13. CI/CD와 GitHub 연동

GitHub Actions 기본 구조

1️⃣ 개념

- GitHub Actions는 **GitHub 내에서 직접 실행되는 CI/CD 플랫폼**
- 코드 push, PR 생성, 이슈 발생 등 이벤트를 트리거로 해서 자동화된 workflow 실행 가능
- **테스트 → 빌드 → 배포 → 린트 → 알림** 등 모든 자동화 작업을 정의할 수 있음

2️⃣ 구성 요소 (핵심 구조)

구성 요소	설명
workflow	자동화의 전체 단위 (YAML 파일로 정의, 여러 job 포함)
job	병렬 또는 순차로 실행할 수 있는 작업 단위 (여러 step 포함)
step	실제 명령어(스크립트) 또는 액션을 실행하는 단위
action	재사용 가능한 기능 단위 (오픈소스 action도 사용 가능)
runner	workflow를 실제로 실행하는 가상 머신 (GitHub-hosted runner / self-hosted runner)

3️⃣ Workflow 기본 예시

파일 위치:

```
.github/workflows/ci.yml
```

```yaml
name: CI Pipeline

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Node.js
        uses: actions/setup-node@v4
        with:
```

```
          node-version: '18'

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm test
```

4️⃣ 주요 구성 요소 상세 설명

Workflow (`workflow`)

- `.github/workflows/*.yml` 파일에 정의
- 여러 event 에 반응 가능:

```
on: push
on: pull_request
on: schedule    # cron job
on: workflow_dispatch   # 수동 실행
```

Job (`job`)

- 병렬/순차 실행 가능
- 각 job은 runner 환경(ubuntu, windows, macos 등) 선택 가능

```
jobs:
  test:
    runs-on: ubuntu-latest
    steps: ...
```

Step (`step`)

- job 내 실제 실행 단계
- 2가지 유형:
  - uses: 공개된 action 사용
  - run: shell 명령어 실행

```
- name: Checkout code
  uses: actions/checkout@v4

- name: Run linter
  run: npm run lint
```

Action (`action`)

- 재사용 가능한 스크립트
- 예시:
  - `actions/checkout` → 코드 checkout
  - `actions/setup-node` → Node.js 설치
  - `docker/build-push-action` → 도커 빌드 & 푸시

Marketplace: https://github.com/marketplace?type=actions

---

5 Runner (`runner`)

- workflow를 실제로 실행하는 **가상 머신 or 실제 머신**
- 기본 제공 runner:
  - `ubuntu-latest`
  - `windows-latest`
  - `macos-latest`
- self-hosted runner도 등록 가능 → GPU 사용, 특수 환경 구성 가능

```
runs-on: ubuntu-latest
```

---

6 Workflow 실행 흐름

```
GitHub Event 발생 → GitHub Actions engine → Runner 할당 → Workflow 실행 → 로그 기록 → 결과 보고 (PR/커밋에 체크 표시)
```

---

7 typical 사용 예시

사례	적용 workflow
코드 커밋 시 자동 테스트	on: push + test job
PR 생성 시 코드 lint + test + 결과 표시	on: pull_request
주기적 자동 빌드	on: schedule
PR merge 후 자동 배포	on: push to main + deploy job
수동 배포 트리거	on: workflow_dispatch

8️⃣ 요약 구조 정리

```yaml
name: 워크플로우 이름

on: 이벤트

jobs:
  job1:
    runs-on: runner 환경
    steps:
      - step1: uses or run
      - step2: uses or run

  job2:
    ...
```

9️⃣ GitHub Actions의 장점

- GitHub에 내장 → 별도 CI 서버 필요 없음
- 오픈소스 project에 무료 제공 (private repo는 사용량 제한 있음)
- 다양한 marketplace action 활용 가능
- cross-platform 지원
- self-hosted runner로 확장 가능

1️⃣0️⃣ 결론

GitHub Actions는 현대 GitHub 기반 개발의 핵심 자동화 도구야.

기본 구조만 이해해도:

✅ 커밋 → 자동 테스트 → PR 검증 → 자동 배포
✅ 버전 태그 → 릴리즈 빌드 → 배포 → Slack 알림

이런 파이프라인을 쉽게 구성할 수 있어.

