14. 보안과 접근 제어

GitHub 공개/비공개 저장소 설정

1️⃣ 저장소 생성 시 공개/비공개 선택

새 저장소 만들기 (`New repository`)

- 저장소 생성 단계에서 선택 가능:

```
☑ Public (공개)
☑ Private (비공개)
```

유형	기본 설명
Public	누구나 GitHub에서 볼 수 있음 (GitHub 검색, 직접 URL 접근 가능)
Private	초대한 Collaborator만 접근 가능 (기본 비공개, 검색 안 됨)

2️⃣ 기존 저장소 공개/비공개 전환

경로

`Repository` → `Settings` → `General` → `Danger Zone` → **Change repository visibility**

- Public → Private 전환 가능
- Private → Public 전환 가능

주의:

- Public으로 변경 시 → 전체 공개됨 → Google 검색 등 노출 가능
- Private으로 변경 시 → 기존 Fork한 Public 저장소는 그대로 남음 (Fork는 복사본이므로 원본만 Private 해도 Fork는 Public일 수 있음)

3️⃣ 권한 차이

항목	Public 저장소	Private 저장소
GitHub 검색 노출	O	X
누구나 Clone 가능	O	X (권한자만 가능)
GitHub Pages 사용	O	O (Private 저장소도 가능, 단 유료 플랜 필요할 수 있음)
GitHub Actions 사용	O	O

항목	Public 저장소	Private 저장소
Pull Request 외부 Contributor 허용	O	O (권한 설정 필요)
무료 사용 가능 여부	개인/조직 모두 가능	무료 플랜에서도 사용 가능 (유료 플랜은 더 많은 기능 제공)

4️⃣ Private 저장소 Collaborator 추가

경로:

`Repository` → `Settings` → `Collaborators and teams` → **Add Collaborator**

- GitHub 사용자명을 입력하여 초대 가능
- 초대받은 사용자는 해당 저장소에 접근 가능 (clone/push/pull 가능)

5️⃣ 조직(Organization) 저장소

조직에서 생성하는 경우:

```
☑ Public
☑ Private
☑ Internal (Enterprise 전용, 조직 내부에만 공개)
```

→ Internal 은 Enterprise 플랜에서만 사용 가능 → **조직 내부 공유용**

6️⃣ GitHub API 연동 시 유의 사항

- Private 저장소는 API 접근 시 반드시 **Authentication 필요**
  - Personal Access Token 사용 → `repo` scope 포함 필요

```
curl -H "Authorization: token <TOKEN>" https://api.github.com/repos/owner/repo
```

→ Public 저장소는 인증 없이 API로 읽기 가능

7️⃣ 실전 활용 패턴

상황	추천 설정
오픈소스 프로젝트	Public
개인 포트폴리오	Public (단, 민감 정보 없는 경우)
회사/팀 내부 개발	Private

상황	추천 설정
개인 실습용 테스트	Private
상용 서비스 코드	Private 필수

---

8 주의사항

- Private 저장소에 **AWS 키, DB 비밀번호 등 민감 정보**라도 실수로 commit 금지
  - GitHub 직원, 사고 대응 팀은 기술적으로 Private 저장소 접근 가능 (지원 요청 시 등)
  - Private 저장소 → 내부 보안 규정 준수 필요
- 공개 저장소로 전환 시 **민감 정보가 이미 올라가 있으면 복구 불가** → 반드시 전환 전 점검

---

9 정리

설정 항목	위치	특징
Public/Private 설정	Repository Settings → General → Danger Zone	즉시 변경 가능
Collaborator 추가	Repository Settings → Collaborators	Private 저장소에서 접근 권한 관리
조직 저장소 유형	Public / Private / Internal	조직 전용 기능 (Internal은 Enterprise만)
GitHub Actions 사용 여부	Public/Private 모두 가능	Private 저장소에서도 사용 가능 (유료 플랜 일부 제한 주의)

결론:

