

# Git Básico

---

**Arturo Silvelo**

Try New Roads

# Buenas Prácticas

# Tags y Releases

---

## ¿Qué son los Tags?

---

Los **tags** son etiquetas que marcan puntos específicos en el historial de Git, típicamente para versiones.

### Características:

- **Inmutables:** Una vez creados, no cambian
- **Referencias fijas:** Apuntan a un commit específico
- **Versionado:** Marcan releases (v1.0, v2.1, etc.)
- **Navegación:** Fácil acceso a versiones anteriores

## Tipos de Tags

---

**Lightweight tags** (etiquetas ligeras):

```
git tag v1.0.0
```

**Annotated tags** (etiquetas anotadas):

```
git tag -a v1.0.0 -m "Release version 1.0.0"
```

**Diferencias:**

- Lightweight: Solo un puntero al commit
- Annotated: Incluye metadata (autor, fecha, mensaje)

## Crear Tags

---

### Tag en el commit actual:

```
git tag v1.0.0  
git tag -a v1.0.0 -m "First stable release"
```

### Tag en commit específico:

```
git tag v0.9.0 abc1234  
git tag -a v0.9.0 abc1234 -m "Beta release"
```

### Ver información del tag:

```
git show v1.0.0
```

## Gestionar Tags

---

### Listar tags:

```
git tag                # Todos los tags
git tag -l "v1.*"      # Tags que coincidan con patrón
git tag --sort=-version:refname # Ordenados por versión
```

### Eliminar tags:

```
git tag -d v1.0.0      # Local
git push origin --delete tag v1.0.0 # Remoto
```

## Subir Tags al Remoto

---

### Un tag específico:

```
git push origin v1.0.0
```

### Todos los tags:

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

### Push con tags automáticamente:

```
git push --follow-tags
```



## Semantic Versioning (SemVer)

---

**Formato:** MAJOR.MINOR.PATCH

- **MAJOR** (1.x.x): Cambios incompatibles
- **MINOR** (x.1.x): Nuevas funcionalidades compatibles
- **PATCH** (x.x.1): Correcciones de errores

### Ejemplos:

```
git tag v1.0.0    # Primera versión estable
git tag v1.1.0    # Nueva funcionalidad
git tag v1.1.1    # Corrección de bugs
git tag v2.0.0    # Breaking changes
```

## Checkout a Tags

---

**Ver código de una versión específica:**

```
git switch --detach v1.0.0
```

**Crear rama desde tag:**

```
git switch -c hotfix-v1.0.0 v1.0.0
```

**Volver a la rama actual:**

```
git checkout main
```

# Releases en GitHub

---

## ¿Qué son los Releases?

---

Los **releases** en GitHub son versiones empaquetadas de tu software basadas en tags.

**Incluyen:**

- **Tag asociado:** Versión específica del código
- **Notas de release:** Descripción de cambios
- **Assets:** Archivos binarios, documentación
- **Changelog:** Lista de cambios desde la versión anterior

## Crear Release en GitHub

---

1. Ir a la pestaña **Releases** del repositorio
2. **"Create a new release"**
3. **Seleccionar tag** (o crear uno nuevo)
4. **Título del release:** v1.0.0 - Primera versión estable
5. **Descripción:** Cambios, mejoras, correcciones
6. **Assets:** Subir archivos (opcional)
7. **Publish release**

# Release Notes - Ejemplo

## # Release v1.2.0 - New Authentication System

### ## 🚀 New Features

- User authentication with JWT tokens
- Password reset functionality

### ## 🐛 Bug Fixes

- Fixed login form validation
- Resolved memory leak in user sessions
- Corrected timezone handling

### ## 🛠️ Technical Changes

- Upgraded React to v18
- Improved database performance

### ## ⚠️ Breaking Changes

- API endpoints now require authentication
- Changed user object structure

## Flujo completo de Release

---

```
# 1. Finalizar desarrollo
git checkout main
git pull origin main

# 2. Crear tag anotado
git tag -a v1.2.0 -m "Release version 1.2.0"

# 3. Subir tag
git push origin v1.2.0

# 4. En GitHub: Crear release desde el tag
# 5. Escribir release notes
# 6. Publicar release
```

