[Git : Spring Legacy MVC Project 팀원간 버전 관리 방법]

1 단계: 프로젝트를 깃허브에 올리는 방법 (처음 한 번만)

1.1 마스터(팀장)가 해야 할 일

1. **깃허브에 로그인**한다. (https://github.com)
2. **새 저장소(Repository)** 를 만든다. (이름은 예를 들어 team-project 로 하자.)
3. 로컬 컴퓨터에 있는 스프링 프로젝트 폴더를 열고, 명령어를 입력한다:

1. GitHub 에서 새 저장소(Repository)를 만드는 방법

사전 준비

- https://github.com 에 회원가입 및 로그인 완료 상태

1 단계: GitHub 에 로그인

1. 웹 브라우저에서 https://github.com 접속
2. 우측 상단의 **Sign in** 클릭 → 아이디/비밀번호 입력 후 로그인

2 단계: 새 저장소 생성 페이지로 이동

1. 로그인 후, 우측 상단의 **+** 아이콘 클릭
2. 드롭다운 메뉴에서 **"New repository"** 선택

또는 바로가기 URL 사용: https://github.com/new

3 단계: 저장소 정보 입력

아래 항목을 입력합니다:

항목	설명
Owner	저장소 소유자 선택 (개인 또는 조직 계정)
Repository name	저장소 이름 (예: team-project)
Description (optional)	저장소에 대한 설명 (예: "Spring Boot team project")
Public / Private	
◉ **Public**: 누구나 볼 수 있음	무료, 팀원 수 제한 없음
◉ **Private**: 선택된 사용자만 접근 가능	무료로는 최대 3명까지 공동 작업 가능
Initialize this repository with:	
☑ **Add a README file** (README.md 자동 생성)	
☑ **Add .gitignore** (원하는 언어/환경 선택, 예: Java)	Java

항목	설명
☑ **Choose a license** (MIT, Apache 등 선택 가능)	None : 다른 사람이 **법적으로 코드를 사용하거나 수정, 배포할 수 없음**
(옵션이지만 편리함을 위해 체크 권장)	

4 단계: 저장소 생성

- 모든 항목 입력 후, 하단의 **[Create repository]** 버튼 클릭

완료 후 저장소가 다음과 같이 생성됩니다:
https://github.com/사용자명/저장소명
예: https://github.com/chaju-yun/team-project

Git repository 삭제 방법

Git 저장소(Repository)를 삭제하는 방법은 **GitHub 원격 저장소 삭제**와 **로컬 저장소 삭제** 두 가지로 나뉩니다.

1. GitHub 원격 저장소 삭제 (GitHub 웹사이트에서)

주의: 이 작업은 되돌릴 수 없습니다!
저장소의 코드, 이슈, 커밋 기록 등 모든 내용이 삭제됩니다.

◆ 삭제 방법

1. GitHub 로그인 후 삭제할 저장소 페이지로 이동 예: https://github.com/yun-chaju/team-project
2. 상단 메뉴에서 **[Settings]** 클릭
3. 좌측 메뉴에서 **[General]** 또는 아래쪽으로 스크롤
4. 가장 아래로 내려가서 **"Danger Zone"** 섹션 찾기
5. **"Delete this repository"** 클릭
6. GitHub 이 저장소 이름을 입력하라고 요구함
   정확하게 저장소 이름 입력 (yun-chaju/team-project)
7. 확인 후 **[I understand the consequences, delete this repository]** 클릭

2. 로컬 Git 저장소 삭제 (PC 에서)

로컬에서 Git 저장소를 완전히 삭제하려면 해당 폴더 자체를 삭제하면 됩니다.

◆ 방법 1: 폴더 전체 삭제

예: 현재 경로가 /d/spring_workspace/team-project 일 때

```
cd ..
rm -rf team-project
```

Windows 에서는 탐색기에서 해당 폴더 우클릭 → "삭제" 해도 됩니다.

◆ 방법 2: Git 설정만 삭제 (.git 폴더만 삭제)

Git 저장소만 초기화하고 폴더는 유지

```
rm -rf .git
```

이렇게 하면 해당 폴더는 **Git 저장소가 아닌 일반 폴더**로 바뀝니다.

2. 명령어 입력 위치(환경)

: 프로젝트가 위치한 폴더에서 터미널(CMD)이나 Git Bash 를 열고 입력합니다.

예를 들어 :

스프링 레거시 프로젝트가 D:₩spring-workspace₩team-project 에 있다고 할 때,

- 해당 프로젝트 폴더에서 Git 명령어를 실행해야 합니다.

방법 1: Git Bash 로 실행 (추천)

Git Bash 를 설치한 후, Spring Legacy (또는 일반 Java/Spring 프로젝트) 를 GitHub 에 업로드하는 전체 명령어 시퀀스입니다.

사전 준비

항목	설명
프로젝트 생성 완료	예: C:₩spring-workspace₩team-project
Git Bash 설치 완료	https://git-scm.com
GitHub 계정 & 저장소 생성	예: https://github.com/yourname/team-project.git

---

□ 단계별 Git Bash 설치 과정 (참고)

Git Bash 는 Windows 에서 Git 명령어를 사용할 수 있도록 해주는 도구입니다.
아래는 공식 Git 사이트인 https://git-scm.com 에서 **Git Bash** 를 다운로드하고 설치하는 과정을 설명한 것입니다.

✓✓ 1단계: Git Bash 설치 파일 다운로드

1. 웹 브라우저를 열고 다음 주소로 접속합니다:
   ☞ https://git-scm.com
2. 메인 화면에서 자동으로 운영체제(Windows)의 최신 버전이 감지되며, **"Download for Windows"** 버튼이 보입니다.

예: Download 64-bit Git for Windows

3. 해당 버튼을 클릭하면 .exe 설치 파일이 자동으로 다운로드됩니다.
   (예: Git-2.49.0-64-bit.exe)

✓✓ 2 단계: 설치 마법사 실행

1. 다운로드된 .exe 파일을 **더블 클릭**하여 설치를 시작합니다.
2. **사용자 계정 컨트롤(UAC)** 메시지가 뜨면 "예(Yes)"를 눌러 설치를 허용합니다.

✓✓ 3 단계: 설치 옵션 설정 (권장 설정대로 진행)

설치 마법사의 화면에서 각 단계를 아래와 같이 설정합니다.

