

## 아래는 우리 회사의 실제 개발 및 배포 프로세스와 겪고 있는 문제 사례입니다.

---

### [현재 프로세스]

- GitHub Desktop 프로그램을 사용해 소스 이력/협업을 관리하고 있습니다.
  - 개발은 각자 feature 브랜치에서 진행, develop 브랜치로 병합(merge) 후 테스트 서버에 반영합니다.
  - 테스트 서버에는 지금까지 개발된 모든 기능이 반영되어, 클라이언트가 테스트합니다.
  - 운영 서버에는 클라이언트가 "운영 반영"을 요청한 기능만 따로 추려서, 필요 없는 소스는 삭제 후 수동으로 배포합니다.
  - 자동화 배포, 서버 Git 연동, SSH/토큰 인증 등은 보안 정책상 모두 불가합니다.
  - war 파일을 직접 빌드해서, 서버에 원격접속(VDI) 후 수동 업로드 방식만 사용 가능합니다. [실제 겪은 문제/불편]
  - 테스트 서버에 모든 개발 기능이 섞여 올라가, 운영에는 반영되지 않아야 할 소스(기능)까지 섞여 들어가 서비스 오작동 및 민원이 발생했습니다.
  - 운영에 일부 기능만 반영하려다 소스 삭제/추리기 과정에서 실수 발생.
  - 브랜치 병합(Merge), 롤백(복구), 협업 충돌 관리도 어렵습니다. [보충설명]
1. 운영/테스트 서버의 구조 운영 서버와 테스트 서버는 각각 어떤 방식으로 관리되나요? (예: 물리적으로 다른 서버인지, 같은 Tomcat 환경 내에서 포트만 다른지 등) => 같은 서버에 DEV 톰캣과 REAL 톰캣으로 따로 톰캣 서버가 존재. 해당 톰캣 webapp 폴더 밑에 war 파일을 붙여넣고 서비스. 아파치 웹서버와 연동. 각각의 도메인으로 접속.
  2. 현재 배포 방식에서의 워크플로 WAR 파일을 빌드해서 수동 업로드한다고 하셨는데, 이 때 정확히 어떤 디렉토리에 배포하나요? => 개발 서버는 DEV 톰캣 폴더 하위의 webapp 폴더, 운영 서버는 REAL 톰캣 폴더 하위 webapp 폴더에 배포 테스트 서버에는 어떤 기준으로 반영되며, 배포 시점은 누가/언제 결정하나요? => 개발자가 개발이 완료된 이후 클라이언트에게 테스트 요청하기 전에 개발 서버에서 war파일을 만들어 복사 붙여넣기 함.
  3. 현재 브랜치 전략 관련 "운영에 반영해야 할 기능만 따로 추리는" 작업은 어떤 식으로 하고 있나요? 예: develop에서 작업된 기능 중 일부만 main에 cherry-pick 하는 식인지, 아니면 수동 복붙? => 개발서버에서 개발된 소스만 관리. 운영 및 테스트에는 개발자기 직접 수동으로 복붙.
  4. 협업 인원 몇 명이 동시에 개발에 참여하고 있는지, GitHub 사용 경험은 모두 있는지 알고 싶어요. => 2명이 참여. GitHub에 대한 경험은 없음.
  5. GitHub Desktop 사용 범위 오직 GitHub Desktop만 사용하고 있는지, CLI나 VSCode Git 플러그인은 아예 사용하지 않는 건가요? => 오직 GitHub만 사용. STS4에서 작업한 다음 GitHub Desktop 프로그램을 실행하여 수정 및 추가한 소스를 push함.
- 

이런 상황에서,

1. **GitHub Desktop** 프로그램을 기준으로,  
개발/테스트/운영 서버 소스 관리와 war 파일 배포 프로세스를  
초보자도 이해할 수 있게, 한글로 단계별로 재정의해 주세요.
2. 특히,
  - feature/develop/main 브랜치 관리 및 테스트/운영 서버 분리 배포 전략
  - 운영에 일부 기능만 반영 시 소스 혼입/오작동 방지법
  - 협업 중 실수, 충돌, 롤백(복구) 등 문제 해결 전략
  - 자동화/서버 Git/배포툴 없이 GitHub Desktop만으로 안전하게 협업/관리하는 방법
  - (개발환경: STS4.5, Zulu-8, Tomcat 9, Maven, MSSQL, MyBatis, JQGrid 등)
3. 그리고, 현재 프로세스의 **비효율/위험요소**와,  
**우리 환경에서 실현 가능한 구체적 개선/최적화 절차**도 같이 안내해 주세요.

---

**\*\*GitHub Desktop을 활용한 “실전 소스/배포 관리 프로세스 개선안”\*\*을  
보다 자세하고 꼼꼼하게 한글로, 단계별로, 예시/팁/가능하면 관련 이미지까지 포함해서 설명해 주세요.**

---

알려주신 정보가 아주 상세해서 도움이 많이 됐어요! 아래 내용만 추가로 확인되면, 바로 실전 중심의 단계별 개선 프로세스를 정리해드릴 수 있어요.

1. "운영에 반영할 기능만 반영" 시에, 기능 단위로 개발이 분리되어 있는 구조인가요?
  - 예: 기능마다 독립적인 Controller/Service/Mapper 파일 구조로 관리되고 있는지?
  - 아니면 하나의 소스파일 내에서 여러 기능이 혼합되어 작성되고 있는지?
2. war 파일은 Maven을 통해 STS에서 빌드하시나요? (예: `mvn clean package` 등)
  - 아니면 수동으로 export해서 만드시나요?
3. 배포할 때, 운영/개발 톱캣의 war 파일 외에 추가로 건드리는 설정 파일이나 리소스가 있나요?
  - 예: `application.properties`, DB 연결, 보안 설정 파일 등
4. 운영 서버에서 자주 롤백(기능 복구)이 필요한 일이 발생하나요?
  - 예: 이전 버전의 war를 따로 백업해두고 수동으로 다시 덮어쓰는 식인가요?

