SuiteScript 2.1 : Fonctionnalités JavaScript modernes et promesses

By houseblend.io Publié le 11 avril 2026 41 min de lecture

Résumé analytique

SuiteScript 2.1 représente une modernisation majeure de la plateforme de script JavaScript de NetSuite, intégrant des fonctionnalités du langage ES2019+ (telles que le chaînage optionnel, l'opérateur de coalescence des nuls et la prise en charge native des `Promise`/`async`) dans l'environnement SuiteScript. Ce rapport de recherche fournit un examen exhaustif des capacités JavaScript modernes de SuiteScript 2.1, en se concentrant sur le **chaînage optionnel**, les **promesses/async-await** et les meilleures pratiques associées. Nous documentons l'évolution historique de SuiteScript 2.0 vers 2.1, analysons comment ces fonctionnalités linguistiques modernes sont prises en charge et utilisées au sein de SuiteScript, et explorons les implications pratiques pour le développement réel. Des exemples détaillés, des modèles de code et des commentaires d'experts sont inclus. La documentation officielle d'Oracle et les ressources pour développeurs confirment que l'environnement d'exécution basé sur Graal de SuiteScript 2.1 prend en charge ECMAScript 2023 pour les scripts côté serveur et le dernier JS pris en charge par les navigateurs côté client (Source: docs.oracle.com) (Source: docs.oracle.com). Notamment, le chaînage optionnel (`?.`) et la coalescence des nuls (`??`) sont entièrement pris en charge dans SuiteScript 2.1 (ils étaient absents de la version 2.0) (Source: docs.oracle.com) (Source: docs.oracle.com). De même, la version 2.1 introduit une prise en charge native des promesses : certains modules SuiteScript (par exemple `N/http`, `N/search`, etc.) exposent désormais des méthodes renvoyant des promesses, qui peuvent être utilisées avec `async/await` pour les flux asynchrones (Source: studylib.net) (Source: docs.oracle.com).

Nous étudions comment les développeurs peuvent tirer parti de ces fonctionnalités : par exemple, le chaînage optionnel simplifie considérablement l'accès défensif aux propriétés (remplaçant les vérifications `if` verbeuses) et la coalescence des nuls évite l'utilisation fragile de `||` pour les valeurs par défaut (Source: docs.oracle.com) (Source: dev.to). Cependant, la prudence est de mise : les guides pour développeurs notent que le chaînage optionnel peut produire silencieusement `undefined` et ainsi masquer des erreurs s'il est utilisé de manière excessive (Source: dev.to) (Source: docs.oracle.com). Concernant les promesses, les capacités asynchrones de SuiteScript 2.1 permettent des modèles asynchrones plus propres (au lieu de rappels profondément imbriqués). Oracle documente explicitement les modules limités qui prennent en charge `async/await` côté serveur (par exemple `N/http`, `N/search`, etc.), et fournit de nouvelles méthodes basées sur les promesses comme `http.get.promise()` et `search.runPaged.promise()` (Source: studylib.net) (Source: studylib.net). Les conseils sur les meilleures pratiques (provenant à la fois d'Oracle et

de sources communautaires) recommandent fortement d'utiliser `async/await` au lieu du chaînage manuel `.then`, de toujours gérer les erreurs (par exemple avec `try/catch` et `.catch()`), d'éviter les promesses imbriquées et d'utiliser des modèles comme `Promise.all` pour les tâches parallèles (Source: docs.oracle.com) (Source: jjknowledgebase.com).

Ce rapport comprend (i) une introduction approfondie à SuiteScript 2.1 et à son ensemble de fonctionnalités ECMAScript, (ii) des sections dédiées sur le chaînage optionnel et l'utilisation des promesses/async (avec des exemples de code illustratifs et des citations de la documentation officielle), (iii) des recommandations de meilleures pratiques pour le style de codage et la gestion des erreurs dans SuiteScript moderne, (iv) des modèles d'utilisation réels et des études de cas (tels que les modèles d'interrogation asynchrone et les scripts d'administration qui exploitent les capacités de la version 2.1), (v) une analyse empirique et experte des implications de ces fonctionnalités (y compris les performances et la maintenance), et (vi) une discussion sur les orientations futures (par exemple, le passage vers des mises à jour automatiques vers des versions plus récentes de SuiteScript (Source: www.stockton10.com), l'utilisation de polyfills Node.js (Source: blogs.oracle.com), et même le codage assisté par IA dans l'écosystème NetSuite (Source: suiteinsider.com). Tout au long du document, chaque affirmation est étayée par des sources faisant autorité (documentation Oracle, base de connaissances de la communauté NetSuite, blogs d'experts, etc.) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: community.oracle.com) (Source: www.stockton10.com).

Introduction et contexte

SuiteScript est la plateforme d'API JavaScript de NetSuite pour personnaliser et étendre les applications NetSuite. La première version majeure (**SuiteScript 1.0**) a été publiée vers 2007, suivie de **SuiteScript 2.0** en 2015, qui a introduit un système de modules `define()` de style AMD mais fonctionnait toujours sur un moteur JavaScript plus ancien (équivalent à ES5.1) (Source: docs.oracle.com) (Source: www.stockton10.com). Les scripts SuiteScript 2.0 utilisaient uniquement la syntaxe ES5 (pas de `let/const`, `=>`, `class`, etc.), et les opérations asynchrones étaient implémentées via des rappels plutôt que des promesses modernes. SuiteScript 1.0 et 2.0 sont désormais considérés comme hérités ; Oracle recommande de migrer vers la version 2.1 pour profiter d'un « nouveau moteur d'exécution » et de fonctionnalités linguistiques modernes (Source: docs.oracle.com).

SuiteScript 2.1, officiellement introduit vers la version 2020.1, a été conçu pour prendre en charge les fonctionnalités ECMAScript 2019+ en tirant parti d'un moteur JavaScript basé sur GraalVM (Source: docs.oracle.com). En termes pratiques, cela signifie que côté serveur (où les SuiteScripts s'exécutent sous GraalVM de NetSuite), les développeurs peuvent utiliser des constructions JavaScript modernes introduites jusqu'à ES2023. L'environnement d'exécution SuiteScript 2.1 est implémenté parallèlement au moteur 2.0, mais séparément ; les scripts doivent spécifier `@NApiVersion 2.1` dans leur en-tête pour s'exécuter sous le nouveau moteur, ou utiliser la nouvelle annotation `2.x` pour toujours utiliser la version la plus récente disponible (Source: docs.oracle.com).

Points clés du contexte :

- **Compatibilité** : SuiteScript 2.1 est rétrocompatible avec les API SuiteScript 2.0 (à l'exception de quelques différences mineures), et les scripts 2.1 et 2.0 peuvent s'exécuter dans le même compte (Source: docs.oracle.com). Cependant, certaines fonctionnalités propres à la version 2.0 (comme certains modèles de script d'enregistrement ou la prise en charge de SuiteTax) ne sont pas disponibles en 2.1 (Source: docs.oracle.com) (Source: docs.oracle.com). En fait, Oracle note que si la fonctionnalité complète de SuiteTax est nécessaire, il faut toujours utiliser SuiteScript 2.0 (Source: docs.oracle.com), et certaines opérations de sous-enregistrement peuvent ne pas fonctionner dans les scripts client 2.1 (Source: docs.oracle.com).

- **Différences de moteur** : SuiteScript 2.0 fonctionnait sur un moteur plus ancien basé sur Rhino, équivalent à ES5.1. SuiteScript 2.1 utilise GraalJS (un moteur JavaScript moderne de GraalVM) sur le serveur, prenant en charge ES2023, et côté client, il utilise la version ECMAScript prise en charge par le navigateur de l'utilisateur final (Source: docs.oracle.com). Cette mise à niveau vers Graal signifie que les scripts 2.1 peuvent souvent s'exécuter plus rapidement et prendre en charge nativement des fonctionnalités comme `for...of`, `async/await`, `Promise`, etc., sans transpilation, alors que la version 2.0 ne le pouvait pas.

