ChatGPT

GitHub Desktop 기반 실전 소스/배포 관리 프로세스 개선안

우리 회사의 현재 개발/테스트/운영 프로세스에서는 **모든 새로운 기능을 한데 모아 테스트 서버에 배포**하고, 그 중 일부 기능만을 운영 서버에 수동 반영하는 방식을 취하고 있습니다. 이로 인해 **테스트 서버에 포함되었지만 운영 배포 대상이 아닌 코드가 운영으로 유입**되어 서비스 오류가 발생하거나, 운영 배포 시 **원치 않는 소스를 수동으로 제거하다 실수**가 발생하는 문제가 있었습니다. 또한 브랜치 병합, 충돌 해결, 롤백 등의 Git 활용 미숙으로 협업에 어려움이 있었습니다.

위 문제들을 해결하기 위해 **Git 브랜치 전략을 재정의하고 GitHub Desktop을 활용한 단계별 협업/배포 프로세스**를 마련하였습니다. 초보자도 따라할 수 있도록 **그림과 함께 상세 단계**를 정리하였으니 참고하시기 바랍니다.

1. 새로운 Git 브랜치 전략 개요 (feature/develop/main 분리)

① **브랜치 종류 및 역할**: 현재 `develop` 브랜치에 모든 기능이 섞여 있는 문제를 해결하기 위해, **Git Flow** 방식의 브랜치 전략을 도입합니다. 주요 브랜치는 다음과 같습니다.

- **main 브랜치**: **운영 배포용 주요 브랜치**입니다. 운영 서버에 반영된 **검증된 코드만** 모입니다. 항상 **운영 서버의 상태와 동일**하게 유지하며, 안정적인 코드(history)를 관리합니다.
- **develop 브랜치**: **통합 개발(branch)** 브랜치로, 개발된 **모든 새로운 기능을 테스트하기 위한 코드**를 모읍니다. 개발자들은 각 기능을 develop에 병합하여 **테스트 서버에 배포**하고, 통합 테스트를 진행합니다.
- **feature 브랜치들**: 각 기능별로 분기한 **개발용 브랜치**입니다. 새로운 기능이나 버그 수정 작업은 main이나 develop에서 분기(branch)하여 **독립된 feature 브랜치**에서 진행합니다. 개발 완료 후 해당 feature 브랜치를 **develop에 먼저 병합**하여 통합 테스트를 거칩니다.
- **(필요시) release 브랜치**: 여러 기능을 묶어 한번에 운영 배포할 때 사용합니다. 예컨대 이번 배포에 포함될 기능들이 develop에서 충분히 테스트되었다면, develop에서 **release 브랜치**를 분기하여 최종 안정화 과정을 거친 뒤 main에 병합합니다. 소규모 팀이라면 release 브랜치 없이 곧바로 develop에서 필요한 커밋만 main으로 선택(cherry-pick)하는 방법도 있습니다.
- **(필요시) hotfix 브랜치**: 운영 중 긴급 수정 사항 발생 시 사용하는 브랜치입니다. main에서 분기하여 수정 후 다시 main에 병합하고, develop에도 반영합니다 (이번 주제에서는 주로 feature/develop/main만 다룹니다).

② **브랜치 흐름 정리**: 위 구조를 따르면 **기능 개발 → 테스트 서버 통합 → 운영 반영** 흐름이 명확히 분리됩니다. 아래는 간단한 흐름도입니다.

- feature 브랜치 (기능A, B 등) → **develop 브랜치** (통합 후 테스트 서버 배포) → **main 브랜치** (운영 서버 배포)

이렇게 하면, **테스트 서버**에는 여러 신규 기능을 모두 올려서 고객이 테스트하더라도, **main(운영)**에는 그 중 승인된 기능만 선택하여 배포할 수 있습니다. 아직 검증되지 않았거나 승인 보류된 기능 코드는 main에 병합하지 않으면 되므로, **원치 않는 코드가 운영에 섞여들 위험이 감소**합니다. 또한 수동으로 WAR에서 코드를 제거하는 등의 작업 없이 **Git 브랜치로 포함/제외를 제어**할 수 있어 실수를 줄입니다.