- # workflow, job, step

📌 전체 계층 구조

```
Workflow (workflow 파일 1개 = 1 workflow)
 ├── Job 1
 |    ├── Step 1
 |    ├── Step 2
 |    ├── ...
 ├── Job 2
 |    ├── Step 1
 |    ├── ...
```

- **Workflow** → `.github/workflows/*.yml` 에 정의 (최상위 개념)
- Workflow 안에 여러 **Job** 존재 가능
- 각 **Job** 안에 여러 **Step** 존재 가능
- **Step**은 실제로 명령어 실행 or 외부 Action 실행

1️⃣ Workflow

개념

- 자동화의 "전체 흐름"을 구성하는 단위
- 이벤트 기반으로 실행됨
- GitHub 레포에 `.github/workflows/*.yml`로 저장

구조

```yaml
name: CI Pipeline  # 워크플로우 이름

on:  # 트리거 이벤트
  push:
    branches: [main]
  pull_request:
    branches: [main]
  workflow_dispatch:  # 수동 실행 지원
```

주요 트리거 예시

트리거	설명
`push`	특정 브랜치에 push 발생 시 실행
`pull_request`	PR 생성/업데이트 시 실행
`workflow_dispatch`	사용자가 수동으로 실행
`schedule`	cron 형식 주기적 실행
`release`	Release 생성 시 실행

2️⃣ Job

개념

- Workflow 내에서 실행되는 "작업 단위"
- 서로 다른 **환경**에서 병렬 or 순차 실행 가능
- 기본적으로 **서로 다른 runner 환경에서 격리되어 실행**됨

기본 구조

```
jobs:
  build:
    name: Build Job
    runs-on: ubuntu-latest  # 어떤 Runner 환경에서 실행할지

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Build project
        run: make build
```

runs-on

- GitHub Hosted Runner 지정

옵션	설명
ubuntu-latest	최신 Ubuntu 버전
windows-latest	최신 Windows 버전
macos-latest	최신 macOS 버전
self-hosted	직접 구축한 runner에서 실행

Job 병렬 실행 예시

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps: ...

  test:
    runs-on: ubuntu-latest
    steps: ...
```

- `build` 와 `test` job은 병렬로 실행됨 (기본)

Job 종속성 설정

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps: ...

  deploy:
    runs-on: ubuntu-latest
    needs: build  # build가 끝나야 deploy 실행
    steps: ...
```

3️⃣ Step

개념

- Job 내에서 실행되는 **구체적 실행 단계**
- 두 종류가 있음:
  - `uses` → GitHub Action 실행 (공식/Marketplace에서 가져온 기능)
  - `run` → 직접 Shell 명령어 실행

기본 구조

```
steps:
  - name: Checkout code
    uses: actions/checkout@v4

  - name: Install dependencies
    run: npm ci

  - name: Run tests
    run: npm test

  - name: Upload artifacts
    uses: actions/upload-artifact@v4
    with:
      name: test-results
      path: test-results.xml
```

Step 간 특징

- **동일한 Runner 컨텍스트에서 순차적으로 실행**됨 (이전 step의 상태 유지)
- 예를 들어 Step 1에서 코드를 clone → Step 2에서는 그 디렉토리 사용 가능

4️⃣ 전체 예시

```yaml
name: Node.js CI

on:
  push:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Use Node.js 18
        uses: actions/setup-node@v4
        with:
          node-version: '18'

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm test
```

5️⃣ 정리

구성 요소	역할	특징
Workflow	전체 자동화 단위	GitHub 이벤트 기반으로 실행
Job	병렬/순차 작업 단위	서로 다른 runner에서 실행 가능
Step	실제 실행 단계	각 job 안에서 순차 실행

6️⃣ 권장 설계 패턴

- **Workflow → Job → Step → Action 또는 명령어** 구성으로 항상 체계적으로 작성
- **Job 간 병렬성 적극 활용** → 빌드/테스트 나누기
- **Step 재사용 시 Action 적극 활용** → actions/setup-node, actions/checkout 등
- **환경 변수 (env), secrets 사용으로 보안 유지** (ex: 배포 키, API 토큰)

push/pull request 트리거 자동화

1 기본 개념

- GitHub Actions는 **이벤트 기반**으로 동작함.
- 가장 많이 쓰는 이벤트가 바로:

이벤트 이름	언제 발생?
`push`	브랜치에 새로운 커밋이 push될 때
`pull_request`	Pull Request가 생성되거나 업데이트될 때

→ 이걸 이용해서 **CI(테스트), 빌드, 린트, 배포, PR 검증 등 거의 모든 자동화 작업**을 할 수 있음.