Step 1: License 확인

- **"Next"** 클릭

Step 2: 설치 위치 선택

- 기본 위치 그대로 두고 **"Next"**

Step 3: 선택할 컴포넌트

- 'Add icon desktop' 체크하고, 나머지 기본 선택 상태 그대로 두고 **"Next"**

Step 4: 시작 메뉴 폴더

- 기본값 그대로 두고 **"Next"**

Step 5: 기본 편집기 선택

- 기본값은 **Vim** 입니다.
- 다른 편집기를 원한다면 드롭다운에서 선택
- **"Next"**

Step 6: 초기 브랜치 이름 설정

- **"Let Git decide"** 그대로 두고 **"Next"**

Step 7: 환경변수 설정 (PATH)

- **"Git from the command line and also from 3rd-party software"** 선택 (권장)
- **"Next"**

Step 8: HTTPS 전송 방식 설정

- **"Use the OpenSSL library"** 선택 (기본값)
- **"Next"**

Step 9: 줄 끝 처리 방식 설정

- **"Checkout Windows-style, commit Unix-style line endings"** 선택 (권장)
- **"Next"**

Step 10: 터미널 에뮬레이터 설정

- **"Use MinTTY (the default terminal of MSYS2)"** 선택 (권장)
- **"Next"**

Step 11: Git pull 동작 방식

- **"Default (fast-forward or merge)"** 선택 (권장)

- **"Next"**

Step 12: 자격 증명 관리

- **"Git Credential Manager Core"** 선택 (기본값)
- **"Next"**

Step 13: 추가 옵션

- "Enable file system caching"에 체크되어 있음 → 유지
- **"Next"**

Step 14: 실험적 기능 선택

- 실험 기능은 선택하지 않고 **"Install"**

✓ 4 단계: 설치 진행 및 완료

- 설치가 완료되면 **"Git Bash Here"** 옵션이 바탕화면 또는 우클릭 메뉴에 추가됩니다.
- **"Launch Git Bash"**에 체크한 후 **"Finish"** 클릭

✓ 5 단계: Git Bash 실행 확인

- 바탕화면에서 **Git Bash 아이콘**을 더블 클릭하거나
- 빈 폴더에서 **우클릭 → Git Bash Here** 클릭하여 실행
  단축 메뉴에 'Git Bash Here' 가 보이지 않는다면, 단축 메뉴 하단의 '추가 옵션 표시'를 클릭함
- 다음과 같이 터미널이 나타나면 설치 완료:
- MINGW64 ~

---

3. 단계별 Git Bash 명령어 시퀀스

1 단계: Git Bash 실행

1. 파일탐색기에서 D:₩spring_workspace₩team_project 폴더로 이동
2. 빈 공간에서 **우클릭 > Git Bash Here** 클릭

Git Bash 터미널이 열립니다.

2 단계: Git 초기화

```
git init
```

.git 폴더가 생성되며 Git 이 이 프로젝트를 추적 시작함

3 단계: 원격 저장소 연결 (GitHub 에 만든 저장소 주소 연결)

```
git remote add origin https://github.com/yourname/team-project.git
```

- yourname → 본인의 GitHub 계정명
- team-project → 만든 저장소 이름

4 단계: .gitignore 파일 작성 (중요)

```
touch .gitignore
```

아래처럼 Git Bash 에서 직접 파일을 만들어도 되고, 메모장으로 저장해도 됩니다.
프로젝트 폴더 안에 저장하면 됩니다. (team-project/.gitignore)
그 다음 nano .gitignore 또는 VS Code 로 열어서 아래 내용 입력:

```
# Eclipse 관련 설정 파일
/.settings/
.project
.classpath

# IntelliJ 관련 (공동 사용 시 포함)
.idea/
*.iml

# 빌드 결과물
/target/
bin/
/build/

# 로그파일
*.log

# 운영체제별 불필요 파일
.DS_Store
Thumbs.db

# 환경 변수 설정 파일 (필요시)
*.env
application-*.properties
application-*.yml
pom.xml
*.xml
```

저장 후 닫기 (nano 사용 시 Ctrl + O → Enter, Ctrl + X)

5 단계: Git 에 파일 추가 (추적 등록)

```
git add .
```

. 은 현재 폴더의 모든 파일/폴더를 Git 에 추가하겠다는 의미

6 단계: 첫 번째 커밋 만들기

```
git commit -m "초기 커밋: 스프링 레거시 프로젝트 업로드"
```

"커밋 메시지"는 팀 내 규칙에 맞춰 작성하세요

7 단계: 브랜치 이름 설정 (main 또는 master)

최근 Git 은 기본 브랜치가 main 입니다.

```
git branch -M main
```

만약 master 브랜치로 관리한다면 이 단계는 생략해도 됩니다.

8 단계: GitHub 로 푸시

```
git push -u origin main
```

최초 푸시이므로 -u 옵션을 사용해 기본 브랜치와 연결

✅ GitHub 로그인 정보가 필요할 수 있음 →

Personal Access Token 을 비밀번호 대신 입력하세요 (2021 년 이후부터 비밀번호 사용 불가)

에러

```
CONFLICT (add/add): Merge conflict in .gitignore Automatic merge failed; fix conflicts and then commit the result.
```

.gitignore 파일에서 충돌이 발생!

이유:

.gitignore 파일이 **로컬에도 있고**, **GitHub에도 있었기 때문에 충돌**이 발생했습니다.

Git은 이 두 내용을 어떻게 합칠지 모르기 때문에 수동으로 해결해야 합니다.

해결 순서

◆ **1 단계: 원격 저장소에서 .gitignore 내려 받은 다음, 파일 열기**

1. 원격 저장소 내용 먼저 받아오기

```
git pull origin main --allow-unrelated-histories
```

편집기로 열면 다음과 같은 내용이 보입니다:

```
<<<<<<< HEAD
# 로컬에 있던 내용
*.log
=======
# 원격(GitHub)에 있던 내용
*.class
```

```
>>>>>>> origin/main
```

◆ 2 단계: 충돌 해결

아래처럼 수정하면 두 내용을 모두 살릴 수 있습니다:

```
*.log
*.class
```

그리고 <<<<<<<, =======, >>>>>>> 구문은 **반드시 삭제**해야 합니다.

◆ 3 단계: 수정된 .gitignore 파일을 스테이지에 추가

```
git add .gitignore
```

◆ 4 단계: 병합 커밋 완료

```
git commit
```

자동으로 병합 메시지가 나타납니다. 그대로 저장하고 종료하면 됩니다.

◆ 5 단계: GitHub 로 푸시

```
git push -u origin main
```

결과

```
Enumerating objects: 13, done. Counting objects: 100% (13/13), done. Delta compression using up to 24 threads Compressing objects: 100% (6/6), done. Writing objects: 100% (11/11), 1.21 KiB | 1.21 MiB/s, done. Total 11 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0) remote: Resolving deltas: 100% (1/1), done. To https://github.com/yun-chaju/team-project.git e401e05..0b75c6e main -> main branch 'main' set up to track 'origin/main'.
```

GitHub 저장소 웹사이트에 가보면 파일들이 올라가 있고(시간이 좀 걸릴 수 있음), README와 같은 내용도 같이 확인할 수 있습니다.