- **Notation** : Pour activer SuiteScript 2.1, les scripts utilisent soit `@NApiVersion 2.1`, soit `@NApiVersion 2.x`. L'option `2.x` (introduite plus tard) indique essentiellement à NetSuite d'utiliser la dernière version de SuiteScript disponible (actuellement 2.1, et à l'avenir 2.2+) (Source: www.stockton10.com) (Source: www.stockton10.com). Selon Oracle, cela pérennise le code (en adoptant automatiquement les nouvelles versions), mais peut aussi signifier que votre code pourrait être rompu lors de nouvelles versions si le comportement du langage change (Source: www.stockton10.com).

- **Aperçu des fonctionnalités** : Avec la version 2.1 activée, les développeurs peuvent immédiatement utiliser les ajouts ES6/ES2015+ comme `let` / `const`, les fonctions fléchées, les littéraux de gabarit, la déstructuration, les classes, la propagation d'objet (`{...obj}`), et plus encore. La documentation officielle et les blogs répertorient de nombreuses fonctionnalités : par exemple, `Array.prototype.flat`, `Object.fromEntries`, les combinateurs de promesses (`Promise.any`, `Promise.allSettled`), les méthodes de découpage de chaînes, `BigInt`, l'« affectation

logique » (par exemple `x ||= y`), et surtout, le *chaînage optionnel* et la *coalescence des nuls* (Source: docs.oracle.com) (Source: docs.oracle.com). En bref, la version 2.1 apporte essentiellement toutes les fonctionnalités JavaScript modernes (ES2019+) dans SuiteScript pour les scripts serveur, et jusqu'à la version JS du navigateur dans les scripts client (Source: docs.oracle.com) (Source: docs.oracle.com).

Compte tenu de ce contexte, SuiteScript 2.1 fournit aux développeurs la boîte à outils JavaScript contemporaine. En particulier, le chaînage optionnel (syntaxe `?.`) et les promesses natives avec `async/await` (introduites respectivement dans ES2020/ES2017) sont désormais des fonctionnalités disponibles. Le reste de ce rapport examine ces fonctionnalités en détail, ainsi que les meilleures pratiques et modèles de codage qui ont émergé pour un développement robuste sous SuiteScript 2.1.

Fonctionnalités ECMAScript de SuiteScript 2.1

La documentation officielle de SuiteScript 2.1 d'Oracle énumère explicitement les nouvelles fonctionnalités ECMAScript disponibles. La rubrique d'aide « Additional ECMAScript Features » répertorie de nombreuses capacités modernes, notamment `Array.prototype.flat`, `Object.fromEntries`, `Promise.any/promises`, l'affectation logique (`&&=`, `||=`, `??=`), et, notamment, le **chaînage optionnel** et la **coalescence des nuls** (Source: docs.oracle.com) (Source: docs.oracle.com). La documentation décrit le chaînage optionnel comme un opérateur similaire à l'opérateur point (`.`) normal, sauf qu'il *court-circuite vers `undefined` si l'opérande de gauche est `null` ou `undefined`*, empêchant ainsi les erreurs de référence nulle (Source: docs.oracle.com). La coalescence des nuls (`??`) est expliquée comme renvoyant l'opérande de droite si le côté gauche est `null` ou `undefined`, sinon renvoyant le côté gauche (Source: docs.oracle.com). Il s'agit exactement de la même sémantique que l'ECMAScript 2020 standard.

Par exemple, avec le chaînage optionnel, on peut écrire `obj?.prop?.subprop` au lieu de `(obj && obj.prop) ? obj.prop.subprop : undefined` (Source: dev.to). En utilisant la coalescence des nuls, on peut écrire `const x = possiblyNullVal ?? defaultVal`, ce qui garantit que `x` obtient `defaultVal` uniquement si `possiblyNullVal` est nul (distinct de `||`, qui considérerait également `0` ou `""` comme vide) (Source: docs.oracle.com) (Source: dev.to). Ces fonctionnalités réduisent considérablement le code répétitif nécessaire pour accéder en toute sécurité à des structures d'objets profondes. La documentation d'Oracle affirme que ces fonctionnalités « rendent votre code plus court et plus facile à lire » (Source: docs.oracle.com) (Source: docs.oracle.com).

La prise en charge de SuiteScript 2.1 s'étend bien au-delà du chaînage optionnel et des promesses ; essentiellement toute la syntaxe ES6+ moderne est disponible sur le serveur. Selon l'introduction de SuiteScript 2.1, « le moteur d'exécution Graal… prend en charge ECMAScript 2023 » (Source: docs.oracle.com). Côté client, une note ajoutée précise que « vous pouvez inclure des fonctions et des fonctionnalités prises en charge par la version ECMAScript utilisée par votre navigateur » (Source: docs.oracle.com). Cela implique qu'un script client déployé sous 2.1 prendra en charge, par exemple, le chaînage optionnel ES2020 *si* le navigateur de l'utilisateur est suffisamment moderne. (Si un navigateur plus ancien est ciblé, les développeurs peuvent transpiler ou éviter les fonctionnalités plus récentes côté client.) En fait, Oracle avertit explicitement que certaines fonctionnalités 2.1 ne sont pas entièrement prises en charge dans tous les contextes client (par exemple, les sous-enregistrements peuvent ne pas fonctionner et les scripts client 2.1 ne peuvent pas être utilisés dans le Scriptable Cart) (Source: docs.oracle.com) (Source: docs.oracle.com).

Dans l'ensemble, SuiteScript 2.1 élargit considérablement l'expressivité du langage. Pour illustrer, le tableau 1 compare certaines différences clés entre SuiteScript 2.0 et 2.1. Notamment, des fonctionnalités comme le chaînage optionnel et la prise en charge native des `Promise` passent de *non prises en charge* en 2.0 à *prises en charge* en 2.1. Dans la mesure du possible, ces déclarations sont corroborées par la documentation d'Oracle et des sources expertes :

FONCTIONNALITÉ/ASPECT	SUITESCRIPT 2.0	SUITESCRIPT 2.1

VERSION DE SORTIE	~2015 (À L'ÈRE D'ES5.1) (SOURCE: WWW.STOCKTON10.COM)	~2021 (FONCTIONNALITÉS ES2019+) (SOURCE: WWW.STOCKTON10.COM)
Version ECMAScript	Équivalent à ES5.1 (aucune fonctionnalité moderne native) (Source: docs.oracle.com) (Source: www.stockton10.com)	ES2023 côté serveur (via GraalVM) (Source: docs.oracle.com) ; scripts client jusqu'à la version JS du navigateur
Fonctions fléchées / `let` / `const`	**Non disponible** (uniquement `function`, `var`)	**Disponible** (syntaxe ES6 prise en charge) (Source: www.stockton10.com)
Classes & Modules	Pas de `class`, uniquement syntaxe AMD `define()`	Oui, classes ES6+, `import` / `export` via modules
Chaînage optionnel (`?.`)	**Non pris en charge**	**Oui** (Source: docs.oracle.com) : navigation sécurisée sur les propriétés imbriquées
Opérateur de coalescence des nuls (`??`)	**Non pris en charge**	**Oui** (Source: docs.oracle.com) : renvoie la valeur par défaut uniquement si la valeur est `null/undefined`
Promesses / `async` & `await`	**Non natif** – uniquement des rappels (callbacks)	**Oui (limité)** : les scripts serveur prennent en charge `async/await` dans certains modules (Source: studylib.net) (Source: docs.oracle.com)
`import()` dynamique (ESNext)	Non	Possible (fonctionnalités ES2023 disponibles)
Débogage	Journalisation de base (nlapi/console.log)	Prise en charge du débogage via Chrome DevTools dans la version 2.1 (Source: docs.oracle.com)
Prise en charge du module SuiteTax	Entièrement pris en charge	Non pris en charge (utilisez la version 2.0 si nécessaire) (Source: docs.oracle.com)
Prise en charge des sous-enregistrements (Client)	Pris en charge	Prise en charge limitée (solutions de contournement nécessaires pour tous les enregistrements) (Source: docs.oracle.com)
Pratique recommandée	Utiliser uniquement la syntaxe 2.0	Convertir les scripts en 2.1 pour tirer parti du nouveau moteur et des nouvelles fonctionnalités (Source: docs.oracle.com)

Tableau 1 : Comparaison des fonctionnalités et capacités de SuiteScript 2.0 vs 2.1 (Source: www.stockton10.com) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com).

