ChatGPT

GitHub Desktop을 활용한 소스/배포 관리 프로세스 개선안

본 문서는 GitHub Desktop만 사용할 수 있는 환경(STS 4.5, Zulu-8, Tomcat 9, Maven, MSSQL, MyBatis, JQGrid 등)에서 **feature → develop → release → main** 구조로 소스 관리를 개선하고, 테스트/운영 서버에 안전하게 수동 배포하는 절차를 초보자도 이해할 수 있도록 정리하였다. 각 단계는 실제 겪은 문제를 기반으로 설계되었으며, GitHub Desktop 공식 문서를 참고하여 캡처와 함께 설명한다.

1 현재 프로세스와 문제점 요약

항목	현재 상황	문제점
브랜치 관리	각자 **feature** 브랜치에서 개발 후 **develop**으로 병합하여 테스트 서버에 반영	테스트 서버에 모든 기능이 혼합되어 운영에 포함되어서는 안 될 소스까지 포함됨. 특정 기능만 운영에 반영할 때 소스를 삭제/추리는 과정에서 실수 발생
운영 배포	클라이언트가 요청한 기능만 선택하여 운영 서버에 수동 반영. war 파일을 직접 빌드해 VDI로 접속 후 업로드	수동 추려내기가 번거롭고 누락이나 오동작 위험이 높음. 브랜치 롤백, 병합 충돌 관리가 어렵고 협업 과정에서 실수가 잦음

이러한 문제를 해결하기 위해 **브랜치를 통한 격리**와 **release 브랜치를 활용한 선택적 배포**를 적용한다. 또한 GitHub Desktop의 기능만으로 커밋/푸시/병합/되돌리기 등을 안전하게 수행하도록 단계별 매뉴얼을 제시한다.

2 개선된 소스 관리 프로세스

2.1 브랜치 전략

GitHub Desktop으로 Git 저장소를 관리할 때, **기능 개발(feature) → 통합(develop) → 릴리스 준비(release) → 운영(main)** 순서의 브랜치 전략을 사용한다. Atlassian Gitflow의 정의에 따르면, 기능 개발은 develop 브랜치에서 분기한 feature 브랜치에서 진행하고 기능 완료 후 develop 브랜치로 병합한다 [1]. 충분한 기능이 모이면 develop 브랜치에서 release 브랜치를 생성하여 버그 수정과 문서화 등 릴리스 준비를 진행하고, 릴리스가 준비되면 release 브랜치를 main 브랜치에 병합하고 develop에도 병합하여 동기화한다 [2]. 이러한 구조는 다음과 같은 장점을 제공한다.

- **격리와 안정성** – feature 브랜치에서 개별 기능을 개발하므로 다른 기능에 영향을 주지 않으며, develop 브랜치에는 테스트 대상 기능만 포함된다. 운영(main) 브랜치는 릴리스가 확정된 안정 소스만 유지한다.
- **선택적 배포** – release 브랜치를 통해 특정 기능만 운영에 반영할 수 있으며, 준비되지 않은 기능은 develop 브랜치에 머무른다. 운영에 필요 없는 소스가 섞이는 문제를 줄인다.
- **롤백 용이성** – release 또는 main 브랜치에서 문제 발생 시 이전 릴리스 태그나 커밋으로 즉시 되돌릴 수 있다.

본 문서 후반의 브랜치 전략 도식화를 참고하면 구조를 쉽게 이해할 수 있다.

2.2 GitHub Desktop 기본 설정 및 리포지터리 준비

1. **설치 및 계정 인증** – GitHub Desktop을 설치하고 GitHub 계정으로 로그인한다. 계정 인증은 GitHub Desktop 메뉴의 **Settings**에서 수행한다.

2. **리포지터리 추가/복제** – 기존 프로젝트를 처음 사용할 때 `File → Add Local Repository`로 로컬 저장소를 연결하거나 `Clone Repository`로 GitHub의 저장소를 복제한다.
3. **기본 브랜치 확인** – 프로젝트의 기본 브랜치(main/master)를 확인하고 develop 브랜치가 없으면 main에서 새 브랜치를 생성한다. GitHub Desktop에서 현재 브랜치 드롭다운을 클릭 후 **새 분기**를 선택하고 이름(`develop`)을 입력하여 생성한다 [3].

