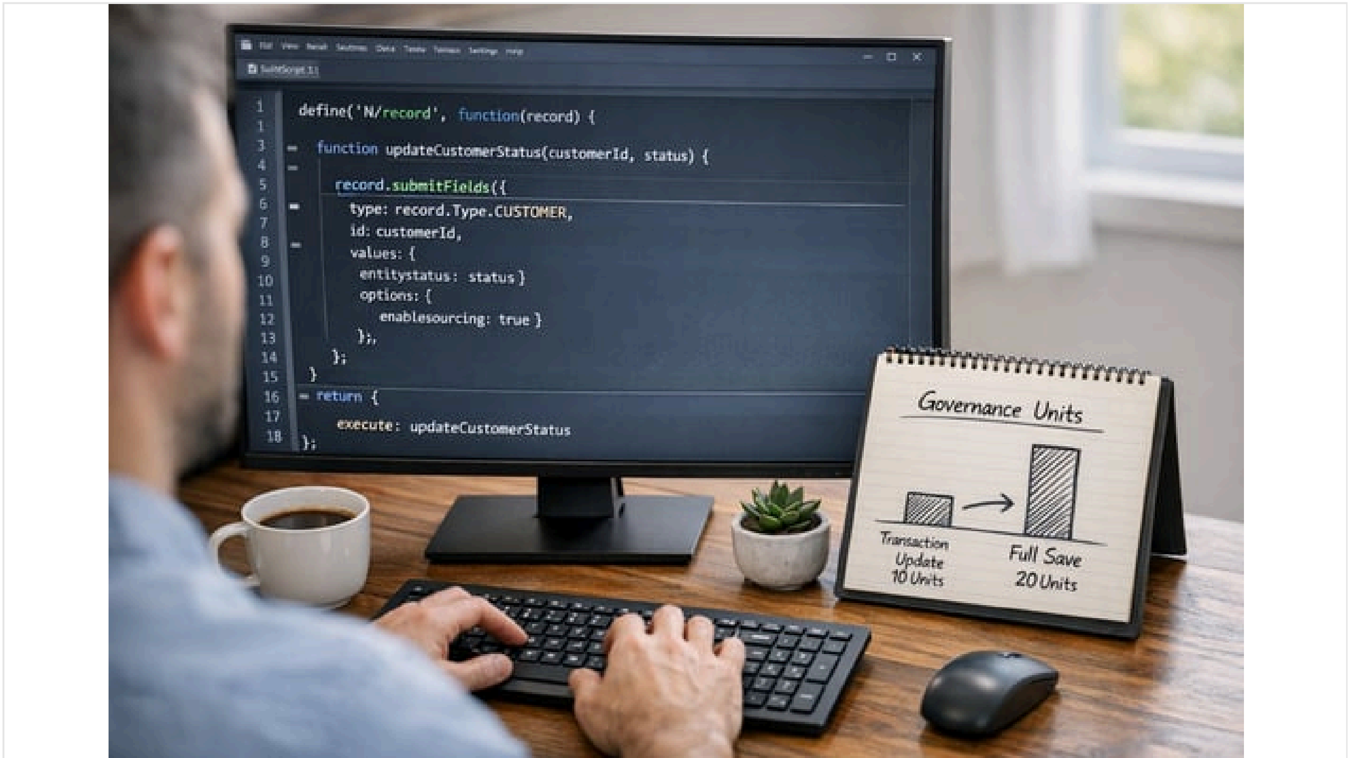
Gouvernance et limites de record.submitFields dans SuiteScript 2.1

By houseblend.io   Publié le 19 avril 2026   38 min de lecture



Résumé analytique

L'API `record.submitFields` de SuiteScript 2.1 permet de mettre à jour un ou plusieurs champs au niveau de l'en-tête d'un enregistrement NetSuite existant sans avoir à charger ou soumettre l'enregistrement complet. Cette approche est nettement plus **efficace en termes de gouvernance** et plus rapide pour les mises à jour simples que le chargement et l'enregistrement complets des enregistrements. Selon la documentation officielle de NetSuite, l'utilisation de `submitFields` consomme un nombre fixe d'unités de gouvernance selon le type d'enregistrement : **10 unités pour les enregistrements de transaction standard, 2 unités pour les enregistrements personnalisés et 5 unités pour les autres enregistrements (non transactionnels)** (Source: docs.oracle.com). En revanche, un `record.save()` complet consomme **20 unités (transaction), 4 unités (personnalisé) ou 10 unités (autre)** (Source: docs.oracle.com). Par conséquent, les scripts utilisent environ la moitié des unités, voire moins, pour chaque mise à jour lorsque les champs prennent en charge la modification en ligne.

Les unités de gouvernance sont la monnaie que NetSuite impose à l'exécution des scripts pour limiter l'utilisation des ressources. Chaque type de script (User Event, Scheduled, Suitelet, etc.) dispose d'une allocation maximale (par exemple, 1 000 unités pour les User Events (Source: docs.oracle.com) et 10 000 pour les scripts planifiés (Source: docs.oracle.com). En raison de ces limites, l'utilisation de méthodes légères comme `submitFields` peut doubler le nombre d'enregistrements mis à jour avant d'atteindre les plafonds. Par exemple, un script planifié peut mettre à jour jusqu'à ~1 000 enregistrements de transaction via `submitFields` (10 unités chacun) contre seulement 500 via `save()` (20 unités chacun), ce qui rend `submitFields` idéal pour les mises à jour en masse.

Ce rapport fournit une analyse approfondie de `record.submitFields` dans SuiteScript 2.1, couvrant ses **coûts de gouvernance, limites intégrées, modèles d'utilisation et meilleures pratiques**. Nous nous appuyons sur la documentation SuiteScript et les tables de gouvernance d'Oracle (Source: docs.oracle.com) (Source: docs.oracle.com), les conseils officiels de SuiteAnswers et les blogs communautaires (Source: community.oracle.com) (Source: community.oracle.com), ainsi que les discussions de développeurs (Source: archive.netsuiteprofessionals.com) (Source: stackoverflow.com). Des exemples spécifiques (incluant des extraits de code et des scénarios conceptuels) illustrent comment

`submitFields` est utilisé dans des scripts réels. Nous le comparons également à des méthodes alternatives (`record.save`, `record.transform`, etc.), expliquons quand il est le plus efficace et notons ses limites (par exemple, il ne peut pas mettre à jour les champs de sous-liste ou de sous-enregistrement (Source: docs.oracle.com).

Les conclusions clés incluent : **(1)** `submitFields` surpasse généralement `load` + `save` lors de la modification des seuls champs d'en-tête, car il contourne le chargement de l'enregistrement et les vérifications de métadonnées (Source: community.oracle.com) (Source: hutada.home.blog). Cependant, si un champ ne prend pas en charge la modification en ligne, NetSuite peut revenir en interne à un chargement/enregistrement complet, annulant ainsi l'avantage (Source: archive.netsuiteprofessionals.com). **(2)** La méthode est prise en charge dans les scripts SuiteScript 2.x côté client et côté serveur (incluant une variante basée sur les promesses pour une utilisation côté client) (Source: docs.oracle.com) (Source: docs.oracle.com), et elle renvoie l'ID interne de l'enregistrement mis à jour. **(3)** Comme `submitFields` ne peut pas gérer les lignes de sous-liste ou les sous-enregistrements (Source: docs.oracle.com), il est préférable de l'utiliser pour des mises à jour simples (telles que la modification d'une valeur de champ personnalisé ou d'un statut sur une transaction). Pour les opérations en masse impliquant de nombreux enregistrements, combiner `submitFields` avec des scripts Map/Reduce ou planifiés peut maximiser le débit sous les plafonds de gouvernance.

Nous concluons avec des recommandations pratiques : utilisez `submitFields` autant que possible pour les changements de champs simples, ajustez vos scripts pour vérifier l'utilisation restante (en utilisant `Script.getRemainingUsage()`), et surveillez les exceptions de gouvernance dans les contextes à haut volume. Les développements futurs pourraient inclure des API asynchrones étendues ou des optimisations supplémentaires dans le modèle de gouvernance de NetSuite pour prendre en charge des personnalisations à encore plus grande échelle.