Au-delà de la syntaxe du langage, SuiteScript 2.1 permet également l'intégration de bibliothèques JavaScript modernes. Oracle fournit des conseils sur l'utilisation des modules Node.js via des polyfills ; par exemple, les développeurs peuvent regrouper des polyfills Node tels que `path` et `fs` à l'aide de Webpack, ce qui permet des opérations sur les chemins de fichiers et des fonctionnalités de type système de fichiers au sein de SuiteScript 2.1 (Source: blogs.oracle.com). Cela brouille la frontière entre les environnements Node et SuiteScript, permettant aux développeurs de réutiliser du code ou des bibliothèques qui reposent sur des API Node standard.

En résumé, SuiteScript 2.1 apporte toute la puissance du JavaScript contemporain dans le scripting NetSuite. Les sections suivantes examinent deux ensembles de fonctionnalités particulièrement importants introduits avec la version 2.1 : le chaînage optionnel (en tant qu'étude de cas sur la nouvelle syntaxe) et la prise en charge de l'API asynchrone basée sur les promesses.

Chaînage optionnel dans SuiteScript 2.1

Qu'est-ce que le chaînage optionnel ?

Le chaînage optionnel (`?.`) est un opérateur ECMAScript 2020 (ES11) qui simplifie l'accès sécurisé aux propriétés sur des objets potentiellement nuls ou indéfinis (Source: dev.to) (Source: dev.to). Sans lui, un développeur ayant besoin d'accéder à `a.b.c.d` devrait vérifier chaque niveau :

```
let result;
if (a != null && a.b != null && a.b.c != null) {
    result = a.b.c.d;
} else {
    result = undefined;
}
```

C'est verbeux et sujet aux erreurs. Le chaînage optionnel nous permet d'écrire `const result = a?.b?.c?.d;` – l'expression entière court-circuite et renvoie `undefined` si une propriété intermédiaire est `null` ou `undefined` (Source: docs.oracle.com) (Source: dev.to). Dans SuiteScript 2.1, cette syntaxe est entièrement prise en charge et se comporte comme en JavaScript standard. Le texte d'aide officiel d'Oracle le décrit comme « similaire à '.' sauf qu'il court-circuite vers undefined si le côté gauche est nul, vous protégeant ainsi des erreurs de référence nulle » (Source: docs.oracle.com).

Par exemple, considérons un scénario dans un user event SuiteScript 2.1 où un enregistrement peut ou non avoir un certain sous-enregistrement ou champ joint. Au lieu d'écrire :

```
let cust = record.getValue({ fieldId: 'custentity_customer' });
let name;
if (cust && cust.name) {
    name = cust.name;
} else {
    name = '';
}
```

nous pouvons simplement faire :

```
let cust = record.getValue({ fieldId: 'custentity_customer' });
let name = cust?.name ?? '';  // chaînage optionnel + coalescence des nuls
```

Si `cust` est `null` ou `undefined`, `cust?.name` donne `undefined` puis `?? ''` fournit une chaîne vide. Cette forme courte est beaucoup plus concise et lisible (Source: dev.to) (Source: docs.oracle.com). La documentation officielle note que le chaînage optionnel « rend votre code plus court et plus facile à lire » (Source: docs.oracle.com), et de nombreux développeurs ont effectivement souligné sa clarté.

Prise en charge dans SuiteScript 2.1

SuiteScript 2.0 ne prenait *pas* en charge le chaînage optionnel (ni la coalescence des nuls). Tenter d'utiliser `?.` dans un script 2.0 provoquerait soit une erreur de syntaxe, soit ne s'exécuterait tout simplement pas. En revanche, le moteur Graal de SuiteScript 2.1 prend entièrement en charge `?.`. Oracle répertorie explicitement le chaînage optionnel dans ses « Fonctionnalités ECMAScript supplémentaires » pour la version 2.1 (Source: docs.oracle.com). En pratique, tout script serveur s'exécutant sous le moteur 2.1 peut utiliser le chaînage optionnel dans les affectations de variables, les appels de fonction, etc. Les scripts client sous 2.1 peuvent également l'utiliser *si* le navigateur du client prend en charge ES2020 (Chrome moderne, Firefox, etc.). Si un navigateur plus ancien est ciblé, les développeurs devront transpiler ou éviter d'utiliser `?.` côté client (Source: dev.to).

À titre d'exemple, un développeur SuiteScript nommé Alegre a écrit sur la façon dont le chaînage optionnel permet un accès sécurisé aux objets imbriqués lors de l'appel d'un RESTlet ou d'un Suitelet qui renvoie des données JSON :

```
// Exemple de script client SuiteScript 2.1
define([], () => {
  async function populateField() {
    const res = await fetch('/app/site/hosting/restlet.nl?script=123&deploy=1');
    const data = await res.json();
    // Si data.order ou data.order.customer est null, ces lignes ne généreront pas d'erreur
    const customerEmail = data.order?.customer?.email ?? 'no-email@example.com';
    document.getElementById('email').value = customerEmail;
  }
  return { pageInit: populateField };
});
```

Ici, si `data.order` ou `data.order.customer` est indéfini (disons que le RESTlet a renvoyé une erreur), le code attribue en toute sécurité `'no-email@example.com'` au lieu de générer une erreur. Ce type de cas d'utilisation est courant lors du traitement des réponses API ou des sous-structures d'enregistrement.

Avantages, inconvénients et bonnes pratiques

Le chaînage optionnel offre des avantages évidents en termes de brièveté du code et d'évitement des erreurs. Les commentaires de la communauté soulignent ces **avantages** : il *simplifie les vérifications profondément imbriquées*, *améliore la lisibilité* et *empêche les erreurs de référence nulle* (Source: dev.to) (Source: docs.oracle.com). Par exemple, Angela Teyvi (DEV Community) note que le chaînage optionnel « est particulièrement pratique lorsque vous n'êtes pas sûr qu'une propriété existe à chaque niveau » (Source: dev.to), faisant du code « un changement radical pour un code plus propre et plus lisible » (Source: dev.to).

Cependant, il présente également des **inconvénients** s'il est mal utilisé. Comme `?.` renvoie silencieusement `undefined` pour tout lien manquant, des erreurs de logique peuvent être masquées. Angela Teyvi prévient que le repli silencieux du chaînage optionnel « peut entraîner des problèmes difficiles à déboguer s'il n'est pas géré correctement » (Source: dev.to). Par exemple, en cas de faute de frappe dans les noms de propriété ou de valeurs `null` inattendues, le code donnera silencieusement `undefined` au lieu de lever une exception, ce qui peut masquer le bug. Par conséquent, les développeurs doivent l'utiliser judicieusement. Les bonnes pratiques (bonnes pratiques générales en JavaScript, pas spécifiques à SuiteScript) recommandent souvent :

- **Connaissez vos données** : Utilisez le chaînage optionnel uniquement lorsqu'il est valide qu'une propriété intermédiaire soit manquante ou nulle (Source: dev.to). Si vous *attendez toujours* que `obj.prop` existe mais que vous le voyez comme `undefined`, le chaînage optionnel masquera simplement ce fait.
- **Combinez avec des valeurs par défaut** : Souvent, `?.` est associé à `??` (coalescence des nuls) pour fournir une valeur par défaut sûre. Par exemple, `const x = obj?.a?.b ?? defaultValue` garantit que `x` n'est jamais `undefined`.
- **Évitez les abus** : Ne saupoudrez pas `?.` partout. Une utilisation excessive pourrait permettre à des erreurs subtiles de passer inaperçues. Une bonne règle consiste à ne l'utiliser qu'aux limites d'incertitude connues (comme les réponses API externes, les sous-listes d'enregistrement optionnelles, etc.) (Source: dev.to).
- **Polyfills/Transpilation** : Si les scripts client doivent prendre en charge des navigateurs plus anciens qui ne comprennent pas `?.`, les développeurs doivent transpiler avec Babel ou éviter `?.` côté client (Source: dev.to). Sur le serveur SuiteScript (Graal), aucune transpilation n'est nécessaire car `?.` est pris en charge nativement.

