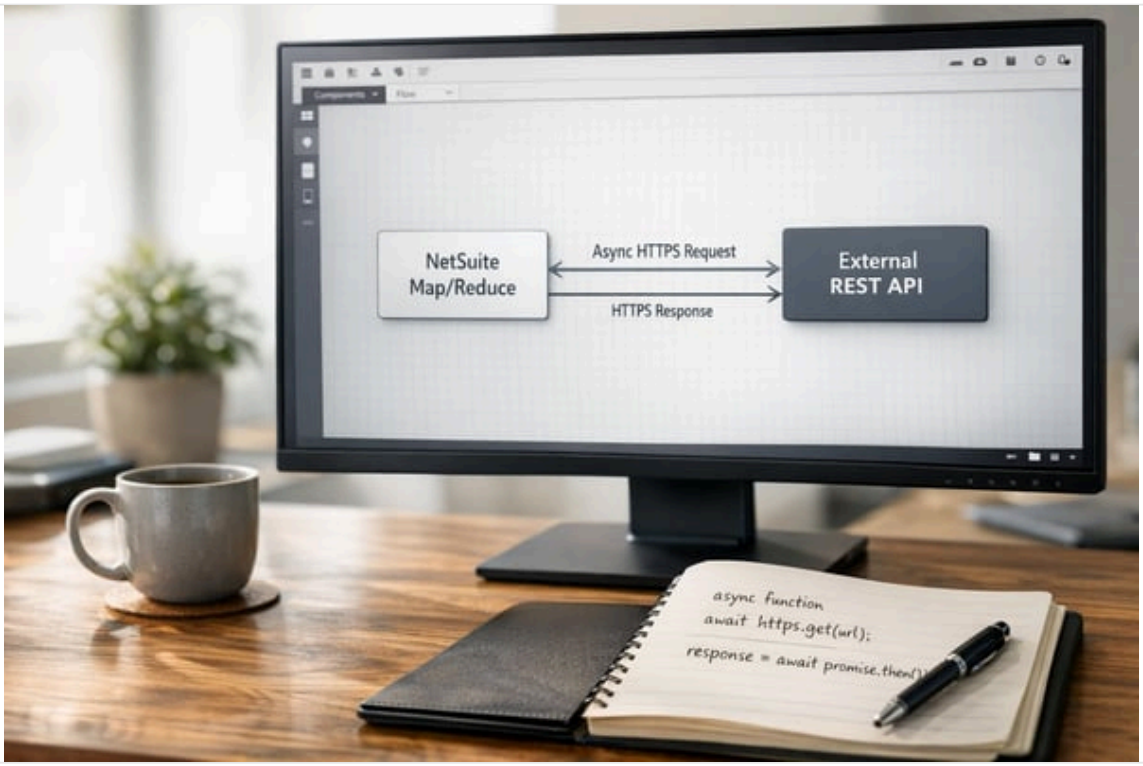
Promesses HTTPS SuiteScript 2.1 : Modèles asynchrones Map/Reduce

By houseblend.io Publié le 19 avril 2026 30 min de lecture



Résumé analytique

SuiteScript 2.1 représente une modernisation majeure de la plateforme de script de NetSuite, ajoutant des fonctionnalités ES2019+ (async/await, Promises, etc.) au code SuiteScript 2.0 précédemment synchrone (Source: www.houseblend.io). En particulier, le module `N/https` en 2.1 expose des méthodes basées sur les promesses (par exemple `https.get.promise(options)`) qui renvoient des Promesses JavaScript pour les appels HTTP asynchrones, en plus des API synchrones héritées (Source: www.houseblend.io) (Source: docs.oracle.com). La documentation officielle de NetSuite confirme que les modules côté serveur clés (tels que `N/https`, `N/http` / `N/https`, `N/search`, `N/query`, `N/transaction`, etc.) prennent désormais en charge les méthodes renvoyant des promesses, permettant des flux asynchrones de type `await` (Source: docs.oracle.com) (Source: www.houseblend.io). Cela permet à un développeur SuiteScript d'appeler des points de terminaison REST, des recherches enregistrées et d'autres API externes sans rappels (callbacks) profondément imbriqués – par exemple, on peut écrire dans un contexte map/reduce :

```
const resp = await https.get.promise({ url: 'https://api.example.com/data' });
const data = JSON.parse(resp.body);
```

comme illustré dans un exemple SuiteScript 2.1 (Source: www.houseblend.io).