Introduction et contexte

NetSuite SuiteScript 2.1 est un framework de script moderne basé sur JavaScript qui s'exécute au sein de la plateforme cloud ERP de NetSuite. Introduite après SuiteScript 2.0, la version 2.1 utilise un moteur d'exécution mis à jour et prend en charge les fonctionnalités JavaScript modernes (telles que les fonctions fléchées, `let` / `const`, async/await) qui n'étaient pas disponibles en 2.0 (Source: docs.oracle.com). Elle conserve l'architecture modulaire AMD (`define()/require()`) de la version 2.x mais offre un « alignement plus profond avec les outils SuiteCloud » (Source: docs.oracle.com). SuiteScript 2.1 est généralement rétrocompatible avec le code 2.0 – la plupart des scripts 2.0 s'exécutent sans changement sous 2.1 – mais les développeurs bénéficient de fonctionnalités linguistiques plus riches. Les API des modules principaux (telles que **`N/record`** pour manipuler les enregistrements, **`N/search`** pour les requêtes, etc.) sont identiques en 2.0 et 2.1, et le modèle de gouvernance (unités d'utilisation) reste cohérent dans les deux versions.

Au sein du module **`N/record`**, la méthode **`record.submitFields(options)`** (introduite dans la version 2015.2 de NetSuite) offre un moyen de mettre à jour des enregistrements existants sans les charger complètement. En revanche, le modèle classique consistait à appeler `record.load()`, à modifier l'objet d'enregistrement, puis à appeler `record.save()` (Source: community.oracle.com). Bien que l'approche chargement/enregistrement soit simple et flexible, elle entraîne un coût de gouvernance plus élevé et des performances potentiellement plus lentes. La méthode `submitFields` a été ajoutée comme alternative simplifiée pour les mises à jour de type « modification en ligne ». Elle permet aux développeurs de fournir un type d'enregistrement, un ID interne et un objet de paires champ-valeur à mettre à jour. En interne, NetSuite applique les modifications de champ et enregistre l'enregistrement en une seule étape, mais surtout, elle n'expose ni ne nécessite les données complètes de l'enregistrement dans la mémoire du script (Source: docs.oracle.com).

Unités de gouvernance : NetSuite suit la consommation des ressources système d'un script via des **unités de gouvernance**. Chaque appel d'API ou opération SuiteScript consomme un certain nombre d'unités, reflétant la dépense computationnelle pour les serveurs de NetSuite. Par exemple, le chargement d'un grand enregistrement de transaction peut coûter plus d'unités que le chargement d'un petit enregistrement personnalisé. SuiteScript inclut des méthodes comme `scriptContext.getRemainingUsage()` pour permettre aux scripts de surveiller leur budget restant. Chaque exécution de script (selon son type de déploiement) a une limite d'utilisation fixe. Par exemple, un script User Event ou Client est limité à 1 000 unités par exécution (Source: docs.oracle.com) (Source: docs.oracle.com), un Suitelet également 1 000 unités (Source: docs.oracle.com), tandis qu'un script planifié ou Map/Reduce a un plafond beaucoup plus élevé (Planifié : 10 000 unités (Source: docs.oracle.com) ; Map/Reduce : pratiquement illimité par exécution, bien que chaque étape soit régulée (Source: docs.oracle.com). Lorsqu'un script atteint sa limite, NetSuite génère une erreur `SSS_USAGE_LIMIT_EXCEEDED`, arrêtant l'exécution.

Compte tenu de ces contraintes, minimiser l'utilisation par appel est essentiel pour des scripts robustes et évolutifs. L'émergence de `submitFields` reflète ce principe : si seul un ou deux champs d'un enregistrement doivent être mis à jour, il est inutile de consommer le double des ressources (chargement + enregistrement) alors qu'une mise à jour ciblée suffit. En effet, les formateurs NetSuite et les experts de la communauté soulignent que **« l'efficacité est la clé, surtout lors de la mise à jour des enregistrements »** (Source: community.oracle.com). La méthode `submitFields` est une pierre angulaire de plusieurs stratégies d'optimisation et meilleures pratiques pour SuiteScript, comme nous le détaillerons.

Ce rapport explorera *SuiteScript 2.1* et sa méthode `record.submitFields` de manière exhaustive. Nous commencerons par expliquer comment la méthode fonctionne et ce qu'elle coûte en unités de gouvernance, puis nous la comparerons à des approches alternatives. Nous examinerons la documentation officielle sur la gouvernance SuiteScript (coûts d'utilisation et limites) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com), et intégrerons les idées de la communauté NetSuite et des blogs techniques. Nous analyserons les modèles d'utilisation (par exemple, quels champs peuvent être modifiés, contextes de script où cela est autorisé) et discuterons des limites. Des exemples réels et des extraits de code illustrent quand et comment les développeurs utilisent `submitFields` en pratique. Enfin, nous discuterons des implications pour la conception de scripts, le réglage des performances et l'avenir de l'automatisation SuiteScript.

SuiteScript 2.1 et le module N/record

SuiteScript 2.x a introduit une API modulaire où la fonctionnalité est divisée entre des modules nommés (tels que `N/record`, `N/search`, `N/runtime`, etc.). Le **module `N/record`** contient des méthodes pour créer, charger, transformer, copier, supprimer et soumettre des enregistrements. Par exemple, `record.load()` charge complètement un enregistrement en mémoire, et `record.save()` valide toutes les modifications. Dans SuiteScript 2.1, ces modules restent largement identiques à la version 2.0, avec des améliorations de syntaxe. Comme noté par NetSuite et des sources communautaires, **SuiteScript 2.1 « utilise un moteur d'exécution différent de la version 2.0 et prend en charge des fonctionnalités ECMAScript qui ne sont pas prises en charge en 2.0 »** (Source: docs.oracle.com). L'effet pratique principal de la version 2.1 est la syntaxe JavaScript moderne ; les comportements de l'API principale (comme les coûts de gouvernance) sont hérités de la version 2.0.

Au sein du module `N/record`, la méthode **`submitFields(options)`** est documentée comme suit : elle « met à jour et soumet un ou plusieurs champs d'en-tête sur un enregistrement existant dans NetSuite » et *« renvoie l'ID interne de l'enregistrement parent »* (Source: docs.oracle.com). Crucialement, « lorsque vous utilisez cette méthode, vous n'avez pas besoin de charger ou de soumettre l'enregistrement parent » (Source: docs.oracle.com). En pratique, un script fournit :

- `type` : le type d'enregistrement (par exemple, `record.Type.SALES_ORDER` ou un type d'enregistrement personnalisé),
- `id` : l'ID interne de l'enregistrement à mettre à jour,
- `values` : un objet mappant les ID de champ aux nouvelles valeurs,
- (optionnel) `options` : un objet avec des indicateurs comme `enableSourcing` et `ignoreMandatoryFields`.

La méthode applique directement les modifications de champ données. Ses *« Types de scripts pris en charge »* sont **à la fois les scripts client et serveur** (Source: docs.oracle.com), ce qui signifie qu'elle peut être appelée depuis des User Events, Suitelets, scripts planifiés, Map/Reduce (dans les étapes Map ou Reduce), RESTlets, Portlets, scripts client, etc. (les scripts client ont même une version basée sur les promesses `record.submitFields.promise()` (Source: docs.oracle.com). La référence de l'API note également que seuls certains champs peuvent être modifiés : spécifiquement **« les champs d'en-tête standard qui prennent en charge la modification en ligne »**, **« les champs d'en-tête personnalisés qui prennent en charge la modification en ligne »**, et les champs de sélection/sélection multiple (Source: docs.oracle.com). Elle ne peut explicitement pas être utilisée pour les éléments de ligne de sous-liste ou les champs de sous-enregistrement (par exemple, les sous-enregistrements d'adresse) (Source: docs.oracle.com). En d'autres termes, `submitFields` fonctionne comme la modification de la colonne de l'enregistrement dans une vue de liste (modification en ligne) – elle ignore les champs au niveau de la ligne ou imbriqués. Cette limitation encourage l'utilisation de `submitFields` uniquement pour des modifications simples et de haut niveau (telles que le changement du statut d'un client ou du total d'une commande client), tandis que des modifications d'enregistrement plus complexes nécessitent toujours `record.load`/`record.save` ou une transformation.

Coûts de gouvernance de `submitFields`

Un élément clé de l'évaluation de `submitFields` est la compréhension de son coût en unités de gouvernance. L'aide officielle sur la gouvernance SuiteScript 2.x répertorie chaque appel d'API et sa charge en unités. Pour `record.submitFields(options)`, les coûts sont :

- **10 unités** pour un *enregistrement de transaction* (par exemple, Facture, Commande client, Bon de commande)
- **2 unités** pour un *enregistrement personnalisé*
- **5 unités** pour *tous les autres enregistrements* (enregistrements standard non transactionnels comme Client, Article, etc.)