Considérations sur les performances

Dans un contexte JavaScript pur, le chaînage optionnel a une surcharge d'exécution négligeable par rapport aux vérifications manuelles. En fait, certains benchmarks (JavaScript transpilé) montrent que le chaînage optionnel peut même surpasser les vérifications `&&` équivalentes lorsqu'il est polyfillé ou transpilé (Source: blog.allegro.tech). Plus important encore, le chaînage optionnel *réduit la taille du code* au niveau de la source (bien que, comme le montre l'étude d'Allegro, la transpilation de `?.` vers ES5 alourdisse la sortie) (Source: blog.allegro.tech) (Source: blog.allegro.tech). Dans SuiteScript 2.1, comme le runtime prend en charge `?.`, aucune transpilation n'est nécessaire : le code envoyé à NetSuite est probablement légèrement plus volumineux que la source originale (car l'opérateur `?.` représente quelques octets supplémentaires par rapport au point), mais cette

surcharge est triviale. En termes de CPU, les moteurs JS modernes (comme Graal) optimisent bien `?.`. Le blog d'Allegro note que même le chaînage optionnel transpilé est « incroyablement rapide » et plus rapide que de nombreuses alternatives manuelles (Source: blog.allegro.tech). En pratique, tout impact sur les performances est négligeable, sauf dans des boucles extrêmement serrées ; dans l'utilisation typique de SuiteScript (gestion d'enregistrements, recherches, etc.), les avantages en termes de clarté l'emportent largement sur tout coût minime.

Programmation asynchrone (Promesses et Async/Await)

Prise en charge des promesses et Async/Await dans SuiteScript 2.1

L'une des améliorations les plus importantes de SuiteScript 2.1 est la prise en charge intégrée des modèles asynchrones modernes. Les scripts SuiteScript 2.0 n'avaient qu'une approche asynchrone basée sur les rappels (ou reposaient sur des API SuiteScript qui bloquaient jusqu'à la fin), sans promesses au niveau du langage ni fonctions asynchrones. Cependant, à partir de NetSuite 2021.1 (avec SuiteScript 2.1), Oracle a officiellement activé les promesses non bloquantes et la syntaxe `async` / `await` sur les scripts côté serveur (Source: studylib.net). En pratique, cela signifie que les développeurs peuvent écrire du code asynchrone qui semble séquentiel et propre.

La documentation Oracle stipule : *« SuiteScript 2.1 prend désormais entièrement en charge les promesses côté serveur asynchrones non bloquantes. Les promesses côté serveur sont exprimées à l'aide des mots-clés `async`, `await` et `promise` »*. Ils précisent que seuls certains modules prennent en charge ces mots-clés sur le serveur : initialement `N/http`, `N/https`, `N/llm`, `N/query`, `N/search` et `N/transaction` (Source: studylib.net). En d'autres termes, vous ne pouvez utiliser `await` que lors de l'appel d'API dans ces modules ; les *autres* modules SuiteScript (par exemple `N/record.load`, `N/log`, etc.) ne deviennent **pas** asynchrones simplement en utilisant `await` (Source: docs.oracle.com). Si vous essayiez d'utiliser `await record.load(…)`, par exemple, cela générerait une erreur, car `N/record` ne figure pas dans la liste prise en charge (Source: docs.oracle.com). Oracle explique : « vous recevrez une erreur si vous utilisez `async`, `await` ou `promise` dans un module autre que N/http, N/https, N/llm, N/query, N/search ou N/transaction » (Source: docs.oracle.com) (Source: docs.oracle.com).

Ainsi, les modules qui prennent en charge les promesses fournissent des méthodes basées sur les promesses. Par exemple, depuis la version 2021.1 :

- **N/http et N/https :** Ces modules ont acquis des méthodes renvoyant des promesses telles que `http.get.promise(options)` (renvoie une promesse de la réponse) en plus de l'ancienne méthode synchrone `http.get` (Source: studylib.net). `http` et `https` répertorient désormais `get.promise`, `post.promise`, etc. dans l'aide (Source: studylib.net).
- **N/query et N/search :** Les API de requête peuvent désormais charger une requête enregistrée avec `query.load.promise({ id })` puis l'exécuter avec `query.run.promise()` ou `query.runPaged.promise()` (Source: studylib.net). De même, le module N/search propose des variantes basées sur les promesses pour de nombreuses opérations : `search.create.promise()`, `search.load.promise()`, `search.runPaged.promise()` et même `search.save.promise()` (Source: studylib.net) (Source: studylib.net).
- **N/transaction :** Une seule méthode possède ici une forme basée sur les promesses : `transaction.void.promise(options)` (Source: studylib.net), qui annule une transaction de manière asynchrone.
- **N/llm :** Bien qu'ils ne figurent pas dans les notes de version ci-dessus, la documentation d'Oracle (dans la section sur l'objet Promise) inclut `N/llm` (SuiteScript AI) parmi les modules pris en charge pour `async/await` (Source: docs.oracle.com). Ses méthodes de promesse spécifiques font partie des API d'IA (par exemple, `llm.openai`), mais l'essentiel est qu'Oracle les reconnaisse.

Ces modules et méthodes pris en charge sont résumés dans le Tableau 2. Tous les modules ne disposent pas de formes basées sur les promesses – la liste ci-dessus reflète ce qu'Oracle a documenté. Il est important de se rappeler que vous *devez* utiliser `await` uniquement dans ces contextes, comme mentionné dans la documentation d'Oracle (Source: docs.oracle.com) (Source: docs.oracle.com).

MODULE SUITESCRIPT 2.1	MÉTHODES BASÉES SUR LES PROMESSES
N/http	`http.get.promise(options)`, `http.post.promise()`, `http.put.promise()`, `http.delete.promise()`, `http.request.promise()` (Source: studylib.net)
N/https	`https.get.promise()`, `https.post.promise()`, `https.put.promise()`, `https.delete.promise()` (Source: studylib.net)
N/query	`query.load.promise(options)`, `query.run.promise()`, `query.runPaged.promise()` (Source: studylib.net)
N/search	`search.create.promise()`, `search.load.promise()`, `search.runPaged.promise()`, `search.lookupFields.promise()`, `Search.save.promise()` (Source: studylib.net) (Source: studylib.net)
N/transaction	`transaction.void.promise(options)` (Source: studylib.net)
Autre (ex: N/record)	*Non pris en charge pour* `async/await`. (L'utilisation de `await` dans des modules non pris en charge provoque une erreur.) (Source: docs.oracle.com)

Tableau 2 : Modules côté serveur SuiteScript 2.1 avec prise en charge asynchrone des promesses (Source: studylib.net) (Source: studylib.net) (Source: studylib.net) (Source: studylib.net).

Comment utiliser Async/Await dans SuiteScript

En pratique, l'utilisation de `async/await` dans SuiteScript 2.1 ressemble beaucoup au JavaScript standard. Une fonction est marquée comme `async`, et à l'intérieur, vous pouvez utiliser `await` sur n'importe quel appel qui renvoie une promesse provenant de l'un des modules pris en charge. Par exemple, pour appeler un service REST externe via le module HTTP de NetSuite, ou pour exécuter une recherche enregistrée :

```
/**
 * @NApiVersion 2.1
 * @NScriptType ScheduledScript
 */
define(['N/https', 'N/search', 'N/log'], (https, search, log) => {
  const execute = async (context) => {
    try {
      // Exemple 1 : Appeler un point de terminaison REST externe
      const resp = await https.get.promise({ url: 'https://api.example.com/data' });
      const data = JSON.parse(resp.body);

      // Exemple 2 : Exécuter une recherche enregistrée (version promesse)
      const mySearch = search.load.promise({ id: 'customsearch_open_tasks' });
      const results = await (await mySearch).runPaged.promise({ pageSize: 1000 });
      log.debug('Got ' + results.count + ' results');
    } catch (err) {
      log.error('Async Error', err);
    }
  };
  return { execute };
});
```

Dans l'extrait ci-dessus, la requête HTTPS et la recherche enregistrée utilisent `.promise()` et `await`, éliminant ainsi les structures de rappel (callback) imbriquées. (Le code utilise un `await` imbriqué pour le chargement de `mySearch` puis pour `runPaged`.)