---

2 `push` 트리거

기본 구조

```yaml
on:
  push:
    branches:
      - main
      - release/*
    paths:
      - '**.js'
      - 'src/**'
```

설명

설정	의미
`branches`	어떤 브랜치에 push될 때만 실행
`tags`	태그 push 시 실행 가능
`paths`	특정 파일/디렉토리 변경 시만 실행 가능 (선택적)

예시

```yaml
on:
  push:
    branches:
      - main
    tags:
      - 'v*.*.*'  # 버전 태그 push 시 실행 (ex: v1.0.0)
```

→ 이런 식으로 **정식 배포 태그가 push될 때만 배포 job 실행**도 가능.

3️⃣ `pull_request` 트리거

기본 구조

```yaml
on:
  pull_request:
    branches:
      - main
    types: [opened, synchronize, reopened, closed]
    paths:
      - '**.js'
```

설명

옵션	의미
`branches`	PR 대상 브랜치 (PR → 어느 브랜치로 머지하는가)
`types`	어떤 PR 이벤트에 반응할 것인가
`paths`	PR에서 수정한 파일 중 지정 경로가 포함된 경우만 실행

types 주요 값

값	의미
`opened`	PR 생성 시
`synchronize`	PR에 새로운 커밋 push 시
`reopened`	PR 다시 열었을 때
`closed`	PR이 closed 되었을 때 (머지 포함)

예시

```yaml
on:
  pull_request:
    branches:
      - main
    types: [opened, synchronize, reopened]
```

→ PR 생성/업데이트 시 테스트/빌드 자동 실행
→ 코드 리뷰 단계에서 **PR이 항상 통과한 상태인지 확인 가능**

4️⃣ push vs pull_request 차이

항목	push	pull_request
대상	직접 브랜치에 push	PR 생성/업데이트
사용 목적	배포, main 브랜치 자동화	PR 단계에서 테스트/빌드/품질 검증
실행 타이밍	커밋 push 직후	PR 관련 이벤트 발생 시

실전 조합 패턴:

- **push → main**: 배포 파이프라인
- **pull_request → main**: PR 검증용 빌드/테스트 → PR 머지 시점에 품질 확인

5️⃣ 실전 예시 (조합 사용)

```yaml
on:
  push:
    branches:
      - main
    tags:
      - 'v*.*.*'

  pull_request:
    branches:
      - main
    types: [opened, synchronize, reopened]
```

→ 동작 흐름:

```
[push to main] → 배포 workflow 실행
[push 태그(v1.0.0)] → 배포 workflow 실행
[PR → main] → PR 테스트 + 빌드 실행 (코드 리뷰 전 품질 검증)
```

6️⃣ 요약 정리

트리거	주요 용도
`push`	main/release 브랜치 변경 시 자동 배포
`push` + tags	Release 태그 생성 시 자동 배포
`pull_request`	PR 생성/업데이트 시 자동 테스트/빌드
`pull_request.closed`	PR 머지 후 후속 처리 (예: 슬랙 알림)

중요 포인트:

- `push` 와 `pull_request` 를 **함께 사용하는 것이 가장 일반적인 패턴**임
- `pull_request`는 반드시 PR 단계에서 테스트/빌드 필수로 설정 → 코드 리뷰 품질 확보
- `push`는 **머지 후 main/release 브랜치에 대한 배포**나 후처리용으로 사용

테스트/빌드/배포 자동화

1 전체 흐름

대표적인 CI/CD 파이프라인 예시

```
[push or pull_request 발생]
          ↓
[CI 단계]  - 코드 체크아웃
           - 의존성 설치
           - 코드 린트 검사
           - 테스트 실행
           ↓
[Build 단계] - 빌드 artifact 생성 (ex: JAR, Docker image 등)
           ↓
[Deploy 단계] - Release tag 시 배포 자동 실행 (서버, 클라우드 등)
```