✓✓ 전체 요약 (명령어만 모아 보기)

```
# 1. Git Bash 열기
cd /c/spring-workspace/team-project

# 2. Git 초기화
git init
```

```
# 3. 원격 저장소 등록
git remote add origin https://github.com/yourname/team-project.git

# 4. .gitignore 생성 및 설정
touch .gitignore
# → 편집기로 열어 설정 입력

# 5. Git 추적 시작
git add .

# 6. 커밋 생성
git commit -m "🚀 초기 커밋: 스프링 레거시 프로젝트 업로드"

# 7. 브랜치 이름 설정 (선택사항)
git branch -M main

# 8. GitHub 로 푸시
git push -u origin main
```

방법 2: CMD (명령 프롬프트) 사용

1. **시작 → cmd → 우클릭 → 관리자 권한 실행 (권장)**
2. 프로젝트 디렉토리로 이동:

cd /d D:₩spring-workspace₩team-project

3. 그리고 Git 명령어 입력:

```
git init
git remote add origin https://github.com/계정명/team-project.git
git add .
git commit -m "처음 프로젝트 업로드"
git push -u origin master
```

방법 3: VS Code 또는 IntelliJ, Eclipse 의 내장 터미널 사용

- VS Code 나 IntelliJ 등 IDE 에서 프로젝트를 연 경우:
    - **터미널 탭**을 열고 거기서 Git 명령어를 입력해도 됩니다.

○ 단, **현재 경로가 프로젝트 루트**인지 확인하세요.

```
git init  # 깃 시작하기
```

```
git remote add origin https://github.com/계정명/team-project.git  # 연결
git add .  # 모든 파일 저장 준비
git commit -m "처음 프로젝트 업로드"  # 저장 설명
git push -u origin master  # 깃허브에 올리기
```

---

2 단계: 팀원들이 프로젝트 받기 (복제하기)

팀원들이 해야 할 일

1. 깃허브에서 team-project 저장소 주소를 복사한다.
2. 자신의 컴퓨터에서 아래 명령어로 복사(클론)한다:

```
git clone https://github.com/계정명/team-project.git
```

3. 이러면 team-project 폴더가 생기고, 안에 프로젝트가 들어있음!

팀 프로젝트를 할 때 **Git 로컬 저장소 위치와 STS4(Spring Tool Suite 4)의 워크스페이스 위치를 같게 지정해도 되는지**, 또는 **분리하는 것이 좋은지**는 사용 목적과 협업 방식에 따라 다를 수 있습니다.

선택	가능 여부	추천 여부	설명
같은 폴더 사용	가능함	**비추천**	관리가 복잡해질 수 있음 (Git 관련 파일이 워크스페이스 전체에 노출됨) **워크스페이스에 프로젝트 1 개만 두고 작업할 경우에는 권장함**
다른 폴더 사용	가능함	**추천**	워크스페이스는 관리 폴더고, Git 프로젝트는 개별 폴더로 분리하는 것이 협업 및 유지보수에 유리함 **(관리가 복잡함)**

설명

① 워크스페이스와 로컬 저장소를 *같게 지정한 경우*

- 예: D:₩spring_workspace 를 워크스페이스로 설정하고, 그 위치에서 Git clone 실행
- 문제점:
  - .git 폴더와 Git 관련 파일이 워크스페이스 루트에 생성됨
  - 여러 개의 Git 프로젝트를 사용할 경우, **워크스페이스에 다 섞이게 됨**
  - .metadata, .settings, .classpath, .project 등 STS 의 내부 설정과 충돌할 수 있음

② 워크스페이스와 Git 프로젝트 폴더를 *분리한 경우* (추천)

- 워크스페이스: D:₩sts4_workspace
- Git clone: D:₩git_projects₩team-project
  team-project 폴더 안에 pom.xml 파일을 복사 붙여넣기함

- STS 에서 File → Import → Existing Maven/Gradle Project 또는 File → Open Projects from File System 으로 team-project 폴더만 가져오면 됨
- 장점:
    - 여러 프로젝트가 깔끔하게 관리됨
    - Git 이력과 충돌 가능성 ↓
    - STS 내부 설정 파일과 Git 설정이 독립적으로 유지됨

추천 워크플로우

1. **Git 프로젝트 복제**
2. cd /d/git_projects
3. git clone https://github.com/yun-chaju/team-project.git
4. **STS4 실행 → 워크스페이스는 D:₩sts4_workspace 등으로 지정**
5. **STS4 에서 프로젝트 불러오기**
    - File > Import > Existing Maven Projects (또는 Gradle 프로젝트)
    - 경로: D:₩git_projects₩team-project (import 전에 pom.xml 복사해 넣을 것)

---

3 단계: 각자 브랜치 만들기 (자신만의 작업 공간 만들기)

브랜치란?

"브랜치"는 마치 복사본처럼 **혼자만의 실험공간임**. 여기서 마음껏 수정하고, 나중에 합칠 수 있음.

만드는 방법

1. 워크스페이스 작업중인 프로젝트(예: first) 폴더 안(.git 폴더가 있는 위치)에서 git bash 열기 > git baxh 열기에서 다음을 입력:

```
git checkout -b 본인이름-작업내용
예:
git checkout -b joo-login-feature
Switched to a new branch 'joo-login-feature'  # 성공 메시지

# 확인
git branch
```

2. 현재 브렌치에 * 표시됨, 이렇게 하면 이제 그 브랜치에서 작업하는 중임.

4 단계: 파일 수정 후 깃허브에 올리기

순서 (매우 중요!!)

```
git add .  # 바뀐 파일들 준비
git commit -m "로그인 기능 추가"  # 설명 쓰기
```

```
git push origin 브랜치이름  # 깃허브에 올리기
예:
git push origin joo-login-feature
```

커밋 메시지 작성 팁

- 짧고 명확하게 작성 (30 자 이내)
- 무엇을 **왜** 했는지 드러나면 좋음

예시:

- "로그인 입력 폼 유효성 검사 추가"
- "관리자 대시보드 UI 개선"
- "회원가입 시 중복 이메일 체크 구현"

주의사항

체크포인트	설명
git add 누락	아무 것도 커밋되지 않음
브랜치 이름 실수	main 에 잘못 올릴 수 있음 (팀 작업 시 주의!)
커밋 없이 push	아무 것도 업로드되지 않음 **(커밋은 필수!)**
깃허브 권한 없음	403 또는 Permission denied 오류 발생 가능

---

push 전에 **내가 변경한 내용을 확인(diff)** 하는 것은 **팀 협업 시 꼭 필요한 습관**입니다.
아래에 Git 에서 diff 를 보는 여러 방법을 **상황별로 정리하였습니다.**

1. 작업 중인 파일의 변경 내용 보기

```
git diff
```

- 워킹 디렉토리(현재 작업 중인 파일)와 스테이징 영역(git add 안 한 상태)의 **차이점**을 보여줍니다.
- 아직 **git add 하지 않은 파일들의 코드 변경 내역을 확인**할 수 있습니다.