위 내용만 확인해 주시면, GitHub Desktop만 사용하는 환경에서도 실수 없이 안전하게 협업하고 배포할 수 있는 구체적인 개선 전략을 정리해드릴게요!

---

운영에 반영할 기능만 반영" 시에, 기능 단위로 개발이 분리되어 있는 구조인가요?  
예: 기능마다 독립적인 Controller/Service/Mapper 파일 구조로 관리되고 있는지?  
아니면 하나의 소스파일 내에서 여러 기능이 혼합되어 작성되고 있는지?  
=> MVC 모델이 적용되어 있고 해당 방식으로 관리

war 파일은 Maven을 통해 STS에서 빌드하시나요? (예: `mvn clean package` 등)  
아니면 수동으로 export해서 만드시나요?

```
\=> 수동으로 export해서 만들
```

배포할 때, 운영/개발 톰캣의 war 파일 외에 추가로 건드리는 설정 파일이나 리소스가 있나요?

```
\=> 없음.
```

운영 서버에서 자주 롤백(기능 복구)이 필요한 일이 발생하나요?

```
\=> 이전 버전의 war를 따로 백업해두고 수동으로 다시 덮어쓰는 식
```

좋아요. 이제 현재 구조와 사용 도구, 제약 조건을 모두 고려한 GitHub Desktop 기반 실전 배포/소스 관리 개선안을 단계별로 정리해서 안내드릴게요. 초보자도 따라 할 수 있도록 예시와 함께 설명드릴 예정입니다. 곧 정리해서 보여드릴게요!

## GitHub Desktop을 활용한 실전 소스/배포 관리 프로세스 개선안

### 현재 프로세스의 문제점과 개선 필요성

현재 귀사의 개발/배포 프로세스에서는 테스트 서버에 모든 신규 기능이 한꺼번에 반영되고, 운영 배포 시 특정 기능만 골라내어 코드를 수동 삭제 및 추려서 배포하는 방식입니다. 이로 인해 다음과 같은 문제가 발생했습니다:

- 테스트 서버에 운영에 나가면 안 되는 기능 코드까지 포함되어 있어, 실수로 해당 코드가 운영 WAR 파일에 섞여 들어가 서비스 오작동이 발생하고 클라이언트 민원이 제기되었습니다.
- 운영에 일부 기능만 반영하려고 코드를 골라내는 과정에서 실수가 발생하여, 필요한 파일을 누락하거나 불필요한 변경이 포함되는 위험이 있습니다.
- Git 브랜치 병합(Merge) 과정에서 여러 개발자의 작업 충돌을 관리하기 어렵고, 문제가 생겼을 때 롤백(복구) 절차도 번거롭습니다.
- 자동화된 배포 도구를 사용할 수 없는 환경이기에, 수동으로 WAR를 빌드하고 VDI 원격접속으로 파일을 올리는 과정에서 휴먼 에러 가능성이 높습니다.

→ 이러한 문제를 해결하기 위해서는, Git 브랜치 전략을 통한 소스 관리 체계화와 배포 절차 개선이 필요합니다. GitHub Desktop을 활용하여 초보자도 쉽게 따라할 수 있는 방식으로 개발(테스트)용 코드와 운영 배포 코드를 명확히 분리하고, 수동 배포 환경에서도 최대한 실수를 방지하는 프로세스를 구축해야 합니다.

### 개선 방향 및 브랜치 전략 (feature/develop/main 분리)

개선된 프로세스의 핵심은 Git 브랜치 전략을 도입하여 개발용 소스와 운영 배포용 소스를 분리 관리하는 것입니다. 이를 위해 일반적으로 많이 사용하는 Git-Flow 기반 전략을 응용하되, 현재 환경에 맞게 단순화하여 적용합니다:

**그림: Git-Flow 브랜치 모델** – 운영 배포용 메인(main) 브랜치와 개발 통합용 develop 브랜치, 그리고 기능별 feature 브랜치를 사용하는 브랜치 전략 예시입니다 (main=운영 코드, develop=전체 개발 코드, feature=개별 기능 작업용). 이러한 구조로 코드 관리를 체계화하면, 운영에 포함하지 않을 기능 코드를 분리하고 병합 내역을 명확히 추적할 수 있습니다.

- **main 브랜치 (운영용):** 운영 서버에 배포되는 최종 릴리스 코드만 모아두는 브랜치입니다. 배포 가능한 안정된 코드(프로덕션 코드)만 유지하며, 여기의 코드는 항상 \*\*운영 서버(REAL Tomcat)\*\*와 동기화된 상태를 의미합니다. 현재 프로세스에서는 별도 분리가 없었지만, 앞으로는 main 브랜치를 운영용 소스의 기준선으로 삼습니다.

- **develop 브랜치 (개발 통합용):** 여러 기능 개발 작업이 **통합되는 브랜치**로서, 향후 배포를 준비 중인 **전체 기능**을 포함합니다. 두 명의 개발자가 각자 작업한 기능을 develop 브랜치에 병합하고, 이 브랜치를 기반으로 **\*\*테스트 서버(DEV Tomcat)\*\***에 배포합니다. 즉, develop 브랜치의 코드는 **테스트 환경에 배포되는 최신 개발 상태**입니다.
- **feature 브랜치 (기능별 분기):** 새로운 기능이나 버그 수정을 할 때마다 개별적으로 분기하여 작업하는 **기능 단위 브랜치**입니다. feature 브랜치는 **develop 브랜치로부터** 분기하여 (혹은 필요에 따라 main에서 분기) 작업하며, 개발 완료 후 **develop에 병합**합니다. 이렇게 하면 각 기능이 **독립적으로 개발**되어 서로 영향 없이 진행되고, 병합 시에 충돌을 최소화할 수 있습니다.