---

2 기본 Workflow 구조 (스켈레톤)

```yaml
name: CI/CD Pipeline

on:
  push:
    branches: [main]
    tags:
      - 'v*.*.*'
  pull_request:
    branches: [main]

jobs:
  ci:
    name: Test & Build
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '18'

      - name: Install dependencies
```

```
        run: npm ci

      - name: Run lint
        run: npm run lint

      - name: Run tests
        run: npm test

      - name: Build project
        run: npm run build

  deploy:
    name: Deploy (Only on Tag)
    runs-on: ubuntu-latest
    needs: ci
    if: startsWith(github.ref, 'refs/tags/v')
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Deploy to server
        run: ./deploy.sh
```

3️⃣ 단계별 설명

🚀 1) CI 단계 (테스트 자동화)

```
      - name: Run lint
        run: npm run lint

      - name: Run tests
        run: npm test
```

- PR 생성/업데이트 시 테스트가 자동 실행됨
- → **코드 리뷰 전 반드시 테스트 통과 보장**
- 실패 시 PR에 X 표시되어 머지 차단 가능

🚀 2) Build 단계 (빌드 자동화)

```
      - name: Build project
        run: npm run build
```

- 테스트 통과 후 빌드 실행
- 빌드 결과물 (JAR, WAR, Docker Image, static files 등)을 생성
- 필요 시 artifact 저장 가능:

```
      - name: Upload build artifact
        uses: actions/upload-artifact@v4
        with:
          name: build-result
          path: dist/
```

🚀 3) Deploy 단계 (배포 자동화)

```
deploy:
  if: startsWith(github.ref, 'refs/tags/v')
```

- → **버전 태그가 push될 때만 배포 단계 실행** (ex: v1.0.0)
- → 실수 방지 (PR 머지했다고 바로 배포되는 실수를 방지)
- deploy 단계에서 하는 작업 예:

```
      - name: Deploy to server
        run: ./deploy.sh
```

→ deploy.sh 안에서:

```
scp dist/* user@myserver:/var/www/html
ssh user@myserver 'sudo systemctl restart myservice'
```

or

→ Docker 기반이라면:

```
      - name: Build and push Docker image
        uses: docker/build-push-action@v5
        with:
          context: .
          tags: my-dockerhub-user/myapp:${{ github.ref_name }}
          push: true
```

→ AWS S3 배포 예:

```
      - name: Deploy to S3
        uses: jakejarvis/s3-sync-action@v0.5.1
        with:
          args: --acl public-read --delete
        env:
          AWS_S3_BUCKET: ${{ secrets.AWS_S3_BUCKET }}
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          AWS_REGION: 'ap-northeast-2'
          SOURCE_DIR: './dist'
```

---

4 실전 적용 패턴

PR 생성 시

```
✅ Lint 검사
✅ Unit Test
✅ Build 테스트
```

Main 브랜치 push 시

```
✅ Lint 검사
✅ Unit Test
✅ Build 테스트
✅ (선택) 자동 배포 → staging 서버
```

Release tag push 시

```
✅ Lint 검사
✅ Unit Test
✅ Build 테스트
✅ Production 서버 자동 배포
✅ Slack 알림 전송
```

---

5 고급 팁

- `needs:` 키워드로 job 간 실행 순서 정의
- `if:` 조건으로 배포 단계 안전하게 설정
- `matrix` 전략으로 다양한 Node.js / Java / Python 버전 테스트
- Artifact 저장 및 다운로드 활용해 build → deploy 간 결과물 전달

---

6 정리

단계	목적	주요 명령
테스트	품질 보증	`npm test`, `gradle test`, `mvn test`
빌드	결과물 생성	`npm run build`, `gradle build`, `mvn package`, Docker build
배포	서버 배포	SCP, SSH, AWS CLI, Docker push, Kubernetes

7️⃣ 결론

- GitHub Actions 기반으로 테스트/빌드/배포 자동화 구성 시:

✅ PR → 품질 보증 자동화
✅ Main 브랜치 → staging 배포 자동화
✅ Release tag → Production 배포 자동화

→ **완벽한 DevOps 기반을 구축할 수 있음.**

secret 관리

1️⃣ 왜 필요한가?

- Workflow에서 보통 **민감한 값(비밀번호, API 키, 클라우드 접근키, SSH key 등)** 을 사용해야 함:

```
aws s3 cp ... --access-key=AKIA... --secret-key=...
```

→ 이런 값을 **그냥 YAML에 노출하면 매우 위험**
→ GitHub Actions는 **Secrets** 기능을 통해 안전하게 관리할 수 있도록 제공함.