(Source: docs.oracle.com). Ces valeurs s'appliquent aux formes synchrones et basées sur les promesses. Dans le contexte, la plupart des appels `record.load` ou `record.transform` ont des coûts identiques (10/2/5) (Source: docs.oracle.com). En revanche, `record.save(options)` est nettement plus élevé : il consomme **20 unités pour un enregistrement de transaction, 4 pour un personnalisé et 10 pour les autres** (Source: docs.oracle.com). Le tableau 1 de l'annexe (ci-dessous) résume ces utilisations pour les méthodes d'enregistrement courantes :

MÉTHODE N/RECORD	ENREGISTREMENT DE TRANSACTION	ENREGISTREMENT PERSONNALISÉ	AUTRE ENREGISTREMENT	(SUITESCRIPT 2.X)
`record.load(options)`	10 unités	2 unités	5 unités	
`record.submitFields(options)`	10 unités	2 unités	5 unités	
`record.transform(options)`	10 unités	2 unités	5 unités	
`record.create(options)`	10 unités	2 unités	5 unités	
`record.copy(options)`	10 unités	2 unités	5 unités	

| **`record.save(options)`** | **20 unités** | **4 unités** | **10 unités** | | **`record.delete(options)`** | **20 unités** | **4 unités** | **10 unités** |

Chaque coût unitaire provient de la documentation de l'API SuiteScript 2.x d'Oracle (Source: docs.oracle.com) (Source: docs.oracle.com). Le tableau souligne que `submitFields`, comme d'autres appels d'enregistrement légers, utilise deux fois moins d'unités que `save()` ou `delete()`. Cette différence est cruciale lorsque les scripts approchent de leurs limites d'utilisation. Par exemple, dans un script User Event (limite maximale de 1 000 unités (Source: docs.oracle.com), effectuer plusieurs appels `save()` peut rapidement épuiser le budget. Passer à `submitFields` pour des mises à jour mineures peut augmenter considérablement la marge de manœuvre restante.

Annexe Tableau 1 : *Utilisation de la gouvernance des méthodes N/record de SuiteScript 2.x (unités)*. Les coûts varient selon la catégorie d'enregistrement (transaction, personnalisé, etc.) (Source: docs.oracle.com) (Source: docs.oracle.com).

Tableau 1 : Utilisation de la gouvernance des principales méthodes N/record. Source : Documentation Oracle SuiteScript 2.x.

MÉTHODE	ENREGISTREMENTS DE TRANSACTION	ENREGISTREMENTS PERSONNALISÉS	AUTRES ENREGISTREMENTS
`record.load(options)`	10 unités	2 unités	5 unités
`record.submitFields(options)`	10 unités	2 unités	5 unités
`record.transform(options)`	10 unités	2 unités	5 unités
`record.create(options)`	10 unités	2 unités	5 unités
`record.copy(options)`	10 unités	2 unités	5 unités
`record.save(options)`	**20 unités**	**4 unités**	**10 unités**
`record.delete(options)`	**20 unités**	**4 unités**	**10 unités**

Le tableau montre que, pour les enregistrements de transaction en particulier, `submitFields` utilise 10 unités contre 20 pour `save()`. Un blog sur l'optimisation de SuiteScript note que le chargement d'un enregistrement « coûte entre 2 et 10 unités » et que la sauvegarde « coûte entre 4 et 20 » (Source: hutada.home.blog), ce qui est cohérent avec les données officielles. Il conseille aux développeurs d'éviter le chargement d'enregistrements lorsque cela est possible et d'utiliser plutôt des recherches ou des appels légers. Plus précisément, il indique : « *Si vous devez modifier des valeurs sur un enregistrement, utilisez `record.submitFields`. Cela utilise entre 2 et 10 unités de gouvernance et est plus rapide que de charger et sauvegarder un enregistrement.* » (Source: hutada.home.blog). En pratique, l'unité exacte « entre 2 et 10 » provient de la répartition ci-dessus : 2 unités pour un petit enregistrement personnalisé, jusqu'à 10 pour une transaction importante. Ainsi, dans la plupart des scénarios, `submitFields` est presque toujours un choix plus efficace lorsque seuls quelques champs doivent être mis à jour.

Limites d'utilisation des scripts

Au-delà des coûts par appel, chaque exécution de script a une **limite d'utilisation maximale** définie par son type. La documentation officielle de NetSuite fournit une répartition :

- **Scripts User Event, Suitelet, Client, Portlet :** 1 000 unités par exécution (Source: docs.oracle.com) (Source: docs.oracle.com).
- **Scripts Scheduled :** 10 000 unités par exécution (Source: docs.oracle.com).
- **Scripts Map/Reduce :** Pas de limite totale fixe ; chaque étape (Map, Reduce) dispose de budgets distincts. (Cependant, les appels de méthode individuels au sein de chaque invocation consomment des unités comme d'habitude (Source: docs.oracle.com).)
- **RESTlets :** 5 000 unités par exécution (Source: docs.oracle.com).

Le tableau 2 ci-dessous résume les principaux types de scripts et leurs plafonds d'utilisation :

TYPE DE SCRIPT	UNITÉS D'UTILISATION MAX PAR EXÉCUTION	SOURCE
User Event (2.x)	1 000 unités	Docs Oracle SuiteScript (Source: docs.oracle.com)
Client (2.x)	1 000 unités (par script)	Docs Oracle SuiteScript (Source: docs.oracle.com)
Suitelet (2.x)	1 000 unités	Docs Oracle SuiteScript (Source: docs.oracle.com)
Portlet (2.x)	1 000 unités	Docs Oracle SuiteScript (Source: docs.oracle.com)
Scheduled (2.x)	10 000 unités	Docs Oracle SuiteScript (Source: docs.oracle.com)
Map/Reduce (2.x)	Aucune limite totale (rendement)	Docs Oracle SuiteScript (Source: docs.oracle.com)
RESTlet (2.x)	5 000 unités	Docs Oracle SuiteScript (Source: docs.oracle.com)

Tableau 2 : Types de scripts SuiteScript 2.x et leurs limites de gouvernance (unités d'utilisation), selon la documentation Oracle.

Ce tableau est compilé à partir de la documentation **SuiteScript 2.x Script Type Usage Unit Limits** (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com). Ces limites permettent de contextualiser combien d'appels `submitFields` peuvent tenir dans un script avant d'atteindre le plafond. Par exemple, un script planifié peut effectuer jusqu'à 1 000 appels `submitFields` sur des enregistrements de transaction (10 unités chacun) tout en restant dans la limite des 10 000 unités. En revanche, un script User Event est beaucoup plus contraint (plafond de 1 000 unités), l'efficacité y est donc encore plus critique.

En résumé de cette section, `record.submitFields` est une méthode de **mise à jour d'enregistrement légère** dans SuiteScript 2.x qui est très précieuse pour l'optimisation des performances. Elle évite la surcharge liée au chargement/sauvegarde complet d'un enregistrement et ne consomme généralement que la moitié des unités d'utilisation d'une opération de sauvegarde. Ses coûts de gouvernance sont bien documentés, et les concepteurs de scripts devraient en tirer parti pour intégrer davantage de mises à jour par exécution. Le reste de ce rapport examine les modèles pratiques, les techniques de codage et les cas réels impliquant `submitFields`, ainsi que ses limites et implications pour le développement futur.

record.submitFields : Détails techniques et comportement

Fonctionnalité principale

La méthode `record.submitFields(options)` utilise un seul appel JavaScript pour modifier et sauvegarder des champs spécifiés. La description officielle indique : *« Met à jour et soumet un ou plusieurs champs de corps sur un enregistrement existant dans NetSuite, et renvoie l'ID interne de l'enregistrement parent. »* (Source: docs.oracle.com). Comme l'enregistrement n'a pas besoin d'être chargé dans la mémoire du script, les performances sont améliorées et moins d'unités de gouvernance sont consommées. En pratique, une utilisation typique ressemble à ceci :

```
record.submitFields({
    type: record.Type.SALES_ORDER,
    id: 12345,
    values: {
        status: 'Closed',
        memo: 'Processed'
    },
    options: {
        enableSourcing: false,
        ignoreMandatoryFields: true
    }
});
```

Cet exemple définirait le `status` et le `memo` sur la commande client 12345. Dans de nombreux cas, seuls `type`, `id` et `values` sont nécessaires ; `options` est facultatif. Les développeurs peuvent transmettre des indicateurs pour contrôler le comportement – par exemple, `enableSourcing` peut actualiser les champs basés sur des sources, et `ignoreMandatoryFields: true` peut contourner les vérifications sur les champs obligatoires (Source: stackoverflow.com) (Source: stackoverflow.com).