# Flujos de Trabajo en Git

---



## ¿Por qué necesitamos flujos de trabajo?

---

Sin un flujo definido, los equipos enfrentan:

- **Conflictos constantes** entre desarrolladores
- **Código inestable** en ramas principales
- **Releases caóticas** sin control
- **Falta de trazabilidad** de cambios

Un flujo de trabajo define **cómo y cuándo** se integran los cambios. No hay una estrategia perfecta, simplemente se usa la que mejor se adapte al proyecto:

- Git Flow
- Github flow
- GitLab flow
- Trunk Based
- Shop/Shop/Ask

# Git Flow

---

El flujo más tradicional y estructurado para equipos grandes.

## Ramas principales:

- `main`: Código en producción (estable)
- `develop`: Rama de desarrollo principal

## Ramas de soporte:

- `feature/*`: Nuevas funcionalidades
- `release/*`: Preparación de releases
- `hotfix/*`: Correcciones urgentes

## Git Flow - Características

---

### Ventajas:

- Control total sobre releases
- Historial muy organizado
- Ideal para software con versiones

### Desventajas:

- Muy complejo para equipos pequeños
- Releases lentos
- Muchas ramas que mantener

## Git Flow - Flujo Feature

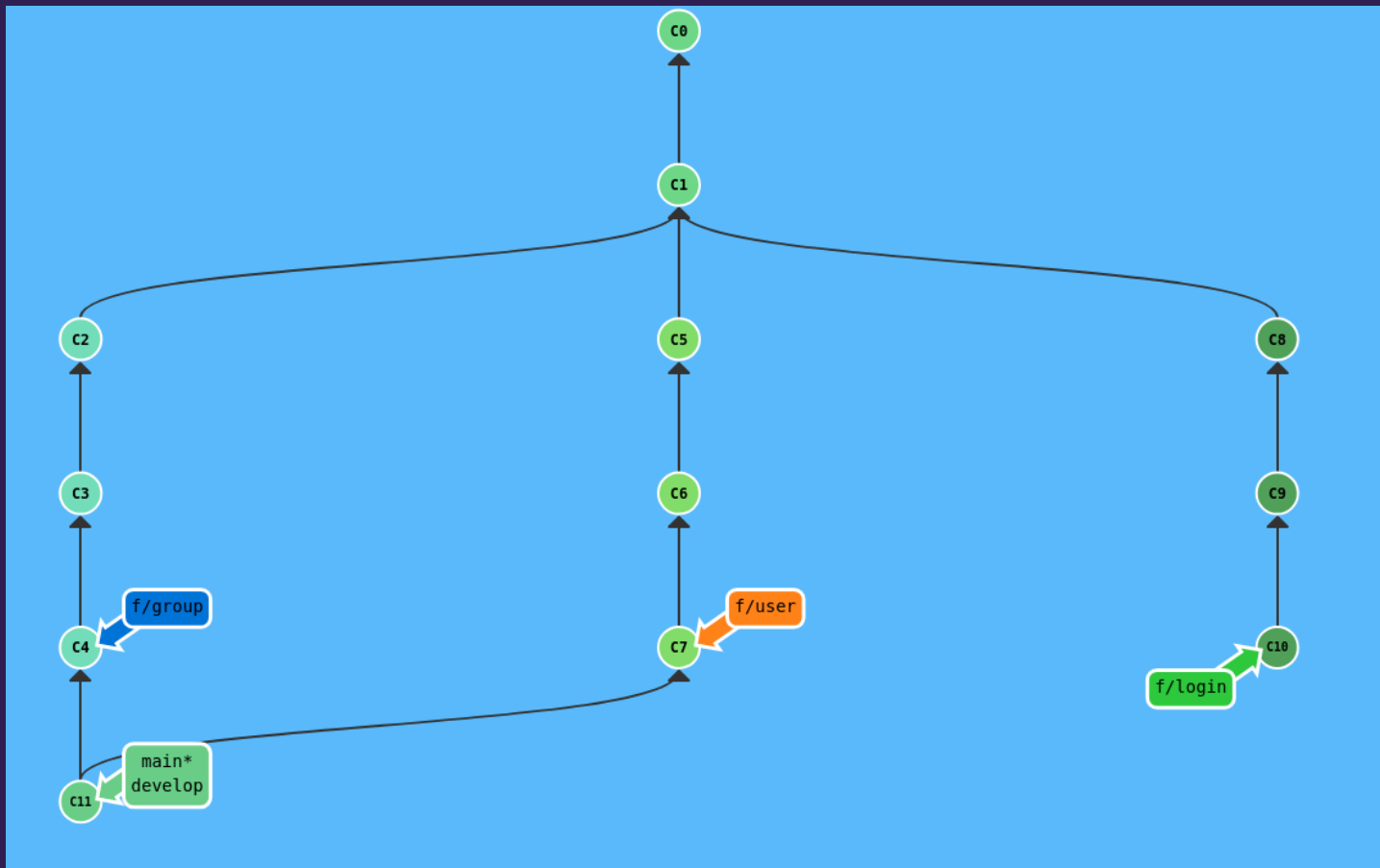
---

```
# Crear feature desde develop
git switch -c develop
git switch -b feature/login

# Desarrollo y commits
git commit -m "feat: add login form"
git commit -m "feat: add validation"

# Merge a develop
git switch develop
git merge feature/login-usuario
```

