

Guía de Semantic Versioning y Git Flow

Semantic Versioning (SemVer)

Formato: MAJOR.MINOR.PATCH (ejemplo: 1.2.3)

¿Cuándo incrementar cada número?

MAJOR (1.x.x → 2.x.x)

Cambios que ROMPEN compatibilidad hacia atrás (Breaking Changes)

Ejemplos:

- Cambiar la firma de un método público
- Eliminar un método o clase pública
- Cambiar el comportamiento de un método de forma incompatible
- Renombrar propiedades públicas

```
// v1.x.x
public function base100Attributes(): array

// v2.0.0 - BREAKING CHANGE
public function base100Attributes(): Collection // Cambió el return type
```

Impacto: Los usuarios DEBEN revisar su código antes de actualizar.

MINOR (x.1.x → x.2.x)

Nuevas características que SÍ son compatibles hacia atrás

Ejemplos:

- Añadir un nuevo método público
- Añadir un nuevo trait
- Añadir parámetros opcionales a métodos existentes
- Añadir nuevas opciones de configuración

```
// v1.1.0
class Base100 implements CastsAttributes
{
    // Métodos existentes...

    // NUEVO método añadido
    public function withPrecision(int $decimals): self
    {
        // ...
    }
}
```

Impacto: Los usuarios pueden actualizar sin cambios en su código.

PATCH (x.x.1 → x.x.2)

Correcciones de bugs que NO cambian funcionalidad

Ejemplos:

- Corregir un bug
- Mejorar rendimiento sin cambiar API
- Actualizar documentación
- Refactoring interno sin cambios en API pública
- Corregir tests

```
// v1.1.0 - Bug: redondeo incorrecto
return (int) $value * 100; // Error

// v1.1.1 - Patch: corregir redondeo
return (int) round($value * 100); // Correcto
```

Impacto: Actualización segura, solo mejoras y correcciones.

Estrategia de Branches

Branches Principales

1. `main` (Branch Principal)

- **Propósito:** Código estable y listo para producción
- **Protección:** Protected (no push directo)
- **Contiene:** Solo código que ha pasado tests y revisión
- **Tags:** Todas las releases se taguean desde aquí

2. `develop` (Branch de Desarrollo) - OPCIONAL

- **Propósito:** Integración de features antes de release
 - **Uso:** Si tienes múltiples features en paralelo
 - **Para este proyecto:** NO necesario (proyecto pequeño)
-

Branches de Trabajo

Feature Branches: `feature/*`

Para nuevas características

```
feature/add-precision-option  
feature/base1000-support  
feature/custom-rounding
```

Ejemplo de uso:

```
# Crear feature branch desde main  
git checkout main  
git pull origin main  
git checkout -b feature/add-precision-option  
  
# Trabajar en la feature...
```

```
git add .
git commit -m "feat: add precision option to Base100 cast"

# Cuando esté listo
git push origin feature/add-precision-option
# Crear Pull Request en GitHub
```

Bugfix Branches: `fix/*`

Para correcciones de bugs

```
fix/rounding-precision
fix/null-handling
fix/trait-initialization
```

Ejemplo de uso:

```
# Crear bugfix branch
git checkout main
git checkout -b fix/rounding-precision

# Corregir el bug
git add .
git commit -m "fix: correct rounding precision in Base100 cast"

# Push y PR
git push origin fix/rounding-precision
```

Hotfix Branches: `hotfix/*`

Para bugs CRÍTICOS en producción

```
hotfix/security-vulnerability
hotfix/data-corruption
```

Diferencia con `fix/*`:

- Hotfix: Bug crítico que necesita release inmediata
- Fix: Bug normal que puede esperar al próximo release

Release Branches: `release/*` - OPCIONAL

Para preparar una release

```
release/1.2.0  
release/2.0.0
```

Uso: Solo si necesitas "congelar" features antes de release.