2. 스테이징된 파일의 변경 내용 보기

```
git diff --cached
또는:
git diff --staged
```

- 이미 git add 한 파일(=스테이징 영역)에 대해서 변경된 내용을 확인
- 이 상태에서 커밋하면 들어갈 내용이 무엇인지 정확히 보여줌

3. 파일 하나만 diff 보기

```
git diff 파일명
예시:
git diff first/src/main/webapp/WEB-INF/views/test/testAjaxView.jsp
```

→ 해당 파일의 변경사항만 확인 가능

4. 최근 커밋과의 변경 내용 보기

```
git diff HEAD
```

- 가장 최근 커밋(HEAD)과 현재 작업 파일 전체를 비교

5. 커밋 전 변경 사항 전체 확인하기 (조합)

보통 커밋 직전 확인 루틴은 이렇게 사용합니다:

```
git status        # 어떤 파일이 변경되었는지 먼저 확인
git diff          # 수정한 파일 내용 확인
git diff --cached   # add 한 파일만의 변경 내역 확인
```

6. (보너스) 커밋 간 차이 보기

```
git diff 커밋 ID1 커밋 ID2
예:
git diff abc123 def456
```

→ 커밋들 간의 코드 차이도 볼 수 있습니다.

커밋 ID 확인 방법

◆ 1. 가장 일반적인 방법: git log

```
git log
```

예시 출력:

```
commit 74a8f3d14f8c5f567d933fdf92c416f8c89b51d9 (HEAD -> main, origin/main)
Author: Cha-joo Yoon <chaju@example.com>
Date:   Mon Apr 21 13:45:00 2025 +0900

    로그인 기능 추가 및 테스트 완료

commit d23ab7a84cebb34eac0dc38c11a6f85d1b2a1f8a
Author: ...
```

```
Date: ...

    초기 세팅 완료
```

→ 여기서 commit 뒤에 나오는 40 자리 문자열이 커밋 ID 입니다.

(보통 **앞의 7~10 자리만 입력해도 인식**됩니다)

예:

```
git diff d23ab7a 74a8f3d
```

◆ 2. 간단하게 한 줄씩 보기 (--oneline 옵션)

```
git log --oneline
```

예시 출력:

```
74a8f3d (HEAD -> main, origin/main) 로그인 기능 추가 및 테스트 완료
d23ab7a 초기 세팅 완료
```

→ **간단하고 보기 좋아서 팀에서도 자주 사용됩니다**

◆ 3. 특정 파일에 대한 커밋 기록 확인

```
git log 파일명
```

예시:

```
git log first/src/main/webapp/WEB-INF/views/test/testAjaxView.jsp
```

→ 이 파일과 관련된 커밋만 추적 가능

◆ 4. GitHub 에서 확인하는 방법

1. GitHub 저장소 접속
2. Commits 메뉴 클릭 (브랜치별 커밋 목록)
3. 각 커밋 오른쪽에 보이는 짧은 해시 클릭하면 전체 커밋 ID 확인 가능

커밋 ID 확인 후 diff 예시

```
git diff d23ab7a 74a8f3d
```

→ d23ab7a 커밋과 74a8f3d 커밋 사이에서 **어떤 코드가 어떻게 변경되었는지** 보여줍니다.

요약

명령어	의미
git diff	작업 중인 변경 내용을 확인 (add 전)
git diff --cached	스테이징된 변경 내용을 확인 (add 후)
git diff 파일명	특정 파일만 변경 내용 확인
git diff HEAD	현재 상태와 마지막 커밋 간 차이 확인

변경된 파일 목록만 git add 하는 방법

◆ 1. 변경된 파일 목록 확인

```
git status
```

예시 출력:

```
On branch chaju-login-feature
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
        modified:   login.jsp
        modified:   loginController.java

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        loginValidator.java
```

◆ 2. 원하는 파일만 개별적으로 add 하기

```
git add login.jsp
git add loginController.java
git add first/src/main/webapp/WEB-INF/views/test/testAjaxView.jsp
```

→ 이렇게 하면 특정 파일만 스테이징되고, 나머지는 아직 대기 상태입니다.

◆ 3. 폴더 단위로 add 하기 (선택적으로)

```
git add src/com/project/controller/
```

→ 해당 폴더 내의 변경 파일만 추가됨

◆ 4. 변경 중 일부 줄(Line)만 선택적으로 추가 (--patch 모드)

```
git add -p loginController.java
```

→ 이 명령은 변경된 코드 줄을 하나씩 보여주면서 스테이징할지 여부를 물어봅니다.

선택지 예:

- y: 이 변경 포함
- n: 이 변경 제외
- q: 종료
- s: 더 잘게 나누기

- ?: 도움말 보기

고급 사용자용이지만, 매우 유용한 기능입니다.

요약

명령어	설명
git add 파일명	개별 파일만 선택
git add 디렉토리명/	특정 폴더만 선택
git add -p 파일명	변경된 코드 중 줄 단위로 선택
git status	현재 변경된 파일 목록 확인
add 취소	git reset HEAD first/src/main/webapp/WEB-INF/views/test/testAjaxView.jsp

Git 에서 브랜치 병합(Merge)은 항상 전체 프로젝트 기준입니다.
Git 은 전체 파일 시스템의 변경 이력을 추적하므로, 특정 폴더에만 병합되는 기능은 기본적으로 지원하지 않습니다.

필요한 폴더만 병합하게 해 주는 명령어 (우회 방법임)

```
git checkout main
git pull origin main
git checkout chaju-login-feature -- src/main/webapp/WEB-INF/views/login/
git add .
git commit -m "login 기능만 병합"
git push origin main
```

보너스 팁: 푸쉬한 파일 하나만 원하는 프로젝트 위치에 병합되게 하는 방법

```
git checkout chaju-login-feature -- src/main/webapp/WEB-INF/views/login/login.jsp
```

Branch 만들 때 폴더와 파일 경로를 지정함 (우회방법임)

---

5 단계: 마스터가 브랜치를 병합(Merge)하는 방법 (마스터만 함)

병합이란?

팀원들이 각자 만든 브랜치의 작업 내용을 **마스터 브랜치(master/main)** 로 **합치는 작업임**

깃허브에서 병합하는 방법

1. 깃허브 웹사이트로 가기
2. Pull Request 버튼을 누르기
3. **어떤 브랜치 → master 로 병합**할 것인지 확인
4. Create pull request 누르기
5. 코드 확인 후 Merge pull request 클릭

주의사항

- 충돌(conflict)이 나면? → 코드가 서로 달라서 충돌남! 팀원과 상의해서 어느 쪽으로 할지 정하고 수정 필요
- 반드시 **코드 확인하고** 머지해야 합니다.! 에러 나면 안 되므로.