Dans les scripts Map/Reduce, qui s'exécutent par étapes parallèles sur les serveurs de NetSuite, ces modèles asynchrones doivent être utilisés avec précaution. Chaque appel `https` (synchrone ou basé sur des promesses) consomme toujours 10 unités de gouvernance (Source: docs.oracle.com) (Source: docs.oracle.com), et Oracle avertit que les promesses côté serveur « ne sont pas destinées aux cas d'utilisation de traitement par lots » (Source: docs.oracle.com). Les meilleures pratiques (issues d'Oracle et de sources communautaires) soulignent l'utilisation de `async/await` au lieu de `.then()` chaînés, la gestion des erreurs avec `try/catch` et `.catch()`, et le contrôle de la concurrence. Par exemple, il faut éviter de lancer des milliers d'appels parallèles à la fois (en utilisant des outils comme `BluebirdPromise.map` avec des limites de concurrence) (Source:

www.houseblend.io). Un modèle courant est « planifier d'abord, attendre plus tard » : par exemple, utiliser `task.create().submit()` dans un Suitelet pour démarrer un ou plusieurs travaux Map/Reduce, puis boucler avec `await sleep()` et `task.checkStatus()` pour interroger leur achèvement (Source: www.houseblend.io) (Source: www.houseblend.io).

Les mesures de performance indiquent que même avec du code asynchrone, le débit de Map/Reduce reste relativement modeste : les rapports communautaires montrent un traitement de l'ordre de seulement 2 à 3 enregistrements par seconde en pratique, même avec un parallélisme modéré (Source: archive.netsuiteprofessionals.com) (Source: archive.netsuiteprofessionals.com). Cela concorde avec l'analyse de Stockton10 selon laquelle SuiteScript 2.1 n'est *« pas significativement »* plus rapide que la version 2.0 (Source: www.stockton10.com) ; les gains se situent au niveau de la clarté et de la maintenabilité du code, et non de la vitesse brute. Notre analyse détaillée des tables de gouvernance de NetSuite confirme que les méthodes asynchrones comportent le même coût d'utilisation que leurs homologues synchrones (par exemple, `https.get` et `https.get.promise` utilisent tous deux 10 unités (Source: docs.oracle.com), donc l'utilisation de promesses ne réduit pas la consommation d'unités. Cependant, la syntaxe `await` plus propre simplifie l'écriture d'une logique Map/Reduce complexe (par exemple, l'agrégation des résultats d'API ou la combinaison de données de recherche) et rend la gestion des erreurs plus directe (Source: www.houseblend.io) (Source: www.stockton10.com).

Ce rapport fournit un examen exhaustif des capacités HTTPS et Promise de SuiteScript 2.1 au sein des scripts Map/Reduce. Nous passons en revue l'évolution historique de SuiteScript 2.0 à 2.1, étudions les nouvelles méthodes N/https et leur utilisation, détaillons les modèles asynchrones courants (y compris les pièges), analysons les données de gouvernance et de débit, et présentons des exemples concrets et des meilleures pratiques. Nous discutons également des implications plus larges et des orientations futures (telles que les polyfills Node.js et le développement assisté par IA. Chaque affirmation est étayée par la documentation officielle de NetSuite et des sources expertes.

Introduction et contexte

SuiteScript est la plateforme d'API JavaScript de NetSuite pour personnaliser et automatiser le système ERP/CRM NetSuite (Source: www.houseblend.io). Le premier SuiteScript (1.0) remonte à ~2007 et a finalement été remplacé par SuiteScript 2.0 en 2015, qui a introduit un système de modules de type AMD `define([...], ...)` mais fonctionnait toujours sur un JavaScript plus ancien de l'ère ES5. SuiteScript 2.1 (publié en 2021) fonctionne sur un moteur GraalVM/V8 et prend en charge les fonctionnalités ECMAScript modernes jusqu'à ES2023 (Source: www.houseblend.io) (Source: www.stockton10.com). Les nouvelles fonctionnalités linguistiques clés de la version 2.1 incluent `let` / `const` à portée de bloc, les fonctions fléchées, les littéraux de gabarit et, surtout, les promesses natives et les mots-clés `async/await` (Source: www.stockton10.com). Comme le note un expert, « SuiteScript 2.0 n'a pas de support asynchrone natif... 2.1 prend en charge les promesses et async/await » (Source: www.stockton10.com) (Source: www.stockton10.com). Cette modernisation vise à améliorer la lisibilité et la maintenabilité du code – par exemple, le code de « pyramide de la mort » avec rappels imbriqués en 2.0 peut être remplacé par des flux linéaires `async/await` en 2.1 (Source: www.stockton10.com) (Source: www.stockton10.com).

Le code SuiteScript est déployé dans divers **types de scripts** (Client, Événement utilisateur, Planifié, Suitelet, Restlet, etc.), chacun avec son propre contexte d'exécution. Il est important de noter que le type de script **Map/Reduce** (introduit en 2016) est conçu pour traiter de grandes quantités de données en parallèle (Source: docs.oracle.com) (Source: www.thenetsuitepro.com). Un script Map/Reduce se compose de quatre points d'entrée : *getInputData*, *map*, *reduce* et *summarize* (Source: medium.com) (Source: docs.oracle.com). Lors de l'exécution, NetSuite divise automatiquement les données d'entrée en plusieurs morceaux et exécute des tâches « map » parallèles sur ceux-ci ; les résultats de l'étape map sont ensuite regroupés et traités par des tâches « reduce » ; enfin, une étape *summarize* peut signaler les résultats globaux et les erreurs. Cette architecture (souvent décrite comme le « cheval de bataille de l'évolutivité » de NetSuite (Source: www.thenetsuitepro.com) (Source: medium.com) permet à Map/Reduce de gérer des millions d'enregistrements en toute sécurité, avec une gouvernance intégrée qui cède si les limites sont atteintes (Source: docs.oracle.com) (Source: www.thenetsuitepro.com).