이러한 구조에서는 *운영에 포함되지 않을 기능*은 develop에는 남아있어도 **main 브랜치에 병합되지 않으면 운영 코드로 절대 반영되지 않으므로**, 수동으로 코드를 삭제하는 등의 번거로운 작업이 불필요해집니다. 또한 main과 develop 브랜치를 분리함으로써, **테스트용 최신 코드**와 **운영 배포용 안정 코드**를 명확히 관리할 수 있습니다.

※ **Tip:** GitHub에서 기본(default) 브랜치를 `develop`으로 지정해 두면, 실수로 main에 잘못 커밋하거나 병합하는 실수를 줄일 수 있습니다. (GitHub 저장소 설정에서 default branch를 develop으로 변경 가능)

## 개발 단계: 기능 개발과 테스트 서버 배포 프로세스

이제 Git 브랜치를 활용한 **개발 및 테스트 서버 반영 절차**를 단계별로 재정의해 보겠습니다. 두 명의 개발자가 **GitHub Desktop**으로 협업한다는 전제하에, **feature 브랜치 생성 → 개발 → develop 병합 → 테스트 배포** 순서로 진행됩니다. 아래는 초보자도 따라할 수 있도록 상세한 단계와 팁입니다:

1. **새 기능 개발을 위한 feature 브랜치 생성:** 먼저 GitHub Desktop에서 현재 `develop` 브랜치를 기준으로 새 feature 브랜치를 만듭니다. GitHub Desktop 우측 상단의 **“Current Branch”** 버튼을 클릭하고 `develop`을 선택한 뒤, **“New Branch”** 버튼을 눌러 브랜치 이름을 지정합니다. 예를 들어 새로운 결제 기능을 개발한다면, 브랜치 이름을 `feature/payment` 처럼 짓습니다. (**GitHub Desktop UI에서 브랜치 생성 예시: GitHub Desktop에서 `develop` 기반으로 새 브랜치 만드는 화면.**) 이렇게 분기된 feature 브랜치에서는 다른 동료와 겹치지 않고 **독립적으로 코딩 작업**을 할 수 있습니다.
2. **STS에서 코딩 및 커밋:** STS4(Eclipse 기반 IDE)에서 방금 만든 feature 브랜치를 체크아웃(checkout)하여 해당 브랜치에서 코드를 작성합니다. (GitHub Desktop에서 브랜치를 만들면 자동으로 해당 브랜치로 전환됩니다.) 예를 들어 `feature/payment` 브랜치 상에서 새로운 Controller, Service, Mapper 등을 생성/수정합니다. **기능 개발이 완료되면**, GitHub Desktop을 열어 변경된 파일들을 확인하고 **의미 있는 커밋 메시지와 함께 커밋(commit)**합니다. 커밋 시점마다 **적절한 단위로 자주 커밋**하는 것이 좋습니다. 그 후 **Push** 버튼을 눌러 원격(remote) 저장소의 해당 feature 브랜치로 변경 내용을 업로드합니다. (협업자도 이 브랜치를 원격에서 pull 받아 함께 작업하거나, 코드 리뷰를 진행할 수 있습니다.)
3. **feature 브랜치 변경 내용 검토 (코드 리뷰):** 만약 협업자 간에 코드 리뷰를 하고 싶다면, GitHub Desktop에서 바로 **Pull Request**를 생성할 수도 있습니다 (GitHub Desktop 메뉴의 `Create Pull Request` 기능 이용). 두 명이 함께 작업하므로, 간단히 **GitHub 웹**에서 Pull Request를 만들어 코드 변경내역을 서로 확인하는 것도 권장합니다. 이 단계는 필수는 아니지만, 리뷰 과정을 거치면 **실수나 버그를 미리 발견**할 수 있고, 병합 전에 코드 품질을 높일 수 있습니다.
4. **develop 브랜치로 기능 병합(Merge):** feature 브랜치에서 충분한 개발 및 자체 테스트를 마쳤다면, 해당 기능을 통합하기 위해 develop 브랜치에 병합합니다. **GitHub Desktop**에서 현재 브랜치를 `develop`으로 전환한 뒤, **“Current Branch”** 드롭다운에서 **“Choose a branch to merge into develop”** 옵션을 선택합니다. 여기서 방금 작업한 feature 브랜치를 선택하고 **“Merge {브랜치명} into develop”** 버튼을 클릭하면, feature 브랜치의 변화가 develop 브랜치에 병합됩니다. (**GitHub Desktop UI에서 브랜치 병합 예시: `develop` 브랜치에 다른 브랜치를 병합하는 화면.**) 병합 후 GitHub Desktop이 자동으로 **merge commit**을 생성해 주며, 이 내용을 **Push**하여 원격 develop 브랜치에 업로드합니다.

- 만약 병합 과정에서 **충돌(conflict)** 이 발생하면, GitHub Desktop에서 경고를 표시하며 병합 버튼이 비활성화됩니다. 이 경우 **충돌난 파일을 열어서 수정**해야 합니다. STS나 선호하는 에디터에서 `<<<<`, `=====`, `>>>>` 표시를 찾아 **두 브랜치의 변경사항 중 올바른 부분을 취사선택**하고 저장합니다. 수정 후 GitHub Desktop으로 돌아와 **Commit merge**를 완료하면 병합이 완료됩니다. 충돌 해결이 어렵다면, **동료와 상의하여** 어느 쪽 변경을 살릴지 결정하고 진행합니다. (충돌은 보통 **같은 파일의 같은 부분을 동시에 수정**할 때 발생하므로, 평소에 역할 분담을 명확히 하고, 자주 pull 받아 동기화하면 충돌을 줄일 수 있습니다.)