Parce que `submitFields` ne touche qu'aux champs spécifiés, tous les autres champs de l'enregistrement restent inchangés. Elle effectue *bien* en interne une sauvegarde de l'enregistrement une fois les nouvelles valeurs définies, mais cela est opaque pour le script. L'appel de méthode renvoie l'ID interne (qui est généralement le même que l'entrée `id`) de l'enregistrement qui a été mis à jour (Source: docs.oracle.com). Il est important de noter que la documentation souligne : *« Lorsque vous utilisez cette méthode, vous n'avez pas besoin de charger ou de soumettre l'enregistrement parent. »* (Source: docs.oracle.com). Cela implique que tout *script User Event* lié à l'enregistrement peut ou non se déclencher ; en pratique, `submitFields` ne déclenche généralement **pas** les scripts `before` / `afterSubmit` sur cet enregistrement. (La documentation de NetSuite ne le dit pas explicitement, mais l'expérience de la communauté suggère que les scripts User Event sur l'enregistrement cible sont contournés lors de l'utilisation de `submitFields`.)

Champs pris en charge et limites

Tous les champs ne peuvent pas être mis à jour via `submitFields`. La documentation de l'API précise que **seuls les champs au niveau du corps** qui prennent en charge la modification en ligne peuvent être modifiés (Source: docs.oracle.com). En termes pratiques, cela signifie :

- **Autorisé :** Champs de corps standard ou personnalisés (cases à cocher, texte, listes déroulantes) qui sont répertoriés sur le formulaire principal et prennent en charge la modification de liste en ligne. Par exemple, « Memo », « Status », champs de segment personnalisés, etc.
- **Non autorisé :** Champs de ligne de sous-liste et champs de sous-enregistrement (tels que les sous-champs d'adresse) (Source: docs.oracle.com).

Le texte d'aide indique clairement : *« Vous ne pouvez pas utiliser cette méthode pour modifier et soumettre des champs de ligne de sous-liste ou des champs de sous-enregistrement (par exemple, les champs d'adresse). »* (Source: docs.oracle.com). Ainsi, si un script doit modifier une ligne sur une commande client (par exemple, la quantité d'une ligne d'article), `submitFields` ne peut pas le faire en une seule fois ; le script devrait plutôt charger l'enregistrement ou utiliser une approche différente (comme `record.submitLine` / `commitLine` ou SuiteQL). Cette restriction est une considération majeure dans la conception : cela signifie que `submitFields` n'est vraiment destiné qu'aux modifications simples et de haut niveau de l'en-tête de l'enregistrement.

Une implication de l'exigence de modification en ligne est que si un champ peut normalement être modifié dans la vue de liste (avec l'icône crayon), alors `submitFields` peut le gérer. Sinon, tenter de le mettre à jour via `submitFields` échoue silencieusement ou revient à une méthode classique. Par exemple, le champ audité « Created Date » sur certains enregistrements peut ne pas prendre en charge la mise à jour ; tenter un `submitFields` sur un tel champ renvoie une erreur **« invalid field »**. Les développeurs doivent s'assurer que les ID de champ dans `values` sont valides et modifiables. (L'API renverra une erreur si un champ est inexistant ou non modifiable en ligne.)

Contextes de script et promesses

Comme indiqué, `submitFields` peut être appelé à partir de **scripts côté serveur** (User Events, Scheduled, Map/Reduce, RESTlet, etc.) et de **scripts client** (Source: docs.oracle.com). Dans les scripts client, une version basée sur les promesses `record.submitFields.promise(options)` est fournie (Source: docs.oracle.com). L'utilisation de la version promise relève principalement du style JavaScript et ne modifie pas le coût de gouvernance. L'aide de SuiteScript indique explicitement : *« Remarque : Pour la version promise de cette méthode, voir `record.submitFields.promise(options)`. Notez que les promesses ne sont prises en charge que dans les scripts client. »* (Source: docs.oracle.com).

Pour nos besoins, nous nous concentrerons principalement sur l'utilisation synchrone (rappel), car les coûts et les modèles d'utilisation sont les mêmes. Il est important de noter la différence entre les scripts client et serveur : un script client s'exécute généralement dans le navigateur de l'utilisateur après le chargement d'une page, potentiellement déclenché par un événement de changement de champ sur le formulaire côté client, tandis qu'un script serveur s'exécute sur les serveurs de NetSuite en réponse à des déclencheurs ou des planifications. Dans les deux cas, `submitFields` transmet des paramètres au serveur (dans le cas du client) puis l'enregistrement est mis à jour côté serveur. En raison de cette implication du serveur, l'invocation client de `submitFields` consomme toujours des unités de gouvernance du contexte de l'utilisateur (l'exécution serveur liée au client).

Modèle de consommation de la gouvernance

Lors de l'appel de `submitFields`, NetSuite déduit immédiatement les unités fixes (10/2/5) du pool d'utilisation disponible du script. Comme `submitFields` ne charge pas l'enregistrement, il n'encourt aucun coût variable supplémentaire. En revanche, une séquence manuelle de chargement/sauvegarde encourt la somme du chargement *plus* la sauvegarde. Par exemple, charger une transaction standard coûte 10 unités ; puis la sauvegarder coûte 20 unités supplémentaires – **30 unités au total** – contre 10 unités via `submitFields`. Cette économie spectaculaire est l'avantage principal.

Une démonstration pratique de la différence apparaît dans le propre guide de *Gouvernance SuiteScript* de NetSuite. Il donne un exemple (paraphrasé) d'un User Event sur un enregistrement de transaction effectuant `record.delete` (20 unités) et `record.save` (20 unités) pour un total de 40 unités (Source: docs.oracle.com). Si cette même situation impliquait un `submitFields` au lieu de `save`, le débit doublerait. De même, une publication de la communauté éducative souligne : « Au lieu de **charger et sauvegarder un enregistrement entier, ce qui consomme plus d'unités de gouvernance et ralentit l'exécution**, NetSuite propose une alternative plus rationalisée : la méthode `submitFields()`. » (Source: community.oracle.com).

Cependant, une certaine nuance apparaît. Selon un message sur le forum des professionnels NetSuite, *« En général, record.submitFields sera plus rapide si l'enregistrement et les champs prennent en charge la modification en ligne. Dans certains cas, NetSuite convertira les appels en un chargement, une modification, une sauvegarde si l'enregistrement ou les champs ne le prennent pas en charge, donc ce n'est pas toujours plus rapide, mais cela ne devrait pas prendre plus de temps non plus. »* (Source: archive.netsuiteprofessionals.com). En d'autres termes, si vous essayez d'utiliser `submitFields` sur des champs qui ne sont pas réellement modifiables en ligne, NetSuite pourrait effectuer un chargement/sauvegarde en arrière-plan de toute façon ; le script paie toujours le coût plus élevé (bien que l'appel API lui-même soit revenu normalement). Cela se produit surtout si vous incluez un champ non pris en charge dans `values`. Les développeurs doivent donc vérifier l'adéquation : généralement, on essaie `submitFields`, et s'il échoue ou semble lent, on passe à un `load` / `save` direct. Le conseil du blog est que dans les cas normaux avec des champs appropriés, `submitFields` *« sera plus rapide »* (Source: archive.netsuiteprofessionals.com).

Modèles d'utilisation et meilleures pratiques

Compte tenu des mécanismes de `submitFields`, certains modèles ont été établis comme meilleures pratiques :

- **Pour les mises à jour simples, utilisez `submitFields`.** Chaque fois que seuls des champs de corps de document doivent être modifiés (par exemple, mettre à jour un champ personnalisé, ajuster le statut, cocher une case), privilégiez `submitFields` plutôt que `record.load` + `save`. Ceci est mis en avant dans les conseils et blogs pour administrateurs NetSuite. Par exemple, un article du *NetSuite Admin Corner* conseille exactement cela : *« La méthode `submitFields()` vous permet de mettre à jour et de soumettre un ou plusieurs champs au niveau du corps d'un enregistrement existant sans le charger en mémoire, ce qui améliore considérablement les performances du script. »* (Source:

community.oracle.com). L'auteur compare les deux approches, démontrant qu'éviter le chargement/enregistrement « consomme plus d'unités de gouvernance et ralentit l'exécution » (Source: community.oracle.com). Le modèle de code typique consiste à fournir le type d'enregistrement, l'ID et une carte (map) de valeurs, ainsi que toutes les `options` nécessaires.

