
  - Do not expose internal errors

Protecting Critical Resources

- Some resources should not be modified or deleted. APIs should enforce these rules consistently.
  - Check resource attributes before deletion
  - Return 403 for forbidden actions
  - Never rely on client behavior

In-Memory Data Stores

- This lab uses in-memory dictionaries as a stand-in for a database. All data is lost when the server restarts.
  - Fast and simple for learning
  - No persistence between runs
  - Common for prototypes and demos

Why This Matters

- Understanding raw HTTP handling builds strong fundamentals. Frameworks automate this work, but the concepts remain the same.
  - Better debugging skills
  - Clear understanding of frameworks
  - Stronger API design intuition

Simulating Network Routes

- Not all networking data is easily accessible on every system. For learning purposes, routes can be simulated in memory.
  - Represent routes as dictionaries
  - Assign IDs for CRUD operations
  - Treat simulated data like real resources

Designing Route Objects

- Consistent data structures make APIs predictable and easier to use. Each route should follow the same schema.
  - Include destination, gateway, and interface
  - Add an ID field for identification
  - Validate required fields

POST: Adding a Route

- POST requests create new resources on the server. Servers should validate input before storing data.
  - Parse JSON input
  - Check required fields
  - Return 201 on success

PUT: Updating a Route

- PUT requests replace or update existing resources. Clients must specify which resource to update.
  - Extract ID from the URL
  - Verify the resource exists
  - Return 200 after updating

DELETE: Removing a Route

- DELETE requests permanently remove a resource. Servers should confirm the resource exists before deletion.
  - Check for valid ID
  - Return 204 on success
  - Avoid deleting protected entries

Error Handling Strategy

- Errors should be predictable and consistent across endpoints. Clients rely on status codes more than messages.
  - 400 for bad input
  - 404 for missing resources
  - 403 for forbidden actions

Server Stability

- Uncaught exceptions can crash your API server. Handlers should protect against unexpected failures.
  - Wrap logic in try/except
  - Return 500 on unexpected errors
  - Do not expose stack traces

Testing with curl

- curl is a simple way to test HTTP APIs from the command line. It allows you to send requests exactly like a client would.
  - Use -X to specify HTTP methods
  - Use -H to set headers
  - Use -d to send JSON bodies

Lab 1 Summary

- You built a REST API using only Python's standard library. This required handling routing, parsing, and responses manually.
  - Full control over HTTP behavior
  - No external dependencies
  - Strong foundation for frameworks

POP QUIZ: Lab 1

POP QUIZ

- Which HTTP method is used to update an existing resource?
- A. GET
- B. POST
- C. PUT
- D. DELETE

POP QUIZ

- Which status code indicates a new resource was created?
- A. 200
- B. 201
- C. 204
- D. 400

POP QUIZ

- Which header tells the server how to parse a request body?
- A. Accept
- B. Host
- C. Content-Type
- D. User-Agent

POP QUIZ

- Where does http.server store data in this lab?
- A. Database
- B. Disk
- C. Memory
- D. Cloud service

POP QUIZ

- Which request should NOT change server state?
- A. POST
- B. PUT
- C. GET
- D. DELETE

POP QUIZ ANSWER

- Which HTTP method is used to update an existing resource?
- A. GET
- B. POST
- C. PUT ✓
- D. DELETE

POP QUIZ ANSWER

- Which status code indicates a new resource was created?
- A. 200
- B. 201 ✓
- C. 204
- D. 400

POP QUIZ ANSWER

- Which header tells the server how to parse a request body?
- A. Accept
- B. Host
- C. Content-Type ✓
- D. User-Agent

POP QUIZ ANSWER

- Where does http.server store data in this lab?
- A. Database
- B. Disk
- C. Memory ✓
- D. Cloud service

POP QUIZ ANSWER

- Which request should NOT change server state?
- A. POST
- B. PUT
- C. GET ✓
- D. DELETE

Lab 2: Building a REST API with Flask

Why Use Flask?

- Flask is a lightweight Python web framework designed to simplify HTTP handling. It removes boilerplate while keeping API behavior explicit and readable.
  - Automatic routing based on decorators
  - Built-in request and response objects
  - Common choice for internal services and APIs

Frameworks vs Standard Library

- Frameworks abstract repetitive HTTP logic into reusable components. The underlying REST and HTTP concepts remain the same.
  - Routing is declarative instead of manual
  - Request parsing is handled automatically
  - Error handling is centralized

Creating a Flask Application

- A Flask application starts by creating a Flask object. This object coordinates routing, configuration, and request handling.
  - from flask import Flask
  - app = Flask(__name__)
  - The app object owns routes and settings

Routing with Decorators

- Flask uses decorators to map URLs to Python functions. Each route explicitly defines which HTTP methods it accepts.
  - @app.route('/users', methods=['GET'])
  - @app.get and @app.post shortcuts
  - Routes are evaluated top-down

Request Object

- Flask provides a request object for accessing incoming HTTP data. It exposes headers, query parameters, and JSON bodies.
  - request.args for query parameters
  - request.get_json() for JSON bodies
  - request.headers for metadata

Returning Responses

- Flask converts Python return values into HTTP responses. You control both the response body and the status code.
  - Use jsonify() to return JSON
  - Return (response, status_code) tuples
  - Headers can be customized when needed

Error Handling in Flask

- Flask allows centralized handling of HTTP errors. This keeps endpoint logic clean and consistent.
  - abort() triggers HTTP errors
  - @app.errorhandler defines custom handlers
  - Error responses should still return JSON

Logging Requests

- Flask supports hooks that run before and after each request. These hooks are useful for logging and timing.
  - @app.before_request
  - @app.after_request
  - Access request metadata for auditing