- 테스트 서버 배포:** develop 브랜치에 새로운 기능이 병합되었으므로, 이제 이 코드를 **\*\*테스트 서버(DEV 톰캣)\*\***에 배포합니다. **STS4에서 WAR 파일 생성:** 프로젝트를 마우스 우클릭하여 **Export -> WAR 파일**로 패키징하거나, Maven 사용 시 `mvn package` 명령으로 WAR 빌드를 합니다 (GUI에 익숙하다면 Export 방식을 사용). 이 때 반드시 **develop 브랜치의 최신 코드로 빌드**하도록 합니다. 빌드된 WAR 파일 (예: `myapp.war`)을 **DEV 톰캣 서버의 `webapps` 디렉터리**에 업로드합니다. (기존에 동일한 이름의 WAR나 폴더가 있다면 미리 삭제하거나 덮어쓰기합니다.) 톰캣을 재구동하거나 자동 전개(auto-deploy)를 통해 **테스트 서버에 최신 기능 반영**이 완료됩니다.

- 테스트 서버에는 develop 브랜치의 모든 누적 기능이 반영되므로, 클라이언트는 **현재까지 개발된 모든 기능을 한 곳에서 테스트**할 수 있습니다. 개발자는 클라이언트에게 테스트를 요청하고, **요류나 개선사항 피드백**을 받으면 다시 해당 feature 브랜치나 새 브랜치에서 수정 후 develop에 반영합니다. 이 과정을 통해 운영 배포 전 충분한 검증을 거칩니다.

요약하면, **기능별 브랜치로 개발 → develop에 병합 → 테스트 서버 배포** 순으로 진행하여, 코드 이력은 Git으로 관리하고 배포 대상 WAR 파일은 항상 명확한 브랜치 (develop 혹은 main)의 코드를 사용하도록 하는 것입니다. 이렇게 하면 협업 충돌을 줄이고, 어떤 기능이 어느 배포에 포함되는지 추적하기 쉬워집니다.

## 운영 배포 단계: main 브랜치 릴리스 및 일부 기능만 배포하기

운영 서버에 코드를 배포할 때는 **main 브랜치의 코드를 기반으로 WAR 파일을 만들어 배포**하는 방식으로 전환합니다. 이렇게 하면 앞서 개발된 여러 기능 중 **운영에 내보낼 기능만 선별**하여 포함시킬 수 있습니다. 시나리오별로 배포 절차를 정리하면 다음과 같습니다:

- 전체 기능 배포 (develop의 모든 변경 반영):** 이번 배포에 develop 브랜치에 있는 **모든 새로운 기능을 운영에 반영**하기로 했다면, 간단히 **develop 브랜치의 변경을 main 브랜치에 병합**하면 됩니다. GitHub Desktop에서 `main` 브랜치를 체크아웃하고 **\*\*\*"Choose a branch to merge into main"**으로 develop을 선택, **\*\*\*"Merge develop into main"**을 수행합니다. 병합 커밋을 **Push**하여 원격 main을 업데이트한 뒤, **STS에서 main 브랜치 코드로 WAR 빌드**를 합니다. 이 WAR를 **\*\*REAL 톰캣의 `webapps` \*\***에 업로드하면 운영 배포 완료입니다. 이 방식은 **일괄 릴리스**에 해당하며, develop상의 기능이 모두 충분히 테스트되고 안정적인 때 주로 사용합니다. (병합 전 develop 브랜치에서 한번 더 통합테스트를 완료하고 오는 것을 권장합니다.)
- 부분 기능 배포 (일부 기능만 선별 반영):** 문제가 되는 부분은 운영에 특정 기능만 넣고, 나머지 개발 중인 기능은 배제해야 할 때입니다. 기존에는 코드를 수동 삭제했지만, 개선된 프로세스에서는 **Git의 이력 관리 기능을 활용**합니다. 대표적인 방법 두 가지는 **Cherry-pick**과 **릴리즈 브랜치(release branch)** 활용입니다:

**a) Cherry-pick으로 특정 커밋만 가져오기:** GitHub Desktop의 **Cherry-pick** 기능을 사용하면, 다른 브랜치의 특정 커밋을 선택하여 현재 브랜치에 적용할 수 있습니다. 예를 들어 develop 브랜치에 5개의 기능 커밋이 있지만, 그 중 2개 기능만 운영에 내보내려 한다면, **main 브랜치에서 develop의 해당 커밋 2개를 cherry-pick**하여 가져오는 식입니다. GitHub Desktop에서 `main` 브랜치를 체크아웃한 뒤, 왼쪽 **History** 목록에서 필요한 커밋을 선택하고 **오른쪽 클릭 → Cherry pick commit**을 선택합니다. 그리고 **해당 커밋을 적용할 브랜치로 main**을 지정하면, 선택한 변경분이 main 브랜치에 복사됩니다. (여러 커밋인 경우 Shift/Ctrl로 복수 선택하여 한꺼번에 cherry-pick 할 수도 있습니다.) Cherry-pick을 마치면 GitHub Desktop에서 main 브랜치에 새로운 커밋으로 추가되며, 이를 **Push**하여 원격 main에 반영합니다. 이렇게 하면 develop에 섞여 있던 다른 기능들은 가져오지 않고, 원하는 변경만 **운영 브랜치에 적용**할 수 있습니다. Cherry-pick한 후에는 **main 브랜치 기준으로 WAR 파일을 빌드 및 운영 배포**하면 됩니다. (참고: **cherry-pick**은 특정 변경만 선별 적용하고 싶을 때 유용한

Git 기능입니다. GitHub Desktop에서도 GUI로 제공되어 초보자도 비교적 쉽게 사용할 수 있지만, 적용한 커밋이 나중에 develop과 충돌하지 않도록 유의해야 합니다.)

