Git Provider Integration Guide

Comprehensive guide for integrating with Git providers (GitHub, GitLab, Bitbucket, Azure DevOps) using **OAuth 2.0** and **Personal Access Tokens (PAT)**.

Table of Contents

OAuth Integration

Overview

OAuth 2.0 enables secure, delegated access to Git resources without sharing credentials. Users authorize your app with scoped, time-limited tokens.

OAuth Flow:

1. Redirect user to provider's authorization page
2. User authenticates and grants permissions
3. Provider redirects back with authorization code
4. Exchange code for access token
5. Use token for API requests

When to Use

Use OAuth	Use PAT Instead
☑ User-facing web apps	❌ CI/CD pipelines
☑ Browser-based auth	❌ Server automation
☑ Multi-user platforms	❌ Personal scripts
☑ User-specific actions	❌ Backend services

Required Components

Component	Description
Client ID	Public identifier (can be exposed)
Client Secret	Private key (never expose to frontend)

Component	Description
Redirect URI	Callback URL after authentication
Scopes	Permissions (e.g., `repo`, `read:user`)

Provider Setup

Provider	Setup Location	Key Scopes
GitHub	Settings → Developer Settings → OAuth Apps	`repo`, `read:user`
GitLab	User Settings → Applications	`api`, `read_repository`, `write_repository`
Bitbucket	Personal Settings → OAuth Consumers	`repository`, `repository:write`
Azure DevOps	Azure Portal → Azure AD → App Registrations	`vso.code_write`, `vso.project`

Environment Variables

```
# GitHub
GITHUB_CLIENT_ID=Iv1.abc123def456
GITHUB_CLIENT_SECRET=secret123abc456def789
GITHUB_REDIRECT_URI=http://localhost:8000/auth/github/callback
GITHUB_OAUTH_SCOPES=repo read:user

# GitLab
GITLAB_CLIENT_ID=abc123
GITLAB_CLIENT_SECRET=xyz789
GITLAB_REDIRECT_URI=http://localhost:8000/auth/gitlab/callback
GITLAB_SCOPES=api read_user

# Bitbucket
BITBUCKET_CLIENT_ID=AbCdEf123
BITBUCKET_CLIENT_SECRET=xyz789
BITBUCKET_REDIRECT_URI=http://localhost:8000/auth/bitbucket/callback

# Azure DevOps
AZURE_CLIENT_ID=abc-123-def-456
AZURE_TENANT_ID=xyz-789
AZURE_CLIENT_SECRET=secret~123
AZURE_REDIRECT_URI=http://localhost:8000/auth/azure/callback
```

PAT Integration

Overview

Personal Access Tokens are user-generated credentials that act as password substitutes for API and Git operations over HTTPS.

How it Works:

1. User generates PAT from provider settings
2. User selects permissions/scopes
3. Application uses PAT in API requests
4. Provider authenticates based on PAT

When to Use

☑ Ideal for:

- CI/CD pipelines
- Backend automation
- Server-to-server operations
- Scripts without user interaction
- Headless environments

❌ Not suitable for:

- Browser-based applications (security risk)
- Multi-user platforms (single-user scope)
- Scenarios requiring user consent flows

Generating PATs

Provider	Location	Token Prefix	Scopes
GitHub	Settings → Developer Settings → Personal Access Tokens	`ghp_`	`repo`, `workflow`, `admin:org`
GitLab	Preferences → Access Tokens	`glpat-`	`api`, `read_repository`, `write_repository`
Bitbucket	Personal Settings → App Passwords	(none)	Repository: read/write
Azure DevOps	Profile → Personal Access Tokens	(none)	Code: Read & Write