# learn git branching



# GitHub Flow

---

El flujo más simple y directo para desarrollo continuo.

## Rama principal:

- `main`: Siempre desplegable y estable

## Ramas de trabajo:

- `feature/*`: Nuevas funcionalidades
- `bugfix/*`: Corrección de errores
- `hotfix/*`: Correcciones urgentes

**Principio:** Todo sale y entra de `main`

## GitHub Flow - Características

---

### Ventajas:

- Muy simple de entender y aplicar
- Deploy continuo natural
- Ideal para equipos ágiles
- Menos overhead de gestión

### Desventajas:

- Requiere CI/CD robusto
- `main` debe estar siempre estable
- No ideal para releases planificados



## GitHub Flow - Flujo básico

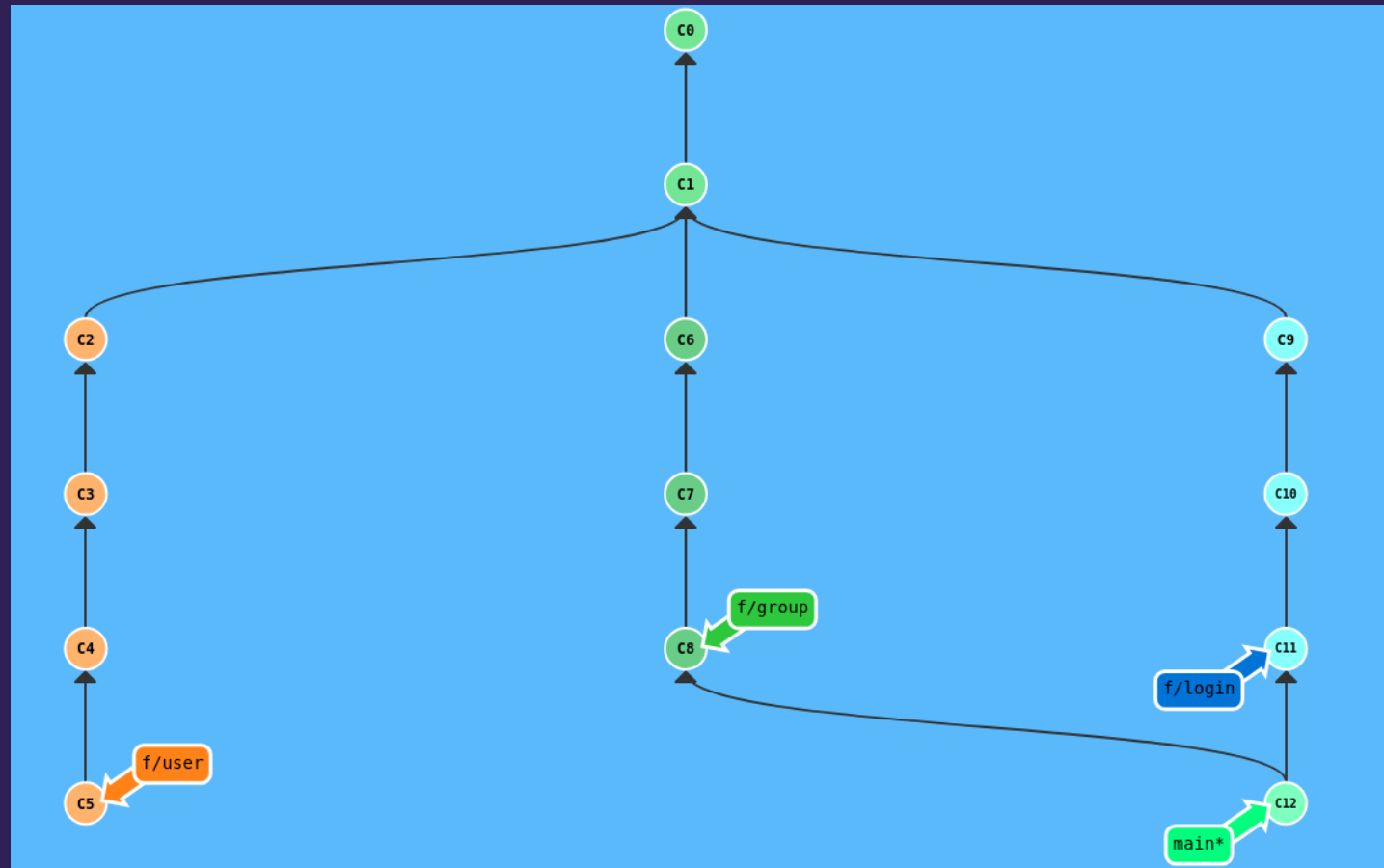
---

```
# Crear feature desde main
git switch main
git pull origin main
git switch -c feature/user-profile

# Desarrollo
git commit -m "feat: add user profile page"
git push -u origin feature/user-profile

# PR → Review → Merge → Deploy
```

# learn git branching



# Trunk-based Development

---

Un tronco principal donde todos integran frecuentemente.