b) Release 브랜치로 배포 준비하기: 또 다른 방법은 릴리즈 브랜치를 활용하는 것입니다. 이는 Git-Flow에서 권장되는 방식으로, 운영 배포를 위해 develop에서 분기한 임시 브랜치( `release/x.y` )를 만들어 배포 대상만 포함되도록 조정한 후, main에 병합하는 절차입니다. 예를 들어 현재 develop에 5개 기능이 있고 그 중 2개만 출시한다면, develop으로부터 `release-1.0` 브랜치를 생성합니다. 그 다음 release-1.0 브랜치에서 제외할 기능 관련 코드를 revert하거나 해당 커밋을 되돌리는 작업을 합니다 (이는 수동 삭제 대신 Git 이력으로 취소하는 것임). release 브랜치에서 배포 준비가 완료되면, release 브랜치를 main에 병합하고 main을 업데이트합니다. 그리고 main에서 WAR를 빌드해 운영 배포합니다. 배포 후에는 release 브랜치를 삭제하고, (만약 release 동안 수정한 사항이 있으면 develop에도 반영합니다.) 이 방법은 cherry-pick보다 절차가 약간 복잡하지만, 다음 릴리즈를 준비하는 동안에도 develop 브랜치는 계속 개발을 진행할 수 있는 장점이 있습니다.

대부분의 소규모 프로젝트에서는 Cherry-pick 방식이 간편하고 빠를 것입니다. 두 명이 협업하는 현재 규모에서는, cherry-pick으로 main에 필요한 커밋만 골라 적용한 뒤 WAR를 배포하는 방식을 추천합니다. 중요한 점은 수동으로 소스 파일을 잘라내지 않고 Git 이력을 통해 선별한다는 것입니다. 이를 통해 실수를 크게 줄이고, 나중에 어떤 커밋들이 운영에 반영됐는지 명확히 추적할 수 있습니다.

- TIP: 운영에 내보내지 않은 기능들은 여전히 develop 브랜치에 남아 있으므로, 다음 번 배포 때 해당 기능들을 출시할 수 있습니다. 단, 이전에 cherry-pick으로 일부 커밋만 가져왔다면, 나중에 develop을 통째로 main에 병합할 때 중복 커밋 또는 충돌이 생길 수 있습니다. 이때는 Git이 자동 인식해 대부분 해결해주지만, 혹 충돌이 나면 수동으로 해결해야 합니다. Cherry-pick한 커밋을 develop에서 나중에 병합할 때 충돌 가능성이 있다는 점만 기억하고, \*\*배포 후에도 develop과 main의 차이를 주기적으로 정리(동기화)\*\*해주는 것이 좋습니다. 예를 들어, cherry-pick으로 main에 반영한 기능 커밋들을 나중에 develop에서 제거하거나(main에 반영됐으니 중복 방지 차원) 또는 main -> develop 병합하여 동일한 상태로 맞춰두는 방법이 있습니다. 이것은 약간 고급 주제이므로, 초반에는 Cherry-pick 후 가능한 빨리 main의 변경을 develop에도 반영해서 두 브랜치 차이를 줄여두는 것을 권장합니다.

6. 운영 WAR 빌드 및 배포: main 브랜치에 최종적으로 운영에 내보낼 코드가 모두 준비되었으면, STS에서 main 브랜치의 코드를 사용하여 WAR 파일을 빌드합니다. (GitHub Desktop에서 main 브랜치를 최신 상태로 체크아웃한 뒤, IDE에서 Export WAR 수행). 이 WAR 파일을 운영 서버(REAL 톰캣)의 `webapps` 폴더에 업로드합니다. 운영 톰캣에 적용할 때는 테스트 때와 마찬가지로, 기존 WAR/폴더 정리 -> 신규 WAR 복사 -> 톰캣 재시작 단계를 거칩니다. 이제 운영 서버는 main 브랜치와 동기화된 최신 코드로 동작하게 됩니다.

- 배포 전에 \*\*버전 태그(tag)\*\*를 지정해두면 좋습니다. 예를 들어 이번 배포본에 `v1.0` 태그를 main 브랜치 커밋에 붙여두면, 나중에 이 버전으로 되돌아가야 할 때 쉽게 해당 커밋을 찾을 수 있습니다. GitHub Desktop에서는 태그 관리 기능이 없으므로 Git Bash 등을 사용해야 하지만, 태그 없이도 커밋 해시나 메시지로 구분하면 됩니다. 또한 운영 배포 전/후로 릴리스 노트나 배포 기록을 남겨두면, 어떤 기능이 언제 배포되었는지 파악하기 쉽습니다.

## 협업 중 충돌/실수 대응 및 안전한 운영 관리

두 명이 협업하면서 겪을 수 있는 충돌, 실수, 긴급 복구(rollback) 상황에 대비하는 방법을 정리합니다:

- Merge 충돌 해결: 앞서 언급한 대로, GitHub Desktop 병합 시 충돌이 나면 수동으로 해결해야 합니다. 충돌 메시지가 나오면 "뷰 파일"을 클릭하여 충돌난 파일을 열고, `<<<<<< HEAD` 와 `>>>>>> branch_name` 사이에 겹친 코드를 정리합니다. 필요한 코드만 남기고 불필요한 부분은 삭제한 후 저장하고 닫으면 됩니다. 그런 다음 GitHub Desktop에서 커밋하여 병합을 완료합니다. 혹시 UI로 어려움이 있다면, VS Code에서 소스 제어를 이용하면 충돌 부분을 선택적으로 취합할 수 있는 도구도 있습니다. Tip: 충돌을 줄이려면, 하나의 작업 단위를 한 사람이 전담하고, 작업 중간중간 원격 저장소의 최신 develop을 자주 pull 받아 리베이스하거나 병합해서 내 로컬 브랜치를 최신으로 유지하는 습관이 필요합니다. 또한 커밋 전에 GitHub Desktop의 변경 내역(diff) 화면을 통해 내 변경사항과 다른 사람의 변경사항을 확인하면 미리 충돌을 예측할 수 있습니다.