- **Public**: 오픈소스, 포트폴리오
- **Private**: 내부 개발, 민감 코드, 상용 서비스
- → 설정은 언제든지 변경 가능하지만 **변경 전/후 보안 점검 필수**

Collaborator 권한 관리

1 Collaborator란?

- **Repository 단위**로 외부 사용자를 초대해서 협업할 수 있게 해주는 기능
- 초대받은 사용자는 **Repository 접근 + 권한 수준에 따른 기능 사용 가능**
- → Organization 구성원이 아니어도 Collaborator로 초대 가능

주요 용도

✅ 회사 외부 Freelancer, 외부 Vendor 참여
✅ 개인 Public 저장소에 공동 Maintainer 추가
✅ Private 저장소에 공동 개발자 초대

2️⃣ Collaborator 추가 방법

경로

```
Repository → Settings → Collaborators and teams → Add collaborator
```

절차

1️⃣ GitHub 사용자명 입력
2️⃣ 권한 수준 선택
3️⃣ 초대 → 상대방이 이메일 또는 GitHub 알림으로 수락해야 적용됨

3️⃣ 권한 수준 (Repository Collaborator 기준)

권한 수준	설명
Read	코드 clone + Issues 보기 가능, PR 생성 가능 (Fork 기반), 코드 수정/푸시는 불가
Triage	Issue/PR 라벨링, close, reopen 가능, merge 불가, 코드 푸시는 불가
Write	코드 clone + push 가능, PR merge 가능, Issues 관리 가능
Maintain	관리 권한(릴리즈 생성, Secrets 관리 등), Settings은 수정 불가, 코드 푸시는 가능
Admin	Repository 설정 전체 수정 가능 (Visibility, Webhook, Secrets 등 포함), Collaborator 관리 가능

→ 일반적인 개발 협업에서는 보통:

- **개발자** → Write 권한
- **QA, PO** → Read or Triage 권한
- **리드 개발자 / 운영자** → Maintain or Admin 권한

4️⃣ Public 저장소에서 외부 기여

- Public 저장소는 기본적으로 **Fork** → **PR** 방식으로 외부 기여
- **Collaborator 추가**하면 → 원본 저장소에 직접 Push/PR 가능
- 그렇지 않으면 PR만 보내고 리뷰 + merge는 Maintainer가 수행

---

5 Private 저장소에서 Collaborator 역할

- Private 저장소는 기본적으로 **Collaborator 추가를 하지 않으면 접근 자체 불가** (Clone도 안 됨)
- Collaborator로 추가하면 설정된 권한 수준에 따라 Push, Issue 관리, Settings 접근 등이 가능해짐

→ **Collaborator 수를 최소화하고 정확한 권한 관리가 매우 중요**

---

6 주의사항

- Collaborator는 Repository 단위로 설정됨 → Organization Member는 Organization 단위 정책이 따름 (Team 사용 가능)
- Collaborator는 Organization 관리자가 아닌 Repository Admin 권한자가 직접 추가 가능
- Collaborator 초대 시 Organization 정책(SSO 등)에 따라 제한 있을 수 있음
- **비정기적 Collaborator 권한 점검 권장** → 프로젝트 종료 후에도 그대로 남아있는 경우가 많음

---

7 실전 패턴

상황	권장 권한
외부 개발자 참여 (코드 개발)	Write
외부 QA/PM 참여	Read or Triage
Maintainer 공동 관리	Maintain
보안/운영 책임자	Admin

---

8 관리 Best Practice

- Collaborator 추가 시 반드시 **권한 최소화 원칙 적용**
- 프로젝트 종료 시 Collaborator 주기적 검토/삭제
- 민감 Private 저장소는 Write 권한 이상 부여 시 반드시 담당자 확인 후 승인 절차 적용
- **GitHub Audit Log (유료 플랜)** → 누가 언제 Collaborator로 추가/삭제했는지 추적 가능

---

9 조직(Organization)과의 차이점

항목	Collaborator	Organization Member / Team
적용 단위	Repository 단위	Organization 단위
권한 설정	개별 사용자별 설정	Team 단위 설정 가능
관리 주체	Repository Admin	Organization Owner/Admin
권장 사용 사례	개인 저장소, 외부 Contributor	조직 내 프로젝트 관리

결론

Collaborator 권한 관리는 협업 품질과 보안 모두에 영향을 주는 핵심 관리 포인트다.