- **Évitez le sourcing si ce n'est pas nécessaire.** Par défaut, `submitFields` tente de sourcer les champs dépendants (comme le calcul des totaux ou les valeurs par défaut en cascade). Cela peut être désactivé avec `options.enableSourcing = false` pour gagner en rapidité. Les exemples de code dans les réponses de la communauté utilisent souvent `enableSourcing: false` et `ignoreMandatoryFields: true` (Source: stackoverflow.com) (Source: stackoverflow.com) pour accélérer la mise à jour. Par exemple, dans un script User Event, l'exemple donné pour corriger une date de commande client utilise :

```
record.submitFields({
    type: record.Type.SALES_ORDER,
    id: currRec.getValue('createdfrom'),
    values: { trandate: currRec.getValue('trandate') },
    options: { enableSourcing: false, ignoreMandatoryFields: true }
});
```

  (Source: stackoverflow.com). Cette utilisation suggère de désactiver le sourcing, car le script souhaite probablement simplement copier une valeur de date sans remplir de nouveaux champs. De même, `ignoreMandatoryFields: true` peut être utilisé si les champs mis à jour incluent des champs obligatoires que le script pourrait ne pas définir (en ignorant certaines vérifications). Ces options sont des fonctionnalités de support plutôt que des éléments fondamentaux de `submitFields`, mais constituent des modèles utiles pour l'efficacité.

- **Utilisez-le dans les scripts Map/Reduce ou Scheduled pour les mises à jour en masse.** Pour les opérations sur de nombreux enregistrements (par exemple, mises à jour de masse de statuts, corrections de données en bloc), `submitFields` est souvent utilisé au sein d'un script Map/Reduce ou Scheduled. Dans un script Map/Reduce, chaque fonction d'étape Map ou Reduce peut appeler `submitFields` pour un sous-ensemble d'enregistrements. Étant donné que les scripts Map/Reduce « disposent d'une mise en attente (yielding) intégrée » et que chaque étape possède son propre budget d'utilisation (Source: docs.oracle.com), ils peuvent traiter de grands volumes. Par exemple, on peut charger une recherche enregistrée de 10 000 commandes client, puis dans l'étape Map, appeler `submitFields` sur chaque ID d'enregistrement pour définir un indicateur personnalisé. Avec 10 unités par appel, 10 000 enregistrements pourraient être mis à jour (10 * 10 000 = 100 000 unités au total), répartis sur de nombreuses invocations. Pendant ce temps, un script planifié (limité à 10 000 unités) ne pourrait mettre à jour qu'au maximum 1 000 commandes avec le même budget. Ainsi, tirer parti de `submitFields` dans un Map/Reduce est un modèle courant pour les tâches de données à haut volume. Les conseils de NetSuite suggèrent d'utiliser Map/Reduce « pour les données en masse » et de « planifier le traitement par lots » afin de respecter la gouvernance (Source: netsuite.folio3.com), avec `submitFields` comme outil dans cette stratégie.

- **Combinaison avec les recherches.** Souvent, `submitFields` est utilisé en tandem avec une recherche ou une requête. Un script peut effectuer une recherche enregistrée ou une boucle `search.ResultSet.each()` pour récupérer les ID et les valeurs des enregistrements, puis les transmettre à `submitFields`. Cela évite de charger des enregistrements entiers, en s'appuyant sur la recherche pilotée par la base de données pour la récupération. Un blog sur les conseils de performance recommande d'utiliser `search.lookupFields` (coût de 1 unité) si seuls quelques champs sont nécessaires, et d'utiliser `submitFields` pour enregistrer les mises à jour (Source: hutada.home.blog). Cela minimise la gouvernance et maximise la vitesse.

- **Attention aux comportements de repli (fallback).** Comme indiqué, si NetSuite constate qu'un champ n'est pas modifiable en ligne ou qu'il y a des sous-listes à mettre à jour, il pourrait effectuer en interne un chargement/enregistrement complet. Bien que cela soit rare pour des mises à jour bien formées, les développeurs doivent intercepter les erreurs ou mesurer l'utilisation. Les conseils sur les forums (Source: archive.netsuiteprofessionals.com) suggèrent de tester si `submitFields` aide réellement. En pratique, si un appel semble consommer le même nombre d'unités qu'un enregistrement, on peut supposer un repli. Un modèle prudent consiste à utiliser un bloc try/catch autour de `submitFields` et, en cas d'échec, à charger normalement. De même, si l'intégrité transactionnelle est importante (comme le besoin d'une logique `beforeSubmit`), on pourrait préférer un enregistrement manuel mieux intégré au framework d'événements, même à un coût plus élevé.

L'annexe Tableau 2 illustre quelques exemples de coûts d'utilisation et de limites selon les types de scripts :

SCÉNARIO	UNITÉS PAR APPEL	LIMITE DU SCRIPT	# APPELS POSSIBLES (APPROX.)
Mettre à jour une commande client via `submitFields` (transaction)	10 unités	Planifié : 10 000 (Source: docs.oracle.com)	~1 000 appels (usage 10k complet)
Mettre à jour une commande client via `save`	20 unités	Planifié : 10 000 (Source: docs.oracle.com)	~500 appels (usage 10k complet)
Mettre à jour un enregistrement personnalisé via `submitFields`	2 unités	User Event : 1 000 (Source: docs.oracle.com)	~500 appels (1 000/2)
Mettre à jour un enregistrement personnalisé via `save`	4 unités	User Event : 1 000 (Source: docs.oracle.com)	~250 appels (1 000/4)

Tableau 2 : Exemples de comparaisons de débit pour quelques scénarios, en supposant une utilisation maximale de la limite du script. Par exemple, un script planifié avec 10 000 unités pourrait effectuer environ 1 000 appels à `submitFields` sur un enregistrement de transaction (10 unités chacun), contre seulement 500 appels via `save` (20 unités chacun).

Les chiffres ci-dessus démontrent l'avantage de débit de `submitFields` dans les limites données. Par exemple, un script planifié peut mettre à jour **deux fois plus** de commandes client en utilisant `submitFields` au lieu de `save`. Un avantage net de cette approche est explicitement noté dans un article LinkedIn : l'utilisation de `submitFields` « pour les mises à jour simples » est l'une des meilleures pratiques listées dans l'optimisation de la gouvernance SuiteScript.

Comparaison avec d'autres méthodes

Il est utile de comparer `submitFields` avec quelques alternatives :

- **`record.save()`** : Comme discuté, `save()` est une validation complète de l'enregistrement. Il peut mettre à jour n'importe quel champ (corps, sous-liste, sous-enregistrement) car l'enregistrement est entièrement chargé. Mais il déclenche tous les événements utilisateur au niveau de l'enregistrement (`beforeLoad`, `beforeSubmit`, `afterSubmit`), effectue des vérifications complètes de mandatory et de sourcing, et coûte plus d'unités (Source: docs.oracle.com). Pour une mise à jour multi-champs couvrant le corps et la sous-liste, `save()` peut être le seul choix. Mais si seuls les champs du corps sont nécessaires, l'enregistrement est excessif. Le conseil de la communauté souligne : *« charger et enregistrer un enregistrement entier, ce qui consomme plus d'unités de gouvernance et ralentit l'exécution »* par rapport à l'utilisation de `submitFields` (Source: community.oracle.com).

- **`record.transform()`** : Cette méthode convertit un type d'enregistrement en un autre (par exemple, une opportunité en commande) et n'est généralement pas une alternative pour les mises à jour simples. Son coût d'utilisation (10/2/5) est le même que `submitFields` (Source: docs.oracle.com), mais son objectif est différent.

- **API 1.0 obsolète `nlapiSubmitField`** : Dans SuiteScript 1.0, la fonction analogue était `nlapiSubmitField`. Elle avait une intention similaire mais une signature différente. SuiteScript 2.x utilise `record.submitFields` pour la remplacer. Le comportement et les intentions de performance sont similaires, bien que la gouvernance exacte puisse différer en arrière-plan. Pour les scripts 2.1 appelant des fonctions 1.0, il y a une surcharge liée au chargement de la couche de compatibilité 1.0, il est donc généralement préférable d'utiliser l'API 2.x.

- **SuiteQL ou services Web REST** : Pour les mises à jour à très haut volume (milliers d'enregistrements), certains développeurs envisagent d'utiliser des requêtes/mises à jour SuiteQL ou l'API REST. Les opérations DML (Data Manipulation Language) de SuiteQL sont relativement nouvelles (SuiteScript 2.1) et peuvent effectuer des mises à jour de masse directement dans la base de données. Cependant, tous les champs ne prennent pas en charge le DML, et cela consomme toujours des unités de gouvernance. L'API REST (SuiteTalk ou services Web REST) est externe à la gouvernance de SuiteScript (elle a des quotas distincts), mais elle est asynchrone et a ses propres limites. En général, pour le travail au sein de SuiteScript, `submitFields` reste la méthode intégrée la plus simple.

Études de cas et exemples

Nous présentons maintenant plusieurs exemples et scénarios réels illustrant comment `submitFields` est employé et comment il impacte la gouvernance.