2. GitHub Desktop을 활용한 브랜치/커밋/병합 관리 – 실무 매뉴얼

이제 GitHub Desktop을 기준으로 **실제 작업 단계**를 살펴보겠습니다. 아래 단계에서는 **GitHub Desktop UI 스크린샷** 예시와 함께, 브랜치 생성, 커밋, 푸시(push), 병합(merge) 등의 작업 방법을 상세히 설명합니다.

2-1. GitHub Desktop 설정 및 기본 화면 이해

- **설치 및 연결**: GitHub Desktop을 설치하고 GitHub 계정/저장소(repository)에 연결합니다. 처음 실행하면 로컬 경로에 저장소를 Clone하거나 추가할 수 있습니다. 이미 Git으로 관리 중인 프로젝트라면 GitHub Desktop에서 **"Open a repository from your computer"** 메뉴로 해당 폴더를 열어버전 관리를 시작합니다.
- **기본 화면 구성**: 좌측 상단에는 현재 선택된 **브랜치명**이 표시됩니다. 우측에는 **변경 파일 목록**과 **diff 뷰어**가 있어, 수정된 내용을 한눈에 볼 수 있습니다. 아래쪽 **커밋 메시지 입력칸**에서 메시지를 적고 `Commit to <branch>` 버튼을 누르면 현재 변경분을 해당 브랜치에 커밋할 수 있습니다. 상단에는 **Fetch origin / Push / Pull** 버튼이 있어 원격과 동기화합니다.

2-2. 새로운 기능 개발 – Feature 브랜치 만들기, 작업, 커밋, 푸시

Step 1: Feature 브랜치 생성 및 전환
1. GitHub Desktop 좌측 상단 **브랜치 드롭다운**을 클릭하면 브랜치 목록이 표시됩니다. 여기서 **"New Branch"** 버튼을 눌러 새로운 브랜치를 만듭니다.
 기준(Base) 브랜치 선택: 새로운 feature 브랜치는 **develop 브랜치로부터 분기**하는 것이 일반적입니다. 대화창에서 "Create branch from:" 항목이 있다면 `develop`을 선택하고 브랜치 이름을 입력합니다 (예: `feature/login-api`, 기능명으로 명명).
2. **브랜치 생성 확인**: 새 브랜치가 생성되면 GitHub Desktop 좌측 상단에 현재 브랜치가 해당 이름으로 바뀝니다 (예: `feature/login-api`). 이제부터 이 브랜치에서 하는 모든 커밋은 develop과 분리되어 독립적으로 관리됩니다.

Step 2: 코드 작업 후 커밋(Commit)
1. STS4(Eclipse) IDE에서 해당 기능 구현 코딩을 합니다. 파일을 생성/수정한 뒤 저장하면, GitHub Desktop 화면에 **Changes** 목록이 실시간으로 표시됩니다.
2. **변경 내용 확인**: GitHub Desktop에서 각 파일을 클릭하면 오른쪽에 변경된 내용(diff)이 하이라이트되어 보여집니다. 초록색(+)은 추가된 코드, 빨간색(-)은 삭제된 코드를 의미합니다.
3. **스테이징 & 커밋**: GitHub Desktop에서는 변경된 파일이 자동으로 스테이징(staged)되므로 추가 단계 없이 바로 커밋이 가능합니다. 하단에 **커밋 메시지**를 간단하고 명확하게 작성합니다 (예: `feat: 로그인 API 구현 완료`). 그 후 `Commit to feature/login-api` 버튼을 클릭하면 해당 변경사항이 로컬 저장소에 커밋됩니다.
- Tip: 커밋 메시지는 나중에 협업자가 변경 이력을 이해하는 데 중요합니다. "무엇을, 왜" 수정했는지 요약해서 적어주세요.
4. **푸시(Push)**: 커밋 후 상단의 `Push origin` 버튼이 활성화됩니다. 이를 눌러 로컬 커밋을 원격 GitHub 저장소의 해당 feature 브랜치로 업로드합니다. 푸시가 완료되면 **GitHub 서버에도 동일한 브랜치와 커밋 내역이 생성**됩니다.