- ✅ Private 저장소는 Collaborator가 없으면 접근 자체 불가
- ✅ Public 저장소는 기본 Fork PR → Collaborator 추가 시 직접 Push 가능
- ✅ 권한 수준을 정확히 이해하고 최소 권한 원칙을 지키는 것이 매우 중요

GitHub Team, Organization 관리

1️⃣ 기본 개념

개념	설명
Organization	여러 개발자(User)를 하나의 조직으로 묶어 관리
Team	Organization 내부에서 역할별/프로젝트별 그룹 구성
Member	Organization에 속한 사용자
Outside Collaborator	Organization에 소속되지 않고 특정 Repository만 권한 부여받은 외부 사용자

→ 개인 User 계정으로 관리하면 권한/구성원/보안 관리가 어려움 → Organization 사용 추천

2️⃣ Organization 만들기

절차

1. GitHub → Profile → Your organizations → New organization
2. 플랜 선택 (Free, Team, Enterprise)
3. Organization 이름 / Billing 설정
4. 구성원 초대

주요 특징

- ✅ Organization 별로 Repository 관리 가능
- ✅ Organization-level Settings 존재
- ✅ Team 기능 사용 가능
- ✅ Billing (유료 플랜 시) Organization 단위로 과금됨

3️⃣ Team 구성

구조

```
Organization
 ├── Team A
 |    ├── User 1
 |    ├── User 2
 ├── Team B
```

```
|    ├── User 3
|    ├── User 4
```

Team 특징

✅ 역할/부서별 관리 가능 (ex: Backend Team, Frontend Team, DevOps Team 등)
✅ Repository 단위 권한을 Team 단위로 설정 가능 → **권한 관리가 매우 편리해짐**
✅ Team 단위로 Mention 가능 (`@team-name`) → Issue/PR에서 사용 가능

Team 관리 경로

```
Organization → Teams → New team
```

Team 내부 설정

- Parent Team 설정 가능 (계층 구조 지원)
- Team Maintainer 지정 가능 (Team 관리 전담자)

4️⃣ 권한 구조

권한 단위

계층	권한 적용 대상
Organization	Member 관리, Organization Settings
Team	Repository 권한 부여, 구성원 구성
Repository	Team 또는 User에 대해 Read/Write/Admin 권한 부여

Repository 권한 수준 (Team or User 적용 동일)

권한 수준	설명
Read	코드 clone, Issue/PR 보기
Triage	Issue/PR 관리(라벨링 등), merge는 불가
Write	Push 가능, PR merge 가능
Maintain	릴리즈 생성, 설정 일부 가능
Admin	Settings 수정, Team/Collaborator 관리 가능

5️⃣ Outside Collaborator

- Organization 내부 Member가 아닌 외부 사용자
- 특정 Repository 단위로만 권한 부여
- 사용 예:
  - 외부 Freelancer
  - Vendor
  - 파트타임 Contributor

관리 주의사항

✅ 최소한 권한만 부여 (주로 Read 또는 Write)
✅ 주기적 검토 및 제거 필요 → 보안 사고 예방
✅ Organization-level Audit Log에서 활동 추적 가능 (Enterprise 플랜)

6️⃣ 관리 Best Practice

조직 관리

- 구성원 초대 시 반드시 Team 할당 → 관리 용이
- Personal Repo → Organization Repo로 이전 권장
- Organization Secrets 활용 (Secrets scope 설정 가능)

권한 관리

- Team 단위로 Repository 권한 관리 → User 직접 권한 설정 최소화
- Outside Collaborator는 필수 Repository에만 적용
- Public Repository의 경우 Contributor 정책 명확화

