Programming for Automation


TEKsystems
Global Services
WORKFORCE
DEVELOPMENT

API Rate Limiting

- Rate limiting controls how frequently clients can call an API over time. It protects services from abuse while ensuring fair access for all users.
  - Limits requests per client or API key
  - Prevents denial-of-service attacks
  - Protects backend resources

Why Rate Limiting Matters

- Without limits, a single client can overwhelm your API and impact others. Rate limiting enforces fairness and stability.
  - Avoids resource exhaustion
  - Controls unexpected traffic spikes
  - Keeps APIs reliable

HTTP 429 Too Many Requests

- When a client exceeds the allowed request rate, the API should reject the request. HTTP 429 signals the client to slow down.
  - Returned when rate limit is exceeded
  - Client request is valid but not allowed yet
  - Used instead of generic 400 errors

Retry-After Header

- The Retry-After header tells the client how long to wait before retrying. This enables polite, predictable client behavior.
  - Sent with HTTP 429 responses
  - Value is time in seconds
  - Improves client reliability

Rate Limiting Scope

- Rate limits are usually applied per client identity, not globally. API keys are commonly used as the identifier.
  - Per API key
  - Per IP address
  - Per user account

Where Rate Limiting Lives

- Rate limiting can exist at multiple layers of a system. This lab focuses on application-level rate limiting.
  - Application code
  - API gateways
  - Reverse proxies

In-Memory Rate Limiting

- This lab uses in-memory data structures to track request rates. This is simple but limited to a single process.
  - Easy to implement
  - Resets on restart
  - Not shared across servers

Common Algorithms

- Different algorithms balance fairness, burst handling, and memory usage. Two common ones are sliding window and token bucket.
  - Sliding window log
  - Token bucket
  - Fixed window (not used here)

Sliding Window Concept

- Sliding window tracks exact request timestamps within a rolling time window. It provides precise rate enforcement.
  - Tracks recent request times
  - Removes expired timestamps
  - Fair but memory-heavy

Token Bucket Concept

- Token bucket allows controlled bursts while enforcing a long-term average rate. Tokens refill over time.
  - Each request consumes a token
  - Tokens refill gradually
  - Allows short bursts

Client-Side Behavior

- Rate limiting requires cooperation from both server and client. Clients should slow down instead of retrying immediately.
  - Read Retry-After header
  - Wait before retrying
  - Avoid hammering the API

Idempotent Requests

- Retries are safest for requests that do not change server state. GET requests are typically idempotent.
  - GET is safe to retry
  - POST can cause duplicates
  - PUT requires caution

Sliding Window Log

- Sliding window log enforces limits by tracking exact request timestamps. It gives precise control over request rates.
  - Uses timestamps per request
  - Maintains rolling window
  - Accurate but memory-heavy

Sliding Window Tradeoffs

- This approach is fair but can consume memory for busy clients. It is best for low to moderate traffic.
  - Stores many timestamps
  - Per-key tracking
  - Not ideal at massive scale

Token Bucket Model

- Token bucket smooths traffic while allowing short bursts. It is commonly used in production systems.
  - Tokens refill over time
  - Each request consumes a token
  - Allows controlled bursts

Token Bucket Tradeoffs

- Token bucket uses less memory and supports bursts. It is slightly less precise than sliding window.
  - Lower memory usage
  - Burst-friendly
  - Approximate enforcement

Choosing an Algorithm

- The right algorithm depends on traffic patterns and fairness requirements. Both approaches are valid.
  - Sliding window = precision
  - Token bucket = flexibility
  - Gateways often choose token bucket

Where to Enforce Limits

- Rate limiting can be enforced at different layers. This lab demonstrates application-level control.
  - Application code
  - Reverse proxy
  - API gateway

Distributed Systems

- In-memory limits do not work across multiple servers. Production systems require shared state.
  - Redis-backed counters
  - Centralized gateways
  - Consistent enforcement

Lab Focus

- This lab keeps rate limiting simple to focus on concepts. The same ideas apply at larger scale.
  - Per API key limits
  - HTTP 429 responses
  - Retry-After handling

Implementing Sliding Window

- Sliding window uses timestamps to decide whether a request is allowed. Old timestamps are removed before evaluating the limit.
  - Track requests per API key
  - Remove expired timestamps
  - Allow or deny request

Time Sources

- Using a monotonic clock avoids issues with system clock changes. This makes rate limiting more reliable.
  - time.monotonic()
  - Unaffected by NTP changes
  - Safer for timing logic

Retry Calculation

- When the limit is exceeded, the server calculates how long the client must wait. This value becomes Retry-After.
  - Based on oldest request
  - Window size determines delay
  - Communicated to client