- **잘못된 커밋 취소(Revert):** 개발을 하다 보면 잘못된 코드나 불필요한 변경을 커밋하는 실수가 생길 수 있습니다. 이 경우 **Git의 revert 기능**을 사용하면 이전 상태로 쉽게 되돌릴 수 있습니다. GitHub Desktop에는 직접적인 UI 버튼은 없지만, 두 가지 방법이 있습니다: (1) 취소하려는 커밋을 선택하고 마우스 오른쪽 클릭하여 **“Revert commit”** 옵션을 사용 (만약 해당 옵션이 없다면 최신 버전 Desktop이 아닌 경우 CLI 사용) 또는 (2) 수동으로 해당 커밋의 반대 변경을 가한 새 커밋을 만드는 것입니다. 예를 들어, 잘못된 코드라면 해당 부분을 올바르게 고쳐 커밋하면 결과적으로 되돌린 효과가 납니다. 만약 이미 **main에 잘못 반영**되어 운영에 문제가 생긴 상황이라면, **신속히 hotfix 브랜치**를 만들어 수정 후 **main에 병합**하는 방법도 있습니다. (hotfix 브랜치는 main에서 분기하여 급한 수정만 진행하고 바로 main에 머지하는 브랜치 전략으로, 운영중 긴급 버그에 대응할 때 쓰입니다.)
- **운영 버전 롤백(Rollback):** 만약 새로 배포한 버전에 중대한 문제가 발견되어 **즉시 이전 버전으로 복구**해야 한다면, **이전 WAR 파일을 다시 배포**하는 방법이 가장 빠릅니다 (현재도 사용 중인 방식). 이를 보다 체계화하려면, **Git에서 이전 버전 커밋으로 체크아웃하여 WAR 빌드**를 할 수도 있습니다. 예를 들어 main 브랜치의 과거 태그( `v0.9` 등)나 커밋 해시를 GitHub Desktop에서 **History**에서 찾아 해당 커밋에서 **\*\*\*New Branch from commit\*\*\***을 실행하면 과거 시점 코드로 브랜치를 만들 수 있습니다. 그 상태에서 WAR를 빌드해 운영에 올리면 코드 수준에서 완벽히 롤백이 됩니다. 다만 이 경우 현재 develop과 main에는 문제가 있던 커밋이 남아있으므로, 이후에 그 부분을 제거하거나 수정하는 별도 조치(revert commit)가 필요합니다. **간단 대처:** 실무에서는 운영 장애 시 우선 **이전 안정 버전의 WAR를 백업에서 복원**하고 (이미 백업 WAR를 보관하고 계시므로 이것으로 즉시 복구), 추후 해당 문제 커밋을 제거(revert)하여 main을 바로잡는 방식을 취하면 됩니다.
- **안전한 협업 습관:** 2인 개발 환경에서는 서로의 작업을 자주 공유하고 확인하는 것이 중요합니다. **GitHub Desktop**만으로 협업할 경우, **커밋 메시지에 작업 내용을 명확히** 적고, 수시로 GitHub 저장소를 확인하여 동료의 커밋을 살펴보세요. 또한 **Pull Request를 통한 코드 리뷰**를 도입하면 실수를 발견하거나 서로의 코드를 이해하는 데 도움이 됩니다. 초보자의 경우 구글 시트나 노션 등에 **간단한 작업 일정 및 브랜치 현황표**를 만들어 놓고 “어느 브랜치에 어떤 기능이 들어있다”를 명시해두면 머지 대상 혼동을 줄일 수 있습니다.
- **테스트 단계 강화:** 운영 배포 전에 **클라이언트 테스트를 충분히 거치는 것**이 가장 기본적인 안전장치입니다. 현재도 개발자들이 직접 테스트 후 클라이언트 UAT를 받고 있지만, 개선된 프로세스 하에서는 **main 브랜치로 릴리즈 준비를 할 때 한 번 더 검증 단계**를 추가할 수 있습니다. 예컨대, main에 cherry-pick이나 release 브랜치로 조정이 끝난 코드베이스를 **테스트 서버에 임시로 올려 최종 리그레션 테스트**를 하는 것입니다. 비록 테스트 서버는 기본적으로 develop용이지만, 중요한 릴리스 전에는 main 브랜치 버전의 WAR를 테스트 서버에 덮어 써서 (혹은 별도의 스테이징 환경이 있다면 이상적) **실제 운영 직전의 상태를 미리 검증**하면 운영 장애를 예방할 수 있습니다. 검증이 끝나면 다시 develop 버전 WAR로 테스트 서버를 원복하거나, 바로 다음 개발 사이클로 진행하면 됩니다.

## 프로세스 최적화 및 추가 개선 방안

개선안을 적용함으로써 기대되는 **효율성과 안정성 향상** 및 현재 환경에서 고려할 점은 다음과 같습니다:

- **불필요한 수작업 감소:** Git 브랜치 분리를 통해 더 이상 운영 배포 시 코드를 일일이 삭제/추리는 작업이 필요 없습니다. 이에 따라 **인적 오류를 크게 줄이고**, 배포 준비 시간도 단축됩니다. 또한 모든 변경 이력이 Git으로 남기 때문에 “어떤 파일을 뺐더니 문제가 생겼다”와 같은 상황에서도 **이력 추적이 용이**합니다.
- **기능 혼입 방지:** main 브랜치에 승인된 기능만 병합하므로, 테스트 단계에서 존재하던 **미완성 기능이나 승인되지 않은 코드는 운영으로 절대 유출되지 않습니다**. 이는 곧 운영 안정성으로 직결됩니다. 만약 특정 기능을 배포 대상에서 제외했다면, main에는 그 커밋이 없으므로 운영 WAR에는 포함될 수 없습니다.
- **협업 생산성 향상:** 두 개발자가 각각 feature 브랜치로 작업하면서 **동시에 여러 기능을 병렬 개발**할 수 있습니다. GitHub Desktop만으로도 브랜치 생성/전환이 쉬워 초보자도 금방 적응할 수 있습니다. 브랜치를 통한 협업은 코드 충돌 가능성을 낮추고, 혹시 문제가 생겨도 개별 브랜치 수준에서 해결할 수 있습니다. 또한, **문제 발생 시 영향 범위가 격리**되므로 한 기능의 오류가 다른 기능 개발에 지장을 줄 확률이 줄어듭니다.
- **효율적인 롤백 및 핫픽스:** 이전에는 WAR 백업에 의존한 롤백을 했지만, 이제 Git 브랜치/커밋을 통해 **정교한 롤백**이 가능합니다. 예를 들어 운영에서 버그 발견 시, hotfix 브랜치를 만들어 긴급 수정 후 main에 반영하면

신속히 패치 적용이 가능하고, 그 사이 문제가 된 배포를 건너뛰고 이전 버전을 유지할 수도 있습니다. Git을 활용하면 여러 버전 코드를 동시에 관리할 수 있어 유연성이 높아집니다. (물론 여전히 즉각 복구를 위해 WAR 백업은 좋은 대비책이며, 병행해서 유지해야 합니다.)

- **형상 관리 도입으로 인한 초기 비용:** 개선된 프로세스를 정착하려면 초반에 **팀원의 Git 이해도 향상**과 **브랜치 전략 숙지**가 필요합니다. 두 분 다 GitHub Desktop을 사용 중이라면 GUI로 대부분 작업이 가능하지만, 용어 (예: cherry-pick, revert 등)가 생소할 수 있습니다. **팀 내부 교육**을 통해 1~2일 정도 Git 사용 방법과 브랜치 전략에 대해 공유하세요. 또한, 실제 적용 초기에 예상치 못한 충돌이나 실수가 있을 수 있으므로, **\*\*테스트 프로젝트로 연습(branch 생성/병합 실습)\*\***해보는 것도 도움이 됩니다.
- **자동화 도구 부재에 따른 대응:** 사내 보안 정책으로 CI/CD나 배포 자동화가 막혀있지만, **형상 관리 수준에서의 자동화**는 시도해볼 수 있습니다. 예를 들어, Maven을 사용하고 있으므로 `pom.xml` 에 **\*\*프로파일(profile)\*\***을 두어 개발용/운영용 빌드 옵션을 달리하거나, 간단한 배치 스크립트를 작성하여 **WAR 빌드 + 원격지 업로드를 한 번에 수행**하도록 할 수 있습니다. (VDI 환경에서 scp나 ftp 업로드 스크립트를 짤 수 있다면 오류 없이 배포하는 데 도움이 될 것입니다.) 이러한 자동화는 서버에 에이전트를 설치하지 않고도, **개발자 PC에서 배포를 좀 더 쉽게 해주는 수준**이므로 보안 정책을 크게 해치지 않으면서 실수는 줄여줄 수 있습니다. 다만, 이런 스크립트 사용이 어려울 경우 현재 방식으로 수동 업로드하되 **체크리스트**를 만들어 “브랜치 확인 -> WAR 빌드 -> 운영 토크트 경로 확인 -> 업로드” 순으로 꼼꼼히 따라가도록 하는 것도 좋습니다.

마지막으로, **GitHub Desktop을 통한 개선 프로세스** 적용 후에도 정기적으로 회고를 통해 무엇이 불편한지 살펴보세요. 예를 들어 브랜치가 너무 많아 관리가 어렵다면 전략을 단순화하고, cherry-pick이 번거롭게 느껴지면 release 브랜치를 사용하는 식으로 팀에 맞게 조정하면 됩니다. **지속적인 프로세스 개선**을 통해 현재의 제약 하에서도 최대한 효율적으로 개발/배포를 진행할 수 있을 것입니다.

위의 개선안을 요약하면: **\*\*\*Git 브랜치를 활용하여 개발용 소스와 운영용 소스를 철저히 분리 관리하고, GitHub Desktop으로 손쉽게 브랜치 생성/병합 및 이력 관리를 수행함으로써, 수동 배포 환경에서도 오류 없이 원하는 기능만 선택 배포한다.\*\*\***는 것입니다. 초기에는 다소 생소할 수 있지만, 작은 팀에서 **규모에 맞는 Git 전략**을 쓰면 충분히 효과를 볼 수 있으니 천천히 적용해 보길 권장드립니다. 필요한 경우 관련 이미지를 보며 한 단계씩 따라하면 금방 익숙해질 것이며, 추후 Git 사용 경험이 쌓이면 자동화 도구 없이도 현재 환경에서 **안전하고 체계적인 배포**를 할 수 있게 될 것입니다.

**참고 자료:** GitHub 공식 문서 – GitHub Desktop 브랜치/병합 가이드, Cherry-pick 활용 가이드 등. 예시 이미지는 GitHub Desktop 사용 화면 및 Git-Flow 브랜치 모델을 참고용으로 첨부했습니다.

위의 심층분석된 내용을 .md 파일형식으로 받을 수 없어?

네, 바로 가능합니다! 아래는 위 내용을 **Markdown(.md) 파일 형식**으로 보기 좋게 구조화해서 제공합니다. 마크다운 특성상 실제 이미지는 첨부할 수 없으니, [이미지 참고] 부분은 설명으로만 넣어둘게요.