Module N/https : Dans SuiteScript 2.x, le module `N/https` est utilisé pour effectuer des requêtes HTTPS sortantes. (Il « encapsule toutes les fonctionnalités de N/http » mais uniquement en SSL/TLS (Source: docs.oracle.com).) Via `N/https`, les scripts peuvent appeler des API REST externes ou d'autres services Web. Avant la version 2.1, ces appels étaient purement synchrones (par exemple, `https.get(options)` renvoyait la réponse directement). En 2.1, NetSuite a ajouté des variantes basées sur les promesses de toutes les méthodes principales (par exemple, `https.get.promise(options)`, `https.post.promise`, etc.) (Source: www.houseblend.io) (Source: docs.oracle.com). La documentation officielle sur `https.get.promise` note explicitement qu'elle « envoie une requête HTTPS GET de manière asynchrone » et *renvoie un objet Promise*, avec les mêmes paramètres et erreurs que la version synchrone (Source: docs.oracle.com). Les méthodes synchrones et basées sur les promesses consomment la même gouvernance (10 unités par appel) (Source: docs.oracle.com) (Source: docs.oracle.com). Surtout, le moteur 2.1 permet d'utiliser `await` sur ces méthodes renvoyant des promesses, permettant un contrôle de flux asynchrone plus propre dans les scripts NetSuite.

Ce rapport se concentre sur la façon dont ces fonctionnalités de SuiteScript 2.1 (en particulier les promesses HTTPS) peuvent être utilisées dans les scripts Map/Reduce. Nous couvrirons les détails techniques de l'API `N/https`, comment structurer le code asynchrone dans une fonction Map ou Reduce, et les compromis pratiques impliqués. Nous nous appuierons sur la documentation officielle de NetSuite, les guides/blogs de développeurs et des expériences réelles pour fournir une analyse approfondie et fondée sur des preuves.

SuiteScript 2.0 vs 2.1 : Fonctionnalités JavaScript modernes

Différences de langage et de syntaxe. SuiteScript 2.0 utilise uniquement la syntaxe JavaScript ES5 (de l'ère 2015). Il ne prend *pas* en charge les constructions modernes telles que les classes, les fonctions fléchées ou `async/await` (Source: www.stockton10.com) (Source: www.stockton10.com). En revanche, SuiteScript 2.1 utilise ES6+ et autorise les fonctionnalités ES2019+ (Source: www.stockton10.com). Par exemple, en 2.1, on peut utiliser `async function`, `await`, les fonctions fléchées, `const` / `let`, les littéraux de gabarit, le chaînage optionnel (`?.`) et la coalescence nulle (`??`) – qui sont tous interdits en 2.0 (Source: www.houseblend.io) (Source: www.stockton10.com). Comme le note un résumé d'expert, « 2.0 utilise ES5... aucune fonctionnalité moderne », alors que « 2.1 prend en charge les fonctionnalités ES6+ comme async/await, les fonctions fléchées et les littéraux de gabarit » (Source: www.stockton10.com). Cela s'aligne avec les conseils officiels d'Oracle (le runtime GraalVM 2.1 prend en charge les fonctionnalités ECMAScript 2023 côté serveur (Source: www.houseblend.io).

Promesses natives et Async/Await. La différence la plus significative pour le codage asynchrone est que SuiteScript 2.1 **prend en charge nativement les promesses et `async/await`**, alors que la version 2.0 ne le fait pas. En 2.0, toute opération non bloquante devait être effectuée avec un rappel ou en divisant manuellement les scripts. En 2.1, certaines méthodes d'API renvoient des promesses, et les développeurs peuvent déclarer des fonctions asynchrones et utiliser `await`, ce qui permet de lire le code de haut en bas. Par exemple, un script 2.0 devrait imbriquer des rappels comme :

```
https.get(options, function(resp) {
  https.get(otherOptions, function(resp2) {
    // rappel imbriqué...
  });
});
```

En 2.1, cela peut s'écrire ainsi :

```
try {
  const resp = await https.get.promise(options);
  const resp2 = await https.get.promise(otherOptions);
  // structure de code linéaire
} catch (e) {
  // gestion unifiée des erreurs
}
```

Les experts soulignent que async/await « se lit de haut en bas » et permet une gestion des erreurs en un seul endroit, comme le note un blogueur : « les rappels imbriqués sont difficiles à lire... SuiteScript 2.1 prend en charge les promesses et async/await... tout le monde peut comprendre » (Source: www.stockton10.com). Stockton10 qualifie également les promesses/async de « fonctionnalité phare » de la version 2.1 (Source: www.stockton10.com). (Stockton10 confirme en outre que la vitesse de la version 2.1 n'est *pas* significativement supérieure à celle de la version 2.0 (Source: www.stockton10.com) ; l'avantage réside dans la clarté du code.)

