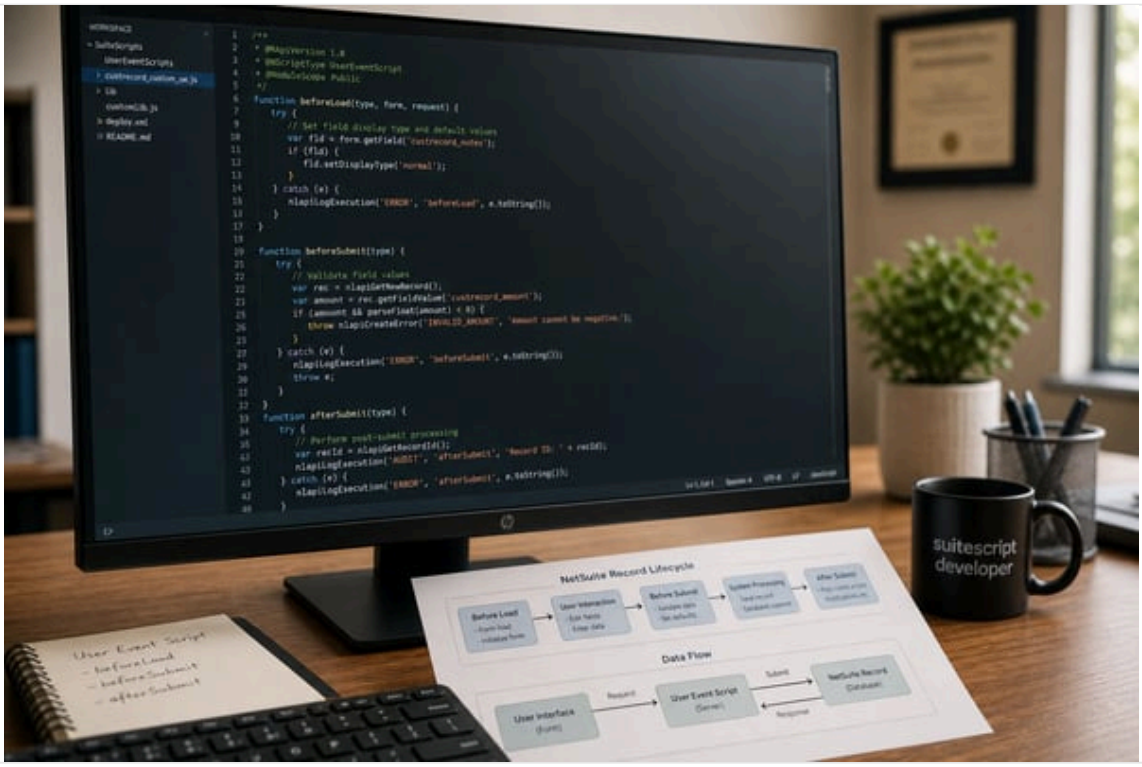
SuiteScript 1.0 User Events : de beforeLoad à afterSubmit

Publié le 14 mai 2026 35 min de lecture



Résumé analytique

SuiteScript est le framework de personnalisation de NetSuite basé sur JavaScript, et **SuiteScript 1.0** était la version originale utilisée pour étendre les enregistrements NetSuite. Un aspect essentiel de SuiteScript 1.0 est le **User Event Script**, qui fournit des **points d'entrée** à des moments clés du cycle de vie d'un enregistrement : *beforeLoad*, *beforeSubmit* et *afterSubmit*. Ces rappels (callbacks) côté serveur permettent aux développeurs de modifier l'interface utilisateur du formulaire, de valider ou d'altérer les données avant leur enregistrement, et d'exécuter une logique post-enregistrement telle que l'envoi d'e-mails ou la création d'enregistrements associés. Chaque point d'entrée remplit un objectif distinct dans le flux de données des transactions NetSuite. Par exemple, la documentation officielle de NetSuite définit **beforeLoad** comme une fonction qui « s'exécute avant le chargement d'un enregistrement — chaque fois qu'il y a une opération de lecture » (Source: docs.oracle.com), tandis que **beforeSubmit** et **afterSubmit** sont invoqués autour des opérations d'écriture en base de données (Source: so.parthpatel.net) (Source: docs.oracle.com).

Ce rapport fournit une référence approfondie sur les points d'entrée des User Event Scripts de SuiteScript 1.0, couvrant leur comportement, leurs déclencheurs, leurs paramètres et les meilleures pratiques. Nous incluons une comparaison détaillée avec le moderne SuiteScript 2.x, des citations faisant autorité de la documentation Oracle et de sources expertes, des exemples de code et des tableaux résumant les aspects clés. Nous intégrons également des modèles d'utilisation, des considérations de performance et des exemples concrets. Bien que SuiteScript 1.0 soit obsolète (Oracle note que la version 1.0 « n'est plus mise à jour » et recommande SuiteScript 2.x/2.1 pour tout nouveau développement (Source: docs.oracle.com) (Source: docs.oracle.com), de nombreux systèmes hérités utilisent encore la version 1.0 ; il est donc essentiel de comprendre ces points d'entrée en profondeur pour maintenir et migrer les solutions NetSuite existantes.

Introduction et contexte

NetSuite est une plateforme ERP/CRM cloud de premier plan, et *SuiteScript* est son langage de script intégré qui permet une logique métier personnalisée. SuiteScript 1.0 (la version originale) permet aux développeurs d'écrire du code JavaScript à la fois côté client et côté serveur. Côté **serveur**, l'un des types de script les plus courants est le *User Event Script*. Un User Event Script est déclenché automatiquement à des points définis

du cycle de vie CRUD (Create, Read, Update, Delete) d'un enregistrement. Plus précisément, SuiteScript 1.0 définit trois points d'entrée pour les User Event Scripts :

- **beforeLoad** : s'exécute *avant* qu'un enregistrement existant ne soit chargé pour affichage ou pour toute opération de lecture.
- **beforeSubmit** : s'exécute *après* qu'un utilisateur a cliqué sur « Enregistrer » (ou qu'un script/ CSV déclenche une écriture), mais *avant* que l'enregistrement ne soit validé dans la base de données.
- **afterSubmit** : s'exécute *après* que l'enregistrement a été enregistré avec succès dans la base de données.

Ces points d'entrée correspondent aux **types d'événements** dans SuiteScript 1.0 (souvent simplement la chaîne « type » dans l'API). Dans SuiteScript 2.x, Oracle a renommé ces « points d'entrée » et fournit un objet de contexte plus riche, mais la séquence fondamentale reste la même. Il est important de noter que NetSuite souligne que **les User Event Scripts ne peuvent pas s'enchaîner entre eux** – un événement utilisateur ne peut pas en déclencher directement un autre sur un enregistrement ou un workflow associé (Source: docs.oracle.com). Cela signifie que chaque User Event gère son propre événement d'enregistrement de manière isolée.

SuiteScript 1.0 est désormais une API héritée. Oracle déclare que « SuiteScript 1.0 est une version précédente de SuiteScript » qui « n'est plus mise à jour » (Source: docs.oracle.com), et que le développement actuel doit utiliser SuiteScript 2.x ou 2.1 (Source: docs.oracle.com). Néanmoins, comme de nombreuses implémentations NetSuite en production ont été construites en utilisant la version 1.0, les projets de maintenance et de migration doivent comprendre le modèle d'événement utilisateur 1.0 en détail. Ce rapport se concentrera sur les points d'entrée et la référence de SuiteScript 1.0, tout en notant les différences applicables avec le framework moderne SuiteScript 2.x.

Tout au long de ce rapport, nous citerons la documentation officielle de NetSuite et les sources pour développeurs. Par exemple, l'aide en ligne d'Oracle explique que **beforeLoad** « exécute une fonction avant le chargement d'un enregistrement — chaque fois qu'il y a une opération de lecture » (Source: docs.oracle.com). De même, le guide des meilleures pratiques d'Oracle stipule explicitement que « les User Event Scripts s'exécutent pendant les événements beforeLoad, beforeSubmit et afterSubmit » (Source: docs.oracle.com). Nous exploiterons ces sources pour garantir que chaque affirmation sur le fonctionnement de ces événements est fondée sur des preuves.