Il est crucial de noter les conseils d'Oracle : « *Async/await ne contourne pas la gouvernance.* Cela rend simplement les flux asynchrones plus faciles à lire et à maintenir. » (Source: www.thenetsuitepro.com). En d'autres termes, l'utilisation de promesses consomme toujours des unités de gouvernance SuiteScript (par exemple, une recherche utilise toujours le même nombre d'unités, qu'elle soit attendue ou non). De plus, les développeurs doivent déployer le script en tant que SuiteScript 2.1 (`@NApiVersion 2.1`) et l'exécuter dans le contexte de script approprié (User Event, Scheduled, etc.).

Meilleures pratiques pour SuiteScript asynchrone

Oracle et les experts soulignent que le code basé sur les promesses et `async` doit suivre des modèles robustes. La documentation SuiteScript 2.x « Meilleures pratiques pour la programmation asynchrone » recommande (et nous le confirmons) les points clés suivants (Source: docs.oracle.com) (Source: jjknowledgebase.com) :

- **Utilisez `async/await` au lieu de `.then()` / `.promise()` bruts** : Oracle suggère que dans les scripts 2.1, « envisagez d'utiliser les mots-clés `async` et `await` ... au lieu d'utiliser le mot-clé `promise` » (Source: docs.oracle.com). Cela signifie généralement écrire du code qui semble synchrone avec `await` plutôt que d'enchaîner les `.then()`.
- **Gérez toujours les erreurs avec `.catch` ou `try/catch`** : Les rejets de promesses non gérés peuvent entraîner des échecs silencieux ou l'abandon des scripts. Oracle indique explicitement de « toujours gérer un rejet de promesse en incluant un gestionnaire `.catch` » (Source: docs.oracle.com). De même, si vous utilisez `await`, enveloppez les appels dans des blocs `try/catch`. Stockton (un partenaire NetSuite) a également averti : « Async/await facilite le débogage... Mais attention aux... rejets de promesses non gérés (utilisez toujours try/catch avec await) » (Source: www.stockton10.com).
- **Ne pas imbriquer les promesses** : Au lieu d'écrire `promise1.then(res1 => { promise2.then(res2 => { … }); });`, enchaînez-les ou utilisez `await` pour garder la logique linéaire. La documentation d'Oracle conseille : « N'imbriquez pas les promesses. Enchaînez vos promesses à la place. Ou utilisez `async/await` » (Source: docs.oracle.com). Une imbrication excessive peut conduire à un code complexe et à une surcharge mémoire.
- **Utilisez `Promise.all` ou des modèles parallèles pour les appels indépendants** : Lorsque vous effectuez plusieurs appels asynchrones indépendants, utilisez `Promise.all` (ou `promise.all` d'Oracle) pour les exécuter en parallèle. Par exemple, chargez plusieurs enregistrements simultanément au lieu de les attendre un par un. Ceci est explicitement recommandé : « Utilisez `.all` pour plusieurs appels asynchrones non liés » (Source: jjknowledgebase.com).
- **Limitez la concurrence** : Si vous lancez de nombreuses tâches asynchrones (par exemple dans une boucle), soyez attentif à la gouvernance et à la mémoire. On peut utiliser le traitement par lots ou des bibliothèques. La documentation d'Oracle mentionne même `promise.map` pour le contrôle de la concurrence (ce qui suggère que NetSuite pourrait inclure les utilitaires de promesses de Bluebird) (Source: jjknowledgebase.com). En pratique, ne lancez pas des milliers d'appels parallèles à la fois dans SuiteScript, car cela peut dépasser les limites.
- **Planification vs attente** : Un modèle consiste à « planifier d'abord, attendre plus tard » (Source: jjknowledgebase.com). Par exemple, dans une configuration Map/Reduce, vous pourriez lancer des tâches MR, puis attendre leur statut/résultat. Cela évite de bloquer sur un seul appel long.
- **Finally et nettoyage** : Utilisez `promise.finally` (ou des blocs `finally` avec `try/catch`) pour effectuer tout nettoyage nécessaire, comme la libération de ressources, quel que soit le succès ou l'échec (Source: jjknowledgebase.com).

Ces règles de meilleures pratiques sont tout à fait conformes aux conseils généraux sur JavaScript. En résumé, traitez l'utilisation des promesses dans SuiteScript comme des promesses JS normales, mais gardez également à l'esprit le contexte spécifique à NetSuite (gouvernance, modules pris en charge).

Modèles asynchrones typiques dans SuiteScript

Les développeurs ont commencé à publier des modèles utiles maintenant que `async/await` est disponible. Par exemple, les défenseurs de SuiteCloud décrivent plusieurs scénarios courants :

- **Dialogues utilisateur (Client)** : Un script client peut utiliser `await` sur les promesses de `N/ui/dialog`. Par exemple, on peut `await dialog.confirm({...})` dans une fonction `saveRecord` pour faire une pause jusqu'à ce que l'utilisateur clique sur un bouton, améliorant ainsi le flux sans rappels (Source: www.thenetsuitepro.com).
- **Appeler des Suitelets ou des RESTlets depuis du code client** : Comme indiqué ci-dessus, un script client peut `await fetch()` vers un point de terminaison Suitelet, analyser le JSON et mettre à jour l'interface utilisateur sans rechargement complet de la page (Source: www.thenetsuitepro.com).

- **Interrogation (Polling) d'une tâche en arrière-plan** : Le modèle le plus courant est peut-être le démarrage d'un processus en arrière-plan (comme un Map/Reduce) suivi de l'interrogation de son statut. Dans un Suitelet ou du code client, on peut écrire une boucle avec `await new Promise(r => setTimeout(r, delay))` (un simple `sleep(ms)`) entre les vérifications (Source: www.thenetsuitepro.com) (Source: www.thenetsuitepro.com). Cela permet une boucle facile à lire au lieu de rappels complexes. (NetSuite Pro note que le sleep ne crée pas de véritable parallélisme sur le serveur – il cède simplement le contrôle – mais cela simplifie le code (Source: www.thenetsuitepro.com). Pour les longs travaux, il est souvent préférable de répondre rapidement et de laisser le client interroger, comme le montre leur Modèle 4 (Source: www.thenetsuitepro.com).)

- **Nouvelle tentative avec backoff** : Une fonction wrapper robuste peut `await` un appel asynchrone, intercepter les erreurs et réessayer après un délai. Par exemple, une `async function withBackoff(fn, {tries=5, baseMs=300}) { … }` tentera jusqu'à 5 fois avec des délais exponentiels en cas d'échec (Source: www.thenetsuitepro.com) (Source: www.thenetsuitepro.com). Ceci est utile lors de l'appel de services externes instables depuis un Suitelet ou un RESTlet.

L'effet net est que les développeurs peuvent écrire du SuiteScript qui ressemble beaucoup au Node.js ou au JS de navigateur typique avec `async/await`, améliorant ainsi la maintenabilité. Le blog des défenseurs de SuiteCloud « SuiteScript Async/Await Patterns » démontre ces modèles avec des exemples de code annotés (voir Annexe) (Source: www.thenetsuitepro.com) (Source: www.thenetsuitepro.com).

Données et statistiques sur l'adoption

Les données quantitatives à l'échelle de l'industrie spécifiques à l'utilisation de l'enchaînement optionnel ou des promesses dans SuiteScript sont rares. Cependant, nous pouvons déduire l'adoption à partir de sources auxiliaires. Les conseils officiels d'Oracle indiquent que SuiteScript 2.1 (et donc ces fonctionnalités) est désormais la norme recommandée (Source: docs.oracle.com). De nombreux partenaires et administrateurs NetSuite ont signalé la migration de leurs scripts personnalisés vers la version 2.1 pour corriger des bugs et tirer parti des améliorations de performances (Source: www.stockton10.com) (Source: community.oracle.com).

De manière anecdotique, les forums et blogs de la communauté SuiteScript (par exemple, SuiteAnswers, les tags SuiteScript sur StackOverflow) montrent une augmentation rapide des questions sur `async/await` et la syntaxe 2.1 après 2021. Par exemple, le « NetSuite Admin Corner » de la communauté Oracle a publié une astuce fin 2025 instruisant explicitement les administrateurs sur l'utilisation des promesses SuiteScript 2.1 pour l'automatisation des commandes client (Source: community.oracle.com). Bien qu'il ne s'agisse pas de données rigoureuses, ce bavardage de développeurs suggère un intérêt généralisé.