Environment Variables

```
# GitHub
GITHUB_USERNAME=your_username
GITHUB_PAT=ghp_abc123def456ghi789

# GitLab
GITLAB_USERNAME=your_username
GITLAB_PAT=glpat-xyz123abc456
```

```
GITLAB_BASE_URL=https://gitlab.com

# Bitbucket
BITBUCKET_USERNAME=your_username
BITBUCKET_APP_PASSWORD=abc123xyz789

# Azure DevOps
AZURE_DEVOPS_ORG=my-org
AZURE_DEVOPS_PROJECT=my-project
AZURE_DEVOPS_PAT=abc123def456
```

OAuth vs PAT Comparison

Feature	OAuth	PAT
Creation	App-initiated browser flow	User manually generates
Security	High (scoped, short-lived)	Medium (long-lived)
Use Case	User-facing apps	Automation, scripts
Expiration	Short-lived	Often doesn't expire
Multi-User	Excellent	Poor (one per user)
Setup	Moderate complexity	Minimal
Best For	Web/mobile apps	CI/CD, servers

Implementation Examples

OAuth Implementation (FastAPI + GitHub)

```python
from fastapi import FastAPI, Depends, HTTPException
from fastapi.responses import RedirectResponse
from pydantic import BaseSettings
import httpx
from urllib.parse import urlencode
import secrets

app = FastAPI()

class Settings(BaseSettings):
    GITHUB_CLIENT_ID: str
    GITHUB_CLIENT_SECRET: str
    GITHUB_REDIRECT_URI: str
    GITHUB_OAUTH_SCOPES: str = "repo read:user"
    GITHUB_AUTHORIZE_URL: str = "https://github.com/login/oauth/authorize"
    GITHUB_TOKEN_URL: str = "https://github.com/login/oauth/access_token"
    GITHUB_API_BASE: str = "https://api.github.com"
```

```
    class Config:
        env_file = ".env"

def get_settings():
    return Settings()

# In-memory state store (use Redis in production)
oauth_states = {}

@app.get("/auth/github/login")
def github_login(settings: Settings = Depends(get_settings)):
    """Redirect user to GitHub for authorization"""
    state = secrets.token_urlsafe(32)
    oauth_states[state] = True  # Store for CSRF protection

    params = {
        "client_id": settings.GITHUB_CLIENT_ID,
        "redirect_uri": settings.GITHUB_REDIRECT_URI,
        "scope": settings.GITHUB_OAUTH_SCOPES,
        "state": state,
    }

    auth_url = f"{settings.GITHUB_AUTHORIZE_URL}?{urlencode(params)}"
    return RedirectResponse(auth_url)

@app.get("/auth/github/callback")
async def github_callback(
    code: str,
    state: str,
    settings: Settings = Depends(get_settings)
):
    """Handle GitHub callback and exchange code for token"""
    # Validate state (CSRF protection)
    if state not in oauth_states:
        raise HTTPException(status_code=400, detail="Invalid state")
    del oauth_states[state]

    # Exchange code for access token
    async with httpx.AsyncClient() as client:
        token_response = await client.post(
            settings.GITHUB_TOKEN_URL,
            headers={"Accept": "application/json"},
            data={
                "client_id": settings.GITHUB_CLIENT_ID,
                "client_secret": settings.GITHUB_CLIENT_SECRET,
                "code": code,
                "redirect_uri": settings.GITHUB_REDIRECT_URI,
            },
        )

    if token_response.status_code != 200:
        raise HTTPException(500, "Failed to get access token")

    token_data = token_response.json()
```

```
    access_token = token_data.get("access_token")

    # Fetch user info
    async with httpx.AsyncClient() as client:
        user_response = await client.get(
            f"{settings.GITHUB_API_BASE}/user",
            headers={"Authorization": f"Bearer {access_token}"},
        )

    if user_response.status_code != 200:
        raise HTTPException(500, "Failed to fetch user")

    user_data = user_response.json()

    # TODO: Store token securely, create session
    return {
        "status": "success",
        "user": {
            "id": user_data["id"],
            "login": user_data["login"],
            "name": user_data.get("name"),
        },
        "access_token": access_token,  # Don't return in production!
    }
```