## Rama principal:

- `main` o `trunk`: Única rama principal

## Ramas de trabajo:

- Muy cortas (< 1 día) o commits directos
- `hotfix/*`: Solo para correcciones urgentes

**Principio:** Integración continua real

## Trunk-based - Características

---

### Ventajas:

- Integración continua real
- Sin merge hell
- Deploy frecuente y rápido
- Feedback inmediato

### Desventajas:

- Requiere desarrolladores senior
- CI/CD muy robusto obligatorio
- Feature flags necesarios
- Tests automáticos críticos

## Trunk-based - Flujo básico

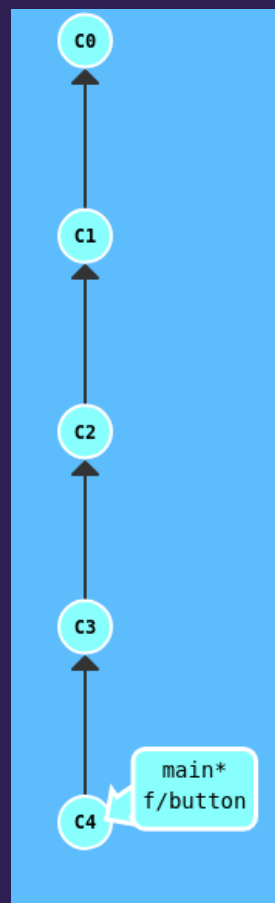
---

```
# Actualizar trunk frecuentemente
git switch main
git pull origin main

# Cambios pequeños directos
git commit -m "feat: add button component"
git push origin main

# 0 ramas ultra-cortas
git switch -c quick-fix
git commit -m "fix: button alignment"
git push origin quick-fix
# PR inmediato → Merge → Delete
```

# learngitbranching



## Ship/Show/Ask

---

Estrategia que combina autonomía con colaboración según el tipo de cambio.