Workflow de Desarrollo

Opción 1: GitHub Flow (RECOMENDADO para este proyecto)

```
main (protegido)  
↑  
└─ feature/nueva-caracteristica (trabajo aquí)  
└─ fix/bug-menor (trabajo aquí)
```

Ventajas:

- Simple y directo
- Ideal para proyectos pequeños-medianos
- Deployments/releases frecuentes

Flujo:

1. Crear branch desde `main`
2. Desarrollar la feature/fix
3. Abrir Pull Request
4. Code review + Tests automáticos (GitHub Actions)
5. Merge a `main`
6. Tag y release

Opción 2: Git Flow (Para proyectos grandes)

```
main (producción)
  ↑
develop (desarrollo)
  ↑
  ├── feature/feature-1
  ├── feature/feature-2
  └── release/1.2.0
```

Ventajas:

- Mejor para equipos grandes
- Releases planificadas
- Múltiples versiones en paralelo

Desventaja:

- Más complejo
- Overkill para proyectos pequeños

Proceso de Release

Release MINOR (nueva feature - 1.1.0 → 1.2.0)

```
# 1. Asegurarte que main está actualizado
git checkout main
git pull origin main

# 2. Actualizar CHANGELOG.md
## 1.2.0 - 2025-10-15
```

Added

- New `withPrecision()` method for custom decimal precision
- Support for negative values in HasBase100 trait

Fixed

- Rounding precision issue in edge cases

3. Commit el changelog

```
git add CHANGELOG.md
```

```
git commit -m "docs: update changelog for v1.2.0"
```

4. Crear tag

```
git tag -a v1.2.0 -m "Release v1.2.0 - Add precision support"
```

5. Push tag y código

```
git push origin main
```

```
git push origin v1.2.0
```

6. Crear GitHub Release

```
gh release create v1.2.0 \
```

```
--title "v1.2.0 - Precision Support" \
```

```
--notes "See CHANGELOG.md for details"
```

Release PATCH (bugfix - 1.1.0 → 1.1.1)

1. Checkout main

```
git checkout main
```

```
git pull origin main
```

2. Actualizar CHANGELOG.md

```
## 1.1.1 - 2025-10-12
```

Fixed

- Correct rounding precision when handling values > 1000

```
# 3. Commit y tag
git add CHANGELOG.md
git commit -m "docs: update changelog for v1.1.1"
git tag -a v1.1.1 -m "Release v1.1.1 - Fix rounding precision"
```

```
# 4. Push
git push origin main
git push origin v1.1.1
```

```
# 5. Release
gh release create v1.1.1 \
  --title "v1.1.1 - Bugfix Release" \
  --notes "Fix rounding precision issue"
```

Release MAJOR (breaking changes - 1.x.x → 2.0.0)

```
# 1. Crear rama para v2
git checkout main
git checkout -b release/2.0.0

# 2. Hacer cambios breaking
# ... código ...

# 3. Actualizar CHANGELOG.md con sección BREAKING CHANGES
```

Ejemplo de CHANGELOG para v2.0.0:

```
## 2.0.0 - 2025-11-01

### BREAKING CHANGES

- `base100Attributes()` now returns `Collection` instead of `array`
- Minimum PHP version raised to 8.4
- Removed deprecated `base100()` method
```

Migration Guide

```
**Before (v1.x):**
```php
protected function base100Attributes(): array
{
 return ['price', 'cost'];
}
```

#### **After (v2.0):**

```
protected function base100Attributes(): Collection
{
 return collect(['price', 'cost']);
}
```

## Added

- New `Base100Collection` class
- Support for custom transformers

Continuación del proceso:

```
```bash
# 4. Mergear a main
git checkout main
git merge release/2.0.0

# 5. Tag y release
git tag -a v2.0.0 -m "Release v2.0.0 - Major overhaul"
git push origin main
git push origin v2.0.0

# 6. GitHub Release con ADVERTENCIA
gh release create v2.0.0 \
```

```
--title "v2.0.0 - BREAKING CHANGES" \  
--notes "See CHANGELOG.md for migration guide"
```

Ejemplos Prácticos

Ejemplo 1: Añadir nueva feature (MINOR)