Modèle d'User Event de SuiteScript 1.0

Définitions des points d'entrée et déclencheurs

Les User Event Scripts de SuiteScript 1.0 définissent des fonctions globales correspondant à chaque point d'entrée. Dans la version 1.0, les signatures et les paramètres sont plus simples que dans la version 2.x. En particulier :

- `function beforeLoad(type, form, request) { ... }`
- `function beforeSubmit(type) { ... }`
- `function afterSubmit(type) { ... }`

Dans chaque cas, le paramètre `type` est une chaîne (ou un objet) indiquant l'**action déclencheuse** (par exemple « create », « edit », « view », « approve », etc.). Le point d'entrée *beforeLoad* possède deux paramètres supplémentaires : `form`, un `nlobjForm` permettant la personnalisation de l'interface utilisateur, et `request`, un `nlobjRequest` pour les requêtes HTTP GET (Source: so.parthpatel.net). La signature de la fonction peut être confirmée dans les exemples SuiteScript. Par exemple, un tutoriel NetSuite montre un gestionnaire beforeLoad minimal comme suit :

```
// Exemple SuiteScript 1.0
function beforeLoad(type, form, request) {
    nlapiLogExecution("DEBUG", "Before Load", "type=" + type);
}
```

(Source: so.parthpatel.net).

SuiteScript 2.x a remplacé ces signatures par un objet `scriptContext` unique. Comme le note la documentation d'Oracle, chaque point d'entrée 2.x gagne des paramètres supplémentaires : le point d'entrée 2.0 `beforeLoad(context)` inclut un `context.newRecord`, et les points d'entrée 2.0 `beforeSubmit(context)` et `afterSubmit(context)` incluent à la fois `context.oldRecord` et `context.newRecord` (Source: docs.oracle.com). En revanche, les `beforeSubmit` / `afterSubmit` de SuiteScript 1.0 n'ont que le `type` (aucun objet d'enregistrement), et `beforeLoad` n'a que la référence à l'interface utilisateur du formulaire (aucun `newRecord`).

La documentation officielle sur les « Différences » de NetSuite résume explicitement ceci :

> - *Le point d'entrée `beforeLoad` dans SuiteScript 2.x inclut un nouveau paramètre : `newRecord`.*
> - *Le point d'entrée `beforeSubmit` dans SuiteScript 2.x inclut deux nouveaux paramètres : `oldRecord` et `newRecord`.*
> - *Le point d'entrée `afterSubmit` dans SuiteScript 2.x inclut deux nouveaux paramètres : `oldRecord` et `newRecord`.*
> - *Le paramètre `type` dans chaque point d'entrée dans SuiteScript 2.x est désormais défini en utilisant l'énumération `context.UserEventType` (Source: docs.oracle.com).*

Cette comparaison montre que les événements utilisateur de SuiteScript 1.0 étaient plus simples dans leur signature et transmettaient moins de contexte au script, tandis que la version 2.x fournit des objets plus riches. Pour plus de clarté, le tableau ci-dessous présente les fonctions de point d'entrée dans SuiteScript 1.0 par rapport à SuiteScript 2.x :

POINT D'ENTRÉE	SIGNATURE SUITESCRIPT 1.0	SIGNATURE SUITESCRIPT 2.X	NOTES
beforeLoad	`function beforeLoad(type, form, request)`	`function beforeLoad(scriptContext)`	1.0 possède `type`, `form` (`nlobjForm`), `request` (`nlobjRequest`) ; 2.x possède `scriptContext.newRecord`, `scriptContext.form`, etc. (Source: so.parthpatel.net) (Source: so.parthpatel.net).
beforeSubmit	`function beforeSubmit(type)`	`function beforeSubmit(scriptContext)`	1.0 ne fournit que `type` ; 2.x fournit `scriptContext.newRecord` et `scriptContext.oldRecord` (Source: docs.oracle.com). Les erreurs dans beforeSubmit empêchent l'enregistrement (Source: www.netsuitediagnostics.com).
afterSubmit	`function afterSubmit(type)`	`function afterSubmit(scriptContext)`	1.0 ne fournit que `type` ; 2.x ajoute `newRecord` & `oldRecord`. AfterSubmit s'exécute après la validation, donc les erreurs n'annulent *pas* l'enregistrement (Source: www.netsuitediagnostics.com) (Source: www.netsuitediagnostics.com).

Chaque User Event Script définit une ou plusieurs de ces fonctions. Oracle souligne que **les User Event Scripts ne se déclenchent que sur les opérations côté serveur**, et non sur le code d'interface utilisateur côté client. En fait, la documentation note que beforeLoad ne se déclenchera pas lors de l'accès à un formulaire en ligne (c'est un scénario de Suitelet) et recommande un script client `pageInit` si vous avez besoin de sourcer des enregistrements standard (Source: docs.oracle.com).

Conditions de déclenchement et opérations prises en charge

Chaque point d'entrée est associé à des activités d'enregistrement spécifiques. Les sources officielles et les guides pour développeurs listent les actions utilisateur qui invoquent chaque événement. En résumé :

- **Before Load** est déclenché lors de *toute opération de lecture*. Par exemple, chaque fois qu'un utilisateur navigue vers un enregistrement dans l'interface utilisateur, ou qu'une API/CSV/service web lit un enregistrement, beforeLoad se déclenche. Comme l'indique un tutoriel, *« L'événement Before Load est déclenché par toute opération de lecture sur un enregistrement. Chaque fois qu'un utilisateur, un script, une importation CSV ou une requête de service web tente de lire un enregistrement de la base de données, l'événement Before Load est déclenché. »* (Source: so.parthpatel.net). Les déclencheurs courants incluent **Create** (chargement d'un nouveau formulaire d'enregistrement), **Edit/View/Load** d'un enregistrement existant, **Copy** d'un enregistrement, **Print**, **Email** et la fenêtre **QuickView**. Le tableau ci-dessous résume les actions d'enregistrement typiques et leurs déclencheurs d'événement utilisateur :

ACTION D'ENREGISTREMENT	BEFORELOAD	BEFORESUBMIT	AFTERSUBMIT	NOTES
Create (Nouvel enregistrement)	✔	✔	✔	Tous les événements se déclenchent lors de la création d'un enregistrement (chargement, puis soumission) (Source: so.parthpatel.net).
Edit (Enregistrement existant)	✔	✔	✔	Tous les événements se déclenchent (chargement pour édition, puis enregistrement).
View/Load (pas d'édition)	✔	—	—	Seul beforeLoad se déclenche (juste pour visualiser l'enregistrement) (Source: so.parthpatel.net).
Copy Record	✔	✔	✔	La copie agit comme une création ; l'événement utilisateur se déclenche en conséquence.
Delete Record	—	✔	✔	Pas de beforeLoad (pas de chargement dans l'interface), mais déclenche les événements de soumission (Source: so.parthpatel.net).
Inline Edit (XEdit)	—	✔	✔	Une édition en ligne de type liste déclenche beforeSubmit/afterSubmit, pas beforeLoad (Source: so.parthpatel.net).
Print/Email/QuickView	✔	—	—	Ce sont des vues en lecture seule, ne déclenchant que beforeLoad (Source: so.parthpatel.net).
Approve/Reject (certains)	—	✔	✔	Les actions d'approbation sur les transactions déclenchent before/afterSubmit, pas beforeLoad (Source: so.parthpatel.net).
Cancel/Pack/Ship	—	✔	✔	Ces actions de transaction déclenchent également à la fois beforeSubmit et afterSubmit (Source: so.parthpatel.net).
Dropship/Special Order	—	—	✔	Types de commande spécifiques : seul afterSubmit se déclenche (Source: so.parthpatel.net).
Mark Complete/Reassign	—	✔	—	Certaines actions de ticket de support ne déclenchent que beforeSubmit (Source: so.parthpatel.net).

Tableau 1 : Résumé des déclencheurs d'événement utilisateur par action d'enregistrement. Chaque « ✔ » indique que l'événement se déclenche lorsque l'action donnée se produit. (Sources : documentation officielle et tutoriels (Source: so.parthpatel.net) (Source: so.parthpatel.net) (Source: so.parthpatel.net).)