- **Ship** 🚢: Cambios que puedes hacer directamente
- **Show** 👁️: Cambios que compartes para feedback
- **Ask** ❓: Cambios que requieren discusión previa
- **Principio**: Balance entre velocidad y control

## Ship/Show/Ask - Características

---

### Ventajas:

- Autonomía para desarrolladores senior
- Velocidad en cambios menores
- Control en cambios críticos
- Cultura de confianza en el equipo

### Desventajas:

- Requiere criterio maduro del equipo
- Necesita desarrolladores experimentados
- Puede crear inconsistencias si no hay criterio
- Difícil para equipos nuevos



## Ship/Show/Ask - Criterios

---

## Ship (Push directo):

- Typos en documentación
- Tests adicionales
- Refactoring menor
- Actualizaciones de dependencias

## Show (PR sin esperar aprobación):

- Features pequeñas
- Cambios de UI menores
- Optimizaciones de rendimiento

## Ask (PR con revisión obligatoria):

- Cambios de arquitectura
- APIs públicas
- Features complejas
- Cambios de seguridad

## Ship/Show/Ask - Flujo

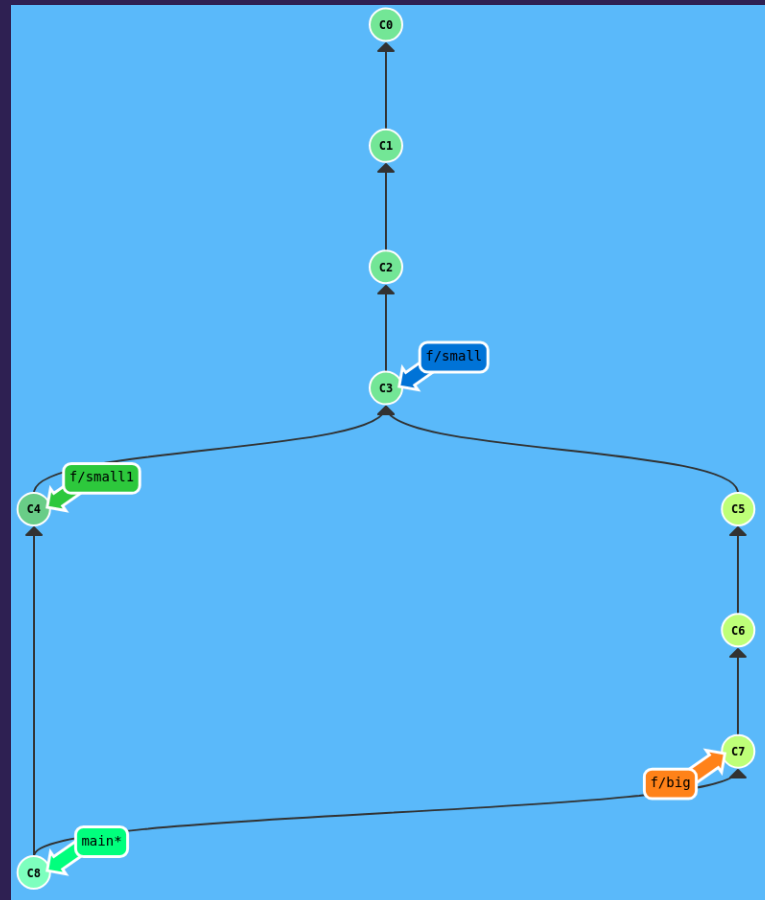
---

```
# SHIP - Fix documentación
git commit -m "docs: fix typo in README"
git push origin main

# SHOW - Feature pequeña
git switch -c small-feature
git push -u origin small-feature
# Create PR → Optional review → Merge

# ASK - Cambio grande
git switch -c big-change
# Discutir primero → PR → Required review
```