API asynchrones prises en charge (modules). Il est important de noter que seuls *certains modules* de SuiteScript 2.1 sont asynchrones. La documentation d'Oracle sur les « Promesses côté serveur asynchrones » répertorie explicitement les modules pris en charge : **`N/http` / `N/https`, `N/query`, `N/search` et `N/transaction`** (Source: docs.oracle.com). Seules les méthodes de ces modules ont des variantes `.promise()` qui fonctionnent avec `await`. (Dans d'autres modules, appeler `await` sur une méthode de style 2.0 n'a aucun effet.) Par exemple, `N/https` a gagné `.get.promise()`, `.post.promise()`, etc., et `N/search` a gagné `runPaged.promise()` (Source: www.houseblend.io) (Source: www.houseblend.io). La documentation sur l'objet Promise de SuiteScript énumère ces méthodes de promesse dans l'aide. En effet, si vous avez besoin d'un comportement asynchrone dans SuiteScript 2.1, vous êtes limité à ces API : par exemple, les appels HTTP, les requêtes SuiteQL ou de recherche enregistrée, ou certains appels de transaction (annuler, transformer, etc.). Cette orientation correspond aux cas d'utilisation réels tels que l'appel d'API externes ou l'exécution de requêtes de données volumineuses.

Le *Tableau 1* ci-dessous résume les unités de gouvernance et la disponibilité des méthodes clés liées à HTTP et aux données dans SuiteScript 2.x :

MÉTHODE	SYNCHRONE (2.0)	PROMESSE (2.1)	UNITÉS D'UTILISATION (ENREG. TRANS.)	NOTES (2.1)

| `https.get(...)` | Oui (requête GET synchrone) | Oui (async `get.promise`) | 10 | Renvoie une promesse de `ServerResponse` (Source: docs.oracle.com). | | `https.post(...)` | Oui (requête POST synchrone) | Oui (async `post.promise`) | 10 | Utilisation identique ; méthode POST asynchrone analogue. | | `https.put(...)`, `delete` | Oui | Oui (avec `.promise`) | 10 chacune | PUT et DELETE disposent de versions `.promise()` (10 unités). | | `https.request(...)` | Oui | Oui (`request.promise`) | 10 | Envoie une requête HTTPS arbitraire (ex. RESTlet) – méthode promise disponible. | | `search.runPaged()` | Oui | Oui (`runPaged.promise`) | 5 | Exécute une recherche paginée (5 unités, même pour une promesse) (Source: docs.oracle.com). | | `search.run()` | Oui | – | 0 | `run()` renvoie un `Iterator` (pas de méthode promise) ; aucune unité consommée. | | `search.save()` | Oui | Oui (`save.promise`) | 5 | L'enregistrement d'une définition de recherche coûte 5 unités (avec ou sans promesse). (Source: docs.oracle.com) | | `record.load()` | Oui | Oui (`load.promise`) | 10 (enr. trans.) | Le chargement d'un enregistrement coûte toujours 10 unités pour les transactions (varie pour les enregistrements personnalisés) (Source: docs.oracle.com). | | `transaction.void()` | Oui | Oui (`void.promise`) | 10 | Annulation d'une transaction (ex. annuler) – 10 unités (Source: docs.oracle.com). |

Tableau 1. Méthodes API sélectionnées dans SuiteScript 2.x, montrant les variantes synchrones vs promesses et les coûts en unités de gouvernance. (La documentation Oracle est la source pour les unités de gouvernance (Source: docs.oracle.com) (Source: docs.oracle.com) ; les méthodes asynchrones sont nouvelles en 2.1 (Source: docs.oracle.com) (Source: docs.oracle.com).)

Notamment, les méthodes basées sur les promesses consomment exactement la même gouvernance que les appels originaux (10 unités par appel HTTP) (Source: docs.oracle.com). Ainsi, l'utilisation de `await https.get.promise()` est fonctionnellement identique en termes de coût à l'utilisation de `https.get()` de manière synchrone. L'avantage réside purement dans le style de codage et le contrôle de l'exécution, et non dans des économies de gouvernance. Par exemple, la documentation officielle pour `https.get.promise(options)` indique explicitement « Gouvernance : 10 unités » (identique à `https.get(options)`) (Source: docs.oracle.com).

Le module N/https et les appels HTTP asynchrones

Le module **N/https** est central pour les intégrations avec des systèmes externes. Il fournit des méthodes pour envoyer des requêtes GET, POST, PUT et DELETE via HTTPS. Dans SuiteScript 2.1, l'API de ce module est augmentée avec des méthodes renvoyant des promesses. Selon la documentation de NetSuite, le module `N/https` « encapsule toutes les fonctionnalités du module N/http » et « vous pouvez effectuer des appels HTTPS à partir de scripts client et serveur » (Source: docs.oracle.com). En pratique, un script SuiteScript peut utiliser `https.get()` pour récupérer une URL (de manière synchrone) ou utiliser `https.get.promise()` pour la récupérer de manière asynchrone.