2.3 기능 개발 – feature 브랜치 작업 흐름

1. **feature 브랜치 생성**
2. GitHub Desktop 상단의 **Current Branch** 드롭다운을 클릭하고 기준 브랜치로 `develop`을 선택한다 [3].
3. 하단의 **New Branch** 버튼을 눌러 브랜치 이름(`feature/기능명`)을 입력하고 생성한다 [4].
4. **Publish Branch**를 클릭하여 원격 저장소에 브랜치를 게시한다 [5].
5. **코드 수정 및 커밋**
6. STS에서 코드를 수정한 후 저장한다.
7. GitHub Desktop의 **Changes** 탭에서 변경 파일을 확인하고, 커밋 메시지를 작성한 뒤 **Commit to feature/기능명**을 클릭한다.
8. 원격 저장소에 공유하려면 **Push origin**을 눌러 푸시한다. GitHub Desktop은 먼저 원격 브랜치의 최신 커밋을 가져오도록 `Fetch origin` → `Pull origin`을 안내한다 [6]. 충돌이 없으면 푸시 버튼이 활성화되고 **Push origin**을 클릭하면 원격에 반영된다 [7].
9. **작업 중인 변경 사항 저장(선택)**
10. 브랜치를 전환해야 하는데 아직 커밋하지 않은 변경이 있는 경우, **Stash** 기능으로 변경을 임시 저장할 수 있다. 메뉴에서 `Repository → Stash Changes`를 선택하면 작업 내용을 안전하게 보존한 뒤 다른 브랜치로 이동할 수 있다.

2.4 기능 완료 후 develop 브랜치로 병합

1. **develop 브랜치로 전환** – `Current Branch`를 클릭하고 develop을 선택하여 체크아웃한다 [8].
2. **feature 브랜치 병합**
3. `Current Branch` 메뉴에서 **Choose a branch to merge into develop**를 클릭한다 [9].
4. 병합할 feature 브랜치를 선택하고 **Merge feature… into develop**를 클릭한다. 충돌이 있는 경우 GitHub Desktop이 경고하고 충돌 해결 전에는 병합할 수 없다 [10].
5. 병합 후 **Push origin**을 클릭하여 원격 develop 브랜치로 푸시한다 [11].
6. **충돌 해결 팁**
7. 병합 시 같은 파일을 여러 사람이 수정했다면 충돌이 발생한다. GitHub Desktop은 병합을 차단하고 텍스트 편집기로 충돌을 해결하도록 안내한다. 에디터에서 충돌 표시(`<<<<<<<`, `=======`, `>>>>>>>`)를 확인하고 수정한 뒤 저장한다. 수정 후 GitHub Desktop에서 **Mark as resolved**를 클릭하고 새 커밋을 생성하여 Push 한다.
8. 충돌을 방지하려면 병합 전 `Fetch origin`과 `Pull origin`을 통해 develop 브랜치를 최신 상태로 유지하고, 자주 작은 단위로 병합하는 것이 좋다.

2.5 릴리스 준비 – release 브랜치 생성 및 테스트 서버 반영

테스트 서버에는 develop 브랜치의 모든 기능이 반영되지만, 운영에 포함할 기능만 선별하려면 release 브랜치가 필요하다.

1. **release 브랜치 생성** – develop 브랜치에서 `Current Branch` 드롭다운 → **New Branch**를 선택하고 이름을 `release/버전`으로 지정한다. Atlassian Gitflow는 develop 브랜치에서 release 브랜치를 포크하여 버그 수정과 문서화 등 릴리스 작업을 진행한 뒤 release 브랜치를 main과 develop에 병합하는 것을 권장한다 [12].
2. **테스트 서버 배포**
3. release 브랜치에서 build 테스트를 수행하고, 테스트 서버에 war 파일을 배포한다. 이는 develop 전체가 아닌 release 브랜치의 코드만 포함하므로 미완성 기능이 섞이지 않는다.
4. **버그 수정** – 테스트 과정에서 발견된 버그는 release 브랜치에서 수정한 뒤 커밋/푸시한다. develop 브랜치에서도 동일한 수정이 필요하다면 release 브랜치를 develop에 병합하여 동기화한다 [13].