2️⃣ 기본 개념

이름	위치	적용 범위
Repository secrets	특정 레포지토리에 설정	해당 레포에만 사용 가능
Organization secrets	조직(Organization) 단위로 설정	선택한 여러 레포에서 공유 사용 가능
Environment secrets	특정 Environment(ex: prod, staging)에 설정	해당 환경에서만 사용 가능 (환경 승인 정책과 함께 사용)

3️⃣ Repository Secrets 설정법

1️⃣ GitHub 레포 → `Settings` → `Secrets and variables` → `Actions` → `Secrets`
2️⃣ `New repository secret` 클릭
3️⃣ Name / Value 입력 → 저장

예시

Name	Value (예시)
`AWS_ACCESS_KEY_ID`	`AKIA**********`
`AWS_SECRET_ACCESS_KEY`	`**********`
`PROD_DEPLOY_SSH_KEY`	`-----BEGIN OPENSSH PRIVATE KEY----- ...`

---

4 Workflow에서 Secrets 사용법

기본 구조: `${{ secrets.<SECRET_NAME> }}`

```
env:
  AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
  AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

or Step에서 직접 사용:

```
- name: Deploy to S3
  run: |
    aws s3 sync ./dist s3://my-bucket --delete
  env:
    AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
    AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

---

5 주의사항

- `secrets` 값은 Workflow 로그에 **자동으로 masking 처리됨** (`***` 로 표시)
- 기본적으로 **read-only** (Workflow 안에서 값 변경 불가)
- 문자열만 저장 가능 (파일 저장 시에는 base64 인코딩 후 저장해서 decode해서 사용)

---

6 Environment Secrets (고급)

- GitHub `Environment` 기능과 함께 사용
- **배포 자동화 시 staging / production 분리 관리 시 유용**

사용 흐름

```
jobs:
  deploy:
    environment: production
    runs-on: ubuntu-latest
    steps:
      - name: Deploy
        run: ./deploy.sh
        env:
          DEPLOY_KEY: ${{ secrets.PROD_DEPLOY_KEY }}
```

→ `environment: production` 으로 설정하면 → `production` Environment에 설정된 Secrets 사용

→ **Environment 별 승인(Reviewer 승인 후 실행)도 설정 가능 → Production 안전 배포에 매우 유용**

7️⃣ Organization Secrets (고급)

- 조직(Organization) 전체 차원에서 관리 가능
- 여러 레포에서 공통으로 사용 가능
- 조직 내 프로젝트가 많을 때 **중복 설정 줄이고 중앙 관리 가능**

설정 위치:

- GitHub Organization → `Settings` → `Secrets` → `Actions`

8️⃣ Secret 값 파일로 사용하기

파일 형태로 필요한 경우 → base64 encode + decode 패턴

업로드 시:

```
base64 -w 0 my_private_key.pem > encoded_key.txt
```

Secrets 에 `PRIVATE_KEY_BASE64`로 등록

Workflow에서 사용:

```
- name: Write private key
  run: echo "${{ secrets.PRIVATE_KEY_BASE64 }}" | base64 -d > private_key.pem
```

→ 이렇게 하면 **private key 파일을 안전하게 runtime에 생성해서 사용** 가능

9️⃣ 정리

구분	용도	주요 특징
Repository secrets	레포 내부에서 민감한 값 관리	가장 일반적 사용
Organization secrets	여러 레포에서 공유 사용	중앙화 관리
Environment secrets	Staging / Production 분리	승인 정책 + 안전 배포

사용 시 핵심 포인트:

✅ 절대 YAML에 직접 노출 X
✅ `secrets.<NAME>` 구문으로 안전하게 참조
✅ 민감한 파일은 base64 encode 패턴 사용
✅ Environment 기반 분리 적극 추천 (배포시 특히 중요)

외부 서비스 연동 (Slack, AWS 등)

전체 원리

- GitHub Actions는 각 Step에서:
  - **외부 서비스용 CLI 사용** (ex: AWS CLI, kubectl 등)
  - **공식/비공식 Action 사용** (ex: Slack 알림 Action)
  - **API 호출 (curl 등 직접 호출)**
- 연동 시 민감 정보는 반드시 **Secrets** 활용

1️⃣ AWS 연동

1. 사용 사례

✅ S3에 정적 파일 배포
✅ EC2 서버에 코드 배포 (SSH or SSM)
✅ Lambda Function 배포
✅ EKS Kubernetes 배포

