Comprendre le Semantic Versioning

Le **Semantic Versioning**, ou versionnage sémantique en français, est une convention de numérotation utilisée pour indiquer les changements dans un projet logiciel. Cette méthode permet de mieux comprendre la compatibilité d'une version avec une autre, ainsi que la nature des modifications apportées. En tant que développeurs ou mainteneurs de projets, il est essentiel d'utiliser cette convention pour assurer la transparence et éviter les conflits entre différentes versions d'un logiciel.

5TTR 6TTR 6TQ Exploration

Qu'est-ce que le Semantic Versioning ?

Le **Semantic Versioning** est une spécification qui utilise une structure à trois nombres pour représenter l'évolution d'un projet. La structure de base est la suivante :

```
MAJEUR.MINOR.PATCH
```

- **MAJEUR** (ou `major`) : représente une version qui introduit des changements incompatibles avec les versions précédentes.
- **MINOR** (ou `minor`) : indique l'ajout de nouvelles fonctionnalités de manière rétrocompatible (c'est-à-dire que les versions précédentes continueront de fonctionner avec la nouvelle version).
- **PATCH** (ou `patch`) : représente une correction de bugs ou une petite amélioration qui ne modifie pas la compatibilité des versions.

Exemples de versionnage

Prenons un exemple simple pour illustrer cette notation :

- **1.0.0** : La première version stable du projet.
- **1.1.0** : Ajout d'une nouvelle fonctionnalité sans casser la compatibilité avec la version 1.0.0.
- **1.1.1** : Correction de bugs sur la version 1.1.0.
- **2.0.0** : Changements majeurs qui rendent la version incompatible avec la version 1.x.x.

Dans cet exemple, la première version stable est 1.0.0, et chaque incrément de numéro signifie quelque chose de spécifique à l'utilisateur du logiciel.

Pourquoi utiliser le Semantic Versioning ?

© 2025 Mr. Lambrechts

Le principal avantage du versionnage sémantique est de clarifier les modifications dans un projet. Grâce à cette convention, les utilisateurs et développeurs savent immédiatement si une mise à jour va casser leur implémentation ou si elle est compatible avec la version actuelle.

Par exemple, si une librairie passe de la version 1.2.3 à la version 2.0.0, il est clair que des changements majeurs ont été apportés, ce qui pourrait nécessiter une adaptation de l'application qui utilise cette librairie.

Scénario pratique : utilisation d'une bibliothèque

Imaginons que vous soyez en train de développer une application en Python et que vous utilisiez une bibliothèque externe pour gérer une base de données. Vous intégrez la version **1.5.0** de cette bibliothèque dans votre projet.

Quelques semaines plus tard, une nouvelle version **2.0.0** est publiée. En tant que développeur, vous devez maintenant décider si vous souhaitez mettre à jour vers cette version. Comme la version **MAJEUR** a changé, vous savez que cela peut signifier des changements incompatibles. Vous devrez probablement modifier une partie de votre code pour qu'il fonctionne avec cette nouvelle version.

Si au contraire la version était passée de **1.5.0** à **1.6.0**, vous seriez en mesure de mettre à jour sans crainte de casser votre application, car cette nouvelle version est rétrocompatible.

Les règles du Semantic Versioning

Les règles du versionnage sémantique sont clairement définies :

1. **Version initiale (0.y.z)** : Les versions 0.x.x sont considérées comme des versions de développement initial. Tout peut changer à tout moment, sans respecter la compatibilité. C'est une période de test avant la version stable.

2. **Changements MAJEURS** : Une fois que la version 1.0.0 est atteinte, tout changement qui casse la compatibilité avec l'API existante doit entraîner un changement du numéro de version MAJEUR. C'est-à-dire que si vous avez une version **1.2.3** et que vous apportez des modifications incompatibles avec la version précédente, vous passerez à **2.0.0**.

3. **Changements MINEURS** : Si vous ajoutez des fonctionnalités sans casser l'API existante, vous incrémentez le numéro de version MINOR. Par exemple, une version **1.2.0** peut introduire de nouvelles fonctionnalités tout en restant compatible avec les versions **1.1.x**.

4. **Corrections de bugs (PATCH)** : Les corrections de bugs, les petites améliorations ou les mises à jour de sécurité qui ne modifient pas l'API sont signalées par une augmentation du numéro PATCH. Passer de **1.2.3** à **1.2.4** ne devrait pas avoir d'impact sur le fonctionnement global de votre projet.

Exemples concrets de versionnage

Exemple 1 : Correctif de sécurité

- **Avant :** 2.3.4
- **Après :** 2.3.5

© 2025 Mr. Lambrechts

Une petite mise à jour de sécurité a été appliquée. Cette correction ne change pas l'API publique, elle se limite à un correctif interne.