Par exemple, la page d'aide SuiteScript pour `https.get.promise(options)` le décrit comme suit : *« Envoie une requête HTTPS GET de manière asynchrone. Remarque : Les paramètres et les erreurs générées pour cette méthode sont les mêmes que ceux pour https.get(options). »* (Source: docs.oracle.com). Elle renvoie une promesse JavaScript de la réponse, permettant l'utilisation de `.then()` ou `await` pour gérer le résultat. La même page note que `https.get.promise` est disponible à la fois dans les scripts client et serveur, et consomme 10 unités de gouvernance (Source: docs.oracle.com). Des pages de documentation analogues existent pour `https.post.promise`, `https.put.promise`, etc. (voir [60]).

D'un point de vue pratique, l'utilisation de `N/https` dans un script Map/Reduce signifie généralement effectuer un appel API externe pour chaque enregistrement d'entrée. Par exemple, une fonction Map pourrait rechercher des données auprès d'un service tiers :

```
define(['N/https','N/log'], (https, log) => {
  const map = async (context) => {
    const recordId = JSON.parse(context.value).id;
    try {
      // Appel HTTP asynchrone vers une API externe
      let response = await https.get.promise({
        url: 'https://api.example.com/data/' + recordId
      });
      let data = JSON.parse(response.body);
      log.debug('Récupéré pour '+recordId, data);
      // Traiter les données ou écrire le résultat...
      context.write({key: recordId, value: data});
    } catch (err) {
      log.error('Erreur HTTP', err);
    }
  };
});
```

Ce code (adapté des principes dans [41]) démontre le nouveau modèle : une fonction map `async` utilisant `await https.get.promise(...)`. Avant la version 2.1, il aurait fallu utiliser un rappel (callback) ou des scripts planifiés séparés pour obtenir le même résultat. L'article de Houseblend sur l'exemple de script planifié (Source: www.houseblend.io) montre le même modèle (il attend `https.get.promise` à l'intérieur d'une fonction asynchrone), confirmant que ce style est officiellement pris en charge.

Comme chaque requête HTTPS coûte 10 unités, les appels externes en masse peuvent rapidement alourdir la consommation. Dans un contexte Map/Reduce, si 1000 enregistrements déclenchent chacun un appel HTTPS, cela utiliserait à lui seul 10 000 unités. Cependant, les scripts Map/Reduce bénéficient de la mise en pause (yielding) : s'ils atteignent les limites de gouvernance, le framework suspendra et redémarrera automatiquement le travail si nécessaire (Source: docs.oracle.com). Néanmoins, les développeurs doivent concevoir leurs solutions avec soin. Par exemple, il est conseillé de récupérer les données externes par lots de taille raisonnable, plutôt que de lancer des milliers d'appels parallèles. Les conseils d'Oracle (via ses blogs de support SuiteCloud) et les experts de la communauté réitèrent qu'il ne faut pas lancer « des milliers d'appels parallèles à la fois dans SuiteScript, car cela peut dépasser les limites » (Source: www.houseblend.io). Dans cette optique, certains développeurs utilisent des techniques comme `Promise.map` de Bluebird (avec des limites de concurrence) ou le séquençage manuel pour réguler les requêtes (Source: www.houseblend.io).

Promesses et modèles asynchrones dans SuiteScript 2.1

SuiteScript 2.1 introduit de véritables promesses JavaScript dans la plateforme. L'objet *Promise* et la syntaxe `async` / `await` fonctionnent pour les modules pris en charge, permettant un code asynchrone plus idiomatique. Officiellement, « SuiteScript 2.1 prend entièrement en charge les promesses asynchrones non bloquantes côté serveur exprimées à l'aide des mots-clés `async`, `await` et `promise` pour un sous-ensemble de modules : N/http, N/https, N/query, N/search et N/transaction » (Source: docs.oracle.com). L'aide SuiteScript recommande de les utiliser dans des « opérations en cours et distinctes » plutôt que dans des boucles de traitement par lots massives (Source: docs.oracle.com).

Il est conseillé aux développeurs de suivre les meilleures pratiques standard pour le code basé sur les promesses. La documentation d'Oracle et les sources communautaires insistent sur l'utilisation de `async/await` au lieu de chaînes `.then()` manuelles, sur l'encapsulation de la logique dans des blocs `try/catch` et sur la gestion explicite des rejets (Source: www.houseblend.io). Par exemple, au lieu d'écrire :

```
https.get.promise(opts)
  .then(resp => { /*...*/ })
  .then(...).catch(err => { /*...*/ });
```

il est plus propre d'écrire :

```
try {
  let resp = await https.get.promise(opts);
  // traiter la réponse...
} catch(e) {
  // gérer l'erreur...
}
```

et c'est le style recommandé (Source: www.houseblend.io) (Source: www.stockton10.com). L'enquête de Houseblend souligne exactement ceci : « *Les conseils sur les meilleures pratiques (d'Oracle et de la communauté) recommandent fortement d'utiliser async/await au lieu de chaînes .then manuelles, de toujours gérer les erreurs (par exemple avec try/catch et .catch(), et d'éviter les promesses imbriquées* » (Source: www.houseblend.io). En bref, il faut traiter les promesses SuiteScript comme des promesses JavaScript normales, mais en étant conscient de la gouvernance et de la journalisation de NetSuite.