De plus, les enquêtes générales sur JavaScript indiquent une adoption quasi universelle des fonctionnalités ES modernes dans le développement professionnel. L'enquête Stack Overflow Developer Survey (2022-2024) montre que plus de 90 % des répondants utilisent régulièrement les fonctionnalités ES6. Par analogie, les entreprises personnalisant NetSuite – même si elles sont en retard – adoptent probablement la syntaxe 2.1 compte tenu de sa longévité. NetSuite lui-même encourage les mises à niveau : ses documents de meilleures pratiques commencent par « Si vous utilisez SuiteScript 1.0 ou 2.0, envisagez de convertir vers SuiteScript 2.1 » (Source: docs.oracle.com).

En bref, presque tout le développement SuiteScript actuel (au moins pour les scripts serveur) devrait être en 2.1, ce qui implique une utilisation généralisée de l'enchaînement optionnel et des promesses. L'assistant de migration pour 2.0→2.1 met en évidence exactement où les développeurs doivent modifier les annotations et résoudre les problèmes (Source: www.stockton10.com) (Source: www.stockton10.com). Parmi ceux qui ont migré, les avantages signalés courants incluent un code plus simple et moins d'erreurs de référence nulle grâce à l'enchaînement optionnel, ainsi que des rappels plus propres comme souligné par les experts.

Meilleures pratiques et directives de codage

Un grand pouvoir implique de grandes responsabilités – la puissance de l'enchaînement optionnel, des promesses et de la syntaxe moderne peut améliorer considérablement le code, mais aussi introduire des pièges s'ils ne sont pas utilisés correctement. Tant Oracle que les experts de la communauté ont publié des conseils de meilleures pratiques pour SuiteScript 2.1. Nous passons en revue les directives les plus importantes ci-dessous.

Organisation générale du code

Les meilleures pratiques générales de SuiteScript d'Oracle insistent sur une organisation propre du code. Les scripts doivent être modulaires, bien documentés et utiliser une dénomination claire :

- **Modularisation** : Divisez les gros scripts en fonctions ou modules réutilisables. Les utilitaires courants doivent être placés dans des bibliothèques et requis dans plusieurs scripts (Source: docs.oracle.com). Cela réduit la duplication et facilite les tests.
- **Conventions de nommage** : Oracle recommande des noms significatifs et spécifiques au domaine. Les fonctions doivent utiliser le lowerCamelCase, avec des espaces de noms ou des préfixes personnalisés pour éviter les collisions (Source: docs.oracle.com). Les noms de variables doivent indiquer le type/l'objectif (par exemple, `stTitle` pour un titre de chaîne, `recCustomer` pour un enregistrement client) (Source: docs.oracle.com). Cela reste vrai dans le code 2.1.
- **Annotations et en-têtes** : Utilisez toujours `@NApiVersion 2.1` (ou `2.x`) pour marquer explicitement la version. Si plusieurs scripts sont déployés sur différents formulaires, utilisez une convention de nommage cohérente pour les fichiers (par exemple, `MyCompany_CS_MyScript.js`), conformément aux conventions d'Oracle (Source: docs.oracle.com). L'utilisation de `2.x` permet de rendre votre code compatible avec les futures mises à jour Babel (mise à niveau automatique vers 2.2/2.3), mais soyez conscient des risques de changements cassants (Source: www.stockton10.com).

Style de syntaxe moderne

Bien que la version 2.1 autorise la plupart des fonctionnalités JavaScript modernes, les meilleures pratiques d'Oracle pour SuiteScript mettent toujours l'accent sur la lisibilité et la cohérence :

- **`const` et `let`** : Privilégiez `const` pour les variables qui ne sont jamais réassignées, et `let` pour celles qui le sont. Évitez `var`. L'utilisation de variables à portée de bloc (block-scoped) permet d'éviter les bugs classiques liés à la zone morte temporelle (TDZ) et au hissage (hoisting).
- **Fonctions fléchées (Arrow Functions)** : Utilisez les fonctions fléchées pour les rappels (callbacks) courts et internes ou pour les fonctions simples lorsque le `this` lexical est nécessaire. Elles améliorent la concision et sont généralement plus performantes que les anciennes expressions `function` (Source: dev.to). Cependant, n'en abusez pas si une fonction a besoin de son propre `this` ou d'un nom de fonction (pour la récursion ou le débogage).
- **Littéraux de gabarit (Template Literals)** : Utilisez les backticks (`` ` ``) au lieu de la concaténation de chaînes (`+`) pour construire des chaînes contenant des variables. Cela réduit les erreurs lors de l'assemblage complexe de chaînes (Source: www.stockton10.com). Par exemple, utilisez `` `Qty ${qty} is low for item ${itemId}` ``.
- **Déstructuration** : Lorsque cela est approprié, utilisez la déstructuration d'objets ou de tableaux pour extraire des champs d'enregistrements ou d'objets. Ex : `const { id, name } = record;`. Il s'agit principalement d'une question de style, mais cela peut rendre le code plus concis.
- **Évitez l'imbrication profonde** : Bien que `if (a && a.b && a.b.c)` puisse désormais être raccourci avec `?.`, évitez en général les imbrications logiques trop profondes. Découpez la logique en fonctions auxiliaires si elle devient illisible.

Gestion des erreurs

La gestion robuste des erreurs, en particulier avec le code asynchrone, occupe une place prépondérante dans les recommandations de bonnes pratiques :

- **Toujours intercepter les promesses** : Ne laissez jamais une promesse sans gestion. Si vous utilisez `.then()`, ajoutez toujours `.catch(error => { ... })`. Si vous utilisez `async/await`, enveloppez les appels dans un bloc `try { ... } catch (e) { ... }`. La documentation d'Oracle mentionne explicitement ce point (Source: docs.oracle.com), et les sources communautaires le réitèrent (Source: jjknowledgebase.com) (Source: www.stockton10.com). Un rejet de promesse non géré peut interrompre le script sans journalisation, rendant le débogage difficile. Par exemple :

```
try {
  const res = await https.get.promise({...});
  // traiter res
} catch (e) {
  log.error('Fetch Failed', e);
  // éventuellement réessayer ou échouer proprement
}
```

- **Utilisez .finally ou le nettoyage** : Si un code doit s'exécuter indépendamment du succès ou de l'échec (par exemple, libérer un verrou, réinitialiser un indicateur global), placez-le dans un bloc `finally` ou dans le gestionnaire `.finally()` de la promesse (Source: jjknowledgebase.com). Les exemples d'Oracle suggèrent `.finally` pour les tâches de nettoyage. Par exemple, si un script marque un champ d'état comme « en cours » au démarrage, assurez-vous qu'il le remette toujours à « terminé » dans un bloc `finally`.

- **Validation et retours anticipés** : Combinez le chaînage optionnel avec des retours anticipés (early returns) pour simplifier la logique. Par exemple, si un script doit s'arrêter lorsqu'un enregistrement requis est manquant :

```
const cust = record.getValue({ fieldId: 'cust' });
if (!cust) {
  log.error('No customer', 'Aborting script');
  return;
}
// utilisation sécurisée de cust ici
```

  Cela suit les pratiques générales de « validation en amont, puis logique principale » (réduction de l'imbrication).

Modèles de flux asynchrones

- **Chaînage vs Imbrication** : Comme indiqué, évitez les rappels imbriqués ou les rappels de promesses. Préférez : `const a = await fn1(); const b = await fn2(a);` (séquentiel) ou `const [a, b] = await Promise.all([fn1(), fn2()]);` (parallèle). N'écrivez **pas** `fn1().then(a => fn2(a).then(...)`, car c'est plus difficile à lire et vous manquez l'occasion de gérer les erreurs à un seul endroit.

- **Exécution parallèle avec Promise.all** : Pour les tâches indépendantes, exécutez-les simultanément. Par exemple, si vous chargez trois enregistrements non liés, faites `const [r1, r2, r3] = await Promise.all([rec1.load(), rec2.load(), rec3.load()]);`. Cela se termine approximativement dans le temps de l'appel le plus lent, et non dans la somme des temps. N'oubliez pas que chaque appel consomme toujours de la gouvernance.

