Cours : Les Conventional Commits pour Développeurs

📚 Table des matières

Introduction

Les **Conventional Commits** sont une convention de nommage pour vos messages de commit Git. C'est comme avoir une grammaire standardisée pour décrire les changements dans votre code. Cette convention rend votre historique Git plus lisible, plus professionnel et permet d'automatiser certaines tâches.

Imaginez votre historique Git comme un journal de bord : avec les Conventional Commits, chaque entrée suit un format précis qui raconte clairement ce qui s'est passé.

Pourquoi utiliser les Conventional Commits ?

🎯 Avantages principaux

Pour vous en tant que développeur :

- Historique Git plus clair et professionnel
- Plus facile de retrouver un changement spécifique
- Meilleure collaboration en équipe

Pour votre projet :

- Génération automatique de changelogs
- Gestion automatique des versions (semver)
- Outils d'intégration continue plus intelligents
- Facilite les reviews de code

Exemple concret : Au lieu de voir des commits comme :

```
- Fix bug
- Update code
- Change stuff
- Oops forgot something
```

Vous aurez :

```
- fix(auth): resolve login timeout issue
- feat(api): add user profile endpoint
- refactor(database): optimize query performance
- docs: update installation guide
```

Structure de base

Format général

```
<type>[étendue optionnelle]: <description>

[corps optionnel]

[pied optionnel]
```

Décomposition

1. **Type** (obligatoire) : le type de changement
2. **Étendue** (optionnelle) : quelle partie du code est affectée
3. **Description** (obligatoire) : description courte du changement
4. **Corps** (optionnel) : explication détaillée
5. **Pied** (optionnel) : informations sur les breaking changes, références, etc.

Exemple simple

```
feat(auth): add password reset functionality
```

- **Type :** feat (nouvelle fonctionnalité)
- **Étendue :** auth (module d'authentification)
- **Description :** add password reset functionality

Les types principaux

Types essentiels à connaître

Type	Description	Exemple
`feat`	Nouvelle fonctionnalité	`feat: add user search`
`fix`	Correction de bug	`fix: resolve crash on startup`
`docs`	Documentation	`docs: update README`
`style`	Formatage, point-virgules, etc.	`style: fix indentation`
`refactor`	Refactoring (ni bug ni feature)	`refactor: simplify auth logic`
`test`	Tests	`test: add login unit tests`
`chore`	Maintenance, outils, etc.	`chore: update dependencies`

Types additionnels utiles

Type	Description	Exemple
`perf`	Amélioration de performance	`perf: optimize image loading`
`ci`	Intégration continue	`ci: add automated testing`
`build`	Système de build	`build: update webpack config`
`revert`	Annulation d'un commit	`revert: remove buggy feature`

Exemples concrets

Exemples simples

bash

```bash
# Nouvelle fonctionnalité
feat: add dark mode toggle

# Correction de bug
fix: prevent memory leak in image gallery

# Documentation
docs: add API usage examples

# Refactoring
refactor: extract validation logic to utils
```

Exemples avec étendue

bash

```
# Préciser le module affecté
feat(api): add user authentication endpoint
fix(ui): resolve button alignment on mobile
docs(readme): add installation instructions
test(auth): add integration tests for login
```

Exemples avec corps et pied

bash

```
feat(search): implement full-text search

Add support for searching across all user content
including posts, comments, and profile information.
Uses Elasticsearch for improved performance.

Reviewed-by: team-lead@company.com
Refs: #123
```

Breaking Changes

bash

```
# Avec !
feat!: migrate to new API version

# Avec BREAKING CHANGE dans le pied
feat(api): update user endpoints

BREAKING CHANGE: user endpoint now requires authentication token
```

Règles détaillées

Règles obligatoires ✅

1. **Le type est OBLIGATOIRE** : `feat:`, `fix:`, etc.
2. **Les deux-points et l'espace sont OBLIGATOIRES** : `feat: ` (pas `feat:` ou `feat :`)
3. **La description est OBLIGATOIRE** et commence juste après l'espace
4. **Tout en minuscules** pour le type : `feat` pas `Feat` ou `FEAT`

Règles optionnelles 📝

1. **L'étendue entre parenthèses** : `feat(auth): add login`

2. **Le corps séparé par une ligne vide**
3. **Le pied séparé par une ligne vide**
4. **Le ! pour les breaking changes** : `feat!: breaking change`

Format des étendues

bash

```bash
# Bonnes étendues
feat(auth): ...          # module d'authentification
fix(ui): ...             # interface utilisateur
docs(api): ...           # documentation API
test(utils): ...         # tests des utilitaires

# Évitez les étendues trop vagues
feat(stuff): ...         # ❌ trop vague
fix(everything): ...     # ❌ pas spécifique
```