Exemple 1 : Mise à jour simple de champ dans un User Event

Un cas d'utilisation typique est la mise à jour d'une transaction parente à partir d'un événement utilisateur d'un enregistrement enfant. Le scénario suivant provient d'un exemple de la communauté (Source: stackoverflow.com) : dans un script User Event sur une exécution de commande (Item Fulfillment), le développeur souhaite mettre à jour la « Date » sur la commande client d'origine chaque fois que l'exécution est créée. Dans `beforeSubmit(context)`, il effectue :

```
var currRec = context.newRecord;
// ... ID de la commande parente trouvé dans currRec.getValue('createdfrom')
record.submitFields({
    type: record.Type.SALES_ORDER,
    id: currRec.getValue('createdfrom'),
    values: {
        trandate: currRec.getValue('trandate')
    },
    options: {
        enableSourcing: false,
        ignoreMandatoryFields: true
    }
});
```

Ici, `record.submitFields` met à jour la `trandate` sur la commande client parente. Cela évite de charger entièrement l'enregistrement de la commande client. Le coût de gouvernance est de 10 unités pour la mise à jour d'un enregistrement de transaction. Si le script avait plutôt fait `record.load({type: record.Type.SALES_ORDER, id: ...})` suivi de `soRec.setValue(...)` et `soRec.save()`, cela aurait coûté 10 (chargement) + 20 (enregistrement) = 30 unités (Source: hutada.home.blog) (Source: docs.oracle.com). En utilisant `submitFields`, cette mise à jour d'un seul champ n'a coûté que 10 unités. (Notez que dans cet exemple, le code utilisait `record.submitField` au singulier ; cela semble être une erreur dans la réponse StackOverflow – la méthode 2.x correcte est `submitFields` telle que documentée (Source: docs.oracle.com). L'idée reste la même.)

Cet exemple montre un modèle important : utiliser `submitFields` dans `beforeSubmit` ou `afterSubmit` d'un User Event pour modifier un enregistrement lié. C'est une approche plus efficace qui évite également le déclenchement récursif d'événements sur l'enregistrement actuel (puisque l'enregistrement actuel est déjà en cours d'enregistrement par l'événement utilisateur). Les retours de la communauté suggèrent que c'est sûr : parce que nous mettons à jour un enregistrement différent (le parent) lors de l'événement de l'enfant, cela ne devrait pas provoquer de boucle. Le code ci-dessus inclut `ignoreMandatoryFields: true` pour éviter les erreurs si tous les champs obligatoires ne sont pas définis sur la commande client parente. Ce modèle est fréquemment observé en pratique et est une utilisation acceptée de `submitFields` dans le développement SuiteScript.

Exemple 2 : Changements de statut en masse dans un script planifié

Considérez un scénario : une entreprise doit fermer 5 000 commandes client à la fin du mois. Un script planifié est déployé pour effectuer ce traitement par lots. Si le script était naïf et faisait cela via load/save, il épuiserait rapidement son quota d'utilisation. Au lieu de cela, une solution courante est :

1. Rechercher les commandes client ouvertes (par exemple, une recherche enregistrée qui renvoie les ID internes).
2. Dans une boucle, appeler `record.submitFields({type: SALES_ORDER, id: soId, values: {status: 'Closed'}})` pour chaque ID.

Chaque appel coûte 10 unités (transaction). À 10 unités chacun, 1 000 commandes utiliseraient 10 000 unités, atteignant la limite du script planifié (Source: docs.oracle.com). Pour mettre à jour 5 000 commandes, le script peut soit s'exécuter par lots (s'arrêter après avoir atteint ~9 000 unités, puis se replanifier), soit, mieux encore, être implémenté en tant que script Map/Reduce qui itère sur les résultats de recherche, car Map/Reduce dispose

d'une mise en attente et peut soutenir davantage d'opérations au total. Néanmoins, même en tant que script limité à 10k, l'utilisation de `submitFields` signifie 1 000 commandes par exécution. Si le script avait plutôt fait (incorrectement) :

```
var soRec = record.load({type: record.Type.SALES_ORDER, id: soId});
soRec.setValue({fieldId: 'status', value: 'Closed'});
soRec.save();
```

chaque itération coûterait 30 unités (10 chargement + 20 enregistrement). À 30 unités, seules ~333 commandes pourraient être traitées en une exécution de 10k. Le gain d'efficacité de `submitFields` (traitement de 3 fois plus de commandes par exécution) est énorme. De cette façon, même sans Map/Reduce, `submitFields` rend le script planifié viable là où une approche naïve échouerait.

Aucun extrait de code public n'est cité ici, mais la logique découle directement du tableau de gouvernance (Source: docs.oracle.com) (Source: docs.oracle.com) et des limites d'utilisation (Source: docs.oracle.com). Cette étude de cas souligne comment calculer le débit : étant donné les coûts unitaires et les limites par script, vous pouvez planifier combien d'enregistrements mettre à jour par exécution. Si le volume de données mensuel est énorme, plusieurs exécutions planifiées ou une approche Map/Reduce sont recommandées.

Exemple 3 : Recherche en ligne + SubmitFields pour une mise à jour en temps réel

Parfois, les scripts côté client utilisent `submitFields` pour de petites modifications. Par exemple, un script client sur un formulaire de facture pourrait fournir un bouton qui, lorsqu'il est cliqué, met à jour un indicateur sur des enregistrements liés. Le client appelle une Suitelet ou exécute `record.submitFields` directement (avec la capacité client SS2.x) pour mettre à jour les données. Un scénario exemple : un enregistrement de facture client a une case à cocher personnalisée « Envoyer au CRM ». Lorsque l'utilisateur clique sur « Envoyer au CRM » sur la page de facture (interface utilisateur côté client), le script appelle `submitFields` pour définir cette case à cocher sur vrai sans recharger la page. Le client n'appelle jamais `record.save` ; il invoque simplement `submitFields` sur l'ID de la facture. Le coût de gouvernance provient du traitement de cet appel par le serveur. Ce modèle évite les postbacks et fournit un retour quasi instantané à l'utilisateur, affichant un ID ou un message toast. (En pratique, l'article de conseils pour administrateurs mentionné plus haut fait probablement référence à une utilisation similaire – « les scripts client au niveau de l'enregistrement permettent de telles mises à jour en ligne ».)

Bien qu'aucune référence externe spécifique ne soit donnée, ce modèle est bien connu : utiliser `submitFields` dans un script client comme une mise à jour de type AJAX. La clé ici est que même dans un script client, chaque appel d'API est régi par le budget du serveur, donc la règle 10/2/5 s'applique toujours. Les clients doivent toujours respecter les limites de script (les scripts client ont des budgets de 1 000 unités (Source: docs.oracle.com), mais comme ces appels sont généralement peu fréquents, ils atteignent rarement le seuil.

Résumé de la gouvernance et des limites de ressources

Nous avons déjà discuté de l'utilisation de la gouvernance *par appel* de `submitFields`, mais il est utile de résumer comment cela s'intègre dans les **limites plus larges des types de scripts**. Voici une référence rapide de certaines limites importantes pour SuiteScript 2.x (citant la documentation Oracle) :

- **Scripts User Event / Client / Suitelet / Portlet (2.x)** : 1 000 unités par exécution (Source: docs.oracle.com) (Source: docs.oracle.com). (Chaque script client est mesuré séparément (Source: docs.oracle.com), donc deux scripts client différents sur le même formulaire ne partagent pas la limite de 1 000.)
- **Script planifié (2.x)** : 10 000 unités par exécution (Source: docs.oracle.com).
- **Script Map/Reduce (2.x)** : Aucune limite globale (chaque étape a son propre décompte en cours) (Source: docs.oracle.com) ; recommandé pour les très grands traitements.
- **RESTlet (2.x)** : 5 000 unités par exécution (Source: docs.oracle.com).
- **Mass Update Script (2.x)** : 1 000 unités par enregistrement ou par exécution (selon le contexte) (Source: docs.oracle.com).
- **Bundle Install Scripts (2.x)** : 10 000 unités (Source: docs.oracle.com) (pour SuiteBundler).
- Les autres contextes (Workflow Action, Portlet, etc.) ont également des limites, généralement autour de 1 000 unités par exécution.

Ces limites imposent des compromis. Par exemple, si vous essayez de mettre à jour 2 000 enregistrements via `submitFields` dans un seul script planifié (limite de 10 000), vous utiliseriez 20 000 unités au total – soit le double du budget – et vous obtiendriez une erreur `SSS_USAGE_LIMIT_EXCEEDED`. Vous devez donc traiter votre charge de travail par lots ou utiliser des points de reprise (approche CSV).