Ce tableau est une compilation des déclencheurs listés par NetSuite. Par exemple, le tutoriel *Before and After Submit* confirme que **beforeSubmit** et **afterSubmit** se déclenchent lors de toute écriture en base de données (création, édition, suppression, édition en ligne, etc.) (Source: so.parthpatel.net). Il note également que certaines actions ne déclenchent qu'un seul de ces événements (par exemple, le dropship ne déclenche que afterSubmit (Source: so.parthpatel.net). Inversement, des actions comme **View** ou **Print** sont purement en lecture seule et ne déclenchent que beforeLoad (Source: so.parthpatel.net).

La documentation SuiteScript d'Oracle souligne en outre que *« la plupart des enregistrements standard NetSuite et des types d'enregistrements personnalisés prennent en charge les User Event Scripts »*, avec quelques exceptions (par exemple, les enregistrements d'identité personnelle et certains enregistrements de feuilles de temps/revenus) (Source: docs.oracle.com). En pratique, les développeurs doivent vérifier que leur type d'enregistrement cible prend en charge les événements utilisateur (voir la liste « SuiteScript Supported Records » de NetSuite (Source: docs.oracle.com) avant de se fier aux événements utilisateur pour cet enregistrement.

Flux d'exécution dans l'application NetSuite

Comprendre **quand** le code d'événement utilisateur s'exécute dans le cycle de vie de la requête est crucial. NetSuite fournit des diagrammes et des descriptions de ces flux (Source: docs.oracle.com) (Source: netsuitedocumentation1.gitlab.io) (Source: netsuitedocumentation1.gitlab.io). En résumé, la séquence est :

1. **Flux beforeLoad :** Lorsqu'un enregistrement est chargé (en naviguant dans l'interface utilisateur, en appelant `nlapiLoadRecord`, via un service web, CSV, etc.), le serveur de NetSuite effectue des vérifications d'autorisation, puis déclenche tous les scripts `beforeLoad` enregistrés *avant* de renvoyer l'enregistrement/formulaire au client. À ce stade, l'enregistrement est toujours sur le serveur d'application, et toute modification de l'interface utilisateur (via `form`) se produit maintenant. Oracle note que **beforeLoad se produit avant que les données ne soient renvoyées au client** (Source: netsuitedocumentation1.gitlab.io).

2. **Flux beforeSubmit :** Lorsqu'un utilisateur clique sur « Enregistrer » (ou qu'un script insère/met à jour un enregistrement), le client envoie les données au serveur. Le serveur vérifie les autorisations et **exécute beforeSubmit sur les données soumises avant qu'elles ne soient validées** (Source: docs.oracle.com) (Source: netsuitedocumentation1.gitlab.io). Pendant l'exécution de beforeSubmit, les données de l'enregistrement ne sont *pas encore* enregistrées dans la base de données (Source: netsuitedocumentation1.gitlab.io). Si un script beforeSubmit génère une exception (via `nlapiCreateError` ou similaire), la sauvegarde est annulée et une erreur est affichée à l'utilisateur (Source: www.netsuitediagnostics.com).

3. **Validation (Commit) dans la base de données :** Si beforeSubmit réussit, NetSuite écrit alors les données dans la base de données. À ce stade, l'enregistrement est officiellement sauvegardé.

4. **Flux afterSubmit :** Une fois les données validées, NetSuite renvoie le nouvel enregistrement (sauvegardé) au serveur pour un post-traitement. Tous les scripts `afterSubmit` sont déclenchés à ce stade (Source: docs.oracle.com) (Source: netsuitedocumentation1.gitlab.io). Étant donné que l'enregistrement est déjà sauvegardé, les erreurs dans afterSubmit n'annulent *pas* la sauvegarde (Source: www.netsuitediagnostics.com). C'est généralement là que vous effectuez des actions non critiques comme l'envoi d'e-mails, la création d'enregistrements liés ou l'appel à des systèmes externes (Source: docs.oracle.com) (Source: docs.oracle.com). (Exemple : « Lorsque afterSubmit s'exécute, il récupère les nouvelles données de la base de données pour un traitement supplémentaire. Les actions afterSubmit sur les données sauvegardées peuvent inclure : l'envoi de notifications par e-mail..., la création d'enregistrements enfants... » (Source: docs.oracle.com).)

La rubrique d'aide d'Oracle « Comment les scripts d'événement utilisateur sont exécutés » confirme que ces points d'entrée ne peuvent pas être appelés arbitrairement par d'autres scripts : *« Les scripts d'événement utilisateur ne peuvent pas être déclenchés par d'autres scripts d'événement utilisateur ou des workflows avec un contexte de script d'événement utilisateur — vous ne pouvez donc pas les enchaîner. »* (Source: docs.oracle.com). En d'autres termes, charger ou enregistrer un enregistrement lié à partir d'un événement utilisateur ne déclenchera pas les événements utilisateur de cet autre enregistrement.

Lors de la conception d'événements utilisateur, ces flux d'exécution dictent l'utilisation appropriée : utilisez **beforeLoad** pour les tâches qui doivent se produire avant l'affichage d'un enregistrement, **beforeSubmit** pour la validation ou la modification des données sur le point d'être enregistrées, et **afterSubmit** pour les actions post-sauvegarde. Les bonnes pratiques d'Oracle énoncent explicitement ces limites : « Les opérations qui dépendent de la validation de l'enregistrement soumis dans la base de données doivent se produire dans un script afterSubmit » (Source: docs.oracle.com), tandis que si vous devez modifier des champs ou appliquer des règles *avant* l'enregistrement, utilisez beforeSubmit (Source: docs.oracle.com).

SuiteScript 1.0 vs SuiteScript 2.x (Contexte)

En 2016, NetSuite a introduit SuiteScript 2.0/2.1, qui a remanié de nombreuses API et introduit des modules. Une perspective utile consiste à comparer les événements utilisateur 1.0 et 2.x (bien que la version 2.x dépasse notre cadre principal). Le guide officiel des différences note que SuiteScript 2.x « inclut tous les types de scripts de SuiteScript 1.0 » (Source: docs.oracle.com), mais avec des concepts renommés et des fonctionnalités supplémentaires :

- En 1.0, le terme « types d'événements » était utilisé, tandis que la version 2.x utilise des « points d'entrée » (correspondance un à un). Pour les événements utilisateur, l'argument « type » de la version 1.0 correspond au `scriptContext` de la version 2.x avec une énumération `UserEventType` (Source: docs.oracle.com).
- Les fonctions de point d'entrée 2.x incluent plus de contexte (comme mentionné, newRecord/oldRecord) (Source: docs.oracle.com).
- Les scripts SuiteScript 2.x utilisent des modules de style AMD (`define([...], function(...) {...})`) plutôt que le style de fonction globale de la version 1.0.
- La plateforme sous-jacente traite à la fois la version 1.0 et la version 2.x comme des SuiteScripts côté serveur, mais les meilleures pratiques privilégient désormais la version 2.x pour son architecture et sa maintenabilité améliorées (Source: docs.oracle.com).

Dans la pratique actuelle, Oracle encourage fortement la migration des scripts 1.0 vers la version 2.x. La documentation officielle indique : *« Vous devez vous connecter pour voir [les PDF de SuiteScript 1.0]. Choisissez SuiteScript 2.0 ou 2.1 pour bénéficier des fonctionnalités les plus récentes et entièrement prises en charge. »* (Source: docs.oracle.com) (Source: docs.oracle.com). Ainsi, bien que ce rapport se concentre sur la version 1.0, nous ferons occasionnellement référence aux normes 2.x pour souligner les fonctionnalités manquantes dans la version 1.0 ou les différences de conception.