PAT Implementation (FastAPI + GitHub)

```
from fastapi import FastAPI, HTTPException, Depends
from pydantic import BaseSettings, BaseModel
import httpx
import base64

app = FastAPI()

class Settings(BaseSettings):
    GITHUB_USERNAME: str
    GITHUB_PAT: str
    GITHUB_API_BASE: str = "https://api.github.com"

    class Config:
        env_file = ".env"

def get_settings():
    return Settings()

class FileContent(BaseModel):
    repo_name: str
    file_path: str
    content: str
    commit_message: str = "Update from API"
```

```
async def github_request(method: str, url: str, settings: Settings,
json_data=None):
    """Make authenticated GitHub API request"""
    headers = {
        "Authorization": f"Bearer {settings.GITHUB_PAT}",
        "Accept": "application/vnd.github+json",
    }
    async with httpx.AsyncClient() as client:
        return await client.request(method, url, headers=headers, json=json_data)

@app.post("/repos/create")
async def create_repo(
    name: str,
    private: bool = False,
    settings: Settings = Depends(get_settings)
):
    """Create a new repository"""
    url = f"{settings.GITHUB_API_BASE}/user/repos"
    response = await github_request(
        "POST", url, settings,
        json_data={"name": name, "private": private, "auto_init": True}
    )

    if response.status_code not in (201, 202):
        raise HTTPException(500, f"Failed to create repo: {response.text}")

    repo_data = response.json()
    return {
        "status": "success",
        "repo": {
            "name": repo_data["name"],
            "url": repo_data["html_url"],
            "clone_url": repo_data["clone_url"],
        }
    }

@app.post("/repos/publish")
async def publish_file(
    file: FileContent,
    settings: Settings = Depends(get_settings)
):
    """Create or update a file in repository"""
    # Ensure repo exists
    repo_url = f"
{settings.GITHUB_API_BASE}/repos/{settings.GITHUB_USERNAME}/{file.repo_name}"
    repo_check = await github_request("GET", repo_url, settings)

    if repo_check.status_code == 404:
        # Create repo
        create_resp = await github_request(
            "POST",
            f"{settings.GITHUB_API_BASE}/user/repos",
            settings,
            json_data={"name": file.repo_name, "private": False, "auto_init":
```

```
True}
        )
        if create_resp.status_code not in (201, 202):
            raise HTTPException(500, "Failed to create repository")

    # Prepare file content (base64 encoded)
    content_b64 = base64.b64encode(file.content.encode()).decode()
    file_url = f"{repo_url}/contents/{file.file_path}"

    # Check if file exists (need SHA for updates)
    existing = await github_request("GET", file_url, settings)
    data = {"message": file.commit_message, "content": content_b64}

    if existing.status_code == 200:
        data["sha"] = existing.json()["sha"]

    # Create or update file
    response = await github_request("PUT", file_url, settings, json_data=data)

    if response.status_code not in (200, 201):
        raise HTTPException(500, "Failed to update file")

    result = response.json()
    return {
        "status": "success",
        "file": {
            "path": file.file_path,
            "url": result["content"]["html_url"],
            "commit_sha": result["commit"]["sha"],
        }
    }

@app.get("/repos")
async def list_repos(settings: Settings = Depends(get_settings)):
    """List user repositories"""
    url = f"{settings.GITHUB_API_BASE}/user/repos"
    response = await github_request("GET", url, settings)

    if response.status_code != 200:
        raise HTTPException(500, "Failed to fetch repos")

    repos = response.json()
    return {
        "count": len(repos),
        "repositories": [
            {
                "name": r["name"],
                "private": r["private"],
                "url": r["html_url"],
            }
            for r in repos
        ]
    }
```