GitHub Desktop에서 **새로운 브랜치를 생성**하고 현재 브랜치를 전환하는 화면 예시입니다. 상단 바의 브랜치 목록에서 `New Branch`를 눌러 분기하며, Base를 `develop`으로 선택합니다.

2-3. 기능 완료 후 통합 – Develop 브랜치로 병합(Merge) 및 테스트 서버 배포

여러 개발자들이 각자 feature 브랜치에서 작업을 마치면, 이제 **develop 브랜치로 코드를 모으는 작업**이 필요합니다. develop은 테스트 서버로 배포될 코드의 집합이므로, 모든 완료된 기능이 이 브랜치에 병합되어야 합니다. GitHub Desktop에서 **로컬 병합**하거나, GitHub에서 **Pull Request**를 생성해 병합하는 두 가지 방법이 있습니다.

방법 A: GitHub Desktop에서 로컬 병합하기 (간단한 협업인원일 경우)
1. **develop 브랜치 최신화**: 먼저 GitHub Desktop 좌측 상단에서 `develop` 브랜치를 선택하여 체크아웃(checkout)합니다. 원격에 다른 개발자가 푸시한 변경이 있을 수 있으므로 **Fetch/Pull**을 눌러 최신 상태를 가져옵니다.
2. **병합 실행**: 메뉴에서 **Branch → Merge into Current Branch…**를 선택합니다. 나타나는 브랜치 목록에서 병합하고자 하는 대상(예: `feature/login-api`)을 선택하면, 현재 체크아웃된 `develop` 브랜치로 해당 feature 브랜치의 커밋들이 병합됩니다.

- 만약 Git 충돌(conflict)이 없다면 바로 병합 완료 메시지가 GitHub Desktop에 나타나고, develop 브랜치의 최신 커밋 히스토리에 feature 브랜치의 커밋들이 합쳐집니다.
- **충돌 발생 시**: GitHub Desktop이 충돌 파일 목록을 보여주며, 해당 파일 옆에 "Conflict" 표시가 나타납니다. 이 경우 **외부 도구를 이용해 충돌을 수동으로 해결**해야 합니다 (아래 "2-4. 충돌 해결" 참조).

1. **병합 결과 커밋**: 충돌이 없었다면 병합 작업 자체가 하나의 새 커밋으로 기록될 수 있습니다('Merge branch feature/login-api into develop' 형태). 이 커밋을 푸시(Push)하여 원격 develop 브랜치에도 반영합니다.

방법 B: Pull Request를 통한 병합 (GitHub에서 코드리뷰를 거치고 싶을 경우)
1. **GitHub에서 PR 생성**: feature 브랜치를 원격 저장소에 푸시한 후, GitHub 웹사이트에서 **Pull Request(PR)**를 생성합니다. PR 생성시 `base:`를 `develop`, `compare:`를 해당 feature 브랜치로 설정합니다.
2. **코드 검토 및 머지**: PR 화면에서 변경 코드를 리뷰하고, 승인되면 **Merge** 버튼을 눌러 develop에 병합합니다. (옵션으로 **Squash and merge**나 **Rebase and merge** 등을 선택 가능하지만, 초보자는 기본 merge commit 방법을 권장).
3. **로컬 동기화**: PR로 병합 후에는 GitHub Desktop에서 `develop` 브랜치를 Fetch/Pull하여 최신 상태(원격에서 병합된 커밋)를 가져옵니다.

테스트 서버에 배포: develop 브랜치에 기능 병합이 모두 끝나고 나면, 이 시점의 develop 코드를 테스트 서버에 배포합니다. **WAR 빌드 & 업로드 절차**는 다음과 같습니다.