Discussion détaillée des points d'entrée

Nous examinons ci-dessous chaque point d'entrée 1.0 en détail : son comportement, les paramètres disponibles, les déclencheurs et les cas d'utilisation courants, étayés par des exemples et des références.

beforeLoad

Définition : Le point d'entrée `beforeLoad` s'exécute *juste avant* qu'un enregistrement ne soit présenté à l'utilisateur (ou renvoyé à partir d'une requête de lecture). Selon Oracle : *« Exécute une fonction avant le chargement d'un enregistrement — chaque fois qu'il y a une opération de lecture, et avant que l'enregistrement ou la page ne soit renvoyé »* (Source: docs.oracle.com). Dans SuiteScript 1.0, vous l'implémentez comme suit :

```
function beforeLoad(type, form, request) {
    // Votre code personnalisé ici
}
```

Ici, `type` est une chaîne indiquant l'opération (par exemple « view », « create », « edit », « copy »), `form` est un objet `nlobjForm` représentant le formulaire d'interface utilisateur en cours de rendu, et `request` est un `nlobjRequest` contenant les paramètres de requête HTTP (présents uniquement pour les actions POST/GET du navigateur) (Source: so.parthpatel.net). Par exemple :

```
function beforeLoad(type, form, request) {
    nlapiLogExecution("DEBUG", "Before Load", "Trigger type: " + type);
    // ex: masquer un champ sur le formulaire
    if (form) {
        var fld = form.getField('custpage_my_field');
        if (fld) {
            fld.setDisplayType('hidden');
        }
    }
}
```

Déclencheurs

Un script beforeLoad se déclenche à chaque *lecture* d'un enregistrement. Les déclencheurs concrets incluent la navigation de l'utilisateur vers un enregistrement (création, consultation ou modification), un script appelant `nlapiLoadRecord`, une importation CSV lisant des données, ou une lecture SOAP/REST externe. Les actions typiques qui invoquent beforeLoad sont *Create (new)*, *Edit*, *View/Load*, *Copy*, *Print*, *Email* et *QuickView* (Source: so.parthpatel.net) (Source: riptutorial.com). Par exemple, le guide SuiteScript liste exactement ces actions :

> *Actions d'enregistrement qui déclenchent un événement `beforeLoad` : Create, Edit, View/Load, Copy, Print, Email, QuickView (Source: so.parthpatel.net).*

Le résultat est que dès que NetSuite dispose des données de l'enregistrement et est sur le point de les rendre, il appelle beforeLoad. À ce stade, les champs de l'enregistrement sont remplis avec les valeurs actuelles, mais vous pouvez toujours modifier le formulaire ou les valeurs par défaut pour les nouveaux enregistrements. La documentation d'Oracle note que **beforeLoad ne permet pas de mettre à jour directement l'enregistrement de la base de données** : *« Vous ne pouvez pas mettre à jour un enregistrement chargé dans un script beforeLoad — si vous essayez, cette logique est ignorée. »* (Source: docs.oracle.com). (En termes modernes, beforeLoad est destiné aux modifications de l'interface utilisateur, pas aux écritures de données.)

Les guides d'utilisation soulignent que beforeLoad est idéal pour modifier l'interface utilisateur. Les cas d'utilisation courants incluent le masquage ou l'affichage de champs, l'ajout d'instructions ou la définition de valeurs par défaut sur un nouveau formulaire (Source: riptutorial.com). Par exemple, dans un beforeLoad, on pourrait faire :

```
// Masquer un champ de formulaire avant que l'utilisateur ne le voie
function beforeLoad(type, form, request) {
    form.getField('custbody_sensitive_data').setDisplayType('hidden');
}
```

Ceci est corroboré par le blog des développeurs SuiteRep : *« Before Load : C'est le moment où un enregistrement se charge... idéal pour remplir automatiquement les champs avec des valeurs par défaut lors de la création d'un enregistrement »* (Source: suiterep.com). Dans SuiteScript 2.x, l'équivalent `beforeLoad(context)` fournit `context.form` pour de telles modifications. Dans la version 1.0, le paramètre `form` remplit ce rôle.

Cependant, notez que si **le formulaire lui-même est chargé via des services Web ou CSV**, l'objet `form` peut ne pas être disponible — le paramètre `request` ne sera présent que pour les chargements basés sur le navigateur (Source: so.parthpatel.net).

Limitations dans beforeLoad

Parce que beforeLoad s'exécute *avant* que les données ne soient affichées, il ne peut pas enregistrer directement les modifications apportées à l'enregistrement. Comme l'indiquent les notes d'Oracle, *« Vous ne pouvez pas mettre à jour un enregistrement chargé dans un script beforeLoad — si vous essayez, cette logique est ignorée »* (Source: docs.oracle.com). Au lieu de cela, si vous souhaitez définir ou modifier des valeurs qui seront enregistrées, utilisez beforeSubmit. De plus, certaines opérations ne déclenchent pas beforeLoad : par exemple, les actions *Approuver/Rejeter* sur les enregistrements de transaction ne provoquent pas de beforeLoad, car elles ne sont pas considérées comme une action de « chargement » (Source: www.netsuitediagnostics.com). (Le blog note que « les types d'événements utilisateur comme `approve`, `cancel`, `xedit` ne sont jamais chargés dans un sens conventionnel et ne déclenchent donc pas le point d'entrée `beforeLoad` » (Source: www.netsuitediagnostics.com).)

Autre limitation : les scripts beforeLoad ne se déclenchent pas lors de l'ajout de champs via **Formulaires > Personnalisation** *par programmation* (contrairement à un Suitelet). Ils ne s'exécutent que lors des chargements réels d'enregistrements. De plus, le formulaire d'interface utilisateur (`nlobjForm`) fourni est en *lecture seule* pour les données d'enregistrement (il représente la mise en page du formulaire, pas les valeurs réelles de l'enregistrement). Vous pouvez manipuler l'affichage des champs, mais pas forcer un changement de valeur de champ ici — le faire n'a aucun effet sur la sauvegarde finale. Pour modifier les données, utilisez beforeSubmit.

Utilisations typiques

- **Personnalisation de formulaire :** Ajout ou suppression de champs/options, modification des types d'affichage des champs, insertion de texte d'aide ou de références de script client (l'objet formulaire peut définir un clientScript).
- **Valeurs par défaut :** Lors des événements *Create* (`type === 'create'`), on définit souvent des valeurs de champ par défaut avant l'affichage du formulaire. (Remarque : certaines valeurs par défaut peuvent également être définies avec la valeur par défaut du champ, mais beforeLoad permet des valeurs par défaut dynamiques.)
- **Contrôle de sécurité ou de contexte :** Masquer des champs ou des sous-listes entières pour certains rôles ou dans certains statuts, car beforeLoad peut vérifier `nlapiGetContext().getRole()` et appeler `form.getField(...).setDisplayType('hidden')`.
- **Prétraitement des données :** Très occasionnellement, on peut extraire des données via `type` et `request` pour filtrer ou augmenter le formulaire.

Développement et débogage : Un développeur peut journaliser le `type` dans beforeLoad pour identifier les déclencheurs. La journalisation est utile car un chargement d'enregistrement donné peut se produire dans différents contextes (par exemple « view » vs « edit »), et le code se ramifie souvent sur le `type`. La chaîne `type` définie pour les actions standard peut inclure des valeurs comme « create », « edit », « view », « approve », etc. (SuiteScript 2.x a introduit `context.UserEventType`, que la version 1.0 n'a pas, donc la version 1.0 utilise des chaînes brutes.)

beforeSubmit

Définition : Le point d'entrée `beforeSubmit` s'exécute *immédiatement avant* que les données de l'enregistrement ne soient validées dans la base de données. La documentation d'Oracle explique : beforeSubmit « exécute la fonction juste *avant* qu'un enregistrement ne soit soumis » (Source: www.netsuitediagnostics.com). Contrairement à beforeLoad, il n'a pas de paramètres `form` ou `request` — il ne reçoit que `type`. Dans SuiteScript 1.0, votre code ressemble à ceci :

```
function beforeSubmit(type) {
    // Validation personnalisée ou manipulation de champ ici
}
```

Ici, `type` (une chaîne comme « create » ou « edit ») vous indique quelle action se produit. Par exemple, nous voyons un beforeSubmit simple journalisant le `type` :

```
function beforeSubmit(type) {
    nlapiLogExecution("DEBUG", "Before Submit", "action=" + type);
}
```

(Source: so.parthpatel.net).

Déclencheurs

Un script beforeSubmit se déclenche sur **toute opération d'écriture** : création, modification, suppression, modification en ligne (xedit), approbation, rejet, annulation, etc., essentiellement chaque fois que NetSuite enregistre un enregistrement (Source: so.parthpatel.net). Plus précisément, le tutoriel parthpatel déclare : *« Ces deux événements (beforeSubmit et afterSubmit) sont déclenchés par toute opération d'écriture de base de données sur un enregistrement. Chaque fois qu'un utilisateur, un script, une importation CSV ou une requête de service Web tente d'écrire un enregistrement dans la base de données, les événements Submit sont déclenchés. »* (Source: so.parthpatel.net). Cela inclut les sauvegardes d'interface utilisateur, les scripts planifiés appelant `nlapiSubmitRecord`, les chargements de données CSV, etc.