---

2. 사전 준비

- AWS IAM 사용자 생성 → **Programmatic access 활성화**
- `AWS_ACCESS_KEY_ID` 와 `AWS_SECRET_ACCESS_KEY` 발급
- GitHub Repository `Secrets`에 등록

---

3. AWS CLI 사용 패턴

```
- name: Configure AWS credentials
  uses: aws-actions/configure-aws-credentials@v4
  with:
    aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
    aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    aws-region: ap-northeast-2

- name: Deploy to S3
  run: aws s3 sync ./dist s3://my-bucket-name --delete
```

→ AWS CLI가 자동으로 인증된 상태로 사용 가능

---

4. 예시: EC2 서버에 SSH 배포

```
- name: Set up SSH key
  run: |
    echo "${{ secrets.EC2_SSH_PRIVATE_KEY }}" | base64 -d > ec2_key.pem
    chmod 600 ec2_key.pem

- name: Deploy to EC2
  run: |
    scp -i ec2_key.pem ./dist/* ec2-user@${{ secrets.EC2_HOST }}:/var/www/html
    ssh -i ec2_key.pem ec2-user@${{ secrets.EC2_HOST }} 'sudo systemctl restart nginx'
```

→ EC2에 직접 파일 복사 후 서비스 재시작

2️⃣ Slack 연동

1. 사용 사례

✅ CI/CD 결과 Slack 채널에 알림
✅ 배포 성공/실패 알림
✅ PR 생성 시 자동 알림

---

2. 사전 준비

- Slack App 생성
- `Incoming Webhook` 기능 활성화 → Webhook URL 발급
- Webhook URL을 GitHub Repository `Secrets`에 등록 (`SLACK_WEBHOOK_URL`)

---

3. 기본 사용 패턴 (curl 직접 사용)

```
- name: Notify Slack
  run: |
    curl -X POST -H 'Content-type: application/json' --data "{
      \"text\": \"✅ Deployment completed on *Production*! 🚀\"
    }" ${{ secrets.SLACK_WEBHOOK_URL }}
```

---

4. 공식 Action 사용

```
- name: Slack Notification
  uses: 8398a7/action-slack@v3
  with:
    status: ${{ job.status }}
    fields: workflow,job,commit,repo,ref,author,took
  env:
    SLACK_WEBHOOK_URL: ${{ secrets.SLACK_WEBHOOK_URL }}
```

→ 배포 성공/실패 여부 자동으로 Slack 알림 가능
→ 커밋 정보, 브랜치 정보도 포함 가능

---

3 기타 서비스 연동

서비스	사용 방법
Discord	Slack과 거의 동일, Webhook 사용
Microsoft Teams	Webhook 사용
Telegram	Bot Token + Chat ID 사용, curl로 API 호출
Notion	API Token 사용 + REST API 호출
PagerDuty / OpsGenie	Incident 알림 API 사용
Google Cloud (GCP)	`google-github-actions/auth` Action 사용
Azure	`azure/login` Action 사용

---

4 실전 패턴 예시

배포 후 Slack 알림

```
deploy:
  runs-on: ubuntu-latest
  steps:
    - name: Checkout code
      uses: actions/checkout@v4

    - name: Deploy to S3
      run: aws s3 sync ./dist s3://my-bucket-name --delete

    - name: Notify Slack
      run: |
        curl -X POST -H 'Content-type: application/json' --data "{
          \"text\": \"✅ *Production* 배포 성공! \nCommit: $GITHUB_SHA \nBranch: $GITHUB_REF\"
        }" ${{ secrets.SLACK_WEBHOOK_URL }}
```

---

5 정리

서비스	연동 방법
AWS	AWS 공식 Action + AWS CLI
Slack	Webhook URL + curl 직접 호출 or 공식 Action

서비스	연동 방법
EC2	SSH Key를 secrets에 저장 + scp + ssh 사용
기타 서비스	대부분 Webhook 기반 or 공식 Action 제공

결론

GitHub Actions = CI/CD 플랫폼 + 통합 자동화 허브

→ AWS, Slack 등 외부 서비스 연동을 통해:

- ✅ 배포 자동화
- ✅ 알림 자동화
- ✅ PR/Issue 기반 알림 자동화
- ✅ Cloud API 활용한 서비스 운영 자동화

→ 실전에서 **완전 자동화된 개발/운영 파이프라인 구축** 가능.