- **WAR 파일 빌드**: STS4(Eclipse)에서 **Maven Build**를 실행하거나, 프로젝트 `pom.xml`이 설정되어 있다면 `mvn package`로 WAR파일을 생성합니다 (예: `target/myproject-1.0.0.war`). 환경에 따라 STS 내 **Export > WAR File** 기능으로 생성할 수도 있습니다.
- **테스트 서버 업로드**: 사내 정책상 자동 배포가 불가능하므로, 개발자는 원격접속(VDI 등)을 통해 **테스트용 Tomcat (DEV 톰캣)** 서버에 접속합니다. Tomcat의 `webapps` 폴더 아래 기존 테스트용 디렉토리를 제거하거나 백업한 후, 새 WAR 파일을 복사합니다. Tomcat을 재기동하거나 **Hot Deploy** 설정 시 자동 전개를 기다립니다.
- **기능 통합 테스트**: 테스트 서버 도메인으로 접속하여 새로운 기능들이 잘 동작하는지, 기존 기능에 영향은 없는지 **클라이언트와 함께 확인**합니다. 여러 기능이 함께 올라갔다면, 각각 정상 동작 여부를 확인하고 문제가 발견되면 해당 브랜치로 돌아가 수정 후 다시 develop에 병합하는 과정을 반복합니다.

  예시 시나리오: 개발자 A와 B가 각각 `feature/결제기능`과 `feature/상품검색` 브랜치에서 작업했다고 가정해봅시다. A, B는 각자 완료 후 자신의 브랜치를 원격에 푸시하고, GitHub Desktop에서 develop 브랜치로 병합합니다. 이제 develop 브랜치에는 두 기능의 코드가 모두 포함되어 있고, 이 코드를 기반으로 WAR를 만들어 테스트 서버에 배포합니다. 클라이언트는 결제 기능과 상품 검색 기능을 테스트 서버에서 모두 시험합니다.

2-4. 운영 서버 배포 – main 브랜치 선별 배포 및 WAR 릴리스

테스트 완료 후 **운영 서버에 실제 배포**할 단계입니다. 핵심은 **운영에 넣을 기능만 main 브랜치에 반영**하는 것입니다. 만약 이번 배포에서 일부 기능은 제외하기로 했다면, develop에는 있어도 main에는 병합하지 않는 방식으로 관리합니다.

Step 1: 운영 배포 대상 선정 및 main 브랜치 병합
1. **운영 반영 대상 결정**: 테스트 결과 클라이언트가 "결제 기능만 우선 운영에 반영하고, 상품검색은 다음 배포에 올리자"고 결정했다고 해봅시다. 이 경우 **결제 기능의 커밋만 main으로 반영**하고, 상품검색 관련 커밋은 main에 반영하지 않습니다.
2. **main 브랜치 업데이트**: GitHub Desktop에서 `main` 브랜치를 체크아웃하고 최신 상태로 Pull 받습니다. 이후 **병합 방법**은 두 가지가 있습니다. - 방법 A: 기능 브랜치 직접 병합: 아직 **feature 브랜치가 삭제되지 않았다면**, 해당 브랜치를 main에 직접 병합할 수 있습니다. `main` 브랜치 선택 → Branch 메뉴의 **"Merge into Current Branch..."** → `feature/결제기능` 선택 → 병합. 이러면 결제 기능만 main에 들어갑니다. 상품검색 브랜치는 병합하지 않고 대