La même source liste les actions d'enregistrement déclenchant à la fois beforeSubmit et afterSubmit : Create, Edit, Delete, XEdit, Approve, Reject, Cancel, Pack, Ship (Source: so.parthpatel.net). (Certaines actions n'en déclenchent qu'un seul, par exemple *Dropship* ne déclenche que afterSubmit ; *Mark Complete* ne déclenche que beforeSubmit.) Surtout, l'entrée beforeSubmit s'exécute *avant* que NetSuite n'enregistre les données. Le diagramme de flux d'exécution d'Oracle indique que **pendant beforeSubmit, « les données soumises n'ont PAS encore été validées dans la base de données. »** (Source: netsuitedocumentation1.gitlab.io).

Comportement et capacités

Dans beforeSubmit, l'enregistrement actuel (avec toutes les modifications) est disponible via `nlapiGetNewRecord()` dans SuiteScript 1.0, et on peut effectuer l'accès aux champs et les modifications. Si un script beforeSubmit génère une erreur, il empêche la sauvegarde. Comme le note le blog netsuitediagnostics : *« Ce script est exécuté avant que l'enregistrement ne soit soumis. Dès que l'utilisateur appuie sur le bouton enregistrer, le script s'exécute. Si le script génère une erreur, l'enregistrement n'est pas sauvegardé. »* (Source: www.netsuitediagnostics.com). Cela signifie que beforeSubmit est l'endroit idéal pour les **validations** et le **nettoyage des données** qui doivent bloquer les entrées incorrectes.

Les tâches courantes de beforeSubmit incluent :

- **Validation des données :** Vérification des valeurs de champ pour les règles métier, et génération de `nlapiCreateError` pour bloquer les données non valides. (Par exemple, si un champ obligatoire est manquant ou si un champ numérique est hors plage, empêcher la sauvegarde.)

- **Mises à jour de champs :** Ajustement des champs en fonction de calculs. Par exemple, mettre à jour les totaux, convertir les unités ou définir des champs personnalisés internes avant la sauvegarde.
- **Stockage de valeurs intermédiaires :** Si l'on doit transférer des données de beforeLoad/UI vers afterSubmit, on peut écrire dans des champs masqués ici afin qu'ils soient enregistrés.
- **Vérifications des permissions/du contexte :** Possibilité de révoquer des formulaires ou de générer des erreurs si un rôle spécifique n'est pas autorisé à enregistrer sous certaines conditions.

Exemple : Supposons que vous souhaitiez garantir que la remise sur une transaction ne dépasse pas une certaine limite. Un `beforeSubmit` pourrait comparer `nlapiGetNewRecord().getFieldValue('discount')` à un seuil et générer une erreur si elle est trop élevée, empêchant ainsi l'enregistrement de la fiche.

Limitations

N'oubliez pas que dans le `beforeSubmit` 1.0, vous ne disposez **pas** encore de l'ID interne de l'enregistrement (il peut ne pas exister pour un nouvel enregistrement). De plus, aucun `oldRecord` n'est fourni par défaut. (Dans SuiteScript 2.0, `context.oldRecord` donnerait les valeurs précédentes ; en 1.0, vous devriez effectuer manuellement un `nlapiLoadRecord` si nécessaire, ce qui est coûteux.) Il est donc difficile de comparer les changements sans interroger la base de données.

Toutes les modifications effectuées dans `beforeSubmit` seront enregistrées lors de la validation (commit) de la base de données, tant qu'aucune erreur ne survient. Cependant, certaines modifications doivent être effectuées avec prudence : par exemple, ajuster les lignes de transaction dans `beforeSubmit` peut entraîner des incohérences si cela n'est pas géré correctement (Oracle avertit spécifiquement que si vous modifiez des lignes, vous devez également ajuster les totaux en conséquence (Source: docs.oracle.com).

Exemple et conseils

D'après Netsuitediagnostics : « En raison de la conception du point d'entrée `beforeSubmit`, l'utilisateur perdra toutes les modifications qu'il a effectuées en cas d'erreur. » (Source: www.netsuitediagnostics.com). Cela souligne que générer une erreur dans `beforeSubmit` annulera les modifications pour revenir à l'état précédent l'édition ; assurez-vous donc de ne générer des erreurs que pour des conditions réellement invalides.

Le guide des meilleures pratiques d'Oracle conseille d'utiliser le paramètre `type` pour différencier la logique : *« Utilisez l'argument `type`, l'objet `context` et l'énumération `context.UserEventType` pour définir et limiter la portée de votre logique d'événement utilisateur. »* (Source: docs.oracle.com). En 1.0, vérifiez simplement la chaîne `type`. Par exemple :

```
function beforeSubmit(type) {
    if (type === 'create') {
        // effectuer une action uniquement lors de la création d'un nouvel enregistrement
    }
}
```

Enfin, notez que comme `beforeSubmit` s'exécute juste avant l'écriture dans la base de données, toute opération coûteuse à ce stade ralentira l'enregistrement pour l'utilisateur. Si certains traitements peuvent être différés, il est préférable de les effectuer dans `afterSubmit` ou via un script planifié.

afterSubmit

Définition : Le point d'entrée `afterSubmit` s'exécute *immédiatement après* que NetSuite a enregistré l'enregistrement. Sa signature est `function afterSubmit(type)`. Oracle le décrit comme *« exécutant la fonction juste après la soumission d'un enregistrement »* (Source: docs.oracle.com). Comme `beforeSubmit`, il ne reçoit que `type` en 1.0. La différence clé est que **les données de l'enregistrement sont déjà dans la base de données** lorsque `afterSubmit` s'exécute. Un exemple de gestionnaire 1.0 serait :

```
function afterSubmit(type) {
    nlapiLogExecution("DEBUG", "After Submit", "action=" + type);
    // ex: envoyer un e-mail de confirmation
}
```

(Source: so.parthpatel.net).

Déclencheurs

`afterSubmit` se déclenche lors de tout enregistrement validé, c'est-à-dire les mêmes actions qui ont déclenché `beforeSubmit` (Création, Édition, Suppression, etc.) (Source: so.parthpatel.net). Il ne se déclenchera *pas* dans les cas où `beforeSubmit` a généré une erreur (c'est-à-dire lors d'enregistrements annulés) (Source: www.netsuitediagnostics.com). Comme l'explique [34], *« si une erreur est levée avant que l'enregistrement ne soit sauvegardé, NetSuite ne déclenchera pas le point d'entrée `afterSubmit`. »* (Source: www.netsuitediagnostics.com). Il ne se déclenche également **pas** lors de simples chargements ou consultations d'enregistrements ; uniquement après une écriture réussie.

Certaines actions sur les enregistrements ne provoquent que `afterSubmit` (pas `beforeSubmit`). Par exemple, le *Dropship* (livraison directe) et la *Commande spéciale* sur les commandes de vente ne déclenchent que `afterSubmit` (comme le note le document parthpatel) (Source: so.parthpatel.net). Il s'agit d'une particularité des flux de travail intégrés de NetSuite pour ces types de commandes : ils effectuent certaines actions uniquement une fois la commande validée.

Capacités et cas d'utilisation

`afterSubmit` est idéal pour le *traitement post-enregistrement*. Puisque l'enregistrement est sauvegardé, il possède désormais un ID interne et est visible par d'autres scripts et flux de travail. Les tâches courantes dans `afterSubmit` incluent :

- **E-mail/Notification :** Envoi d'e-mails ou de notifications concernant l'enregistrement créé ou mis à jour. Oracle liste explicitement « l'envoi de notifications par e-mail, la création d'enregistrements liés ou la synchronisation avec un système externe » comme des actions à effectuer dans `afterSubmit` (Source: docs.oracle.com) (Source: docs.oracle.com).
- **Création d'enregistrements liés :** Par exemple, créer automatiquement des enregistrements enfants, des tâches ou des écritures de journal en réponse à cet enregistrement.
- **Appels d'intégration :** Transférer des données vers des systèmes externes via des appels de services Web ou REST, où vous souhaitez souvent que l'enregistrement soit entièrement validé au préalable.
- **Mises à jour finales de champs :** Si certains champs n'ont de sens qu'après que l'enregistrement a obtenu un ID (par exemple, utiliser le numéro de l'enregistrement ou un horodatage dans un autre processus).