보안 관리

- Organization level에서 **2FA 필수 설정 강력 추천**
- SSO (Enterprise 플랜) 사용 시 Organization 강제 적용 가능
- Audit Log 주기적 확인 → 권한 변경/민감 이벤트 추적

7️⃣ 실전 예시

Backend / Frontend / DevOps Team 구성 예시

```
Organization: my-company

Teams:
- backend-team
- frontend-team
- devops-team

```

```
Repositories:
- backend-api
  - backend-team: Write
  - devops-team: Read
- frontend-app
  - frontend-team: Write
  - devops-team: Read
- infrastructure
  - devops-team: Admin
  - backend-team: Read
  - frontend-team: Read
```

관리 포인트

✅ 신규 Repository 생성 시 → 자동으로 Team 권한 부여 적용
✅ 신규 구성원 추가 시 → Team 할당 → 자동으로 권한 부여
✅ Outside Collaborator 사용 시 최소한 권한만 부여하고 종료 시 삭제

8️⃣ 정리

항목	기능
Organization	전체 협업 구조 단위, 구성원/Team 관리
Team	역할별 구성원 그룹, Repository 권한 단위로 적용
Member	Organization 내부 구성원 (직원, 정식 Contributor)
Outside Collaborator	Organization 외부 사용자의 특정 Repository 권한 설정
권한 관리 Best Practice	Team 단위로 관리, User 직접 권한 최소화, 2FA 필수 설정

결론:

- 개인 프로젝트 수준 → 개인 User + Collaborator 관리
- 회사/팀 개발 → 반드시 Organization 기반 구성
- **Team 중심 권한 관리 + Outside Collaborator 최소화 + 2FA 적용** → 안전하고 체계적인 협업 가능

GPG 서명 커밋 (Verified 배지)

1️⃣ GPG 서명이란?

- **커밋이나 태그에 디지털 서명을 추가**하는 기능
- 본인이 개인 키로 서명 → 다른 사람이 공개 키로 검증 가능
- GitHub에서는 서명된 커밋에 대해 **"Verified" 배지**를 표시해줌

Verified 배지

✅ `Verified` 표시가 있으면:

- **커밋을 한 사람이 진짜 그 사람(해당 키의 소유자)임을 증명**
- 커밋 내용이 변경되지 않았음을 보장

2️⃣ 왜 사용하는가?

이유	설명
보안	커밋 위조 방지 (예: 탈취된 계정에서 임의 커밋 생성 방지)
신뢰성	릴리즈 태그에 반드시 서명 추가 → 릴리즈 진본 여부 검증 가능
규정 준수	기업/오픈소스 프로젝트에서 필수 적용 요구 증가 중

3️⃣ GPG 키 생성 및 등록

1️⃣ GPG 키 생성

```
gpg --full-generate-key
```

- RSA and RSA (default)
- Key size: 4096 추천
- Expiration: 원하는 기간 설정 가능
- Name, Email 입력 → GitHub 계정 Email과 일치해야 Verified 적용 가능

2️⃣ GPG 공개 키 확인

```
gpg --list-keys --keyid-format LONG
```

```
gpg --armor --export <YOUR_KEY_ID>
```

→ 출력된 내용을 GitHub에 등록

3️⃣ GitHub 등록

`Settings` → `SSH and GPG keys` → `New GPG key` → 공개 키 붙여넣기

4️⃣ Git 커밋 서명 설정

기본 설정

```
git config --global user.signingkey <YOUR_KEY_ID>
git config --global commit.gpgsign true
```

→ 이후 커밋 시 자동으로 GPG 서명 추가됨

커밋 시 수동으로 추가도 가능

```
git commit -S -m "My signed commit"
```

`-S` 옵션 → GPG 서명 추가

5️⃣ 사용 예시

```
* a1b2c3d4 (HEAD -> main) Verified My signed commit
```

→ GitHub UI에서도 해당 커밋에 "Verified" 배지가 표시됨

6️⃣ 주의사항

- GitHub는 **GPG 키의 Email 주소가 GitHub 계정에 등록된 Email과 일치해야 Verified 표시** 가능
- GPG 키는 주기적으로 만료 후 재발급/갱신 필요
- GPG Agent 사용 시 패스프레이즈 자동 캐싱 가능

```
gpgconf --launch gpg-agent
```