Escenario: Quieres añadir soporte para Base1000

```
# 1. Crear feature branch  
git checkout main  
git checkout -b feature/base1000-support  
  
# 2. Desarrollar la feature  
# - Crear src/Casts/Base1000.php  
# - Añadir tests  
# - Actualizar README  
  
# 3. Commits durante desarrollo  
git add .  
git commit -m "feat: add Base1000 cast class"  
git add .  
git commit -m "test: add Base1000 tests"  
git add .  
git commit -m "docs: document Base1000 usage"  
  
# 4. Push y crear PR  
git push origin feature/base1000-support  
gh pr create --title "Add Base1000 support" --body "Adds support for base-1000 conversions"  
  
# 5. Después de aprobación y merge  
git checkout main  
git pull origin main  
  
# 6. Release como 1.2.0 (MINOR - nueva feature)  
git tag -a v1.2.0 -m "Release v1.2.0 - Add Base1000 support"  
git push origin v1.2.0
```

```
gh release create v1.2.0
```

Ejemplo 2: Corregir bug (PATCH)

Escenario: Hay un bug en el redondeo

```
# 1. Crear fix branch
git checkout main
git checkout -b fix/rounding-issue

# 2. Corregir el bug
# - Editar src/Casts/Base100.php
# - Añadir test que reproduce el bug
# - Verificar que el test pasa

# 3. Commit
git add .
git commit -m "fix: correct rounding for values > 10000"

# 4. Push y PR
git push origin fix/rounding-issue
gh pr create --title "Fix rounding issue" --body "Fixes #42"

# 5. Después del merge
git checkout main
git pull origin main

# 6. Release como 1.1.1 (PATCH - bugfix)
git tag -a v1.1.1 -m "Release v1.1.1 - Fix rounding issue"
git push origin v1.1.1
gh release create v1.1.1
```

Ejemplo 3: Hotfix crítico (PATCH urgente)

Escenario: Descubriste un bug que causa pérdida de datos

```
# 1. Crear hotfix branch DESDE main
git checkout main
git checkout -b hotfix/data-loss-prevention

# 2. Corregir RÁPIDAMENTE
# - Solo el fix necesario, nada más
# - Test mínimo que demuestre el fix

# 3. Commit
git add .
git commit -m "fix: prevent data loss in null handling (critical)"

# 4. Merge DIRECTO a main (sin PR si es muy urgente)
git checkout main
git merge hotfix/data-loss-prevention

# 5. Release INMEDIATA
git tag -a v1.1.2 -m "Release v1.1.2 - Critical hotfix"
git push origin main
git push origin v1.1.2
gh release create v1.1.2 --title "v1.1.2 - Critical Hotfix"

# 6. Notificar usuarios en GitHub/Packagist
```

Comandos Git Útiles

Gestión de Branches

```
# Ver todas las branches
git branch -a

# Eliminar branch local
git branch -d feature/mi-feature

# Eliminar branch remoto
```

```
git push origin --delete feature/mi-feature
```

```
# Actualizar main desde remoto
```

```
git checkout main && git pull origin main
```

```
# Crear branch desde un commit específico
```

```
git checkout -b fix/bug abc1234
```

Gestión de Tags

```
# Listar todos los tags
```

```
git tag
```

```
# Ver detalles de un tag
```

```
git show v1.0.0
```

```
# Eliminar tag local
```

```
git tag -d v1.0.0
```

```
# Eliminar tag remoto
```

```
git push origin --delete v1.0.0
```

```
# Crear tag desde un commit antiguo
```

```
git tag -a v1.0.1 abc1234 -m "Release v1.0.1"
```

Revertir Cambios

```
# Revertir un commit (crea nuevo commit)
```

```
git revert abc1234
```

```
# Revertir último commit (antes de push)
```

```
git reset --soft HEAD~1
```

```
# Descartar cambios locales
```

```
git checkout -- archivo.php
```