기시킵니다.
- 방법 B: 커밋 체리픽(cherry-pick): feature 브랜치가 여러 개 커밋으로 구성되었거나 이미 develop에 합쳐진 후라면, **체리픽** 기능으로 특정 커밋만 main에 적용할 수 있습니다. GitHub Desktop에는 직접적인 체리픽 버튼은 없으므로, 다음과 같이 수행합니다: - GitHub에서 해당 커밋을 확인하여 커밋 해시(hash)를 복사 → 로컬 터미널에서 `git cherry-pick <커밋해시>` 실행 → GitHub Desktop에 돌아와 보면 main 브랜치에 해당 커밋이 추가된 것을 확인 → Push로 원격 반영.
- (만약 CLI 사용이 어려우면, 임시로 main에서 새로운 브랜치 만들어 해당 커밋의 변경분을 수동 적용 후 커밋하는 방식도 있습니다. 하지만 실수 위험이 크므로 가능하면 cherry-pick이나 PR을 활용하세요.)
- 방법 C: Pull Request 이용: GitHub 웹에서 `compare:`를 `main`, `base:`를 `feature/결제기능`으로 PR을 만든 후 Merge 하는 방법도 있습니다. 소규모 프로젝트에서는 간단히 로컬에서 병합해도 무방합니다.

1. **main 브랜치 배포 버전 준비**: main에 필요한 커밋이 모두 반영되었다면, 이 시점의 main 브랜치 코드를 기반으로 **운영 배포 WAR 파일**을 빌드합니다. (`mvn package` 또는 STS Export 기능 활용)
2. Tip: **버전 태깅(tagging)** – 운영 배포 시점에 Git 태그를 달아두면 나중에 해당 배포 버전을 쉽게 참조할 수 있습니다 (예: `v1.2.0-prod`). GitHub Desktop에서는 직접 태그 생성은 지원하지 않으므로, GitHub 웹 UI에서 Release를 생성하며 태그를 달거나 CLI로 `git tag` 명령을 사용할 수 있습니다.

Step 2: 운영 서버에 WAR 배포
1. **백업 및 업로드**: 운영 서버(Remote 접속한 REAL Tomcat)에 기존 운영 애플리케이션 폴더를 백업(폴더명 변경 등)합니다. 그리고 새로 생성한 WAR 파일을 Tomcat `webapps` 폴더에 업로드합니다.
2. **배포 및 검증**: Tomcat을 재시작하거나 자동 전개를 기다려 **운영 서비스에 새로운 WAR가 배포**됩니다. 운영 도메인으로 접속해 배포된 기능(우리 예시에서는 결제 기능)이 정상 작동하는지 확인합니다. 개발되지 않은(보류된) 기능은 당연히 보이지 않습니다.
3. **모니터링**: 배포 후 초기에는 서버 로그와 서비스 상태를 면밀히 모니터링합니다. 문제 발생 시 신속히 원인 파악이 가능하도록, **Git commit 히스토리와 배포 버전을 연결지어**두는 것이 좋습니다 (앞서 언급한 Git 태그 활용).

> **예시 시나리오 계속:** 결제 기능만 병합된 `main` 브랜치에서 WAR 파일을 빌드하여 운영 Tomcat에 배포했습니다. 상품검색 기능은 main에 안 들어갔으므로 운영에는 나타나지 않습니다. 며칠 후 상품검색 기능이 승인되면, 그때 해당 브랜치를 main에 병합하고 다시 WAR를 만들어 배포하면 됩니다. 이처럼 **main 브랜치를 통해 원하는 기능만 선택적으로 배포**할 수 있어, 운영에 섞여들면 안 되는 미완성 기능을 차단할 수 있습니다.

2-5. 협업 중 발생하는 문제 대응 (충돌 해결, 실수 복구, 롤백 전략)

두 명 이상의 개발자가 동시에 작업하다 보면 **충돌(conflict)**이나 **실수(commit 실수, 잘못된 merge)** 등이 발생할 수 있습니다. 아래는 GitHub Desktop으로 협업 시 일어날 수 있는 문제 상황들과 해결 방법입니다.