Le diagramme « comment ça marche » d'Oracle note qu'à l'étape 3, `afterSubmit` reçoit les données fraîchement validées (Source: docs.oracle.com). Cela est cohérent avec la documentation de SuiteScript 2.x qui recommande `afterSubmit` pour « tout ce que vous voulez voir se produire après une opération d'écriture », incluant explicitement l'e-mail et la création d'enregistrements liés (Source: docs.oracle.com).

Comportement en cas d'erreur

Comme la validation a déjà eu lieu, les erreurs générées dans `afterSubmit` ne provoquent **pas** l'annulation de l'enregistrement. Netsuitediagnostics précise : *« Si le script lève une erreur, contrairement au point d'entrée `beforeSubmit`, l'enregistrement est sauvegardé. »* (Source: www.netsuitediagnostics.com). En pratique, on intercepte et journalise souvent les erreurs dans `afterSubmit` plutôt que de les lever, à moins de vouloir bloquer un processus ultérieur. (Des erreurs drastiques ici pourraient prêter à confusion puisque l'enregistrement est déjà persistant.)

Exemple et meilleures pratiques

Les meilleures pratiques d'Oracle stipulent : *« Effectuez toutes les opérations de post-traitement de l'enregistrement actuel dans un événement `afterSubmit`. »* (Source: docs.oracle.com). De plus, si vous devez transférer une valeur de `beforeLoad` vers `afterSubmit`, Oracle suggère d'utiliser un champ masqué comme intermédiaire (définissez-le dans `beforeLoad`, puis lisez-le dans `afterSubmit`) (Source: docs.oracle.com).

Netsuitediagnostics résume `afterSubmit` ainsi : « *`afterSubmit` est un bon endroit pour effectuer une logique après que l'enregistrement a été sauvegardé.* » (Source: www.netsuitediagnostics.com). Par exemple, lors de la création d'une facture, vous pourriez vouloir envoyer une notification par e-mail ou mettre à jour un enregistrement CRM lié uniquement après que la facture est entièrement dans NetSuite. Dans `afterSubmit` 1.0, vous pouvez utiliser `nlapiSendEmail`, `nlapiSubmitRecord` pour d'autres types, etc. Cependant, gardez à l'esprit la note de performance d'Oracle : si un script `afterSubmit` effectue un travail lourd (boucles ou appels externes), il retarde toujours l'achèvement de l'action d'enregistrement de l'utilisateur. Pour les post-traitements très lourds, il est préférable de déléguer à un script planifié ou à un travail Map/Reduce (un concept 2.x).

Un point important connexe : **`afterSubmit` asynchrone**. NetSuite autorise un événement utilisateur `afterSubmit` *asynchrone* (c'est-à-dire mis en file d'attente plutôt qu'en ligne), mais uniquement dans le contexte de paiement Webstore (Source: docs.oracle.com). Il s'agit d'un cas relativement rare pour les sites SuiteCommerce ; lors d'un enregistrement normal via l'interface utilisateur, `afterSubmit` est synchrone.

Paramètres et accès à l'API

Dans SuiteScript 1.0, l'API au sein d'un script d'événement utilisateur inclut :

- **nlapiGetNewRecord()/nlapiGetOldRecord() :** En 1.0, `nlapiGetOldRecord()` n'est pas disponible. Pour comparer l'ancien et le nouveau, vous devriez recharger les anciennes données via `nlapiLoadRecord` si nécessaire. (La version 2.x transmet automatiquement l'objet `oldRecord` (Source: docs.oracle.com).) En 1.0 pur, le script utilise généralement `nlapiGetNewRecord()` pour référencer l'enregistrement en cours de sauvegarde.
- **nlapiGetContext() :** Le contexte (informations utilisateur, rôle, contexte d'exécution) est accessible. Une bonne pratique consiste à vérifier `nlapiGetContext().getExecutionContext()` pour filtrer l'invocation du script (par exemple, ne l'exécuter que pour l'interface utilisateur et non pour les fichiers CSV) (Source: docs.oracle.com).
- **Formulaire & Requête (beforeLoad uniquement) :** Dans `beforeLoad`, vous obtenez `form` (objet UI) et `request` (requête serveur). Ces API sont respectivement `nlobjForm` et `nlobjRequest` en 1.0 (Source: so.parthpatel.net).
- **nlapiLogExecution() :** La journalisation s'effectue via `nlapiLogExecution('DEBUG'|'ERROR', titre, détails)`. Les meilleures pratiques suggèrent de journaliser les informations clés, surtout dans `beforeSubmit` / `afterSubmit` pour le dépannage. (Les exemples ci-dessus utilisent `nlapiLogExecution` pour démonstration (Source: so.parthpatel.net) (Source: so.parthpatel.net).)
- *Autres API nlapi :** Un script d'événement utilisateur peut effectuer n'importe quelle opération NetSuite : recherche d'enregistrements, recherche, appel d'API externe (avec XMLHttpRequest), etc. Un modèle courant consiste à créer des enregistrements liés dans `afterSubmit` via `nlapiCreateRecord()` / `nlapiSubmitRecord()`, ou à définir des champs dans `beforeSubmit` (`newRec.setFieldValue()`).

Référez-vous au guide officiel de l'API SuiteScript 1.0 (désormais uniquement en PDF (Source: docs.oracle.com) ou aux références complètes de la version 1.0 pour les spécificités de chaque fonction `nlapi`.

Dépendances et ordre des scripts d'événement utilisateur

Lorsque plusieurs scripts d'événement utilisateur sont déployés sur le même type d'enregistrement, NetSuite les exécute dans un ordre défini. Oracle note que **vous pouvez définir l'ordre dans la fiche de déploiement du script** (Source: docs.oracle.com). Par exemple, si vous avez deux scripts `beforeSubmit` sur une commande de vente, vous pouvez choisir lequel s'exécute en premier. Les scripts s'exécutent de manière synchrone dans cette séquence configurée.

Meilleure pratique : Évitez de déployer trop de scripts d'événement utilisateur sur un seul enregistrement. Comme Oracle avertit explicitement, « s'il y a dix scripts `beforeLoad` qui doivent se terminer avant que l'enregistrement ne se charge, le temps nécessaire au chargement de l'enregistrement peut augmenter de manière significative » (Source: docs.oracle.com). Dans des cas extrêmes, cela peut dégrader l'expérience utilisateur. Il est préférable de consolider la logique dans moins de scripts ou d'utiliser `afterSubmit`/scripts planifiés pour les tâches non urgentes. Les directives de performance suggèrent de maintenir chaque événement utilisateur en dessous de 5 secondes (Source: docs.oracle.com). Utilisez la SuiteApp de NetSuite pour la gestion de la performance des applications (APM) afin de tester les temps d'exécution (Source: docs.oracle.com).