2.6 운영 배포 – main 브랜치 병합 및 war 빌드

1. **main 브랜치로 병합** – release 브랜치의 검증이 완료되면 main 브랜치로 전환하여 release 브랜치를 병합한다(`Current Branch` → `Choose a branch to merge into main`). 병합 후 **Push origin**을 수행한다.
2. **태그 지정** – 배포된 버전을 식별하기 위해 main 브랜치 커밋에 태그를 생성한다(`Repository → Create Tag`). 태그는 차후 롤백 지점으로 활용할 수 있다.
3. **release → develop 동기화** – release 브랜치에서 수정한 버그 수정 사항이 develop에도 반영되도록 develop 브랜치에 release 브랜치를 병합한다 [13].
4. **war 파일 빌드**
5. 로컬에서 `mvn clean package` 또는 STS의 **Run As → Maven build**를 이용해 war 파일을 생성한다. 생성된 war는 `target/프로젝트명.war`에 위치한다.
6. 운영 서버 접속(VDI) 후 기존 war와 exploded 폴더(예: `webapps/프로젝트명`)를 삭제한다.
7. 새 war 파일을 서버의 Tomcat webapps 폴더에 업로드한다. Tomcat 공식 문서는 Host의 `appBase`(기본값 `$CATALINA_BASE/webapps`)에 압축된 .WAR 파일을 복사하면 Tomcat을 재시작할 때 자동 배포된다고 설명한다 [14].
8. Tomcat이 실행 중이라면 `autoDeploy`가 `true`인 경우 새 war 파일을 webapps에 넣으면 자동으로 배포/재배포가 수행된다 [15]. 운영 정책상 재시작 배포가 필요하면 Tomcat을 중지 후 다시 시작한다.

2.7 운영 서버 수동 배포 절차 요약 (Tomcat)

1. **백업** – 기존 war 파일과 exploded 디렉터리를 백업(다른 폴더로 이동)한다.
2. **배포 준비** – Tomcat을 중지하거나 `autoDeploy`를 비활성화한 경우 톰캣 재시작이 필요하다.
3. **새 war 업로드** – 빌드한 war 파일을 `$CATALINA_BASE/webapps`에 업로드한다. Tomcat은 `deployOnStartup=true`일 때 시작 시 webapps 내 war 파일을 자동으로 배포한다 [14].
4. **배포 확인** – Tomcat 로그나 브라우저에서 `/context-path` 접근하여 정상 구동을 확인한다. 필요 시 `unpackWARs` 설정에 따라 exploded 폴더가 생성된다.
5. **문제 발생 시 롤백** – 배포 버전 태그를 확인하여 이전 버전 war 파일로 되돌리거나, GitHub Desktop에서 revert 기능으로 문제 커밋을 되돌린 뒤 새로 빌드하여 배포한다.

2.8 운영에 일부 기능만 반영하는 방법

1. **release 브랜치 활용** – develop 브랜치에서 필요한 기능만 선택하여 release 브랜치를 생성함으로써 불필요한 기능이 운영에 포함되지 않도록 한다. 준비되지 않은 기능은 develop에 남아 있으며 release 브랜치에는 포함되지 않는다.

2. **cherry-pick** – 만약 긴급하게 특정 커밋만 운영에 반영해야 한다면 release 브랜치에서 `커밋 cherry-pick` 기능을 사용한다. GitHub Desktop에서는 커밋을 우클릭 후 **Cherry-pick**을 선택하여 원하는 커밋만 적용할 수 있다. 하지만 cherry-pick은 히스토리를 복잡하게 만들 수 있으므로 release 브랜치 전략을 우선 사용한다.
3. **직접 코드 삭제 금지** – 운영에 필요 없는 기능을 제거하기 위해 개발된 코드를 수동으로 삭제하지 않는다. 해당 기능은 feature 브랜치에서 개발되고 release 브랜치에는 포함되지 않으면 삭제할 필요가 없다. 잘못된 삭제는 서비스 오작동의 원인이 될 수 있다.

2.9 롤백과 복구 전략

문제가 발생했을 때 GitHub Desktop에는 여러 가지 되돌리기 기능이 있다.