- **변경 충돌 해결**: 예를 들어 개발자 A와 B가 같은 파일의 같은 부분을 수정했다면, feature 브랜치를 develop에 병합할 때 충돌이 발생할 수 있습니다. GitHub Desktop에서 충돌 알림이 뜨면, 충돌난 파일을 찾아 수동으로 편집해야 합니다.
- GitHub Desktop에서 **충돌 파일**을 더블 클릭하면 기본 텍스트 편집기가 열리거나, 우측 상단의 **"Open in External Editor"** 버튼이 제공됩니다. 파일을 열어 보면 `<<<<<<HEAD`와 `======` 등의 구분자가 표시되어, 어느 부분이 어느 브랜치의 내용인지 보여줍니다. **두 버전의 코드를 합쳐서 하나의 최종 내용으로 편집**한 뒤, 충돌 구분자 표식들을 모두 삭제합니다. 저장 후 GitHub Desktop으로 돌아와 **마크(mark) as resolved** 버튼을 눌러 충돌 해결을 완료합니다. 그런 다음 평소처럼 **Commit**하고 **Push**하여 병합 커밋을 마무리합니다.

- 충돌을 예방하려면 **자주 Pull하여 동기화**하고, 같은 파일을 수정할 때는 미리 협업자와 상의하여 작업 범위를 나누는 것이 좋습니다.

- **잘못된 커밋/푸시 실수**: 예를 들어 잘못된 코드를 커밋해서 푸시했다면, 이를 되돌리는 방법은 **Revert**와 **Reset** 두 가지가 있습니다.

- Revert (되돌리는 새 커밋 생성): 현재 브랜치에서 문제 커밋만 **취소하는 새로운 커밋**을 만드는 방법입니다. GitHub Desktop에는 revert 버튼이 없으므로, GitHub 웹 UI에서 해당 커밋을 보면서 "Revert" 버튼을 사용하거나, 로컬에서 `git revert <커밋해시>`를 CLI로 실행할 수 있습니다. 그러면 "Revert '커밋 메시지'"라는 새로운 커밋이 생성되어 변경을 취소합니다. 이 커밋을 푸시하면 실수가 복구됩니다.

- Reset (커밋기록 자체를 삭제): 공개 저장소에서는 권장되지 않지만, 로컬에서 최근 커밋을 아예 없던 일로 만들 수도 있습니다 (`git reset HEAD~1` 등). 이 경우 GitHub Desktop에서는 해당 기능이 GUI로 제공되지 않으므로 CLI 사용이 필요하며, 이미 푸시한 경우 **강제 푸시(force push)**가 필요해 위험합니다. 초보자의 경우 reset보다는 revert 방식을 권장합니다.

- **Merge 후 문제 발생 시 롤백**: 운영 배포 후 문제가 생겨 **이전 버전으로 되돌려야 할 때**는, 가장 최근 배포 시점의 커밋으로 코드베이스를 돌리는 것이 안전합니다. 방법은 **태그**나 **커밋 해시**를 이용해 해당 버전의 코드를 checkout한 후 새 브랜치를 만들어 다시 배포하거나, main 브랜치에 revert 커밋을 만들어 배포하는 것입니다.

- 예를 들어, `v1.2.0-prod` 태그로 운영 배포를 했다가 문제가 생기면, GitHub Desktop에서 그 태그를 기반으로 브랜치를 만든 뒤 (예: `rollback-202508`) WAR를 빌드하여 운영에 배포할 수 있습니다. 또는 문제를 유발한 커밋들을 revert해서 main 브랜치를 수정한 후 다시 배포해도 됩니다. 어떤 방법이든 **철저한 테스트 후 배포**해야 함은 말할 필요도 없습니다.

- **GitHub Desktop만으로 안전하게 협업**: 가능한 GitHub Desktop GUI 내에서 작업을 처리하고, 꼭 필요한 경우에만 GitHub 웹이나 CLI를 사용하는 전략입니다. 예를 들어 **브랜치 보호 설정**(Protection Rules)을 GitHub에서 설정해 두면, main 브랜치에 직접 푸시하지 못하도록 하고 PR만 허용하여 실수를 방지할 수 있습니다. 하지만 소규모 팀이라면, 합의 하에 main에 직접 병합/푸시하되 **항상 이중 확인**하고, 주요 변경 전에 develop에서 충분히 테스트하도록 합니다.