Implementing Token Bucket

- Token bucket refills tokens based on elapsed time. Each request consumes a token if available.
  - Track last refill time
  - Refill proportionally
  - Consume per request

Refill Rate

- The refill rate controls how quickly clients recover capacity. Faster refill allows higher sustained traffic.
  - Tokens per second
  - Controls steady-state rate
  - Independent of burst size

Burst Capacity

- Bucket capacity defines how many requests can be made instantly. This supports short bursts without overload.
  - Initial token count
  - Allows sudden spikes
  - Still rate-limited long-term

Flask Integration

- Rate limiting logic runs inside request handlers or hooks. It executes after authentication but before business logic.
  - Validate API key first
  - Apply rate limit check
  - Return early on failure

Returning 429 Responses

- When denying a request, the response must clearly explain why. HTTP 429 is the standard signal.
  - Status code 429
  - JSON error body
  - Retry-After header

Testing Behavior

- Testing rate limits requires sending many requests quickly. Curl loops are effective for this.
  - Rapid request loops
  - Observe status codes
  - Adjust sleep timing

Lab Completion

- At this point, you should understand how rate limiting works. The challenge applies it to your own API.
  - Per-key enforcement
  - Correct HTTP responses
  - Client-friendly behavior

Lab 2: Nginx as a Reverse Proxy

- In this lab, you place Nginx in front of a Flask web application. This mirrors how real production systems expose web apps safely.
  - Nginx faces the internet
  - Flask stays internal
  - Single entry point

What Is a Proxy?

- A proxy is an intermediary between a client and a server. It forwards requests and responses on behalf of another system.
  - Sits between client and server
  - Can modify requests
  - Can modify responses

Forward Proxy

- A forward proxy sits in front of clients. It controls how clients access the internet.
  - Client → Proxy → Internet
  - Controls outbound traffic
  - Common in corporate networks

Reverse Proxy

- A reverse proxy sits in front of servers instead of clients. Clients never talk to the backend directly.
  - Client → Proxy → Backend
  - Backends stay hidden
  - Clients see one endpoint

Why Reverse Proxies Exist

- Reverse proxies solve problems that applications should not handle directly. They centralize infrastructure concerns.
  - TLS termination
  - Security controls
  - Traffic management

Before and After Architecture

- This lab demonstrates how architecture changes when adding a reverse proxy. The backend becomes internal-only.
  - Before: Browser → Flask
  - After: Browser → Nginx → Flask
  - Flask no longer public

Nginx in This Lab

- Nginx will accept HTTPS traffic from the browser. It forwards requests to Flask over HTTP.
  - Listens on port 443
  - Proxies to localhost:5000
  - Handles TLS

Why Flask Should Not Face the Internet

- Flask’s built-in server is for development only. Production traffic should be handled by proper servers.
  - Not hardened for attacks
  - Single-process model
  - No TLS management

Reverse Proxy Responsibilities

- The proxy becomes responsible for many cross-cutting concerns. Applications can stay simple.
  - TLS certificates
  - Logging and headers
  - Rate limiting and auth

Lab Goal

- By the end of this lab, you will access Flask only through Nginx. Direct backend access should no longer be used.
  - HTML served through proxy
  - API served through proxy
  - Backend stays private

Flask Behind a Proxy

- When Flask runs behind Nginx, it no longer handles direct client connections. It becomes an internal service.
  - Listens on localhost only
  - No public exposure
  - Simpler application logic

Plain HTTP for Backend

- The backend Flask app runs over HTTP, not HTTPS. TLS is terminated at the reverse proxy.
  - HTTPS handled by Nginx
  - Flask runs without SSL
  - Clear separation of concerns

Jinja2 Rendering

- Flask renders HTML templates using Jinja2. Nginx simply forwards requests and responses.
  - Templates rendered by Flask
  - HTML returned to proxy
  - Proxy passes content unchanged

Serving HTML Through Nginx

- The browser only communicates with Nginx. HTML pages appear the same as before.
  - Same URLs
  - Same responses
  - Different architecture

Proxying API Requests

- JSON API endpoints are also proxied through Nginx. Frontend and API share the same origin.
  - /api/data via proxy
  - Backend still on :5000
  - Same-origin behavior

Why CORS Is Avoided

- Because Nginx serves both frontend and API from the same origin, browsers do not enforce CORS.
  - Same scheme
  - Same host
  - Same port

Headers Added by Nginx

- Nginx forwards important metadata to the backend using headers.
  Flask can read these headers.
  - X-Forwarded-For
  - X-Real-IP
  - X-Forwarded-Proto