Bonnes pratiques

1. Soyez spécifique mais concis

bash

```bash
# ✅ Bon
fix(auth): resolve token expiration bug

# ❌ Trop vague
fix: bug fix

# ❌ Trop long
fix(auth): resolve the bug where authentication tokens were expiring too early causing
```

2. Utilisez l'impératif

bash

```bash
# ✅ Bon (impératif)
feat: add user profile page
fix: remove deprecated API calls

# ❌ Mauvais (passé)
feat: added user profile page
fix: removed deprecated API calls
```

3. Une seule responsabilité par commit

bash

```
# ✅ Bon - commits séparés
feat: add user search functionality
fix: resolve login validation bug

# ❌ Mauvais - mélange plusieurs choses
feat: add search and fix login bug
```

4. Utilisez les étendues de manière cohérente

bash

```
# Si votre projet a ces modules
feat(auth): ...
feat(ui): ...
feat(api): ...
feat(database): ...

# Restez cohérent dans toute l'équipe
```

5. Documentez les breaking changes

bash

```
feat!: migrate to TypeScript

BREAKING CHANGE: All JavaScript files have been converted to TypeScript.
Update your imports to use .ts extensions.
```

Erreurs courantes à éviter

❌ Erreurs de format

bash

```
# Mauvais format
Feat: new feature           # Type en majuscule
feat : new feature          # Espace avant les deux-points
feat:new feature            # Pas d'espace après les deux-points
feat(ui) : fix button       # Espace avant les deux-points
```

❌ Descriptions inappropriées

bash

```
# Trop vague
fix: bug
feat: stuff
chore: things

# Pas descriptif
fix: oops
feat: forgot this
chore: whatever
```

❌ Mauvais types

bash

```
# Types inexistants
update: change config       # utilisez plutôt 'chore' ou 'feat'
change: modify layout       # utilisez plutôt 'refactor' ou 'style'
bugfix: resolve issue       # utilisez 'fix'
```

❌ Commits trop gros

bash

```
# Un commit qui fait trop de choses
feat: add search, fix login, update docs, refactor utils

# Mieux : séparer en plusieurs commits
feat: add search functionality
fix: resolve login validation issue
docs: update API documentation
refactor: simplify utility functions
```

Exercices pratiques

Exercice 1 : Corriger ces messages de commit

Transformez ces mauvais commits en Conventional Commits :

```bash
# À corriger :
1. "Fixed the bug"
2. "New search feature"
3. "Updated README"
4. "Refactored some code"
5. "Login improvements"
```

Solutions :

```bash
1. fix: resolve authentication timeout issue
2. feat: add user search functionality
3. docs: update README with installation guide
4. refactor: simplify data validation logic
5. feat(auth): improve login error handling
```

Exercice 2 : Choisir le bon type

Pour chaque changement, choisissez le type approprié :

1. Ajout d'une nouvelle page de profil utilisateur
2. Correction d'un bug de calcul dans le panier
3. Mise à jour de la documentation de l'API
4. Optimisation des requêtes database
5. Ajout de tests unitaires pour le login
6. Changement de l'indentation du code
7. Mise à jour des dépendances npm

Solutions :

```bash
1. feat: add user profile page
2. fix: correct cart calculation bug
3. docs: update API documentation
4. perf: optimize database queries
5. test: add unit tests for login functionality
6. style: fix code indentation
7. chore: update npm dependencies
```

Exercice 3 : Commits avec étendue

Ajoutez des étendues appropriées à ces commits :

```bash
1. feat: add payment processing
2. fix: resolve button styling issue
3. test: add validation tests
4. docs: update deployment guide
```

Solutions possibles :

```bash
1. feat(payment): add payment processing
2. fix(ui): resolve button styling issue
3. test(validation): add validation tests
4. docs(deployment): update deployment guide
```

Conclusion

Les Conventional Commits transforment votre historique Git en un outil puissant et professionnel. En suivant ces conventions :

✅ **Vous améliorerez** la lisibilité de votre code
✅ **Vous faciliterez** la collaboration en équipe
✅ **Vous automatiserez** certaines tâches
✅ **Vous professionnaliserez** vos projets

Mémo rapide

```bash
# Format de base
<type>[étendue]: <description>

# Types principaux
feat:     nouvelle fonctionnalité
fix:      correction de bug
docs:     documentation
style:    formatage
refactor: refactoring
test:     tests
chore:    maintenance

# Exemples
feat(auth): add password reset
fix(ui): resolve mobile layout bug
docs: update installation guide
```

Ressources complémentaires

- Spécification officielle
- Commitizen - Outil pour vous aider à créer des commits
- Husky - Git hooks pour valider vos commits

Prêt à commencer ? Essayez dès votre prochain commit ! 🚀