GitHub 원격 저장소에서 커밋된 브랜치의 변경된 코드를 확인하는 방법

1. [GitHub 웹사이트]에서 브랜치 변경 코드 확인하기

◆ ① **브랜치별 커밋 목록 보기**

1. GitHub 저장소 접속 → 예: https://github.com/yun-chaju/team-project
2. 상단 메뉴에서 **"Code"** 탭 선택
3. 좌측 상단의 **[main ▼] 브랜치 선택 드롭다운** 클릭
4. 확인하고 싶은 브랜치 선택 (예: joo-login-feature)
5. 상단 메뉴 → **"Commits"** 클릭
   → 해당 브랜치의 커밋 리스트 확인 가능

◆ ② **커밋 상세 보기 및 변경된 코드 확인**

- 각 커밋 오른쪽의 </> 또는 커밋 메시지를 클릭하면
- 해당 커밋에서 **변경된 파일 목록**, **추가/삭제된 코드**를 확인할 수 있음

◆ ③ **브랜치 간 변경사항(Pull Request) 비교**

1. GitHub 에서 **Pull Request** 생성 화면으로 이동
2. base: 병합 대상 브랜치 (main)
3. compare: 확인할 작업 브랜치 (예: joo-login-feature)

4. GitHub 이 자동으로 **차이(diff)** 를 보여줌
   → 파일별로 변경된 코드 확인 가능

2. [로컬 Git]에서 브랜치 변경 코드 확인하기

◆ ① 브랜치 간 변경 비교

원격 브랜치 가져오기

```
git fetch origin
```

로컬에서 두 브랜치 간 변경 내용 비교

```
git diff origin/main..origin/joo-login-feature
```

→ main 브랜치와 joo-login-feature 브랜치 사이의 코드 변경사항 출력

◆ ② 특정 커밋의 변경 코드 확인

커밋 목록 보기

```
git log origin/joo-login-feature
```

커밋 하나의 diff 보기

```
git show <커밋 ID>
예:
git show 12f8a34
```

◆ ③ 로컬 브랜치에 checkout 하여 직접 확인

```
git checkout joo-login-feature
```

이후 직접 코드 파일을 열어 비교할 수 있습니다.

요약

방법	위치	설명
GitHub 웹	브라우저	브랜치 선택 → 커밋 → 코드 변경 확인
GitHub Pull Request	브라우저	compare vs base 브랜치 비교
Git 명령어 (CLI)	로컬 터미널	git diff, git log, git show로 비교 확인

GitHub 웹사이트에서 브랜치를 병합하는 전체 흐름

◆ 1 단계: GitHub 웹사이트 접속

- 주소: https://github.com/사용자명/저장소명
- 예: https://github.com/yun-chaju/team-project

◆ 2 단계: Pull Request 버튼 클릭

- 메뉴 상단에서 **"Pull requests"** 탭 클릭
- 우측에 **"New pull request"** 버튼 클릭

◆ 3 단계: 브랜치 선택

페이지 상단의 'Comparing changes' 아래의 base 와 compare 에 브렌치 지정함

항목	설명
base	병합 대상 브랜치 (보통 main 또는 master)
compare	팀원이 작업한 브랜치 (예: joo-login-feature)

→ 예: compare: joo-login-feature → base: main

(즉, joo-login-feature 브랜치의 내용을 main 브랜치에 병합하겠다는 의미)

자주 하는 실수 방지 포인트

실수 유형	설명	해결책
base와 compare를 반대로 선택	main을 compare로, feature를 base로 선택하는 실수	항상 **base = main**, **compare = 작업한 브랜치** 로 지정
병합하려는 브랜치 선택 안 함	GitHub에서 자동 선택된 브랜치 그대로 진행	**풀다운 메뉴에서 원하는 브랜치 수동으로 선택**
아직 커밋 푸시 안됨	compare 브랜치에 푸시된 커밋이 없음	먼저 git push origin 브랜치명 으로 커밋 푸시 필요

◆ 4 단계: Pull Request 제목과 설명 작성

- **제목**: 어떤 기능을 추가했는지 간략히 작성
  예: Add login form and session handling
- **설명**: 상세한 변경사항, 목적, 테스트 결과 등을 작성

그 후, **[Create pull request]** 버튼 클릭

◆ 5 단계: 코드 확인 후 병합

- 변경된 파일(diff)을 확인하여 문제가 없는지 검토
- 리뷰어 또는 마스터가 **[Merge pull request]** 버튼 클릭
- 옵션에 따라:
  - **Create a merge commit**
  - **Squash and merge**
  - **Rebase and merge**

- 보통은 첫 번째 옵션 사용 (기본값)

◆ 6 단계: 병합 완료 후 브랜치 삭제 (선택 사항)

- 병합이 완료되면 Delete branch 버튼이 나옵니다
- 작업이 끝난 브랜치는 삭제해도 무방합니다 (로컬에는 여전히 남아 있음)

병합 작업 후 팀원이 할 일

병합이 완료되었으면, **다른 팀원들도 main 브랜치를 최신 상태로 동기화**해야 합니다:

```
# main 브랜치로 이동
git checkout main

# 원격 저장소의 최신 main 브랜치 가져오기
git pull origin main
```

요약

단계	설명
1. GitHub 접속	https://github.com/프로젝트
2. Pull Request 생성	compare → base 확인 후 생성
3. 코드 확인	변경 내용(diff) 확인
4. Merge pull request	마스터 또는 리뷰어가 병합
5. 브랜치 삭제 (선택)	병합 완료 후 정리 차원에서 삭제 가능

---

6 단계: 병합된 최신 코드 다시 받기 (팀원들)

마스터 브랜치의 최신 코드를 내 컴퓨터로 가져오는 법

```
git checkout main      # 마스터로 이동
git pull origin main      # 최신 코드 가져오기
```

그 다음 작업을 계속하고 싶다면?

```
git checkout -b 새로운브랜치
```

새로운 작업 브랜치 만들어서 또 작업!

브랜치 작업 시 주의사항

주의사항	설명
브랜치는 자주 나눠서 만들기	작업 하나에 하나씩 (예: 로그인 만들기, 회원가입 만들기 등)
커밋 메시지는 명확하게	예: "로그인 기능 추가", "DB 연결 수정" 등
브랜치 이름 규칙 정하기	예: 이름-작업내용 (joo-login, min-signup 등)
마스터에서 직접 작업 ❌	마스터는 최종본만 있어야 해요! 안전하게 관리하기 위해
Pull 받기 전에는 커밋 먼저	로컬에서 수정 중일 때 git pull 하기 전에 꼭 commit 해야 충돌 방지됨

로컬 저장소와 스프링 워크스페이스가 다른 경우 최신 코드를 받아서 실제 Spring 프로젝트(워크스페이스)에 정상 적용하는 방법

상황 예시

- Git 저장소 경로: /d/git_local_repository/team-project
- Spring 워크스페이스 경로: /d/spring_workspace/team-project

즉, **코드 푸시는 Git 저장소에서 진행**,

하지만 **Spring Tool Suite(STS4)는 워크스페이스 폴더에서 프로젝트 실행 중**이라는 의미입니다.

선택지 1: 가장 권장되는 방식 (워크스페이스 자체가 Git 저장소이도록 하기)

Spring 프로젝트 폴더를 Git 저장소로 사용하는 방식입니다.