- **커밋 취소(Undo)** – 푸시하기 전 최근 커밋을 취소하려면 **Changes** 탭에서 **Undo** 버튼을 클릭한다 [16] . 해당 커밋의 변경 내용은 워킹 디렉터리에 되돌아가며 다시 수정할 수 있다.
- **특정 커밋으로 리셋** – 여러 커밋을 한 번에 되돌리고 싶다면 **History** 탭에서 되돌리고 싶은 커밋을 우클릭하여 **Reset to commit**을 선택한다 [17] . 이 기능은 마지막으로 푸시된 커밋까지 되돌릴 수 있으며, 이후 변경 내용은 로컬에서 수정 후 다시 커밋한다.
- **푸시된 커밋 되돌리기(Revert)** – 이미 원격에 푸시된 커밋을 제거하지 않고 변경 내용만 반영 취소하려면 History 탭에서 커밋을 우클릭하고 **Revert Changes in Commit**을 선택한다 [18] . 이 작업은 되돌리기 커밋을 추가하므로 히스토리는 보존된다.
- **브랜치 롤백** – 운영 배포 후 문제가 발생하면 main 브랜치에서 이전 태그나 커밋으로 체크아웃한 뒤 새 release 브랜치를 생성하여 다시 배포할 수 있다.

2.10 협업 충돌 해결 및 팁

- **자주 Pull/Fetched** – 여러 명이 같은 브랜치에서 작업하면 다른 사람이 푸시한 변경을 주기적으로 Fetch/Pull 하여 로컬과 동기화한다 [6] . 이를 소홀히 하면 나중에 대규모 충돌이 발생할 수 있다.
- **작은 단위로 커밋** – 큰 변경을 한번에 커밋하기보다 기능 단위로 자주 커밋하면 충돌 시 해결 범위를 줄일 수 있고 롤백도 용이하다.
- **커밋 메시지 규칙** – `[기능명] 설명`과 같이 일관된 형식을 사용하면 나중에 cherry-pick이나 revert할 때 찾기 쉽다.
- **협업 중 실수 방지** – 잘못된 브랜치에서 작업하지 않도록 매일 작업 전에 현재 체크아웃된 브랜치를 확인한다. 브랜치 이름에 `feature/`, `release/` 등 접두어를 붙여 혼동을 줄일 수 있다.
- **테스트 환경 분리** – develop 브랜치는 항상 테스트 서버와 동기화하고, feature 브랜치는 로컬에서 충분히 테스트 후 develop에 병합한다. 테스트 서버에서 버그가 발견되면 release 브랜치에서 수정 후 develop과 동기화한다.

2.11 보안 정책을 준수하면서 GitHub Desktop만 사용하기

- **토큰/SSH 미사용** – 보안 정책상 서버 Git 연동과 자동화 배포가 불가하므로 GitHub Desktop으로만 원격 저장소에 접속하고, 서버에는 수동으로 war 파일을 업로드한다.
- **권한 관리** – GitHub 저장소의 브랜치 보호 기능을 설정하면 main 브랜치에 대한 직접 푸시를 제한하고 풀 리퀘스트를 통해 검토 후 병합하도록 강제할 수 있다. GitHub Desktop은 보호된 브랜치에서 삭제나 강제 푸시를 허용하지 않는다 [19] .
- **VDI 접근 기록** – 운영 서버에 원격 접속할 때마다 배포 일지에 날짜, 배포자, 배포 버전을 기록하여 추적성을 확보한다.

3 비효율 및 위험 요소와 개선 절차

3.1 비효율/위험 요소

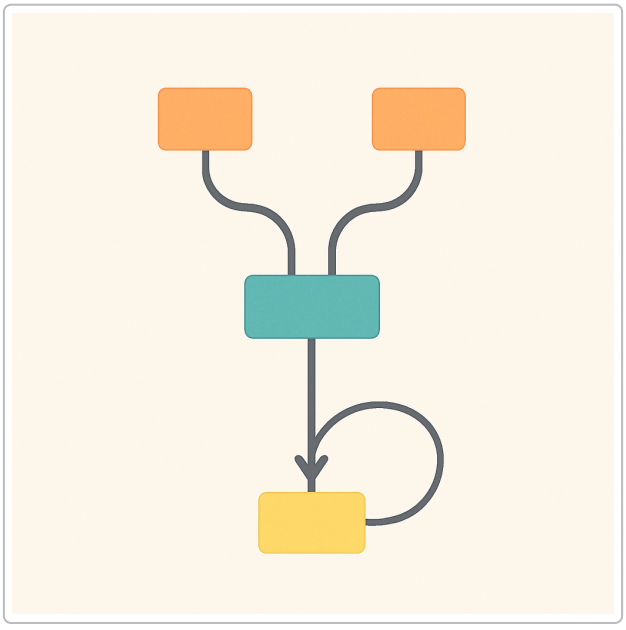
1. **테스트 서버에 모든 기능이 혼합** – develop 브랜치 전체를 테스트 서버에 배포하므로 운영에 포함되지 않아야 할 기능까지 서비스 오작동의 원인이 됐다.
2. **수동 소스 삭제** – 운영에 필요한 기능만 추리기 위해 소스를 삭제/추리는 과정에서 실수가 발생했다. 수동 삭제는 히스토리를 어지럽히고 예기치 않은 오류를 유발한다.
3. **브랜치 롤백 어려움** – 단일 develop 브랜치에 모든 기능이 병합되어 있어 특정 기능만 롤백하거나 선택적으로 반영하기 어렵다.
4. **협업 관리 부족** – 푸시/병합 규칙이 없어 충돌이 빈번하게 발생하고, 커밋 메시지나 브랜치 명명 규칙이 일관되지 않아 이력 관리가 어려웠다.