Trusting Proxy Headers

- Backends must trust headers only from known proxies. Blind trust is dangerous.
  - Proxy controls headers
  - Do not trust direct clients
  - Secure internal network

Observing Client IP

- Flask can see the real client IP via forwarded headers. This enables logging and security controls.
  - request.headers
  - X-Forwarded-For
  - Proxy visibility

Lab Progress Check

- At this stage, traffic flows through Nginx to Flask. Direct backend access should be avoided.
  - Proxy path works
  - Backend internal only
  - Ready for production patterns

Nginx Configuration File

- Nginx behavior is defined by configuration files. A server block defines how requests are handled for a domain.
  - server { } block
  - listen and server_name
  - location routing

TLS Configuration

- Nginx handles HTTPS using certificates and private keys. This centralizes TLS management.
  - ssl_certificate
  - ssl_certificate_key
  - Single TLS termination point

proxy_pass Directive

- The proxy_pass directive forwards requests to the backend service. It defines where traffic is sent.
  - Points to backend URL
  - Supports HTTP backends
  - Transparent to client

Preserving Host Header

- Forwarding the Host header keeps backend routing consistent. Some apps rely on it.
  - proxy_set_header Host
  - Supports virtual hosts
  - Avoids surprises

Forwarded IP Headers

- Nginx forwards client IP information using headers. Backends use these for logging and security.
  - X-Forwarded-For
  - X-Real-IP
  - X-Forwarded-Proto

Reloading Nginx Safely

- Configuration changes should be validated before reload. This prevents downtime.
  - nginx -t
  - systemctl reload nginx
  - Zero-downtime reload

Gunicorn and Flask

- Gunicorn is a WSGI server used to run Flask in production. It handles concurrency properly.
  - Multiple worker processes
  - Binds to localhost
  - Designed for production

Why Not Flask Dev Server

- The Flask development server is not production-ready. It lacks security and performance features.
  - Single process
  - No request hardening
  - Debug-only features

Production Request Flow

- In production, traffic flows through multiple layers. Each layer has a clear responsibility.
  - Browser → Nginx
  - Nginx → Gunicorn
  - Gunicorn → Flask app

Lab 2 Completion

- You now understand how reverse proxies fit into real deployments. This mirrors cloud architectures.
  - Backend hidden
  - TLS centralized
  - Scalable pattern

Pop Quiz 1

- What is the primary role of a reverse proxy?
  - A. Hide clients from servers
  - B. Hide servers from clients
  - C. Replace backend apps
  - D. Encrypt databases

Pop Quiz 2

- Where does TLS typically terminate in this lab?
  - A. Flask app
  - B. Browser
  - C. Nginx
  - D. Gunicorn

Pop Quiz 3

- Why does this setup avoid CORS issues?
  - A. Flask disables CORS
  - B. Same-origin requests
  - C. Nginx rewrites headers
  - D. Browser ignores CORS

Pop Quiz 4

- Which header carries the original client IP?
  - A. Host
  - B. X-Forwarded-Proto
  - C. X-Forwarded-For
  - D. User-Agent

Pop Quiz 5

- Why should Flask not face the internet directly?
  - A. Flask is too slow
  - B. Flask lacks TLS support
  - C. Dev server is not production-ready
  - D. Flask cannot handle JSON

Answer 1

- A. Hide clients from servers
- **B. Hide servers from clients**
- C. Replace backend apps
- D. Encrypt databases

Answer 2

– A. Flask app
– B. Browser
– **C. Nginx**
– D. Gunicorn

Answer 3

- A. Flask disables CORS
- **B. Same-origin requests**
- C. Nginx rewrites headers
- D. Browser ignores CORS

Answer 4

- A. Host
- B. X-Forwarded-Proto
- **C. X-Forwarded-For**
- D. User-Agent

Answer 5

- A. Flask is too slow
- B. Flask lacks TLS support
- **C. Dev server is not production-ready**
- D. Flask cannot handle JSON

OAuth Login with GitHub

- OAuth lets users authorize your app without giving you their password. Your Flask app redirects to GitHub and receives a code to complete login.
  - Authorization, not password sharing
  - User can revoke access anytime
  - Common for “Login with …” flows

Why OAuth Instead of Passwords

- With OAuth, your app never stores user passwords for the provider. Access is scoped and controlled by the provider.
  - No password handling
  - Least-privilege permissions
  - Revocable access

OAuth Roles

- OAuth defines four roles so everyone knows who does what. GitHub acts as both the authorization server and the API resource server.
  - Client: your Flask app
  - Resource Owner: the user
  - Authorization Server: GitHub login + token
  - Resource Server: GitHub API data