Meilleures pratiques

Oracle et les sources communautaires proposent plusieurs directives de meilleures pratiques pour les scripts d'événement utilisateur :

- **Évitez le chaînage :** Comme mentionné, n'essayez pas d'appeler un événement utilisateur depuis un autre. Factorisez plutôt le code partagé dans des modules de bibliothèque (Source: docs.oracle.com).
- **Portée par type :** Vérifiez toujours l'argument `type` et limitez la logique aux cas pertinents (Source: docs.oracle.com). Cela évite un comportement involontaire lors des modifications par rapport aux créations.
- **Filtrage de sécurité/contexte :** Utilisez `nlapiGetContext().getExecutionContext()` ou `nlapiGetContext().getUser()` pour ignorer la logique si elle n'est pas nécessaire. Par exemple, vous pourriez vouloir que les événements utilisateur ne s'exécutent que dans l'interface utilisateur et non via les services Web, ou vice versa (Source: docs.oracle.com).
- **Écritures en base de données :** Modifiez uniquement les champs de l'enregistrement dans **`beforeSubmit`** (pas `beforeLoad`) via `nlapiSetFieldValue`, à moins d'avoir stocké des valeurs intermédiaires dans un champ masqué. Tous les effets post-enregistrement (notifications, mises à jour externes) vont dans **`afterSubmit`** (Source: docs.oracle.com) (Source: docs.oracle.com).
- **Performance :** Testez et optimisez les scripts. Les calculs lourds dans les événements utilisateur ralentissent le traitement des enregistrements pour chaque utilisateur qui enregistre/charge cet enregistrement. Dans la mesure du possible, déléguez à des processus asynchrones.
- **Gestion des erreurs :** Dans `beforeSubmit`, générer une erreur annule l'enregistrement – à utiliser avec prudence. Dans `afterSubmit`, journalisez les erreurs plutôt que de les générer pour permettre à l'enregistrement de l'utilisateur de se terminer.
- **Journalisation et débogage :** Utilisez `nlapiLogExecution` généreusement pour le débogage. SuiteScript 1.0 prend en charge l'instruction JavaScript `debugger;` dans certains cas, mais pour les scripts serveur, il est préférable de déployer dans un compte hors production et de vérifier les journaux d'exécution des scripts. Les meilleures pratiques officielles suggèrent d'utiliser le débogueur SuiteScript (dans l'interface utilisateur ou via IDE) pour déboguer les scripts serveur (Source: docs.oracle.com), bien que les détails pour la version 1.0 soient limités.
- **Utilisez des champs masqués si nécessaire :** Si vous avez besoin que des données persistent de `beforeLoad` à `afterSubmit` dans un seul script, stockez-les dans un champ masqué temporaire comme le suggère Oracle (Source: docs.oracle.com).

Analyse des données et considérations sur la performance

Comme les scripts d'événement utilisateur s'exécutent fréquemment, leur impact sur la performance peut être significatif à grande échelle. Oracle fournit des directives impliquant que l'exécution doit être maintenue en dessous de 5 secondes (Source: docs.oracle.com) et encourage l'utilisation d'un outil APM pour profiler les scripts. En pratique, cela signifie :

- Minimiser les appels coûteux dans `beforeLoad` / `Submit` : évitez d'exécuter des recherches complètes ou des intégrations si possible sur le chemin critique. Utilisez la mise en cache ou des vérifications légères.
- Opérations en masse : Si `afterSubmit` doit traiter des sous-enregistrements liés, considérez si un processus **Map/Reduce** (SuiteScript 2.x) ou **Suitelet** pourrait être plus efficace que de le faire en ligne.
- Journalisation de données volumineuses : Utilisez les journaux de débogage avec discernement ; une journalisation excessive peut ralentir l'exécution (et inondera les journaux, rendant le diagnostic plus difficile).

Il n'existe pas de statistiques publiées sur les temps d'exécution moyens, mais nous pouvons raisonner : si une validation de base de données typique prend ~500ms, et qu'un événement utilisateur ajoute 200ms supplémentaires, cela peut être acceptable. Mais si plusieurs événements utilisateur ajoutent chacun ~1 seconde, le temps d'attente de l'utilisateur augmente. Les métriques APM (telles que recommandées (Source: docs.oracle.com) peuvent fournir des temps réels par action. Par exemple, les journaux de script de NetSuite rapporteront l'utilisation totale (les unités de gouvernance) mais pas le temps réel ; un APM tiers pourrait mesurer le temps réel en sandbox.

Un conseil de performance intéressant : si 10 événements utilisateur effectuent chacun 0,5s de travail, cela représente 5s de temps de chargement supplémentaire, ce qui est significatif. Comme [23] avertit, « soyez conscient du nombre de scripts d'événement utilisateur utilisés... le temps nécessaire [pour charger] peut augmenter de manière significative » (Source: docs.oracle.com). Il est donc sage de consolider ou de différer le travail.

Lors de la conception de grandes personnalisations, une approche consiste à déplacer la logique non essentielle hors des événements utilisateur vers des scripts **planifiés** ou **Map/Reduce** qui peuvent s'exécuter de manière asynchrone. En fait, les meilleures pratiques d'Oracle suggèrent d'utiliser un script planifié pour « nettoyer » ou poursuivre le traitement après l'échec d'un événement utilisateur ou pour effectuer le travail lourd (Source: docs.oracle.com). Pour une logique critique, une sauvegarde planifiée garantit la cohérence des données même si un événement utilisateur est ignoré ou génère une erreur.

Études de cas et exemples concrets

Bien que les études de cas formelles sur SuiteScript soient rares, de nombreuses organisations partagent des exemples d'utilisation des événements utilisateur. Nous décrivons ici quelques scénarios hypothétiques et réels illustrant les concepts :

- **Exemple 1 : Valeurs par défaut et masquage de champs sur une facture (beforeLoad).** Une entreprise souhaite masquer le champ « Commercial » (Sales Rep) sur un formulaire de facture pour les clients de certaines régions, et définir par défaut la « Date d'échéance » (Due Date) à la fin du mois en cours lors de la création d'une facture. Ils déploient un script `beforeLoad` sur l'enregistrement de facture. Le script vérifie `nlapiGetContext().getRole()` et le `type` d'action. Si `type == 'create'` et que le client se trouve dans la région X (via `newRecord.getFieldValue('entity')`, peut-être après le chargement), il appelle `form.getField('salesrep').setDisplayType('hidden')`. Il définit également `newRecord.setFieldValue('duedate', monthEnd)` avant l'affichage du formulaire. Ainsi, lorsque l'utilisateur ouvre le formulaire de nouvelle facture, il voit la date d'échéance par défaut et aucun champ de commercial. (Sources : [19 † L39-L43] pour une utilisation typique, [56 † L17-L24] pour le contexte.) Cette logique appartient clairement à `beforeLoad`, car elle affecte l'interface utilisateur et les valeurs par défaut au chargement du formulaire.

- **Exemple 2 : Application d'une validation sur les bons de commande (beforeSubmit).** Une entreprise a pour règle qu'aucun bon de commande ne peut dépasser un total de 50 000 $. Ils écrivent un script `beforeSubmit` sur le bon de commande. Dans le script, ils utilisent `var total = parseFloat(nlapiGetNewRecord().getFieldValue('total'); if (total > 50000) { nlapiCreateError('PO_OVR_50K','Total exceeds $50K',true); }`. Comme cette erreur est générée dans `beforeSubmit` (et que l'indicateur `isUncaught` est vrai), NetSuite interrompt l'enregistrement et affiche le message d'erreur à l'utilisateur. Cela empêche l'enregistrement de tout bon de commande supérieur à 50 000 $. Le développeur se réfère à [30], s'assurant de générer l'erreur correctement pour arrêter la validation. Ils s'assurent que le code ne s'exécute que lors de la création ou de la modification, et enregistrent peut-être des entrées pour l'audit. Il s'agit d'un cas d'utilisation classique de `beforeSubmit` : une validation de données qui bloque une sauvegarde.

- **Exemple 3 : Envoi de notifications d'expédition (afterSubmit).** Une entreprise de vente au détail souhaite envoyer un e-mail au service expédition chaque fois qu'une commande client est enregistrée avec le statut « Expédié ». Ils utilisent `afterSubmit` sur l'enregistrement de commande client. Le script vérifie si `type == 'edit'` et si le champ de statut est passé à « Expédié » (en chargeant `nlapiGetNewRecord().getFieldValue('status')`). Si c'est le cas, il récupère certains détails de la commande et appelle `nlapiSendEmail(from, to, subject, body, ...)`. Comme cela se produit dans `afterSubmit`, l'e-mail est envoyé une fois que l'enregistrement est en sécurité dans la base de données, garantissant que tous les champs sont définis. Ils peuvent également créer un enregistrement « Tâche d'entrepôt » associé via `nlapiCreateRecord('customrecord_task')`. Cela correspond à la suggestion d'Oracle selon laquelle `afterSubmit` est utilisé pour les notifications et la création d'enregistrements enfants (Source: docs.oracle.com). Si l'e-mail échoue, ils le capturent et le journalisent, mais la commande reste enregistrée.