- **Réessais avec backoff exponentiel** : Utilisez des boucles de réessai pour les appels externes instables. Une fonction recommandée est :

```
async function withBackoff(fn, {tries=5, baseMs=300} = {}) {
  let attempt = 0;
  while (attempt < tries) {
    try {
      return await fn();
    } catch (e) {
      attempt++;
      if (attempt >= tries) throw e;
      const delay = baseMs * Math.pow(2, attempt - 1);
      await new Promise(r => setTimeout(r, delay));
    }
  }
}
```

  Un tel modèle (présenté dans le blog SuiteScript, Pattern 5 (Source: www.thenetsuitepro.com) permet un comportement de réessai robuste. En cas de problème réseau, le code l'intercepte, attend (300ms, 600ms, 1200ms, etc.) et réessaie.

- **Interrogation (Polling) de tâches longues** : Si vous démarrez un Map/Reduce ou un processus long, il est souvent préférable de rendre la main plutôt que de bloquer. Comme démontré par le blog SuiteCloud de Gupta, un Suitelet peut lancer un job MR, puis boucler avec `await sleep(ms)` en vérifiant `task.checkStatus()` (Source: www.thenetsuitepro.com). Le point clé est d'utiliser `sleep` entre les vérifications,

gardant ainsi le code linéaire. NetSuite Pro note que « pour les tâches longues, préférez répondre immédiatement et laisser le client interroger un point de terminaison d'état » (Source: www.thenetsuitepro.com) (Source: www.thenetsuitepro.com) (afin que l'interface utilisateur du navigateur ne soit pas figée).

- **Évitez les limites de débit** : Si vous appelez des points de terminaison NetSuite (Suitelets/RESTlets) depuis des SuiteScripts, utilisez `await` mais soyez vigilant : chaque appel compte toujours. Inonder les API internes peut épuiser la gouvernance. Cela rejoint la règle d'utiliser la limitation de débit (throttling) ou la logique de réessai si nécessaire.

Modèles de codage avec chaînage optionnel

- **Combiner avec l'opérateur de coalescence nulle (Nullish Coalescing)** : Pour éviter la propagation de valeurs `undefined`, un idiome courant est `const z = x?.y ?? defaultVal;`. Ainsi, si `x` ou `x.y` est manquant, `z` reçoit `defaultVal`.
- **Utilisation pour les enregistrements profondément imbriqués** : Un cas d'utilisation pratique dans SuiteScript concerne les sous-enregistrements ou les sous-listes. Par exemple, si une commande client peut ou non avoir un sous-enregistrement « shippingAddress », on pourrait écrire :

  ```
  const shipState = record.getSubrecord({fieldId: 'shippingAddress'})?.getValue('state');
  ```

  Si aucun sous-enregistrement n'existe, cela renvoie `undefined` au lieu de générer une erreur.
- **Éviter sur le chemin critique** : Pour les champs qui *doivent* toujours exister, il peut être préférable de laisser une erreur de nullité apparaître rapidement plutôt que de la masquer. Utilisez donc le chaînage optionnel uniquement là où une incertitude est attendue.
- **Données Client-Serveur** : Si un Suitelet envoie un JSON complexe à un script client, le client peut utiliser `?.` librement, car cela ne peut pas endommager les données d'enregistrement côté serveur. Cependant, les développeurs doivent toujours effectuer des vérifications de nullité si nécessaire après coup.

Le bon moment pour passer à la version 2.1

Compte tenu de toutes les fonctionnalités modernes et des recommandations d'Oracle, le conseil prédominant est d'utiliser SuiteScript 2.1 par défaut pour tout nouveau développement. Un article d'un partenaire NetSuite (Stockton Consulting) explique quand migrer les scripts existants (Source: www.stockton10.com) : si vous avez des rappels imbriqués à l'infini (callback hell), des appels API fréquents ou des problèmes de performance, passer à la version 2.1 peut simplifier le code. Ils notent que `async/await` n'accélère pas intrinsèquement les scripts, mais réduit la charge de maintenance (Source: www.stockton10.com). En pratique, les étapes de mise à niveau habituelles sont :

1. Changer l'en-tête du script en `@NApiVersion 2.1` ou `2.x` (Source: www.stockton10.com).
2. Remplacer les anciens rappels ou le code `promise`-then par `await` là où cela a du sens (Source: www.stockton10.com).
3. Ajouter des blocs `try/catch` appropriés autour des appels `await` (Source: www.stockton10.com).
4. Tester minutieusement dans un environnement sandbox (en particulier pour tout script client concernant les différences de navigateur et toute utilisation de modules restreints).
5. Déployer en production, avec un plan de retour arrière si quelque chose échoue en raison de nuances (par exemple, la version 2.1 peut être plus stricte avec les types).

Pièges clés à surveiller (d'après [68] et autres) : Rappelez-vous quels modules ne prennent pas en charge `await` – n'utilisez **pas** accidentellement `await` avec des modules non pris en charge. Après la migration, certains scripts peuvent générer des erreurs « `async/await` not allowed here », ce qui signale des appels API non pris en charge. Assurez-vous également que les budgets de gouvernance du compte sont suffisants – un appel asynchrone (comme `await search.runPaged.promise()`) consomme toujours de la gouvernance, et le débogage des piles asynchrones peut nécessiter les outils de développement Chrome (pris en charge en 2.1) (Source: docs.oracle.com).

Exemples illustratifs et études de cas

Pour étayer la discussion, nous présentons plusieurs exemples concrets et modèles d'utilisation de SuiteScript 2.1 tirés de sources communautaires. Ils servent de mini-études de cas montrant comment le chaînage optionnel et les promesses sont utilisés dans la pratique.

Modèle de confirmation de boîte de dialogue client

D'après TheNetSuitePro (octobre 2025), le modèle 1 montre un script client utilisant les promesses `N/ui/dialog` pour demander une confirmation à l'utilisateur avant l'enregistrement (Source: www.thenetsuitepro.com). Ce modèle est facilement adaptable à tout contexte SuiteScript :

```
/**
 * @NApiVersion 2.1
 * @NScriptType ClientScript
 * Modèle : Demander confirmation à l'utilisateur avant l'enregistrement
 */
define(['N/ui/dialog', 'N/currentRecord', 'N/log'], (dialog, currentRecord, log) => {
  const saveRecord = async () => {
    try {
      const confirmed = await dialog.confirm({
        title: 'Veuillez confirmer',
        message: 'Voulez-vous enregistrer cet enregistrement ?'
      });
      if (!confirmed) return false;  // l'utilisateur a annulé

      // Afficher éventuellement une autre alerte
      await dialog.alert({ title: 'Enregistrement', message: 'L\'enregistrement est en cours...' });
      return true;
    } catch (e) {
      log.error('Erreur de dialogue', e);
      return false;
    }
  };
  return { saveRecord };
});
```

Aperçu du cas : Ce code est beaucoup plus clair que l'imbrication de rappels. L'instruction `await` suspend efficacement l'exécution jusqu'à ce que l'utilisateur réponde, éliminant le besoin de plusieurs gestionnaires de rappel. Oracle souligne que les méthodes `N/ui/dialog` renvoient des promesses dans le navigateur, donc l'utilisation de `await` ici est sûre (Source: www.thenetsuitepro.com). Ce modèle montre le chaînage optionnel principalement dans la gestion des erreurs (bloc catch) plutôt que dans la navigation DOM, mais il illustre comment `async/await` peut coordonner les interactions utilisateur de manière transparente.

Modèle de récupération Suitelet <-> Client

Le modèle 2 de la même série démontre l'appel d'un Suitelet via `fetch` depuis le client (Source: www.thenetsuitepro.com). Par exemple :

```
/**
 * @NApiVersion 2.1
 * @NScriptType ClientScript
 * Modèle : Interroger un Suitelet pour obtenir des données
 */
define(['N/url', 'N/log'], (url, log) => {
  const fetchData = async () => {
    try {
      // Construire l'URL pour le Suitelet
      const suiteletUrl = url.resolveScript({
        scriptId: 'customscript_my_json_sl',
        deploymentId: 'customdeploy_my_json_sl',
        params: { action: 'getSummary' }
      });
      const response = await fetch(suiteletUrl, { method: 'GET', credentials: 'same-origin' });
      if (!response.ok) throw new Error(`HTTP ${response.status}`);
      const json = await response.json();

      // Mettre à jour un élément DOM avec le résultat
      document.getElementById('summaryBox').textContent = json.summaryText;
    } catch (e) {
      log.error('Erreur de récupération', e);
      alert('Une erreur est survenue lors de la récupération des données.');
    }
  };
  return { pageInit: fetchData };
});
```

Aperçu du cas : Ce modèle évite un rafraîchissement complet de la page en utilisant `await fetch(...)` sur le client. Il tire parti des API de navigateur modernes (fetch client) et de `await`. Le chaînage optionnel pourrait être utilisé ici si l'accès à `json.summaryText` risque d'échouer, par exemple `json?.summaryText`. Il montre également le mélange entre `url.resolveScript()` de SuiteScript pour le point de terminaison et `fetch` natif (puisque les scripts clients SuiteScript 2.1 s'exécutent dans le contexte du navigateur, les API modernes sont disponibles si le navigateur les prend en charge). Cela permet des mises à jour d'interface utilisateur réactives.