# GitHub Desktop 기반 실전 소스/배포 관리 프로세스 개선안

---

## 1. 현재 프로세스의 문제점과 개선 필요성

- **\*\*테스트 서버에 모든 신규 기능이 한꺼번에 반영\*\***

→ 운영에 나가면 안 되는 기능까지 포함되어, 서비스 오작동 및 클라이언트 민원 발생



- **\*\*운영에 일부 기능만 반영하려는 과정에서 수동 실수\*\***  
→ 코드 누락, 불필요한 변경 포함 위험

- **\*\*브랜치 병합/롤백/협업 충돌 관리 어려움\*\***

- **\*\*자동화된 배포 도구 없이 수동 WAR 빌드 및 업로드\*\***  
→ 휴먼 에러 가능성↑

**\*\*→ 해결 방향:\*\***

Git 브랜치 전략을 도입, **\*\*소스 관리 체계화 + 배포 절차 개선\*\*** 필요

---

## ## 2. 개선 브랜치 전략 (feature / develop / main 분리)

- **\*\*main 브랜치 (운영용) :\*\***  
운영 서버 배포 대상 “안정 코드”만 저장
- **\*\*develop 브랜치 (개발 통합) :\*\***  
전체 기능을 통합, 테스트 서버 배포 기준
- **\*\*feature 브랜치 (기능별) :\*\***  
각 기능별로 분리 개발, 개발 후 develop로 병합

> **\*\*이미지 참고:\*\***

> Git-Flow 브랜치 모델: main(운영) / develop(테스트) / feature(기능)로 분리

---

## ## 3. 개발 단계 (feature → develop → 테스트 서버 배포)

### 1. **\*\*feature 브랜치 생성\*\***

- GitHub Desktop → `develop` 기준 새 브랜치 생성
- 예시: `feature/payment`

### 2. **\*\*STS4에서 개발 & 커밋\*\***

- 해당 feature 브랜치에서 작업
- 의미 있는 단위로 자주 커밋 & Push

### 3. **\*\*코드 리뷰/검토\*\***

- Pull Request 생성(권장), 변경 내역 상호 확인

### 4. **\*\*develop 브랜치로 병합\*\***

- GitHub Desktop에서 develop로 머지
- 병합 후 Push
- 충돌 발생 시: 충돌 파일 수정 후 커밋

## 5. \*\*테스트 서버에 배포\*\*

- develop 최신 상태로 WAR 파일 export
- DEV 톱캣에 업로드

> \*\*팁:\*\*

- > develop 브랜치에 있는 기능 모두 테스트 서버에 반영됨.
- > 오류/피드백 발생 시 다시 feature 브랜치에서 수정 후 재반영

---

## ## 4. 운영 배포 단계 (main 기반 WAR 배포, 기능 일부만 반영)

### - \*\*전체 기능 운영 반영\*\*

- develop → main으로 머지 → main에서 WAR 빌드 & 운영 배포

### - \*\*일부 기능만 운영 반영\*\*

#### 1. \*\*Cherry-pick 활용\*\*

- main에서 develop의 특정 커밋만 cherry-pick
- main 기준으로 WAR 빌드 & 운영 배포
- (GitHub Desktop에서 cherry-pick 지원)

#### 2. \*\*릴리즈 브랜치 활용\*\*

- develop에서 release 브랜치 생성, 필요한 커밋만 남기고 정리
- release 브랜치에서 main으로 머지 후 배포

> \*\*TIP:\*\*

- > 운영에 내보내지 않은 기능은 main에 없으므로, 실수로 운영에 반영되는 일 없음

---

## ## 5. 협업 중 충돌/실수/롤백 관리

### - \*\*병합 충돌 해결\*\*

- GitHub Desktop에서 충돌 파일 직접 수정
- 필요 시 VS Code, STS 사용

### - \*\*잘못된 커밋 되돌리기\*\*

- "Revert commit" 또는 수동 수정 후 새 커밋

### - \*\*운영 롤백\*\*

- 이전 WAR 파일로 복구
- 또는 main 브랜치의 과거 커밋/태그에서 새 브랜치 생성 후 WAR 빌드

### - \*\*핫픽스\*\*

- main에서 hotfix 브랜치 분기, 수정 후 머지

### - \*\*협업 실수 방지\*\*

- 커밋 메시지 명확하게, 작업 브랜치/내용 공유
- Pull Request 활용, 작업 내역 주기적 점검

---

## ## 6. 추가 개선/최적화 팁

- **\*\*불필요한 수작업/오류 감소:\*\***
  - 운영 배포 때 수동 소스 삭제 불필요, 히스토리 추적 쉬움
- **\*\*효율적 롤백/핫픽스:\*\***
  - Git 이력 기반으로 손쉽게 복구
  - 여전히 WAR 파일 백업 병행 권장
- **\*\*초기 정착 시 팀원 교육 추천:\*\***
  - branch 전략, cherry-pick, revert 등 실습
- **\*\*테스트 단계 강화:\*\***
  - main 브랜치 코드로 WAR 빌드 후 테스트 서버에 올려 사전 검증 가능

---

## ## 7. 한 줄 요약

- > **\*\*브랜치** (특히 main/feature/develop)로 개발·운영 소스를 명확히 분리 관리,
- > GitHub Desktop으로 GUI 환경에서 손쉽게 병합/이력 추적/일부 기능만 안전하게 운영 반영 및 롤백까지 가능

---

### **\*\*참고:\*\***

- GitHub Desktop 공식 도움말
- Git-Flow 브랜치 전략
- cherry-pick/릴리즈 브랜치 실습 자료



이대로 복사해서 `.md` 파일로 저장하면, 바로 사용 가능합니다! 혹시 추가로 “실제 브랜치 생성/머지/cherry-pick” 화면 예시가 필요하면 캡처 이미지 설명도 첨부해드릴 수 있어요. **다른 포맷이나 항목 강조/축소 요청도 언제든지 말씀하세요!**