◆ **장점**

- 별도 복사 필요 없음
- 브랜치 작업, pull/push 후 즉시 프로젝트에 반영됨

방법

1. GitHub 에서 clone 시, 워크스페이스에 직접 clone

```
cd /d/spring_workspace
```

```
git clone https://github.com/사용자명/team-project.git
```

2. STS4 에서 File > Import > Existing Maven/Gradle Project 로 가져오기
   → **/spring_workspace/team-project 폴더 직접 지정**

이렇게 하면 Git 과 프로젝트가 **동일한 폴더**에서 관리되어 충돌 없이 동기화됩니다.

선택지 2: 현재처럼 Git 저장소와 워크스페이스가 분리된 경우

이 경우엔 Git 저장소에서 최신 코드를 가져온 뒤, 워크스페이스에 복사해야 합니다.

단계별 절차

① Git 저장소에서 최신 코드 가져오기

```
cd /d/git_local_repository/team-project
git checkout main
git pull origin main
```

② 최신 코드를 워크스페이스로 복사

예시: 기존 워크스페이스에 덮어쓰기

```
xcopy /E /Y /I /D "D:₩git_local_repository₩team-project" "D:₩spring_workspace₩team-project"
```

.git 폴더는 복사하지 마세요. 워크스페이스가 Git 저장소로 오해할 수 있습니다.

③ STS에서 새로고침

- STS4 에서 F5 (또는 프로젝트 우클릭 → Refresh)
- Maven > Update Project 수행 (단축키 Alt+F5)
  → 최신 소스 코드 적용 완료

팁: .git 폴더 복사 방지

복사할 때 .git/ 폴더는 포함하지 마세요. 그렇지 않으면 워크스페이스가 또 다른 Git 저장소로 인식됩니다.

```
robocopy "D:₩git_local_repository₩team-project" "D:₩spring_workspace₩team-project" /E /XD .git
```

✓ 전체 요약

역할	할 일 요약
마스터	저장소 만들기, 브랜치 머지, 충돌 해결
팀원	클론 → 브랜치 만들기 → 작업 → 커밋 → 푸시
모두	병합 후 pull 해서 최신 코드 받아오기

[Spring Tool Suite 4 (STS4) 안에서 스프링 레거시(Spring Legacy) 프로젝트를 Git 으로 버전관리하는 방법]

✔ **준비사항**

- STS4 설치 완료
- Git 설치 완료
- GitHub 계정 준비 완료

---

1 단계: STS 에서 Git 연동을 위한 기본 세팅

1. **STS4 실행**
2. 작업 중인 **프로젝트를 열기**
3. 상단 메뉴에서 Window > Show View > Other... 선택
4. 검색창에 **Git** 입력 → Git Repositories, Git Staging 등을 선택해서 창을 열기
   → 하단에 Git 관련 뷰가 생김

2 단계: 로컬 Git 저장소 만들기 (Git 초기화)

1. **패키지 탐색기(Project Explorer)** 에서 프로젝트 폴더 우클릭
2. Team > Share Project... 클릭
3. Git 선택하고 Next
4. Create 클릭해서 새로운 Git 저장소 생성
   - 보통 .git 폴더가 프로젝트 안에 생김
5. Finish 클릭

☞ 이제 프로젝트가 Git 과 연결됨!

3 단계: .gitignore 설정 (꼭 필요!)

Git 에 올리지 않을 파일들을 정함 (예: target, .classpath, .settings 등)

1. 프로젝트 루트에 .gitignore 파일 만들기
2. 아래 내용 추가:

```
# Eclipse 관련 설정 파일
/.settings/
.project
.classpath

# IntelliJ 관련 (공동 사용 시 포함)
.idea/
```

```
*.iml

# 빌드 결과물
/target/
bin/
/build/

# 로그파일
*.log

# 운영체제별 불필요 파일
.DS_Store
Thumbs.db

# 환경 변수 설정 파일 (필요시)
*.env
application-*.properties
application-*.yml
pom.xml
*.xml
```

이 파일은 수동으로 텍스트 편집기로 만들거나 STS 에서 생성 가능

Spring Legacy MVC 프로젝트를 팀원들과 함께 Git 으로 버전 관리할 때, **불필요한 파일이 올라가지 않도록 .gitignore 파일을 잘 설정하는 것이 매우 중요**합니다.
이 설정이 잘못되면 *.class, .settings, target/, .iml 등 프로젝트에 불필요하거나 시스템별로 다른 파일들이 깃허브에 올라가 협업 시 문제를 일으킬 수 있습니다.

✓ .gitignore의 역할

Git 이 무시할 파일 또는 폴더를 지정하는 파일
개발 중 생성되지만 **공유하지 않아야 할 파일**이나 **환경별로 달라지는 설정 파일** 등을 Git 에 포함되지 않게 합니다.

1. Spring Legacy 프로젝트에서 .gitignore 에 포함해야 할 대표 항목

다음은 Spring Legacy MVC 프로젝트에서 꼭 .gitignore 에 추가해야 하는 내용입니다:

```
# Eclipse 관련 설정 파일
```

```
.settings/
.project
.classpath

# IntelliJ 관련 (공동 사용 시 포함)
.idea/
*.iml

# 빌드 결과물
target/
bin/
build/

# 로그파일
*.log

# 운영체제별 불필요 파일
.DS_Store
Thumbs.db

# 환경 변수 설정 파일 (필요시)
*.env
application-*.properties
application-*.yml
pom.xml
*.xml

# 기타
.vscode/
```

항목별 이유 설명

항목	이유
.settings/, .project, .classpath	Eclipse에서 자동 생성되는 설정 파일 (로컬 환경마다 다름)
.idea/, *.iml	IntelliJ 사용 시 생성되는 설정 (공유 필요 없음)
target/, bin/, build/	컴파일 결과물 (빌드마다 재생성됨, 저장소에 불필요)
*.log	로그 파일은 실행 시마다 생기고 불필요함

항목	이유
.DS_Store, Thumbs.db	Mac/Windows 운영체제에서 자동 생성됨
.env, application-*.yml	로컬 또는 운영 환경에 따라 달라지는 설정파일 (보안 위험 가능)

2. .gitignore 생성 및 적용 방법

2.1 .gitignore 파일 생성

1. 프로젝트 루트 디렉토리에 New > File → 이름을 .gitignore 로 생성
2. 위 내용을 복사해서 붙여넣기

2.2 이미 Git 에 올라간 파일 무시 처리

.gitignore 에 추가했지만 이미 Git 에 추적된 파일이라면 다음 명령어로 캐시를 제거해야 합니다:

```
git rm -r --cached .
git add .
git commit -m " .gitignore 적용 및 불필요 파일 제거"
```

→ 이제 .gitignore 이 제대로 적용되어 추적이 제거되고, 이후부터 무시됩니다.