Testing

```
# Install dependencies
pip install fastapi uvicorn httpx pydantic python-dotenv

# Create .env file
echo "GITHUB_CLIENT_ID=your_client_id" > .env
echo "GITHUB_PAT=your_pat_here" >> .env

# Run server
uvicorn main:app --reload

# Test OAuth
# Visit: http://localhost:8000/auth/github/login

# Test PAT - Create repo
curl -X POST "http://localhost:8000/repos/create?name=test-repo"

# Test PAT - Publish file
curl -X POST "http://localhost:8000/repos/publish" \
  -H "Content-Type: application/json" \
  -d '{
    "repo_name": "test-repo",
    "file_path": "README.md",
    "content": "# Hello World",
    "commit_message": "Initial commit"
  }'
```

Security Best Practices

Never Expose Tokens

❌ Don't:

```
# Hardcode tokens
TOKEN = "ghp_abc123..."

# Log tokens
print(f"Token: {token}")

# Return tokens to frontend
return {"token": access_token}
```

☑ Do:

```
# Use environment variables
import os
TOKEN = os.getenv("GITHUB_PAT")

# Mask in logs
print(f"Token: {token[:8]}...")

# Store server-side only
# Return session ID instead
```

Encrypt Token Storage

```
from cryptography.fernet import Fernet
import os

ENCRYPTION_KEY = os.getenv("TOKEN_ENCRYPTION_KEY")
cipher = Fernet(ENCRYPTION_KEY)

def encrypt_token(token: str) -> str:
    return cipher.encrypt(token.encode()).decode()

def decrypt_token(encrypted: str) -> str:
    return cipher.decrypt(encrypted.encode()).decode()
```

CSRF Protection (OAuth)

```
import secrets

# Generate and store state
state = secrets.token_urlsafe(32)
redis_client.setex(f"oauth:{state}", 600, "true")

# Validate in callback
if not redis_client.exists(f"oauth:{state}"):
    raise HTTPException(400, "Invalid state")
redis_client.delete(f"oauth:{state}")
```

Input Validation

```
from pydantic import BaseModel, validator

class FileContent(BaseModel):
    repo_name: str
    file_path: str

    @validator('repo_name')
```

```
    def validate_repo(cls, v):
        if not v.replace('-', '').replace('_', '').isalnum():
            raise ValueError('Invalid repo name')
        return v

    @validator('file_path')
    def validate_path(cls, v):
        if '..' in v or v.startswith('/'):
            raise ValueError('Invalid file path')
        return v
```

Security Checklist

- [ ] Never commit secrets to Git
- [ ] Use `.env` files (add to `.gitignore`)
- [ ] Encrypt tokens at rest and in transit
- [ ] Use HTTPS in production
- [ ] Implement rate limiting
- [ ] Rotate PATs every 90 days
- [ ] Use minimal scopes/permissions
- [ ] Monitor and audit token usage
- [ ] Have a token leak response plan

Troubleshooting

Common Issues

Issue	Cause	Solution
Invalid redirect_uri	URI mismatch	Ensure exact match with registered URI
Invalid state	CSRF validation failed	Check state storage/expiration
401 Unauthorized	Invalid/expired token	Regenerate PAT or refresh OAuth token
403 Forbidden	Insufficient permissions	Add required scopes to PAT/OAuth
Rate limit exceeded	Too many requests	Implement rate limiting, check headers

Token Leaked

If a token is exposed:

1. **Immediately revoke** the token from provider
2. **Generate new** token
3. **Update** environment variables/secrets
4. **Remove** from Git history if committed
5. **Review** logs for unauthorized usage

Resources

- GitHub OAuth Docs
- GitLab OAuth Docs
- Bitbucket OAuth Docs
- Azure DevOps OAuth Docs

Summary

Choose OAuth for: User-facing web apps requiring individual user authorization **Choose PAT for:** Backend automation, CI/CD, and scripts without user interaction

Both methods provide secure access to Git providers when implemented correctly with proper security measures.