Exemple 2 : Ajout d'une fonctionnalité

- **Avant :** 2.3.4
- **Après :** 2.4.0

Une nouvelle fonctionnalité est ajoutée à l'API publique, mais elle est rétrocompatible. Cela justifie une augmentation du numéro MINOR.

Exemple 3 : Changement majeur

- **Avant :** 2.4.0
- **Après :** 3.0.0

Un changement majeur est apporté à l'API, qui n'est plus compatible avec les versions précédentes. Cela nécessite une augmentation du numéro MAJEUR.

Bonnes pratiques du versionnage sémantique

Pour bien utiliser le **Semantic Versioning**, voici quelques recommandations :

1. **Documentez vos versions** : Il est essentiel de tenir un journal de changements (**changelog**) pour informer les utilisateurs des modifications apportées à chaque version. Cela permet à d'autres développeurs de comprendre rapidement les implications d'une mise à jour.

2. **Respectez la compatibilité** : Si vous incrémentez un numéro MINOR ou PATCH, assurez-vous que votre code reste compatible avec les versions précédentes. Si vous cassez la compatibilité, augmentez le numéro MAJEUR.

3. **Utilisez des outils d'automatisation** : Il existe des outils comme Semantic Release qui peuvent automatiser la gestion des versions sémantiques. Ils génèrent automatiquement les numéros de version, les changelogs et les publications basées sur le type de commit dans un projet.

Informations de pré-release : les tags alpha, beta et rc

Dans le cadre du **versionnage sémantique** (GB ***semantic versioning***), tu peux aussi ajouter des tags de pré-release comme **alpha**, **beta** et **rc** pour signaler que certaines versions du logiciel sont encore en phase de développement et ne sont pas prêtes pour une utilisation en production. Dans le format **MAJEUR.MINOR.PATCH** du semver, ces tags sont ajoutés après le numéro de version, **précédés d'un tiret**, pour indiquer que la version est instable ou en phase de test.

Par exemple, une version `1.2.0-alpha` signifie que cette version est une étape préliminaire de la version 1.2.0. Ces tags permettent aux utilisateurs et aux développeurs de comprendre que, bien que certaines fonctionnalités soient présentes, le logiciel n'est pas encore complètement fiable. Ils offrent un cadre clair pour distinguer les versions stables des versions en cours d'élaboration, facilitant la gestion des mises à jour et des déploiements.

- **Alpha** : Une version alpha est une version préliminaire du logiciel qui est encore en phase de développement initial. Elle peut contenir des fonctionnalités non finalisées, des bugs importants et n'est généralement pas destinée à être utilisée par un large public. Elle sert principalement aux développeurs pour tester les nouvelles fonctionnalités.

  Exemple : `Projet_v1.0_alpha`

- **Beta** : Une version beta est plus avancée qu'une alpha. Elle inclut généralement toutes les fonctionnalités prévues pour la version finale, mais peut encore contenir des bugs. Les versions beta sont souvent diffusées à un groupe restreint d'utilisateurs pour effectuer des tests et recueillir des retours.

  Exemple : `Projet_v1.0_beta`

- **RC (Release Candidate)** : Une release candidate est une version qui est potentiellement prête pour la sortie finale, à moins que des bugs critiques ne soient découverts. C'est la dernière étape avant la version officielle du logiciel.

  Exemple : `Projet_v1.0_rc`

Enfin, ces tags peuvent être **suivis d'un numéro**, qui indique le numéro de la release. En effet, en fonction du nombre de corrections ou changements nécessaires suite aux tests d'une release, il peut être nécessaire d'effectuer une nouvelle release. Ex: `v1.4-alpha`, `v1.4-alpha2`, `v1.4-beta`, `v1.4-rc`, `v1.4-rc2`, `v1.4-rc3`, `v1.4`

L'utilisation de ces tags dans vos noms de versions permet de communiquer clairement l'état d'avancement de votre projet. Cela est particulièrement utile si vous partagez votre code avec d'autres personnes ou si vous publiez des versions de test pour recueillir des commentaires. Même en travaillant seul, ces tags vous aident à organiser votre travail et à savoir quelle version du code est stable ou en cours de développement.

Conclusion

Le **Semantic Versioning** est un outil puissant pour gérer les versions de vos projets logiciels. Il offre une convention claire qui permet aux utilisateurs de savoir quand une mise à jour est sécuritaire, quand elle pourrait nécessiter des adaptations, ou quand elle pourrait introduire des fonctionnalités intéressantes. En suivant cette convention, vous facilitez la maintenance, l'évolution et la compatibilité de vos projets.

Qu'il s'agisse d'un petit correctif ou d'une refonte majeure, le versionnage sémantique apporte de la clarté et de la prévisibilité dans la gestion des versions logicielles.

Ressources complémentaires :

- Spécification officielle du Semantic Versioning
- Outil pour générer des changelogs automatiquement

© 2025 Mr. Lambrechts