3. 협업 중 .gitignore 주의사항

주의사항	설명
초기부터 .gitignore을 설정해야 함	추적된 뒤엔 무시되지 않으므로 먼저 설정해야 함
팀원 간 IDE 설정 공유 ✕	Eclipse, IntelliJ 설정 파일은 로컬에서만 유지
target/은 절대 올리면 안 됨	자동 생성된 바이너리 파일로, 충돌 위험 있음
.gitignore도 Git에 포함됨	팀원 간 같은 무시 규칙을 공유하게 됨

보너스: 템플릿 .gitignore 예제 (Spring + Eclipse)

```
# Eclipse
.settings/
.classpath
.project

# IntelliJ
.idea/
*.iml

# Maven
target/
```

```
pom.xml

# Logs
*.log

# OS files
.DS_Store
Thumbs.db

# IDEs
.vscode/

# 환경설정
*.env
application-*.yml
application-*.properties
*.xml
```

이 파일은 프로젝트 루트에 넣으면 Git 이 자동으로 읽고 무시합니다.

필요하다면 .gitignore 자동 생성 사이트도 있어요:
https://www.toptal.com/developers/gitignore

4 단계: GitHub 원격 저장소 연결하기 (Remote 설정)

1. 깃허브에서 새 저장소(repository) 생성 (https://github.com/계정명/저장소명.git)
2. STS 하단 Git Repositories 뷰에서:
    - 우클릭 → Remotes > Create Remote
    - 이름은 origin 으로 설정
    - Configure Push 체크 → Next
    - URL 입력 (예: https://github.com/yourname/my-spring.git)
    - User 와 Password 는 GitHub 계정 정보 (또는 Personal Access Token)
    - Finish

5 단계: Git 으로 커밋(Commit)하기

1. STS 하단 Git Staging 뷰에서:
    - **Unstaged Changes** 창에서 변경된 파일 확인
    - 파일을 선택하고 + 버튼 → **Staged Changes** 로 이동
2. **Commit 메시지** 입력 (예: "처음 커밋")

3. Commit 클릭 (혹은 Commit and Push 바로 사용)

6 단계: GitHub 로 푸시(Push)하기

- 커밋 후 Commit and Push 를 안 했다면:
    1. STS 하단 Git Repositories 뷰에서 저장소 우클릭 → Push
    2. 브랜치 선택 (master 또는 main)
    3. Next > Finish

☞ 이제 GitHub 에서 프로젝트를 확인할 수 있어요!

7 단계: 브랜치 만들고 작업하기

1. STS 에서 Git Repositories 뷰 열기
2. 저장소에서 우클릭 → Switch To > New Branch
3. 브랜치 이름 입력 (예: login-feature)
4. Checkout new branch 체크 → Finish

새로운 브랜치에서 작업 후 커밋/푸시하면 GitHub 에도 브랜치가 생김

8 단계: Pull Request & Merge (웹에서 진행)

STS 에서 머지는 안 되고, **GitHub 웹사이트**에서 Pull Request 를 생성해 머지 가능

9 단계: 팀원과 협업할 때 Pull 로 최신 코드 받기

1. Git Repositories 뷰에서 원격 브랜치 우클릭 → Pull
2. 변경된 내용이 로컬로 내려옴

중요: Pull 전에 변경사항 있으면 반드시 커밋 먼저 해야 충돌 방지

주의사항 요약

항목	설명
.gitignore 꼭 설정	불필요한 파일이 올라가는 것 방지
마스터 브랜치 직접 수정 ❌	브랜치를 만들어서 수정 후 머지
커밋 자주 하기	변경 이력을 잘 남기기 위해
커밋 메시지는 명확하게	"버그 수정", "기능 추가" 등으로 간결하게
충돌 방지를 위해 자주 Pull	다른 사람이 수정한 코드 놓치지 않기 위해

[Spring Tool Suite 4 (STS4) 환경에서 마스터(팀장)가 GitHub 에 올린 프로젝트를 팀원이 받아서, 수정하고 다시 올리는 전체 흐름 설명]

1 단계: GitHub 에서 프로젝트 가져오기 (Clone) (팀원)

1.1 GitHub 저장소 주소 복사

- 마스터가 만든 저장소 페이지로 이동
- Code 버튼 클릭 → HTTPS 주소 복사 (예: https://github.com/teamleader/spring-project.git)

1.2 STS4 에서 Clone 하기

1. STS4 실행
2. 상단 메뉴에서 File > Import 클릭
3. Git > Projects from Git 선택 → Next
4. Clone URI 선택 → Next
5. 복사한 URL 붙여넣기
   → 나머지 정보 자동 완성됨 (계정은 GitHub 로그인 필요 시 입력)
6. Next 클릭 → 브랜치 선택 (보통 master 혹은 main) → Next
7. 저장할 로컬 경로 선택 → Next
8. Import as general project 선택 → Next
9. Finish

1.3 Spring 프로젝트로 변환

1. 프로젝트에서 우클릭 → Configure > Convert to Maven Project 또는 Convert to Spring Project
2. 제대로 인식되면 정상 완료!

2 단계: 브랜치 생성 및 작업 환경 준비 (팀원)

작업은 **항상 새로운 브랜치**에서 진행합니다!

2.1 브랜치 만들기

1. 프로젝트에서 우클릭 → Team > Switch To > New Branch
2. 브랜치 이름 작성 (예: login-feature)
3. Checkout new branch 체크 → Finish

✔ 이 브랜치에서 작업하게 됩니다.

3 단계: 코드 수정 및 커밋 (팀원)

3.1 코드 수정

- 필요한 기능을 개발하거나 기존 코드 수정

3.2 Git Staging 뷰에서 커밋하기

1. 하단 Git Staging 탭 클릭
2. **Unstaged Changes** 영역에서 수정된 파일을 확인
3. + 버튼 클릭해서 **Staged Changes** 로 이동
4. 커밋 메시지 작성 (예: 로그인 기능 구현 완료)
5. Commit 또는 Commit and Push 클릭

4 단계: GitHub 로 푸시 (Push) (팀원)

커밋만 했다면 별도로 푸시 필요!

1. 프로젝트에서 우클릭 → Team > Push Branch
2. 원격 저장소 선택 (보통 origin)
3. 브랜치 이름 확인 후 Next → Finish

이제 GitHub 에 브랜치가 업로드 됐습니다!

GitHub 로그인 창이 나타났을 때 해결 방법

원격 저장소로 push 할 때 로그인 창이 나타나는 것은 windows 에서 github 접속을 위한 자격 인증을 요구하는 것입니다. 각자 본인의 GitHub 로그인 아이디와 각자 발급받은 토큰을 패스워드에 입력하면 됩니다.

🔥 **핵심 원인**

❗ GitHub 은 일반 비밀번호 인증을 더 이상 허용하지 않습니다.

즉, 로그인 창이 뜨더라도:

GitHub 비밀번호를 입력하면 실패합니다.

✅ 대신 **"Personal Access Token (PAT)"을 비밀번호 칸에 입력**해야 합니다.