Blueprints

- Blueprints allow Flask applications to be split into modules. They help organize routes as applications grow.
  - Define routes in separate files
  - Register blueprints with the app
  - Encourage clean separation of concerns

Query Parameters

- Query parameters allow clients to filter or modify requests without changing the URL path. They are commonly used for search, filtering, and pagination.
  - Access via request.args
  - Values are always strings
  - Optional parameters should have defaults

Reading Headers

- HTTP headers provide metadata about the client and request. Flask exposes headers through a dictionary-like interface.
  - request.headers.get('User-Agent')
  - Headers may or may not exist
  - Never assume header presence

Client IP Address

- Knowing the client IP can be useful for logging and auditing. Flask exposes this information directly on the request object.
  - Use request.remote_addr
  - May represent a proxy address
  - Accuracy depends on deployment

Custom Status Codes

- Flask allows explicit control over HTTP status codes. This makes API behavior predictable for clients.
  - Return (json, status_code) tuples
  - Avoid encoding status in JSON bodies
  - Let HTTP do its job

Custom Headers

- Servers can include custom headers in responses. Headers can communicate metadata without changing response bodies.
  - Use make_response()
  - Set headers on the response object
  - Avoid leaking sensitive data

Before Request Hooks

- before_request hooks run prior to route execution. They are ideal for logging and validation.
  - Executed on every request
  - No return value continues processing
  - Returning a response short-circuits routes

After Request Hooks

- after_request hooks run after a response is created. They can modify responses globally.
  - Useful for timing and headers
  - Must return the response object
  - Runs even on errors

Logging Request Duration

- Measuring request time helps identify performance issues. Flask hooks make timing straightforward.
  - Store start time in before_request
  - Compute duration in after_request
  - Log method, path, and duration

CORS Overview

- Browsers enforce cross-origin request rules for security. APIs must explicitly allow cross-origin access when needed.
  - CORS applies to browsers only
  - Not enforced by curl or servers
  - Handled via response headers

Enabling CORS in Flask

- Flask does not enable CORS by default. The flask-cors extension simplifies configuration.
  - from flask_cors import CORS
  - CORS(app) enables defaults
  - Restrict origins in production

In-Memory Data in Flask

- Flask applications often start with in-memory data structures for simplicity. This approach is useful for learning and prototyping.
  - Use dictionaries or lists
  - Data resets on restart
  - Not suitable for production

Implementing CRUD with Flask

- Flask routes can be mapped directly to CRUD operations. Each HTTP method corresponds to a specific action.
  - GET reads data
  - POST creates data
  - PUT updates data
  - DELETE removes data

GET Endpoints

- GET routes retrieve data without changing server state. They should always be safe and repeatable.
  - Return JSON responses
  - Never modify data
  - Use status code 200

POST Endpoints

- POST routes accept data from the client to create new resources. Servers must validate input before storing it.
  - Read JSON with request.get_json()
  - Return 201 on success
  - Reject invalid input

PUT Endpoints

- PUT routes update existing resources identified by the URL. Clients must specify which resource to modify.
  - Extract IDs from paths
  - Check resource existence
  - Return updated object

DELETE Endpoints

- DELETE routes remove resources from the server. They should clearly indicate success or failure.
  - Return 204 on success
  - Return 404 if missing
  - Protect critical data

Searching Across Data

- Search endpoints allow clients to query across multiple resources. They are commonly implemented using query parameters.
  - Read search terms from request.args
  - Compare keys and values
  - Case-insensitive matching

Logging for Auditing

- Logging helps track usage and diagnose issues. APIs should log key request metadata.
  - Log method and path
  - Log client IP and User-Agent
  - Avoid logging sensitive data

Lab 2 Challenge Overview

- In this lab, you will build a Flask-based network API. All data will remain in memory.
  - Expose interfaces, MACs, and routes
  - Implement full CRUD
  - Add logging hooks

Lab 2 Summary

- Flask simplifies REST API development while preserving HTTP fundamentals. The skills learned here transfer directly to other frameworks.

  - Cleaner routing
  - Centralized request handling
  - Foundation for async frameworks

POP QUIZ: Lab 2

POP QUIZ

- What Flask feature replaces manual routing logic from http.server?
- A. Middleware
- B. Decorators
- C. Templates
- D. Threads

POP QUIZ

- Which object provides access to headers and query parameters?
- A. response
- B. session
- C. request
- D. app

POP QUIZ

- Where should request timing start in Flask?
- A. after_request
- B. route handler
- C. before_request
- D. errorhandler

POP QUIZ

- Which status code is best for a successful DELETE?
- A. 200
- B. 201
- C. 204
- D. 404

POP QUIZ

- What Flask feature helps split large apps into modules?
- A. Blueprints
- B. Hooks
- C. CORS
- D. Sessions

POP QUIZ ANSWER

- What Flask feature replaces manual routing logic from http.server?
- A. Middleware
- B. Decorators ✓
- C. Templates
- D. Threads

POP QUIZ ANSWER

- Which object provides access to headers and query parameters?
- A. response
- B. session
- C. request ✓
- D. app

POP QUIZ ANSWER

- Where should request timing start in Flask?
- A. after_request
- B. route handler
- C. before_request ✓
- D. errorhandler

POP QUIZ ANSWER

- Which status code is best for a successful DELETE?
- A. 200
- B. 201
- C. 204 ✓
- D. 404

POP QUIZ ANSWER

- What Flask feature helps split large apps into modules?
- A. Blueprints ✓
- B. Hooks
- C. CORS
- D. Sessions