- **Exemple réel : Traitement des commandes en livraison directe (Drop Ship).** Dans le flux de travail de livraison directe de NetSuite, les commandes peuvent passer par un événement utilisateur `afterSubmit`. Comme noté par le blog d'experts de Prolecto, lorsqu'une ligne de commande client est marquée avec ces types spéciaux, seul `afterSubmit` se déclenchera (en raison de la manière dont les transitions NetSuite se produisent) (Source: so.parthpatel.net). Un développeur pourrait écrire un script `afterSubmit` pour gérer la logique de livraison directe : par exemple, créer un bon de commande fournisseur après la validation de la commande client. Il s'agit d'un modèle courant où `beforeSubmit` ne détecterait pas l'indicateur de commande spéciale, d'où la nécessité de `afterSubmit`. (Bien que le blog de Prolecto dépasse la version 1.0, le concept d'actions comme la livraison directe ne se déclenchant qu'après la soumission est capturé dans le guide parthpatel (Source: so.parthpatel.net).)

Ces exemples illustrent comment les points d'entrée des événements utilisateur sont utilisés dans les personnalisations réelles. Les modèles s'alignent sur la documentation et les meilleures pratiques : ajustements de l'interface utilisateur dans `beforeLoad`, validations dans `beforeSubmit`, et processus externes/enfants dans `afterSubmit`.

Discussion et orientations futures

Les points d'entrée des événements utilisateur de SuiteScript 1.0 ont été fondamentaux pour personnaliser NetSuite depuis leur introduction. Au fil du temps, NetSuite a fait évoluer la plateforme : SuiteScript 2.0/2.1 offrent des améliorations modulaires, une meilleure gestion des erreurs, un traitement asynchrone (Map/Reduce) et une prise en charge par les IDE. Néanmoins, de nombreuses organisations conservent du code SuiteScript 1.0. L'**implication** est claire : bien que les scripts 1.0 fonctionnent aujourd'hui, les futures fonctionnalités de NetSuite (comme les nouveaux types d'enregistrements, les intégrations ou même le retrait d'anciennes API) pourraient ne pas prendre en charge la version 1.0 indéfiniment. Les conseils d'Oracle recommandent de migrer vers la version 2.x pour une « fonctionnalité entièrement prise en charge » (Source: docs.oracle.com).

En pratique, cela signifie que les développeurs doivent suffisamment comprendre la version 1.0 pour la réécrire. Les concepts de `beforeLoad`, `beforeSubmit` et `afterSubmit` restent les mêmes en 2.x ; seule la syntaxe d'appel de l'API change. Par conséquent, les idées données ici sur le moment et la raison d'utiliser chaque point d'entrée restent directement applicables. En fait, les blogs de développeurs Oracle encouragent le passage

au SuiteScript 2.1 typé et même au TypeScript pour les événements utilisateur (Source: blogs.oracle.com), signalant l'avenir : un code plus robuste avec des outils modernes.

En ce qui concerne l'écosystème SuiteScript, les orientations futures incluent :

- **Gouvernance renforcée et sécurité des types :** SuiteScript 2.1 prend en charge TypeScript, permettant une analyse statique des événements utilisateur. Cela réduira les erreurs d'exécution dans `afterSubmit`, par exemple.
- **Flux de travail asynchrones :** Map/Reduce et `afterSubmit` asynchrone (en dehors de la boutique en ligne) restent des domaines où la version 1.0 est limitée. Il est probable que les traitements lourds continueront de basculer vers des frameworks asynchrones.
- **Amélioration des outils :** L'intégration de SuiteCloud IDE et Git permet un meilleur contrôle de version des scripts d'événements utilisateur, ce qui devrait améliorer la qualité du code par rapport à l'utilisation ponctuelle de l'éditeur NetSuite.
- **Obsolescence de la version 1.0 :** Au fil du temps, Oracle pourrait supprimer complètement la prise en charge des scripts de point d'entrée 1.0. Cela rend une documentation comme celle-ci vitale pour les systèmes hérités jusqu'à ce que la migration ait lieu.

D'un point de vue stratégique, les équipes doivent inventorier tous les événements utilisateur SuiteScript 1.0 (via les outils SuiteCloud), mesurer leurs performances et leur impact commercial, et planifier leur chemin de migration. Les tableaux fournis ici (en particulier les différences entre les paramètres 1.0 et 2.x) seront une référence utile dans cet effort.

Même si NetSuite ajoute davantage de fonctionnalités d'automatisation (SuiteFlow, SuiteAnalytics, etc.), les scripts d'événements utilisateur restent uniques pour l'exécution inconditionnelle de code côté serveur sur les événements d'enregistrement. Les futures améliorations pourraient intégrer des capacités avancées (par exemple, le scoring par apprentissage automatique dans `beforeSubmit`, ou des modèles d'e-mail simplifiés dans `afterSubmit`), mais le modèle de base *beforeLoad* / *beforeSubmit* / *afterSubmit* persistera conceptuellement.

Conclusion

Les points d'entrée des événements utilisateur de SuiteScript 1.0 — `beforeLoad`, `beforeSubmit` et `afterSubmit` — constituent la clé de voûte de la personnalisation côté serveur de NetSuite. Ils permettent aux développeurs de se connecter aux opérations de lecture et d'écriture des enregistrements, permettant la modification dynamique des formulaires, la validation des données et les actions après enregistrement. Ce rapport a rassemblé une référence complète sur ces points d'entrée : définition de leurs paramètres et déclencheurs, illustration de l'utilisation avec des exemples, tabulation du comportement selon les actions et notes comparatives avec SuiteScript 2.x. Nous nous sommes appuyés sur une multitude de sources faisant autorité : les pages d'aide officielles d'Oracle (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com), des guides de bonnes pratiques (Source: docs.oracle.com) (Source: docs.oracle.com), et l'expertise de la communauté (Source: so.parthpatel.net) (Source: suiterep.com).

Les points clés à retenir incluent :

- **beforeLoad** se déclenche lors des lectures d'enregistrements et est principalement utilisé pour les ajustements de l'interface utilisateur/formulaire (Source: so.parthpatel.net) (Source: riptutorial.com).
- **beforeSubmit** se déclenche lors de l'enregistrement (avant la validation) et est utilisé pour la validation et les modifications de données (Source: www.netsuitediagnostics.com) (Source: docs.oracle.com).
- **afterSubmit** se déclenche après la validation et est utilisé pour les notifications et la création d'enregistrements associés (Source: www.netsuitediagnostics.com) (Source: docs.oracle.com).
- Il faut être conscient de la façon dont les déclencheurs s'alignent avec les actions de l'utilisateur (voir le Tableau 1 ci-dessus) et du fait que les événements utilisateur ne s'enchaînent pas (Source: docs.oracle.com).
- SuiteScript 1.0 est un système hérité qui manque de certaines commodités modernes (pas d'objets d'enregistrement ancien/nouveau dans les paramètres), mais le modèle d'utilisation reste similaire dans SuiteScript 2.x (Source: docs.oracle.com) (Source: docs.oracle.com).
- Les considérations de performance et les meilleures pratiques (éviter trop de scripts, utiliser des filtres de contexte d'exécution, maintenir l'exécution en dessous de 5s) sont cruciales pour des solutions évolutives (Source: docs.oracle.com) (Source: docs.oracle.com).
- Les développeurs doivent consulter la référence officielle de l'API 1.0 ou ses équivalents (conservés au format PDF par NetSuite (Source: docs.oracle.com) et effectuer des tests approfondis.

En résumé, une compréhension approfondie des points d'entrée des événements utilisateur de SuiteScript 1.0 est vitale pour toute équipe technique NetSuite gérant des personnalisations héritées ou effectuant une transition vers des API plus récentes. Ce rapport vise à servir de référence technique approfondie, étayée par la documentation officielle et des sources expertes, à cette fin.

Références : Toutes les déclarations factuelles ci-dessus sont étayées par la documentation NetSuite et des sources de développeurs crédibles. (Les citations sont fournies en ligne.) Les tableaux 1 et 2 résument les déclencheurs et les différences de signature entre les versions de SuiteScript. Ce rapport devrait permettre aux développeurs et aux architectes de concevoir, dépanner et optimiser les scripts d'événements utilisateur en toute confiance.

Étiquettes: suitescript-10, scripts-user-event, beforeload, beforesubmit, aftersubmit, developpement-netsuite, scripting-cote-serveur, cycle-de-vie-enregistrement

---