7️⃣ 태그 서명도 가능

릴리즈 태그에 반드시 서명 추가 권장:

```
git tag -s v1.0.0 -m "Release v1.0.0"
```

검증:

```
git tag -v v1.0.0
```

→ 릴리즈 태그가 Verified 표시됨 → 릴리즈의 신뢰도 매우 높아짐

8️⃣ 회사/팀에서 적용 권장 사례

대상	권장 정책
릴리즈 태그	반드시 서명 필수 (GPG / S/MIME)
main 브랜치 커밋	관리자/Lead는 서명 커밋 권장
외부 PR	PR 커밋 서명 여부 체크 (정책 적용 가능)
Actions Workflow	자동 커밋 시 GitHub Actions Bot 서명 적용 가능 (CI용 Key 별도 등록 가능)

9️⃣ GPG 외 대안

방식	지원 여부	특징
GPG	GitHub 정식 지원	개인적으로 가장 널리 사용됨
S/MIME	GitHub Enterprise 에서도 지원	기업 내부 인증서 사용 가능
SSH Key 서명 (Git 2.34+)	최신 Git 지원	GitHub에서도 지원 (2023 이후)

→ 최근 GitHub는 **SSH Key 기반 Commit 서명도 지원 시작**
→ GPG Key, SSH Key, S/MIME 중 하나만 사용해도 충분

정리

항목	내용
Verified 배지 의미	커밋 또는 태그가 진본이고 위변조되지 않았음을 보장
GPG 키 필요 여부	필요 (개인 키 생성 후 공개 키 GitHub에 등록)
커밋 서명 활성화 방법	`git config --global commit.gpgsign true`
태그 서명 방법	`git tag -s` 사용
기업/팀 적용 시	태그 필수 서명, main 브랜치 커밋 서명 권장

결론:

- GPG 서명 커밋 → 프로젝트 보안 신뢰성 매우 향상됨
- 특히 **릴리즈 태그는 반드시 서명 추가**하는 게 좋은 습관
- 개인/기업/오픈소스 프로젝트 모두 적극 적용 추천

SSH 키와 Personal Access Token 관리

1️⃣ SSH 키

개념

- **SSH 키 쌍(Public Key, Private Key)** 를 이용해 GitHub에 안전하게 인증 → clone, push, pull 가능
- 비밀번호 대신 사용하는 안전한 인증 방식
- SSH 키는 **Git 작업(Repository 접근)** 용도로 주로 사용됨

SSH 키 생성

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

- ED25519가 최신 알고리즘 (빠르고 안전)
- 생성 시 `~/.ssh/id_ed25519` (Private), `~/.ssh/id_ed25519.pub` (Public)

GitHub에 등록

1️⃣ GitHub → `Settings` → `SSH and GPG keys` → `New SSH key`
2️⃣ `id_ed25519.pub` 내용 붙여넣기 → 저장

사용 시 동작

```
git clone git@github.com:username/repo.git
git push origin main
```

→ SSH 키로 인증 → **패스워드 입력 불필요**

SSH 키 관리 Best Practice

✅ 각 PC/서버 별로 SSH 키 쌍 별도 생성 → 동일한 Private Key 여러 기기 공유 금지
✅ Private Key는 **절대 외부 노출 금지** (`id_ed25519` 자체 절대 공유 X)
✅ GitHub에서 사용하지 않는 SSH 키는 반드시 삭제
✅ SSH Key 만료 관리 (주기적 교체 권장)

2️⃣ Personal Access Token (PAT)