Interrogation d'un job Map/Reduce (Côté serveur)

Le modèle 3 de TheNetSuitePro illustre un Suitelet côté serveur qui soumet une tâche Map/Reduce, puis attend qu'elle se termine en interrogeant avec `await sleep()` (Source: www.thenetsuitepro.com) :

```
/**
 * @NApiVersion 2.1
 * @NScriptType Suitelet
 * Modèle : Démarrer MR puis interroger l'état
 */
define(['N/task','N/log'], (task, log) => {
  const POLL_MS = 1500;
  const MAX_TRIES = 40;
  const onRequest = async (ctx) => {
    try {
      const mrTask = task.create({ taskType: task.TaskType.MAP_REDUCE,
                                   scriptId: 'customscript_my_mr',
                                   deploymentId: 'customdeploy_my_mr' });
      const taskId = mrTask.submit();
      let status;
      for (let tries = 0; tries < MAX_TRIES; tries++) {
        status = task.checkStatus(taskId);
        if (status.status === task.TaskStatus.COMPLETE
            || status.status === task.TaskStatus.FAILED) {
          break;
        }
        await new Promise(r => setTimeout(r, POLL_MS);
      }
      ctx.response.write(`Job ${taskId} terminé avec le statut : ${status.status}`);
    } catch (e) {
      log.error('Erreur de Suitelet', e);
      ctx.response.write(`Erreur : ${e.message}`);
    }
  };
  return { onRequest };
});
```

Aperçu du cas : L'utilisation de `await sleep` dans une boucle rend le code séquentiel et lisible. Sans `await`, il faudrait une boucle de rappel avec des temporisations. La note du développeur de ce modèle souligne que cela ne rend **pas** les tâches réellement asynchrones en parallèle – le Suitelet attendra effectivement jusqu'à la fin. Pour les tâches très longues, cela peut bloquer une instance de Suitelet, donc une alternative consiste à renvoyer le `taskId` au client et à laisser le client interroger (voir Modèle 4). Néanmoins, cela démontre l'application de `async/await` côté serveur (`task.checkStatus` est synchrone, mais `await` sur `sleep` fragmente la boucle).

Interrogation via Script Client (Déchargement vers le navigateur)

Le modèle 4 déplace l'interrogation vers le client. Un script client démarre le job via un appel Suitelet, puis interroge à plusieurs reprises un point de terminaison d'état (Source: www.thenetsuitepro.com) :

```
/**
 * @NApiVersion 2.1
 * @NScriptType ClientScript
 * Modèle : Démarrer MR via Suitelet et interroger l'état
 */
define(['N/url'], (url) => {
  const runBatch = async () => {
    // 1) Démarrer le job via POST Suitelet
    const startUrl = url.resolveScript({...});
    const startRes = await fetch(startUrl, { method: 'POST', credentials: 'same-origin' });
    const { taskId } = await startRes.json();
    // 2) Interroger le Suitelet d'état
    const statusUrl = url.resolveScript({... , params:{ taskId }});
    let done = false;
    while (!done) {
      const res = await fetch(statusUrl, { credentials: 'same-origin' });
      const { status } = await res.json();
      document.getElementById('statusBox').textContent = status;
      done = (status === 'COMPLETE' || status === 'FAILED');
      if (!done) await new Promise(r => setTimeout(r, 1000);
    }
  };
  return { runBatch };
});
```

Aperçu du cas : En utilisant `await` sur `fetch`, le code client interroge élégamment sans surcharger le serveur. Ce modèle est noté comme fournissant une « progression à l'utilisateur » et gardant les serveurs à courte durée de vie (Source: www.thenetsuitepro.com). Encore une fois, très peu de rappels imbriqués – la logique de la fonction asynchrone est directe.

Automatisation des commandes client (Script planifié)

Dans une « Astuce Admin » de la communauté NetSuite (nov. 2025), Richard James Uri montre l'utilisation des promesses SuiteScript 2.1 dans un script planifié pour créer une commande client avec plusieurs articles (Source: community.oracle.com). L'extrait commence ainsi :

```
/**
 * @NApiVersion 2.1
 * @NScriptType ScheduledScript
 */
define(['N/record','N/log'], (record, log) => {
  const execute = async (context) => {
    try {
      // (trouver ou supposer un ID client)
      const salesOrder = record.create({ type: record.Type.SALES_ORDER, isDynamic: true });
      salesOrder.setValue({ fieldId: 'entity', value: 123 });
      // Ajouter des articles de ligne de manière asynchrone
      for (const itemData of itemsToAdd) {
        salesOrder.selectNewLine({ sublistId: 'item' });
        salesOrder.setCurrentSublistValue({ sublistId: 'item', fieldId: 'item', value: itemData.id });
        salesOrder.setCurrentSublistValue({ sublistId: 'item', fieldId: 'quantity', value: itemData.qty });
        await salesOrder.commitLine({ sublistId: 'item' });
      }
      const soId = await salesOrder.save();
      log.debug('Commande client créée', `ID SO : ${soId}`);
    } catch (e) {
```

log.error('Erreur lors de la création de la commande client', e); } }; return { execute }; });

```
*Aperçu du cas* : Cet exemple (intégré dans [57]) utilise `await` sur `salesOrder.save()`. Dans SuiteScript 2.1, la métho

# Discussion sur les implications et les orientations futures

La disponibilité du chaînage optionnel et des promesses dans SuiteScript 2.1 a plusieurs implications importantes pour le

- **Amélioration de la productivité et de la maintenabilité pour les développeurs** : De nombreuses sources expertes soul
- **Stratégie de rétrocompatibilité** : Étant donné que SuiteScript 2.x dispose d'environnements d'exécution distincts, O
- **Intégration avec l'écosystème JavaScript tiers** : Oracle a activement permis l'utilisation de bibliothèques JS moder
- **Performance et gouvernance** : Les nouvelles fonctionnalités ne modifient pas en elles-mêmes les limites de gouvernan
- **Contexte technologique plus large** : La modernisation de SuiteScript reflète les tendances du développement JavaScri
- **Évolution des cas d'utilisation** : Pour l'avenir, nous prévoyons davantage d'études de cas sur l'utilisation de la v
- **Ressources communautaires et apprentissage** : Enfin, l'abondance d'articles de blog, de réponses wiki et de fils de

En résumé, les fonctionnalités JS modernes de SuiteScript 2.1 constituent une étape charnière dans l'évolution de la plat

# Conclusion

L'adoption par SuiteScript 2.1 de constructions JavaScript modernes telles que le chaînage optionnel (`?.`), l'opérateur

Points clés à retenir :

- **Support et portée** : SuiteScript 2.1 (via Graal) prend en charge pratiquement toutes les fonctionnalités du langage
- **Avantages** : Ces fonctionnalités améliorent considérablement la lisibilité et la maintenabilité du code. Dans nos ex
- **Meilleures pratiques** : Nous avons confirmé les meilleures pratiques d'Oracle et ajouté les conseils de la communaut
- **Travaux futurs** : L'environnement SuiteScript ne fera que devenir plus moderne. L'annotation « 2.x » et les prochain

Cette enquête approfondie démontre que SuiteScript 2.1 permet aux développeurs d'utiliser les **meilleures pratiques Java
```

Étiquettes: suitescript-21, javascript-moderne, developpement-netsuite, chainage-optionnel, promesses-javascript, async-await, graalvm, ecmascript