3.2 개선/최적화 절차

1. **브랜치 구조 확립** – feature → develop → release → main 구조를 도입하여 기능 개발, 통합 테스트, 릴리스 준비, 운영 반영을 분리한다.
2. **release 브랜치 활용** – 운영에 포함될 기능만 release 브랜치에 cherry-pick하거나 develop에서 분기하여 선별함으로써 불필요한 코드가 운영에 반영되지 않도록 한다.
3. **자동화 대체 수단** – CI/CD 도구를 사용할 수 없는 환경에서는 배포 스크립트라도 로컬에 작성하여 Maven 빌드, war 파일 백업, 서버 업로드를 표준화한다.
4. **정기적 코드 리뷰** – develop → release 병합 시 코드 리뷰를 진행하여 품질을 확보한다. GitHub Desktop에서도 Pull Request를 생성할 수 있으므로 리뷰 후 병합하는 방식이 좋다.
5. **교육과 문서화** – 신규 인원이 쉽게 따라할 수 있도록 본 문서와 같이 단계별 매뉴얼을 작성하고, 브랜치 생성/병합/배포 과정을 교육한다.

4 참고 이미지

아래 그림은 feature → develop → release → main 브랜치 전략의 흐름을 추상적으로 표현한 것이다. 여러 개의 기능(feature) 브랜치가 develop 브랜치로 병합되고, develop에서 release 브랜치가 갈라져 운영(main)으로 합쳐진다. 도식화는 실제 브랜치 이름 대신 색상과 화살표로 개념을 설명한다.



5 맺음말

GitHub Desktop만으로도 브랜치 전략과 수동 배포 절차를 체계화하면 기존의 비효율과 위험을 크게 줄일 수 있다. feature 브랜치를 적극 활용해 기능을 격리하고, release 브랜치를 통해 선택적 운영 배포를 수행하며, GitHub Desktop의 커밋 관리 기능(Undo, Reset, Revert)을 통해 롤백을 안전하게 수행하자. 또한 Tomcat webapps 디렉터리에 war 파일을 업로드하면 자동으로 배포된다는 점 [14] 을 활용하여 배포 절차를 단순화하되, 배포 일지를 남겨 보안·감사 요구사항을 충족하는 것이 중요하다.

---

[1] [2] [12] [13] Gitflow Workflow | Atlassian Git Tutorial
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[3] [4] [5] [8] [19] GitHub Desktop에서 분기 관리하기 - GitHub Docs
https://docs.github.com/ko/desktop/making-changes-in-a-branch/managing-branches-in-github-desktop

[6] [9] [10] [11] Syncing your branch in GitHub Desktop - GitHub Docs
https://docs.github.com/en/desktop/working-with-your-remote-repository-on-github-or-github-enterprise/syncing-your-branch-in-github-desktop

[7] Pushing changes to GitHub from GitHub Desktop - GitHub Docs
https://docs.github.com/en/desktop/making-changes-in-a-branch/pushing-changes-to-github-from-github-desktop

[14] [15] Apache Tomcat 9 (9.0.107) - Tomcat Web Application Deployment
https://tomcat.apache.org/tomcat-9.0-doc/deployer-howto.html

[16] Undoing a commit in GitHub Desktop - GitHub Docs
https://docs.github.com/en/desktop/managing-commits/undoing-a-commit-in-github-desktop

[17] Resetting to a commit in GitHub Desktop - GitHub Docs
https://docs.github.com/en/desktop/managing-commits/resetting-to-a-commit-in-github-desktop

[18] Reverting a commit in GitHub Desktop - GitHub Docs
https://docs.github.com/en/desktop/managing-commits/reverting-a-commit-in-github-desktop