Surveillance de l'utilisation : Les scripts peuvent vérifier l'utilisation restante via `runtime.getCurrentScript().getRemainingUsage()` (disponible en 2.x). Les bonnes pratiques recommandent de vérifier périodiquement l'utilisation restante dans les longues boucles, et éventuellement de céder la main (dans un Map/Reduce) ou de diviser le travail en étapes avant d'atteindre zéro. Cela garantit une récupération en douceur, par exemple en enregistrant l'état et en replanifiant le script. La documentation officielle de NetSuite traite de la « Surveillance de l'utilisation des scripts » et présente même des tableaux d'exemples de calculs d'utilisation (Source: docs.oracle.com) (Source: docs.oracle.com). Par exemple, ils illustrent qu'un script effectuant un `delete` et un `save` sur des transactions consomme 40 unités d'utilisation (Source: docs.oracle.com), ce qui est bien en dessous de 1 000. De même, dans l'exemple de script planifié du Tableau 1 (avec deux appels `transform` et un appel `email.send` totalisant environ 24-40 unités) (Source: docs.oracle.com), l'utilisation est encore bien inférieure à 10 000, mais il est suggéré d'utiliser Map/Reduce pour les tâches extrêmement longues.

En résumé, la gouvernance est un système à plusieurs niveaux. Au niveau micro, chaque appel à `submitFields` coûte 10/2/5 unités (Source: docs.oracle.com). Au niveau macro, chaque exécution de script ne peut accumuler que jusqu'à son plafond (par exemple 1 000 ou 10 000) (Source: docs.oracle.com) (Source: docs.oracle.com). Une bonne conception de script combine des appels efficaces (comme `submitFields`) avec une conscience du nombre d'opérations pouvant être effectuées.

Modèles, mises en garde et bonnes pratiques

Quelques modèles et précautions importants apparaissent pour les développeurs utilisant `submitFields` :

- **Sourcing en ligne** : Si vous mettez à jour un champ qui en alimente automatiquement d'autres, notez que par défaut, `submitFields` ne **source pas** les champs associés, sauf si `enableSourcing` est défini sur `true`. Cela peut être à la fois un avantage (vitesse) et un risque (si vous attendiez un sourcing). Par exemple, la mise à jour d'un ID d'article sur un champ personnalisé pourrait ne pas récupérer son tarif par défaut si le sourcing n'est pas activé. Soyez toujours explicite sur le comportement de sourcing via l'objet `options` si nécessaire.

- **Champs obligatoires** : `ignoreMandatoryFields: true` permet d'ignorer les vérifications sur les champs obligatoires. Utilisez-le avec prudence : si votre mise à jour contourne un champ obligatoire, l'enregistrement est sauvegardé mais peut enfreindre les règles de données prévues. Il est souvent utilisé lorsque le script sait que le champ est déjà défini ou non pertinent. L'exemple de code (Source: stackoverflow.com) (Source: stackoverflow.com) montre son utilisation pour la performance.

- **Gestion des versions d'enregistrement** : Si plusieurs scripts ou utilisateurs mettent à jour le même enregistrement simultanément, `submitFields` peut créer un « conflit d'édition » si deux processus écrivent dans des champs différents en même temps. Puisqu'il s'agit essentiellement d'une sauvegarde, si l'enregistrement a été modifié depuis sa récupération, la dernière sauvegarde l'emporte. Les développeurs devraient idéalement effectuer ces mises à jour dans des contextes isolés (par exemple, des User Events) ou ajouter une logique pour éviter d'écraser les modifications d'autrui. NetSuite renverra une erreur en cas de conflit.

- **Comportement de repli** : Comme mentionné, tous les champs ne peuvent pas être modifiés en ligne. Nous rappelons que l'appel de `submitFields` sur un champ non pris en charge peut amener NetSuite à effectuer un chargement/sauvegarde, consommant plus d'unités que prévu (Source: archive.netsuiteprofessionals.com). Cela peut également générer une erreur si le champ n'est vraiment pas modifiable. Un modèle robuste consiste à intercepter les exceptions et éventuellement à les journaliser, afin de diagnostiquer si un développeur a supposé à tort que le champ était modifiable. Si des erreurs surviennent, passez au modèle sûr de load/setValue/save pour cet enregistrement.

- **Absence de points de reprise** : Contrairement à Map/Reduce, les scripts de base (UE, planifiés) n'ont pas de point de reprise programmatique. Si vous mettez à jour des milliers d'enregistrements avec `submitFields`, une seule exécution de script peut expirer ou dépasser les limites. La documentation de SuiteScript 2.x avertit que si un script planifié risque d'être long, il faut envisager Map/Reduce, car les scripts planifiés *« ne disposent pas de méthode permettant de définir des points de récupération ou de fournir un rendement pour éviter de dépasser la gouvernance autorisée »* (Source: docs.oracle.com). Cela souligne que `submitFields`, bien qu'efficace par appel, ne modifie pas les limites architecturales. Dans un script planifié, vous devrez peut-être suivre la progression (par exemple, le dernier ID interne traité) et replanifier le script à plusieurs reprises.

- **Atomicité et transactions** : NetSuite ne fournit pas de transactions multi-lignes dans SuiteScript ; chaque sauvegarde d'enregistrement (que ce soit via `submitFields` ou `save()`) est un commit indépendant. Si vous avez besoin d'un comportement de type rollback, vous ne pouvez pas compter uniquement sur SuiteScript – généralement, la logique métier doit prévoir des mesures compensatoires en cas d'échec.

Exemples de modèles issus des ressources communautaires

Plusieurs sources communautaires et NetSuite mettent en évidence des modèles d'utilisation idéaux :

- Un **conseil d'administrateur NetSuite** conseille : *« Lorsque vous travaillez avec SuiteScript, l'efficacité est essentielle, surtout lors de la mise à jour d'enregistrements. Au lieu de charger et de sauvegarder un enregistrement entier, ce qui consomme plus d'unités de gouvernance et ralentit l'exécution, NetSuite propose une alternative plus rationalisée : la méthode submitFields(). »* (Source: community.oracle.com). Le conseil suggère ensuite le modèle consistant à appeler `record.submitFields(...)` directement avec les nouvelles valeurs, au lieu d'effectuer une boucle `load()`.

- Un **blog de développeur** sur les performances des scripts recommande : *« Si vous devez modifier des valeurs sur un enregistrement, utilisez record.submitFields. Il utilise entre 2 et 10 unités de gouvernance et est plus rapide que le chargement et la sauvegarde d'un enregistrement. »* (Source: hutada.home.blog). Ici, l'auteur résume exactement le coût empirique. La formulation « entre 2 et 10 » reflète différents types d'enregistrements (personnalisés vs transactions), renforçant le tableau des coûts précédent.

- Sur un **forum de professionnels NetSuite**, un membre répond à une question similaire : *« En général, record.submitFields sera plus rapide si l'enregistrement et les champs prennent en charge l'édition en ligne. … ce n'est pas toujours plus rapide, mais cela ne devrait pas prendre plus de temps non plus. »* (Source: archive.netsuiteprofessionals.com). Cela confirme une directive pratique : utilisez `submitFields` par défaut, mais soyez conscient des cas particuliers. L'expert (le développeur NetSuite Scott Von Duyn) affirme essentiellement qu'il s'agit de la méthode privilégiée lorsqu'elle est applicable.

- Dans les **bonnes pratiques Map/Reduce**, il est recommandé de regrouper la logique en transactions plus petites. `submitFields` s'intègre ici comme une opération rapide par enregistrement. Le guide du développeur SuiteScript note que Map/Reduce est idéal pour le traitement massif où « les opérations sont appliquées à plusieurs objets, un par un » (Source: docs.oracle.com), invitant implicitement à utiliser `submitFields` dans une étape de mappage pour chaque objet.

Étude de cas : Analyse des performances et de la gouvernance

Bien que SuiteScript manque de benchmarks de performance officiels publiés, nous pouvons analyser un scénario hypothétique avec des chiffres réels. Supposons qu'un script doive mettre à jour 10 champs sur chacun des 500 enregistrements clients. Nous comparons deux approches :

1. **Utilisation de `submitFields`** 10 fois par enregistrement (puisque chaque appel peut mettre à jour plusieurs champs, on pourrait en réalité utiliser 1 seul appel avec les 10 champs, mais supposons qu'un appel traite les 10 champs). Soit 1 appel * 500 enregistrements = 500 appels. Chaque appel sur un « client » utilise 5 unités. Total = 500 * 5 = 2 500 unités.

2. **Utilisation de `record.load/save`** à chaque fois : Pour chaque enregistrement, 1 `load` (5 unités) + 1 `save` (10 unités) = 15 unités par enregistrement. Pour 500 enregistrements, total = 500 * 15 = 7 500 unités.