Quelques modèles spécifiques sont à noter dans le contexte Map/Reduce :

- **Appels parallèles vs séquentiels :** Si vous devez effectuer plusieurs appels HTTP liés pour un enregistrement, vous pouvez utiliser `Promise.all()` ou similaire pour les exécuter en parallèle. Cependant, comme indiqué, ne faites pas exploser la concurrence. Par exemple, si chaque enregistrement nécessite # appels, vous pourriez écrire `let results = await Promise.all([call1, call2, ...])`. Si le nombre d'appels parallèles par enregistrement est faible, cela peut faire gagner du temps. S'il est important, envisagez le traitement par lots ou la division en tâches map supplémentaires.
- **`Promise.map` (Bluebird) :** Les notes de version et le support d'Oracle ont laissé entendre que les méthodes utilitaires de Bluebird (comme `Promise.map`) pourraient être disponibles (Source: www.houseblend.io). Cela peut aider à contrôler les limites de concurrence. Par exemple : `await Promise.map(recordIds, async id => { return https.get.promise({url:apiUrl+id}); }, {concurrency: 5})` ne traiterait que 5 éléments à la fois.
- **Gestion des erreurs dans les promesses :** Attachez toujours `.catch()` ou utilisez `try/catch` autour de `await`. Les rejets de promesses non interceptés pourraient terminer la tâche de manière inattendue. Utilisez l'étape *summarize* de Map/Reduce (`summarize.batchSummary.errors`) pour journaliser tous les enregistrements qui ont échoué en raison d'erreurs.
- **Interrogation (Polling) de tâches longue durée :** Un modèle asynchrone courant consiste à lancer une tâche longue, puis à vérifier périodiquement son statut. Par exemple, un gestionnaire `onRequest` de Suitelet pourrait utiliser `N/task` pour soumettre un script Map/Reduce, puis boucler avec `await sleep(ms)` pour interroger son statut (Source: www.houseblend.io) (Source: www.houseblend.io). Houseblend fournit un exemple de « Modèle 3 » : un Suitelet qui fait `task.create({taskType: MAP_REDUCE, ...}).submit()`, puis dans une boucle `for` appelle `task.checkStatus(taskId)` ; si ce n'est pas terminé, il fait `await new Promise(r => setTimeout(r, POLL_MS)` (Source: www.houseblend.io). Cette technique `await load-sleep` (utilisant une promesse résolue pour `setTimeout`) rend le code de la boucle d'interrogation séquentiel et lisible, au prix de l'immobilisation de l'instance du Suitelet jusqu'à la fin (Source: www.houseblend.io). En pratique, il faut se prémunir contre les interrogations très longues (le code ci-dessus limite à `MAX_TRIES` itérations). Un modèle alternatif (« Modèle 4 ») consiste à renvoyer l'ID de la tâche à un client et à laisser le navigateur interroger via AJAX, ce qui libère le thread serveur (Source: www.houseblend.io).
- **Finally/Nettoyage :** Les blocs `finally` des promesses peuvent être utilisés pour le nettoyage. Par exemple, une fois que toutes les tâches map sont terminées ou que tous les appels HTTP sont finis, vous pourriez vouloir fermer un fichier ouvert ou libérer un verrou externe. Houseblend note que vous devriez utiliser `promise.finally` ou un bloc `finally` avec `await` pour exécuter le code de nettoyage dans les cas de succès ou d'erreur (Source: www.houseblend.io).

Dans l'ensemble, la nouvelle prise en charge des promesses dans SuiteScript 2.1 permet l'utilisation de nombreux idiomes JavaScript asynchrones familiers. Cependant, les développeurs doivent toujours garder à l'esprit le contexte NetSuite : chaque `await` ne met en pause que *ce thread côté serveur* (pas tout le compte), et les limites de gouvernance s'appliquent toujours comme d'habitude. En particulier, le guide officiel avertit : « *Cette capacité n'est pas destinée aux cas d'utilisation de traitement en masse où une solution hors bande... peut suffire* » (Source: docs.oracle.com). En d'autres termes, pour des flux de travail en arrière-plan vraiment massifs, on pourrait toujours préférer les files d'attente de travail natives (ex. mises à jour de masse, files d'attente tierces ou planification répétée) à une seule boucle asynchrone énorme.