# learn git branching



# Git Hooks

---

## ¿Qué son los Git Hooks?

---

Scripts que se ejecutan automáticamente en ciertos eventos de Git:

- **Pre-commit:** Antes de cada commit
- **Pre-push:** Antes de cada push
- **Post-commit:** Después de cada commit
- **Pre-receive:** En el servidor antes de recibir push

**Ubicación:** `.git/hooks/`



## Tipos de Hooks

---

### Hooks del lado cliente:

- `pre-commit`: Validar código antes del commit
- `commit-msg`: Validar mensaje de commit
- `pre-push`: Validar antes de hacer push
- `post-commit`: Acciones después del commit

### Hooks del lado servidor:

- `pre-receive`: Validar en servidor antes de recibir
- `post-receive`: Acciones después de recibir push

## Pre-commit Hook - Ejemplo

```
#!/bin/sh
# .git/hooks/pre-commit

# Ejecutar linter
npm run lint
if [ $? -ne 0 ]; then
    echo "❌ Linting failed"
    exit 1
fi

# Ejecutar tests
npm test
if [ $? -ne 0 ]; then
    echo "❌ Tests failed"
    exit 1
fi

echo "✅ Pre-commit checks passed"
```

## Commit-msg Hook - Ejemplo

```
#!/bin/sh
# .git/hooks/commit-msg

# Validar formato conventional commits
commit_regex='^(feat|fix|docs|style|refactor|test|chore)(\(.+\))?: .{1,50}'

if ! grep -qE "$commit_regex" "$1"; then
    echo "❌ Invalid commit message format"
    echo "Use: type(scope): description"
    echo "Example: feat(auth): add login validation"
    exit 1
fi

echo "✅ Commit message format is valid"
```

## Herramientas para Hooks

---

**Husky:** Simplifica la gestión de hooks

```
{
  "husky": {
    "hooks": {
      "pre-commit": "lint-staged",
      "pre-push": "npm test",
      "commit-msg": "commitlint -E HUSKY_GIT_PARAMS"
    }
  }
}
```

## lint-staged

---

Ejecuta linters solo en archivos staged (preparados para commit):

```
{
  "lint-staged": {
    "*.js": ["eslint --fix", "prettier --write"],
    "*.css": ["stylelint --fix"],
    "*.md": ["markdownlint --fix"]
  }
}
```

**Ventaja:** Solo procesa archivos modificados, no todo el proyecto.

## Instalación y Configuración

---

```
# Instalar Husky
npm install --save-dev husky

# Activar hooks de Git
npx husky install

# Crear hook pre-commit
npx husky add .husky/pre-commit "npm test"

# Hook commit-msg
npx husky add .husky/commit-msg "commitlint --edit $1"
```

## Hooks en el Servidor

---

Pre-receive hook para proteger ramas:

```
#!/bin/sh
# .git/hooks/pre-receive

while read oldrev newrev refname; do
    if [ "$refname" = "refs/heads/main" ]; then
        echo "❌ Push directo a main no permitido"
        echo "Usa Pull Request para cambios"
        exit 1
    fi
done

echo "✅ Push permitido"
```

## Pre-commit:

- Linting y formato de código
- Tests rápidos (unitarios)
- Validación de sintaxis

## Pre-push:

- Suite completa de tests
- Build del proyecto
- Análisis de seguridad

## Commit-msg:

- Formato de mensajes
- Referencias a issues

- Longitud de mensajes



## Saltarse Hooks (Emergencias)

---

```
# Saltar pre-commit (emergencias únicamente)
git commit --no-verify -m "hotfix: critical security patch"

# Saltar pre-push
git push --no-verify
```

**⚠ Usar solo en emergencias reales**