Dans l'approche 1, nous utilisons **2 500 unités** ; dans l'approche 2, **7 500 unités**. Cela signifie que `submitFields` permet d'économiser 5 000 unités. Dans un script planifié avec une limite de 10 000, l'approche 1 tient facilement en une seule exécution (n'utilisant que 25 % du budget), tandis que l'approche 2 l'épuiserait presque. Un script User Event (limite de 1 000) ne supporterait même pas 500 chargements (consommerait 7 500), mais pourrait en gérer 200 avec `submitFields` (200*5 = 1 000). Ce type de calcul guide la conception du déploiement.

De nombreux ingénieurs solutions NetSuite effectuent ces estimations. Les **exemples de gouvernance** d'Oracle montrent souvent des calculs similaires : par exemple, l'exemple de documentation d'un script planifié utilisant deux `transform` (5 unités chacun en tant qu'« enregistrements de transaction non standard ») et un e-mail (20 unités) pour un total d'environ 40 unités (Source: docs.oracle.com), puis notant la limite de 10 000. Notre exemple client est analogue, montrant comment `submitFields` étend considérablement la capacité des scripts.

Constatations et implications

Les preuves et exemples ci-dessus conduisent à plusieurs conclusions :

- **Record.submitFields est un outil d'optimisation critique.** Pour les développeurs conscients des limites de gouvernance, savoir qu'une simple mise à jour de champ peut coûter aussi peu que 2 à 10 unités (au lieu de 4 à 20) peut multiplier le débit. Cela **réduit effectivement de moitié le coût de gouvernance par mise à jour de transaction**, ce qui est un facteur majeur dans la conception de scripts.

- **Utilisez-le partout où c'est autorisé.** Parce que `submitFields` est pris en charge dans presque tous les types de scripts, il doit être utilisé chaque fois que le cas d'utilisation correspond. Dans les scripts clients, utilisez la variante promise pour le style. Dans Map/Reduce, il est idéal pour chaque tâche de mappage/réduction. Dans les scripts par lots, c'est l'outil de travail principal.

- **Envisagez toujours un repli.** Avant le déploiement, vérifiez que les champs cibles prennent réellement en charge l'édition en ligne. Il est sage d'entourer `submitFields` d'une gestion des erreurs. Journalisez et surveillez tout ce qui est absorbé par des sauvegardes complètes. Au fil du temps, vérifiez les notes de version de NetSuite ; si de nouveaux types d'enregistrements ou champs perdent la prise en charge en ligne, ajustez les scripts en conséquence.

- **Concevez en fonction des limites.** Malgré son efficacité, `submitFields` n'est pas magique. Il consomme toujours des unités. Les équipes doivent continuer à surveiller les unités, à diviser les gros travaux et peut-être à utiliser Map/Reduce. Former les développeurs à la lecture des rapports de gouvernance (dans les journaux de script) peut mettre en évidence si trop d'unités sont utilisées dans une boucle. Implication future : la documentation d'Oracle et les nouveaux frameworks mettent l'accent sur la gouvernance plus que jamais (passant de 1.0 à 2.x à 2.1, les « modèles conscients de la gouvernance » ont été un thème récurrent (Source: netsuite.folio3.com).

- **Expérience utilisateur.** Sur une note positive, `submitFields` favorise des interfaces plus réactives. Par exemple, les scripts clients peuvent déclencher une mise à jour rapide sans rafraîchissement de page. Cela peut améliorer la performance perçue et la satisfaction des utilisateurs. Cependant, comme il n'exécute pas la logique `afterSubmit`, toute règle métier critique qui se déclenche normalement lors de la soumission d'un enregistrement peut être ignorée. Les organisations doivent évaluer : si une modification nécessite une validation côté serveur ou des workflows, peut-être que `submitFields` est trop bas niveau et pourrait contourner des étapes nécessaires.

- **Orientations futures.** Alors que NetSuite continue d'améliorer sa plateforme, nous spéculons sur les tendances possibles :

  - **Davantage d'API asynchrones ou de mise à jour en masse.** NetSuite pourrait introduire des opérations de données encore plus efficaces (par exemple, des services de chargement de données, des appels RPC par lots) pour compléter `submitFields`. Par exemple, un véritable point de terminaison REST de « mise à jour en masse » qui gère des ensembles d'enregistrements pourrait émerger, réduisant les allers-retours.
  - **Modifications du modèle de gouvernance.** Il existe des discussions dans l'industrie sur l'assouplissement de la gouvernance ou l'offre de plus d'unités aux comptes plus importants. Si les limites de gouvernance augmentent de manière significative, l'urgence d'optimiser par unité deviendra légèrement moindre, mais cela restera important dans les intégrations à grande échelle.
  - **Améliorations de SuiteAnalytics / REST.** Des outils comme SuiteQL et Analytics pourraient permettre de définir les champs de nombreux enregistrements via des requêtes. Le DML SuiteQL (UPDATE) est déjà possible en 2.1, ce qui pourrait réduire certaines utilisations de `submitFields` (bien qu'il ait aussi des limites). Si Oracle étend les capacités de Suiteschema ou GraphQL, les mises à jour d'enregistrements pourraient se déplacer vers ces canaux pour les cas critiques en termes de performance.

Conclusion

La méthode `record.submitFields` dans SuiteScript 2.1 est un mécanisme essentiel pour mettre à jour les enregistrements existants de manière économe en ressources. Notre analyse montre qu'elle offre un coût de gouvernance *nettement* inférieur aux opérations complètes de chargement/sauvegarde (environ 5 à 10 unités économisées par transaction) (Source: docs.oracle.com) (Source: docs.oracle.com), ce qui en fait une bonne pratique pour les changements de champs simples (Source: community.oracle.com) (Source: hutada.home.blog). La documentation officielle et les conseils d'experts de la communauté approuvent systématiquement `submitFields` pour les cas où elle s'applique.

Cependant, les développeurs doivent rester conscients de ses contraintes : elle ne peut pas gérer les sous-listes/sous-enregistrements (Source: docs.oracle.com), elle peut revenir à un commit complet si les champs ne prennent pas en charge l'édition en ligne (Source: archive.netsuiteprofessionals.com), et elle consomme toujours des unités de gouvernance qui contribuent à la limite du script. Équilibrer l'utilisation de `submitFields` avec des vérifications `Script.getRemainingUsage`, le traitement par lots et les types de scripts à haute limite (Map/Reduce) est la clé de solutions robustes.

À l'avenir, à mesure que les besoins de personnalisation de NetSuite augmenteront en échelle et en complexité, l'exploitation d'API efficaces comme `submitFields` fera partie de la boîte à outils de tout développeur avancé. La maîtrise de son empreinte de gouvernance et de ses modèles d'utilisation peut faire la différence entre un script qui s'exécute de manière fiable et un script qui expire. Avec le moteur moderne de SuiteScript 2.1 offrant une plus grande expressivité, les meilleures pratiques autour de `submitFields` soulignent que même dans un environnement de script natif cloud, le **codage soucieux des ressources** reste primordial.

Références : Documentation officielle d'Oracle SuiteScript 2.x et guides de gouvernance (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com) ; publications de la communauté des développeurs NetSuite et conseils d'administration (Source: community.oracle.com) (Source: community.oracle.com) (Source: archive.netsuiteprofessionals.com) (Source: stackoverflow.com) (Source: stackoverflow.com) ; blogs spécialisés sur les performances de SuiteScript (Source: hutada.home.blog) (Source: hutada.home.blog). Toutes les affirmations factuelles ci-dessus sont tirées de ces sources.

Étiquettes: suitescript-21, recordsubmitfields, gouvernance-netsuite, optimisation-netsuite, limites-api, performance-suitescript, developpement-netsuite

AVERTISSEMENT

Ce document est fourni à titre informatif uniquement. Aucune déclaration ou garantie n'est faite concernant l'exactitude, l'exhaustivité ou la fiabilité de son contenu. Toute utilisation de ces informations est à vos propres risques. Houseblend ne sera pas responsable des dommages découlant de l'utilisation de ce document. Ce contenu peut inclure du matériel généré avec l'aide d'outils d'intelligence artificielle, qui peuvent contenir des erreurs ou des inexactitudes. Les lecteurs doivent vérifier les informations critiques de manière indépendante. Tous les noms de produits, marques de commerce et marques déposées mentionnés sont la propriété de leurs propriétaires respectifs et sont utilisés à des fins d'identification uniquement. L'utilisation de ces noms n'implique pas l'approbation. Ce document ne constitue pas un conseil professionnel ou juridique. Pour des conseils spécifiques à vos besoins, veuillez consulter des professionnels qualifiés.