

DevOps Shack Git Assignment | Task:6

Task 6: Tagging Releases

6.1 Introduction to Git Tags

In Git, **tags** are **pointers to specific commits**. Unlike **branches** (which move with new commits), **tags are fixed** references, typically used to **mark important milestones** like:

- **Releases** (v1.0, v2.0).
- **Patches** (hotfix-2024-04-01).
- **Deployments**.

Think of a **tag** as a **bookmark** in the commit history—it helps you quickly **identify, reference, and return to** significant points in your project timeline.

Why Git Tags Matter in Real-World Projects

In **corporate DevOps environments**, Git tags:

1. **Mark release points:**
 - Example: v1.0, v2.1-beta.
2. **Trigger CI/CD pipelines:**
 - Some CI/CD tools (like **GitHub Actions, GitLab CI, Jenkins**) can detect **tags** and automatically:
 - Build release artifacts.
 - Deploy software to production.
3. **Facilitate rollbacks:**
 - If a release breaks, teams can **checkout the previous tag** (e.g., v1.0) and redeploy a **stable version**.
4. **Provide a clear release history:**
 - Teams can **trace what changes** went into each **release**.

Analogy:

Tags are like **sticky notes in a textbook**—marking **important pages** (releases), so you can find them **easily later**.

6.2 Types of Git Tags

Git supports **two types of tags**:

Tag Type	Description	Use Case
Lightweight	- A simple pointer to a commit.	- Temporary tags.
	- No metadata (author, timestamp, message).	- Quick references.
Annotated	- A full Git object (like a commit) with metadata.	- For releases and public tags .
	- Includes author, date, message, GPG signature (optional).	- Better for long-term reference .

Best practice:

Always use annotated tags for releases.

6.3 How Git Tags Work Internally

- A **tag** is a **reference (pointer)** stored in:
 - `.git/refs/tags/`
 - For **annotated tags**:
 - Git creates a **tag object**:
 - Contains **metadata** (author, timestamp, message).
 - References a **commit hash**.
 - For **lightweight tags**:
 - Just a **pointer to a commit hash** (no object created).
-

6.4 Step-by-Step Implementation of Git Tagging

Scenario Setup:

You've completed several commits in the **main** branch and are ready to **mark a release point** (v1.0).

Step 1: Identify the Commit to Tag

- Generally, this is the **latest commit on main**.
- Use `git log --oneline` to view recent commits.

c4d1a3b Finalize backend logic

9b2e33a Merge feature-frontend into main

e3f8d1c Initial commit with README

Step 2: Create an Annotated Tag

`git tag -a v1.0 -m "Release version 1.0: Includes frontend and backend features"`

- `-a`: Create **annotated tag**.
 - `v1.0`: **Tag name**.
 - `-m`: **Tag message**.
-

What Happens Internally:

- Git creates a **tag object**:
 - Stores:
 - **Tagger information** (author, date).
 - **Message**.
 - **Reference to the commit**.
-

Step 3: Verify Tags

`git tag`

- Lists **all tags**.

To see **tag details**:

`git show v1.0`

- Displays:
 - **Commit referenced**.
 - **Metadata** (author, timestamp).
 - **Tag message**.
-

Step 4: Push Tags to Remote

By default, **tags are not pushed** with `git push`.

To push the **specific tag**:

```
git push origin v1.0
```

To push **all tags**:

```
git push origin --tags
```

Step 5: Checkout a Tag (Detached HEAD)

To **checkout a tag**:

```
git checkout v1.0
```

Important:

- You enter a **detached HEAD state** (not on any branch).
- Useful for **viewing** or **testing** code at that point.

To return to a branch:

```
git checkout main
```

6.5 Lightweight Tags (For Quick References)

Create a **lightweight tag**:

```
git tag v1.0-light
```

- No metadata, no message—just a **commit pointer**.
-

6.6 Visualizing Tags in Commit History

Without Tags:

```
A---B---C---D (main)
```

With Tags:

```
A---B---C---D (main, v1.0)
```

- **v1.0** points to commit **D**.

6.7 Advanced Tagging Techniques

Tag a Specific Commit (Not HEAD)

Find the commit hash:

```
git log --oneline
```

Tag a **previous commit**:

```
git tag -a v0.9 <commit-hash> -m "Release v0.9"
```

Signing Tags (For Security)

If using **GPG keys**:

```
git tag -s v1.0 -m "Signed release v1.0"
```

Why sign tags?

- To **verify authenticity** of releases.
-

Delete Tags

Locally:

```
git tag -d v1.0
```

Remotely:

```
git push origin --delete v1.0
```

6.8 Real-World Best Practices for Tagging

1. **Use annotated tags for releases:**
 - Include **meaningful messages**.
2. **Follow consistent naming conventions:**
 - **Semantic versioning:** v1.0.0, v2.1.3-beta.
3. **Document what's included in the tag:**
 - Example: "Release v1.0: Includes frontend auth module and backend API."
4. **Push tags immediately after creation:**
 - Ensures **CI/CD systems** detect and act on tags.
5. **Never rewrite tags for published releases:**

- Tags should be **immutable** once shared.

6.9 Tagging in CI/CD Pipelines

Tags are commonly used to:

1. **Trigger release pipelines:**
 - Example: GitHub Actions triggers **deployments** when a **tag is pushed**.
2. **Deploy specific versions:**
 - Example: Deploy **v2.1.0** to production.
3. **Version artifacts:**
 - CI/CD systems **label Docker images**, binaries, or packages with **Git tags**.

6.10 Common Mistakes & Pitfalls

Mistake	How to Avoid
Using lightweight tags for releases	Use annotated tags with messages .
Forgetting to push tags	Push tags with <code>git push origin --tags</code> .
Reusing tag names	Tags should be unique and immutable .
Not signing critical tags	Use GPG signing for high-security environments.