Checklist de Release

Antes de Release

- Todos los tests pasan (`composer test`)
- PHPStan sin errores (`composer phpstan`)
- Código formateado (`composer format`)
- CHANGELOG.md actualizado
- README.md actualizado (si hay cambios en uso)
- Versión en composer.json coincide? (NO - Packagist lo maneja)
- Pull Request revisado y aprobado

Durante Release

- Main actualizado (`git pull origin main`)
- Tag creado con mensaje descriptivo
- Tag pusheado a GitHub
- GitHub Release creada con notas
- Packagist se actualizó automáticamente (webhook)

Después de Release

- Verificar que aparece en Packagist
- Badges del README actualizados
- Anunciar en redes/comunidad (si es relevante)
- Crear issues/milestones para próxima versión

Recomendaciones Específicas

Estrategia Recomendada

Para proyecto pequeño:

1. Usar GitHub Flow (simple)
2. Main siempre deployable
3. Feature branches para TODO
4. Pull Requests siempre (aunque seas solo tú - para CI)
5. Tags para cada release

Naming Conventions

```
# Features
feature/add-base1000
feature/custom-precision
feature/collection-support

# Fixes
fix/rounding-precision
fix/null-handling
fix/trait-initialization

# Hotfixes
hotfix/security-vulnerability
hotfix/data-corruption

# Docs
docs/update-readme
docs/add-examples
docs/api-documentation

# Chores
chore/update-dependencies
chore/ci-improvements
```

Commits Convencionales

```
feat: add new feature
fix: bug correction
docs: documentation only
```

```
style: formatting, no code change
refactor: code restructure
test: add/update tests
chore: maintenance tasks
perf: performance improvements
ci: CI/CD changes
```

Ejemplos:

```
git commit -m "feat: add withPrecision() method"
git commit -m "fix: correct rounding for negative values"
git commit -m "docs: add usage examples to README"
```

Recursos Adicionales

- **Semantic Versioning:** <https://semver.org/>
 - **Git Flow:** <https://nvie.com/posts/a-successful-git-branching-model/>
 - **GitHub Flow:** <https://guides.github.com/introduction/flow/>
 - **Conventional Commits:** <https://www.conventionalcommits.org/>
-

Preguntas Frecuentes

¿Cuándo hago MAJOR vs MINOR?

MAJOR (2.0.0): Si un usuario actualiza y su código se ROMPE → MAJOR

MINOR (1.1.0): Si un usuario actualiza y todo sigue funcionando → MINOR

¿Debo crear branch para cada pequeño cambio?

SÍ. Siempre trabaja en branches, incluso para cambios pequeños:

- Permite que GitHub Actions verifique antes de merge

- Historial más limpio
- Puedes descartar fácilmente si algo sale mal

¿Cuándo hacer release?

Flexible, pero algunas guías:

- PATCH: Cuando tengas 1+ bugfix importante
- MINOR: Cuando completes 1+ nueva feature
- MAJOR: Cuando hagas breaking changes (con cuidado)

Frecuencia recomendada:

- Patches: Cada 1-2 semanas
- Minor: Cada 1-2 meses
- Major: Solo cuando sea necesario

¿Puedo cambiar un tag después de crearlo?

NO recomendado una vez pusheado. Si lo haces:

- Los usuarios que ya instalaron la versión tendrán problemas
- Packagist se confunde
- Rompe la confianza

Si **DEBES** hacerlo:

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

# Crear nuevo tag
git tag -a v1.0.0 nuevo_commit -m "..."
git push origin v1.0.0
```

Última actualización: 2025-10-10

Autor: Abdelkarim Mateos Sanchez

Aviso

Esta documentación y su contenido, no implica que funcione en tu caso o determinados casos. También implica que tienes conocimientos sobre lo que trata, y que en cualquier caso tienes copias de seguridad. El contenido el contenido se entrega, tal y como está, sin que ello implique ningún obligación ni responsabilidad por parte de [Castris](#)

Si necesitas soporte profesional puedes contratar con Castris [soporte profesional](#).

Revision #13

Created 2025-10-10 16:55:53 UTC by Abkrim

Updated 2026-01-18 05:51:00 UTC by Abkrim