개념

- GitHub API, Git 작업(HTTPS 방식), GitHub Actions, 외부 툴 연동 시 사용하는 **Access Token**
- 기존에는 GitHub 계정 비밀번호 사용 가능했지만 → 현재는 **PAT 필수**
- PAT는 **scopes(권한 범위)** 를 지정해서 발급 가능 → 최소 권한 원칙 적용 가능

PAT 발급 방법

1️⃣ GitHub → `Settings` → `Developer settings` → `Personal access tokens`
2️⃣ `Fine-grained tokens` 또는 `Tokens (classic)` 선택 → 새 토큰 발급
3️⃣ Scope 설정

주요 Scope 예시

Scope	용도
repo	Private Repository 접근 (Clone/Push/Pull 등)
workflow	GitHub Actions workflow 접근
admin:org	Organization 관리 API 접근
read:packages / write:packages	GitHub Packages 접근

사용 예시

Git HTTPS 사용 시

```
git clone https://github.com/username/repo.git
```

- 처음 인증 요청 시 PAT 입력
- Credential Helper 사용하면 저장 가능

API 사용 시

```
curl -H "Authorization: token <YOUR_PAT>" https://api.github.com/user
```

→ REST API 호출 시 Authorization Header로 사용

PAT 관리 Best Practice

✅ Scope 최소화 → 필요한 기능에만 최소 Scope 설정
✅ 사용 주체별(Personal, CI/CD, Bot 등) 별도 PAT 발급
✅ PAT는 **절대 코드/공개 Repo에 노출 금지**
✅ 토큰 노출 감지 시 즉시 폐기 → 새 PAT 발급

✅ Fine-grained Token 사용 → Repository 단위로 Scope/기간/대상 Repo 설정 가능
✅ Organization에선 **Organization approval 요구 적용 가능** (Enterprise 플랜 등)

---

3 SSH 키 vs PAT 차이

항목	SSH 키	PAT
주요 용도	Git 작업(SSH 기반)	Git 작업(HTTPS 기반), GitHub API, Actions 등
인증 방식	SSH Public/Private Key 쌍	Token 기반 인증
안전성	매우 강함 (비대칭 암호화)	Scope 및 노출 관리가 중요
추천 사용 상황	개발자 로컬 Git 작업	외부 시스템/API 연동, GitHub Actions 연동
발급 주체	GitHub User Profile	GitHub User Profile

---

4 추천 실전 구성

환경	추천 인증 방식
개인 개발 PC → Git 작업	SSH 키
CI/CD (GitHub Actions 내부 Clone/Push)	GitHub Actions default token or PAT
외부 툴(Jenkins, GitLab CI) → GitHub Repo	Fine-grained PAT
REST API 사용	Fine-grained PAT (최소 Scope 설정)
Server → GitHub 자동화 Script	별도 Machine User용 PAT or Deploy Key 사용

---

5 관리 권장 정책

✅ SSH 키는 **기기별 분리, 주기적 갱신, Private Key 노출 금지**
✅ PAT는 **Scope 최소화, 노출 방지, 주기적 폐기 및 재발급**
✅ 조직 차원에서는 Fine-grained PAT 사용 적극 권장
✅ GitHub Actions에서는 **기본 제공** `GITHUB_TOKEN` **사용 우선** → 필요 시 별도 PAT 사용

정리

인증 수단	주요 용도	관리 포인트
SSH 키	Git 작업 (SSH)	Private Key 노출 방지, 기기별 분리
PAT	Git 작업 (HTTPS), API 호출	Scope 최소화, 노출 방지
GITHUB_TOKEN (Actions 기본 제공)	GitHub Actions Workflow 내 작업	기본 사용, 최소한 외부 PAT 사용 최소화

결론:

- 개인 개발자는 SSH 키 + 필요한 경우 PAT 조합 사용
- CI/CD 환경에서는 PAT → Fine-grained PAT 적극 활용
- GITHUB_TOKEN 적극 활용 → 불필요한 PAT 남발 금지
- 보안 정책 수립 → 주기적 감사 및 재발급 프로세스 마련 추천