해결 방법: Personal Access Token (PAT) 발급 → 비밀번호 대신 입력

◆ 1. GitHub 에서 토큰 생성

1. GitHub 로그인 → 오른쪽 상단 프로필 클릭 → **Settings**
2. 좌측 하단에서 **"Developer settings"** 클릭

3. 좌측 메뉴에서 **"Personal access tokens" → "Tokens (classic)"** 클릭
4. **[Generate new token]** 클릭
   (토큰 종류는 classic 또는 fine-grained 가능)

◆ 2. 토큰 생성 시 옵션 예시

- **Note**: STS Git Push
- **Expiration**: 90 일 또는 원하는 기간
- **Scope(권한)**:
  - ✅ repo (저장소 전체 접근 허용)
  - ✅ workflow (워크플로 허용 필요 시)
- 생성 후 **한 번만 보여지는 토큰 문자열 복사** (중요!)

◆ 3. STS Git Push 시 사용

- 로그인 창에서:
  - **Username**: GitHub 아이디
  - **Password**: 생성한 토큰 (복붙)

✔ 이후부터 정상적으로 Push 됩니다.

STS 에서 한번 저장되면 이후 로그인 창 안 뜰 수도 있습니다
그래도 windows 에서 '**자격 증명 관리자**'나 sts4 안의 **Preferences > Git** 에서 확인 가능합니다.

---

5 단계: GitHub 웹사이트에서 Pull Request 만들기 (팀장이 함)

1. GitHub 저장소 페이지로 이동
2. 방금 푸시한 브랜치가 보일 것
3. Compare & pull request 버튼 클릭
4. 머지 설명 작성 → Create pull request
5. 마스터(팀장)가 검토하고 Merge pull request 클릭하면 병합 완료

6 단계: 병합된 최신 코드 받아오기 (Pull)

팀원들 모두 해야 할 일!

1. 브랜치 병합이 완료되면:
2. STS4 에서 master 브랜치로 전환
   - Team > Switch To > master
3. Team > Pull 클릭

이제 병합된 최신 코드가 로컬에도 반영됩니다.

Git 은 브랜치 기반으로 동작하므로,

현재 브랜치(main)에서 git pull 을 해도,

다른 브랜치(feature, login-feature 등)에는 영향이 없습니다.

예를 들어

팀원 상태:

- 현재 작업 중인 브랜치: chaju-login-feature
- 작업 중 파일 일부 수정 완료

```
git checkout ycj-login-feature
```

작업 중...

이때 main 브랜치 최신 코드가 궁금해서 아래를 실행:

```
git checkout main
git pull origin main
```

→ 결과:

- main 브랜치의 최신 코드가 내 로컬로 내려옴
- 현재 작업 중인 ycj-login-feature 브랜치는 **아무 영향 없음**
- 작업 내용이 사라지거나 덮어쓰이지 않음

다시 작업 브랜치로 돌아가려면

```
git checkout ycj-login-feature
```

만약 최신 main 내용을 내 작업 브랜치에도 반영하고 싶다면

```
git checkout ycj-login-feature
git merge main
# 또는
git rebase main
```

→ 이건 **병합 또는 재정렬 작업이므로**, 충돌 가능성도 있으니 주의해야 합니다.

요약

작업	영향
main 브랜치에서 git pull	안전, 다른 브랜치 영향 없음
작업 중인 브랜치로 복귀	git checkout 브랜치명
작업 브랜치에 최신 main 반영	git merge main 또는 git rebase main

전체 흐름 요약 (정리)

단계	설명
1단계	GitHub에서 Clone으로 프로젝트 가져오기
2단계	브랜치 만들어서 본인 작업 환경 만들기
3단계	코드 수정 후 커밋하기
4단계	원격 저장소에 푸시(Push)
5단계	GitHub에서 Pull Request 생성, 마스터가 Merge
6단계	팀원들은 Pull로 최신 코드 반영하기

주의사항 요약

주의사항	설명
master 브랜치에서 직접 수정 ❌	항상 새로운 브랜치에서 작업
푸시 전에 **커밋 필수**	변경사항이 반영되지 않으면 푸시 안 됨
Pull 전에는 **커밋 먼저**	로컬 변경사항이 있을 경우 충돌 방지
브랜치 이름 규칙 정하기	예: 이름-기능 (e.g., joo-login)
자주 Pull 하세요	다른 사람 작업 반영 잊지 말기

브랜치 선택 시 꼭 master 나 main 을 선택해야 하는가?

반드시 master 또는 main 을 선택해야 하는 건 아닙니다.

하지만 **보통은 master 나 main 브랜치가 기준(기준선, mainline)이 되므로 선택하는 게 일반적입니다**.

보통 master 또는 main 을 기준으로 작업하는 이유

이유	설명
기준 브랜치 역할	모든 팀원이 공통으로 사용하는 안정적인 버전이 보통 master 나 main 이에요.
브랜치 파생 용이	새로운 기능 작업 시 master 를 기준으로 브랜치 만들어야 병합 시 충돌이 적어요.
병합(Merge) 목적	대부분의 Pull Request 는 master 로 머지되기 때문에 기준이 중요합니다.

어떤 브랜치를 기준으로 작업 브랜치를 만들 수 있나?

- 아래 브랜치들을 기준으로 작업할 수 있습니다:

상황	기준 브랜치
일반 개발	master 또는 main (가장 보편적)
대규모 기능 개발 중	develop, feature-login, feature-payment 등
긴급 수정	hotfix-버그명 같은 별도 브랜치에서

상황	기준 브랜치
실험용 코드	test-기능명, experiment-브랜치명 등

단, 팀 내에서 "어떤 브랜치를 기준으로 작업할지 규칙을 정해두는 것"이 중요합니다!

GitHub 에서 프로젝트를 **Clone** 할 때 **"저장할 로컬 경로 선택"** 부분은, 말 그대로 **내 컴퓨터의 어디에 프로젝트를 복사해 둘지 정하는 단계**입니다.

✅ 어디를 지정하면 좋을까?

□ 일반적으로 추천하는 위치:

C:₩Users₩사용자이름₩workspace₩

또는

D:₩git-projects₩

- 예시:

D:₩git-projects₩team-spring-project

폴더 선택 시 고려할 점

항목	설명
📌 프로젝트 모아두는 폴더에 저장하기	예: workspace, git-projects, spring-projects 등
❌ 한글 경로 지양하기	경로에 한글이 포함되면 빌드 도중 오류 날 수 있음
❌ 공백 포함 지양하기	예: My Documents → 대신 MyDocuments 또는 다른 폴더 사용
✅ 깊이 있는 폴더 지양	너무 긴 경로는 컴파일러에서 문제 생길 수 있어요 (C:₩Users₩너무₩길게₩폴더₩이름₩지정₩하지₩마세요₩...)

팁: STS 의 Workspace 와 프로젝트 위치는 달라도 괜찮습니다.

- STS 의 Workspace 는 IDE 설정 등이 저장되는 폴더이고,
- Git 프로젝트는 별도 폴더에 있어도 문제 없습니다.
- Clone 후 Import 할 때 "existing project into workspace"로 불러오면 돼요.