Modèles asynchrones dans les scripts Map/Reduce

Dans un script **Map/Reduce**, les points d'entrée `map` et `reduce` peuvent être déclarés comme des fonctions `async` dans SuiteScript 2.1. Cela signifie que vous pouvez utiliser `await` à l'intérieur. (Le framework de script gérera automatiquement les promesses renvoyées.) C'est une meilleure pratique que de renvoyer des promesses manuellement. Par exemple :

```
/**
 * @NApiVersion 2.1
 * @NScriptType MapReduceScript
 */
define(['N/search','N/https','N/log'], (search, https, log) => {
  const map = async (context) => {
    try {
      // getInputData fournit context.value -> data, ex. {id:123}
      let rec = JSON.parse(context.value);
      const resp = await https.get.promise({ url: 'https://api.service/data/' + rec.id });
      const data = JSON.parse(resp.body);
      // écrire les données pour le regroupement reduce
      context.write({ key: rec.groupId, value: JSON.stringify(data) });
    } catch (e) {
      log.error('Erreur Map', e);
      // ne pas lancer d'exception ; laisser le travail continuer, les erreurs seront dans le résumé
    }
  };
  const reduce = async (context) => {
    // context.key = groupId, context.values = tableau de chaînes JSON depuis map
    try {
      let combined = context.values.map(v => JSON.parse(v);
      // par exemple, sommer les valeurs, ou combiner les enregistrements
      log.debug('Reduce ' + context.key, 'Combinaison de '+combined.length+' enregistrements');
      // Enregistrer les résultats ou effectuer des calculs...
    } catch (e) {
      log.error('Erreur Reduce', e);
    }
  };
  const summarize = (summary) => {
    let errors = summary.mapSummary.errors.iterator();
    errors.each((key, err) => {
      log.error('Clé d\'erreur ' + key, err);
      return true;
    });
  };
  return { getInputData, map, reduce, summarize };
});
```

Le pseudocode ci-dessus illustre des points clés : la fonction `map` est `async`, elle utilise `await https.get.promise()`, et elle appelle `context.write()` une fois que l'appel asynchrone a renvoyé un résultat. La fonction `reduce` agrège les valeurs par clé (ici, en journalisant simplement le nombre). La gestion des erreurs utilise `try/catch` afin qu'une défaillance ne bloque pas l'ensemble du travail Map/Reduce (les enregistrements ayant échoué apparaîtront dans `summarize.mapSummary.errors`). Ce modèle (try/catch dans chaque étape) est explicitement recommandé pour éviter qu'un mauvais enregistrement ne tue le travail (Source: www.thenetsuitepro.com).

Quelques observations spécifiques à Map/Reduce :

- **Concurrence :** Le framework Map/Reduce exécute déjà plusieurs tâches map (et reduce) en parallèle, selon le paramètre de concurrence du déploiement du script. Si vous effectuez des appels asynchrones *à l'intérieur* de chaque map, ces appels eux-mêmes sont attendus en série (sauf si vous utilisez une concurrence de promesses supplémentaire comme `Promise.all`). Par exemple, dans le code ci-dessus, chaque tâche map gère son enregistrement l'un après l'autre avec `await`. Si vous le souhaitez, vous pouvez récupérer plusieurs URL par enregistrement en parallèle en utilisant `Promise.all([...])` comme mentionné précédemment. Mais vous devez équilibrer cela avec la gouvernance : même au sein d'une tâche map, paralléliser trop peut faire grimper rapidement les unités de consommation. Comme le prévient un expert de la communauté, *« ne lancez pas des milliers d'appels parallèles à la fois dans SuiteScript »* (Source: www.houseblend.io).

- **Traitement par lots (Batching) :** Si un appel externe peut traiter plusieurs ID à la fois, envisagez de regrouper l'appel. Par exemple, au lieu d'appeler 1000 GET individuels pour des expéditions, appelez une API de lot avec les 1000 ID si possible. `getInputData` de NetSuite peut être écrit pour découper les données par dizaines ou centaines par invocation de map. Houseblend note que vous devriez *« Toujours récupérer les enregistrements par blocs, et non l'ensemble du jeu de données, pour éviter les pics de gouvernance »* (Source: www.thenetsuitepro.com). De même, vous devriez réguler vos appels HTTPS. Par exemple, on pourrait écrire une boucle dans `getInputData` qui ne renvoie que 100 ID à la fois, en exécutant l'étape Map plusieurs fois si nécessaire.

- **Partage de données entre les étapes :** Les données transmises via `context.write()` de l'étape map vers l'étape reduce doivent être sérialisées (format JSON) ou être des types primitifs ; ne transmettez que ce qui est nécessaire à la clé/valeur de l'étape reduce. Dans le code ci-dessus, nous envoyons `value: JSON.stringify(data)`. L'étape reduce reçoit ensuite `context.values` sous forme de tableau contenant ces chaînes JSON pour cette clé. Le tutoriel de NetsuitePro sur le Map/Reduce avancé illustre exactement ce modèle : écrire `{key: customerId, value: salesOrderId}` dans l'étape map, puis gérer dans l'étape reduce des tableaux de valeurs pour chaque client (Source: www.thenetsuitepro.com) . (Toute donnée externe volumineuse doit généralement être enregistrée dans un enregistrement ou un fichier, et non transportée via le contexte.)