- 또한 **커밋 메시지 규칙**, **코딩 컨벤션**을 팀원끼리 정해서 적용하면 협업 품질이 올라갑니다. GitHub Desktop에서는 커밋 이모티콘이나 템플릿을 쓸 순 없지만, 수동으로 `[feat]`, `[fix]` 등의 태그를 붙이는 식으로 메시지 형식을 맞출 수 있습니다.
- 정기적으로 **브랜치 정리**도 필요합니다. 병합 완료된 feature 브랜치는 GitHub에서 PR Merge 시 "Delete branch"를 하면 삭제되고, 로컬 GitHub Desktop에서도 `Discard branch`로 정리할 수 있습니다. 깔끔한 브랜치 관리가 추후 유지보수에 도움이 됩니다.

3. 현재 프로세스의 비효율/위험 요소 및 개선 방안

마지막으로, 기존 프로세스의 문제점을 짚어보고 위 개선안을 적용함으로써 **어떤 효율 향상과 위험 감소**가 이루어지는지 정리하겠습니다.

- **문제 1: 테스트 서버에 모든 기능 혼재 → 운영에 불필요한 코드 유입 위험**
  비효율/위험: 테스트 단계에서 완료되지 않은 기능까지 한 서버에 올라가 있어, 운영 배포 시 포함되면 안 되는 코드가 섞여 들어갈 우려가 있었습니다. 이를 막으려다 보니, 개발자가 WAR 배포 전에 코드를 일일이 골라내거나 삭제하는 수작업이 발생했고, 이 과정에서 실수가 잦았습니다.
  개선 방안: **브랜치로 배포 대상 구분** – 이제 **develop(테스트용)**과 **main(운영용)** 브랜치를 분리하여, 운영에는 main 브랜치 기준으로만 배포합니다. 테스트 서버에는 여전히 여러 기능이 올라가지만, **운영 WAR를 만들 때는 main 브랜치만 사용**하므로 불필요한 기능이 포함되지 않습니다. 수동 삭제 작업이 없어져 인적 오류가 줄고, **Git을 통한 변경 이력 추적**이 가능해 집니다.

- **문제 2: 수동 배포 및 변경 관리의 번거로움**
  비효율: 자동화 도구(예: CI/CD, 서버 Git 연동 등)를 일절 사용할 수 없어, 모든 배포 과정을 사람이 처리해야 했습니다. 특히 **VDI 원격 접속 후 파일 복사**라는 과정은 시간도 걸리고, 실수로 엉뚱한 위치에 배포하거나 권한 문

제를 일으킬 위험이 있었습니다.
개선 방안: **절차 표준화 및 반자동화** – 완전 자동화는 불가하더라도, **일관된 수동 배포 절차**를 수립하여 시행합니다. 예를 들어, WAR 파일 명을 `프로젝트명-버전.war` 형태로 규칙화하고, 배포 전/후에 **체크리스트**(예: 백업 완료 여부, 환경 설정 파일 확인 등)를 두어 매번 동일한 순서로 작업합니다. 또한 **Maven 빌드 스크립트**에 profile을 활용하여 DEV/REAL 환경별 설정을 자동 적용하게 해두면, 빌드 실수를 줄일 수 있습니다. 이처럼 **반(半)자동화** 요소를 도입하여 수동 작업량과 위험을 최소화합니다.