Authorization Code Flow

- The authorization code flow uses browser redirects to safely obtain an access token. The code is exchanged server-to-server.
  - Redirect user to GitHub
  - GitHub returns with ?code=…
  - Server exchanges code for token

Lab Endpoints

- This lab follows a simple set of routes for login, callback, and protected pages. Sessions store the logged-in user.
  - / (login page)
  - /login (redirect to GitHub)
  - /callback (exchange code for token)
  - /dashboard (protected)

Project Structure

- Keep templates separate from app logic. Your app file controls routes while templates render the pages.
  - app_oauth.py
  - requirements.txt
  - templates/layout.html
  - templates/login.html, dashboard.html

requirements.txt

- Only two packages are needed for this lab. Flask handles routing and sessions, and requests handles outbound HTTP calls.
  - Flask
  - requests
  - Install in a virtualenv

Register a GitHub OAuth App

- You must register an OAuth application so GitHub knows where to redirect users after approval. Callback URL must match exactly.
  - Homepage: http://localhost:8000
  - Callback: http://localhost:8000/callback
  - Copy Client ID and Client Secret

Environment Variables

- Store OAuth secrets in environment variables, not in code. This prevents accidental commits and makes configuration portable.
  - GITHUB_CLIENT_ID
  - GITHUB_CLIENT_SECRET
  - GITHUB_REDIRECT_URI

Templates Overview

- Templates render the UI while Flask handles logic. The login page links to /login and dashboard shows the username.
  - layout.html provides a base layout
  - login.html has “Login with GitHub” link
  - dashboard.html shows username + logout

OAuth Redirect Endpoint

- The login route builds a redirect URL that sends the user to GitHub’s authorization server.
  - Includes client_id and redirect_uri
  - Specifies requested scopes
  - Generates a state value for CSRF protection

OAuth State Parameter

- The state parameter prevents CSRF by binding the auth request to the user session.
  - Generated randomly per login
  - Stored in session before redirect
  - Validated on callback

Authorization Code

- After approval, GitHub redirects back with a short-lived authorization code.
  - Code appears as query param
  - Code can be used only once
  - Expires quickly

Token Exchange

- The server exchanges the authorization code for an access token.
  - POST request to GitHub token endpoint
  - Client secret is used server-side
  - Browser never sees the token

Access Token Purpose

- The access token allows the app to call GitHub APIs on behalf of the user.
  - Sent in Authorization header
  - Limited by granted scope
  - Can be revoked by user

Fetching User Info

- The app uses the access token to fetch the authenticated user profile.
  - GET https://api.github.com/user
  - Returns username and id
  - Used to identify user locally

Creating a Session

- After login, user info is stored in the Flask session.
  - Session cookie sent to browser
  - Used for protected routes
  - No GitHub calls needed after login

Protecting Routes

- Protected routes check for authenticated session data.
  - Decorator pattern used
  - Redirects unauthenticated users
  - Matches real web apps

Logout Flow

- Logging out clears the local session but does not log out of GitHub.
  - Session is deleted server-side
  - User must re-authenticate
  - GitHub token remains valid

OAuth in Real Systems

- OAuth login is commonly used for SSO across many platforms.
  - GitHub, Google, Azure AD
  - Avoids password handling
  - Standardized and widely supported

Pop Quiz 1

- What problem does OAuth primarily solve?
  - A. Faster API responses
  - B. Password sharing between apps
  - C. Secure delegated access
  - D. Session encryption

Pop Quiz 2

- Which role verifies the user and issues tokens?
  - A. Client
  - B. Resource Owner
  - C. Authorization Server
  - D. Resource Server

Pop Quiz 3

- Why is the OAuth state parameter used?
  - A. Rate limiting
  - B. CSRF protection
  - C. Token encryption
  - D. Session expiration

Pop Quiz 4

- Which step exchanges the code for an access token?
  - A. /login redirect
  - B. Browser callback
  - C. Server-to-server POST
  - D. Dashboard load

Pop Quiz 5

- Where should OAuth access tokens be handled?
  - A. In browser JavaScript
  - B. In URL query strings
  - C. Server-side only
  - D. Local storage

Answer 1

- A. Faster API responses
- B. Password sharing between apps
- **C. Secure delegated access**
- D. Session encryption

Answer 2

- A. Client
- B. Resource Owner
- **C. Authorization Server**
- D. Resource Server

Answer 3

- A. Rate limiting
- **B. CSRF protection**
- C. Token encryption
- D. Session expiration

Answer 4

- A. /login redirect
- B. Browser callback
- **C. Server-to-server POST**
- D. Dashboard load

Answer 5

- A. In browser JavaScript
- B. In URL query strings
- **C. Server-side only**
- D. Local storage