- **문제 3: 브랜치 병합, 충돌, 롤백 등에 대한 경험 부족**
  위험: Git 사용 미숙으로 인해 협업 중 충돌이 생기면 시간이 지체되고, 잘못 병합했다가 되돌리는 데 어려움을 겪었습니다. 최악의 경우 운영 코드에 실수가 들어가도 롤백 방법을 몰라 당황할 수 있습니다.
  개선 방안: **GitHub Desktop 활용 교육 및 전략 수립** – 위에서 설명한 대로, GitHub Desktop으로 **브랜치별 작업 절차**를 표준화하면 혼란이 줄어듭니다. 또한 팀 내 **모범 사례 공유**(예: "충돌 나면 이렇게 해결하세요", "커밋 잘못했을 땐 revert를 쓰세요")를 통해 모두가 대처 방법을 알도록 합니다. 작은 팀이라도 **코드 리뷰 문화**를 도입하면 오류를 조기에 발견할 수 있고, PR을 통해 병합하면 실수를 한 번 더 걸러낼 수 있습니다. 롤백의 경우 미리 **백업 태그**나 **이전 WAR 파일**을 준비해 두고, 응급시 빠르게 교체하는 시나리오 연습도 추천합니다.

- **문제 4: 현재 환경에서의 한계**
  한계점: 보안 정책상 **CI 도구, 배포 자동화 도입이 어려운 환경**입니다. 또한 팀원들이 GitHub 경험이 적어 고급 워크플로(CI/CD, GitOps 등)를 바로 적용하기 어려운 상황입니다.
  개선 방안: **현실적인 범위 내 최적화** – 당장은 GitHub Desktop과 브랜치 전략만으로도 충분히 개선이 가능합니다. 익숙해지면, 그 다음 단계로 **GitHub Actions** 같은 CI 서비스를 제한된 범위에서 활용하는 방안도 검토할 수 있습니다 (예: PR시 코드 스캔 정도). 또는, **스크립트화**된 배포: 예를 들어 배치 파일이나 Maven 플러그인으로 WAR 파일을 특정 경로로 FTP 업로드까지 자동화하는 간단한 도구를 만들어 쓸 수도 있습니다. 이런 소규모 자동화는 보안 정책을 크게 해치지 않으면서도 편의성을 높일 수 있습니다. 가장 중요한 것은, **모든 절차를 문서화**하여 새로운 팀원이나 초보자도 매뉴얼만 따르면 실수 없이 작업할 수 있게 하는 것입니다.

---

4. 결론 및 한눈에 보는 개선 프로세스

정리하면, **GitHub Desktop을 활용한 개선된 소스/배포 관리 프로세스**는 다음과 같습니다:

1. **기능별 브랜치에서 개발** → 독립적으로 코드 작성, 커밋, 푸시
2. **develop 브랜치로 병합** → 통합된 코드로 테스트 서버에 배포, 전체 기능 테스트
3. **main 브랜치로 선별 병합** → 운영에 반영할 기능만 골라 병합, 운영 서버에 WAR 배포
4. **철저한 변경 이력 관리** → 커밋 기록, 태그, 브랜치 관리로 언제든지 코드 상태 파악 및 복구 가능
5. **표준화된 협업 절차** → GitHub Desktop 매뉴얼대로 작업하여 충돌/실수 감소, 문제 시 대응 방법 공유

이러한 프로세스를 따르면, **운영에는 안정된 코드만 반영**되고 **테스트와 운영 환경이 명확히 분리**되어 품질과 신뢰도가 높아집니다. 비록 자동화 도구는 없지만, **Git 브랜치와 GitHub Desktop만으로도 충분히 안전하고 효율적인 협업**이 가능합니다. 우리 팀의 작은 사례이지만, 차근차근 Git 사용에 익숙해지면 향후 보다 발전된 워크플로도 소화해낼 수 있을 것입니다.

초보자라도 위 가이드에 따라 단계별로 진행해보세요. 처음엔 다소 복잡해 보이지만, **한 단계씩 밟다 보면 Git과 배포 관리에 대한 이해도가 높아지고 실수는 현저히 줄어들 것**입니다. 팀원들과 함께 이 프로세스를 지속적으로 개선해 나가길 바랍니다.

참고 자료: Git 브랜치 전략 (Git Flow), GitHub Desktop 공식 문서, 협업 시 충돌 해결 가이드 등.