- **Considérations sur `context.write()` :** N'oubliez pas que `context.write()` lui-même est soumis à des limites d'utilisation et de taille. Chaque appel à `write()` consomme 1 unité d'utilisation en plus de la mémoire. Le passage de valeurs extrêmement volumineuses ou de nombreuses petites écritures peut augmenter la consommation de mémoire. Dans la mesure du possible, écrivez des clés/valeurs minimales et effectuez les traitements lourds en dehors du Map/Reduce (par exemple, via des recherches enregistrées ou des systèmes externes).

- **Planification des tâches :** Un autre modèle courant dans le SuiteScript asynchrone consiste à utiliser le module `N/task` pour enchaîner les travaux. Par exemple, une fonction map (ou, plus couramment, un Suitelet) peut utiliser `task.create({ taskType: task.TaskType.MAP_REDUCE })` pour lancer un autre Map/Reduce de manière asynchrone (Source: www.houseblend.io). Il est ensuite possible d'utiliser `await` ou d'interroger (polling) le statut de la nouvelle tâche. Cela peut s'avérer utile pour diviser un flux de travail très volumineux en plusieurs phases. (L'exemple de polling [53] montre exactement cela : un Suitelet appelle `task.submit()`, puis boucle jusqu'à l'achèvement.)

Dans tous les modèles asynchrones, gardez à l'esprit que **SuiteScript s'exécute toujours sur un serveur V8 monothread par tâche**. L'utilisation de `await` ne génère pas de threads en arrière-plan au-delà de la concurrence habituelle des tâches map. En d'autres termes, marquer une fonction comme `async` et utiliser `await` rend le contrôle au moteur NetSuite jusqu'à ce que l'appel attendu soit terminé. Comme l'a noté un développeur : *« Les scripts attendent toujours la promesse avant de s'arrêter. Votre fonction peut renvoyer un résultat plus rapidement, mais le moteur de script doit toujours continuer à tourner pour attendre. »* (Source: archive.netsuiteprofessionals.com). Cela signifie que bien que votre code soit plus simple, la tâche Map/Reduce dans son ensemble occupera toujours du temps de traitement pendant l'exécution de `await`. Cela ne libère *pas* d'unités d'exécution de manière globale.

Analyse des performances et de la gouvernance

L'utilisation d'appels HTTPS asynchrones dans un Map/Reduce a plusieurs implications sur les performances. La plus évidente concerne la consommation de gouvernance : chaque appel externe est coûteux. Le tableau 1 ci-dessus montre que chaque `https.get()` ou `https.get.promise()` coûte 10 unités (Source: docs.oracle.com). Ainsi, l'utilisation totale pour les appels externes est de `10 × (nombre d'appels)`. Un travail modéré avec 1 000 enregistrements effectuant un appel API par enregistrement consommerait déjà 10 000 unités rien que pour les appels HTTP. Comparé aux scripts planifiés (10 000 unités au total par exécution), le Map/Reduce est plus tolérant (il dispose de 10 000 unités par étape) (Source: www.stockton10.com), mais le budget peut tout de même être rapidement épuisé. Si votre flux de travail charge ou enregistre également des enregistrements (souvent 10 unités supplémentaires chacun), le total augmente. Il est donc crucial de minimiser les appels inutiles : regroupez les requêtes lorsque c'est possible, ignorez les appels pour les enregistrements qui n'en ont pas besoin et compressez la logique dans chaque étape map.

Nous pouvons illustrer la consommation avec des données réelles : prenons la discussion sur le forum où un utilisateur a traité environ 2 000 transactions simples (une ligne chacune) via un Map/Reduce (Source: archive.netsuiteprofessionals.com). Même **sans aucun appel HTTP externe** et avec une concurrence de 7, le travail ne s'exécutait qu'à environ 2 enregistrements/seconde (Source: archive.netsuiteprofessionals.com) (Source: archive.netsuiteprofessionals.com). L'utilisateur a observé que **la désactivation de tous les scripts/workflows personnalisés** et la simple sauvegarde manuelle d'une facture de 2 000 lignes prenaient environ 5 minutes. Après avoir activé le Map/Reduce 2.0, il n'a atteint qu'environ 2,2 enregistrements/seconde (≈900/min) (Source: archive.netsuiteprofessionals.com) (Source: archive.netsuiteprofessionals.com). Cela indique que le traitement interne de NetSuite (recherche et enregistrement) est intrinsèquement lent ; ajouter de l'HTTP asynchrone par-dessus ajoutera probablement une latence supplémentaire significative. En bref, vous ne pouvez pas vous attendre à une accélération spectaculaire simplement en utilisant `async/await` – le débit sera toujours limité par le backend de NetSuite et par la gestion de la gouvernance. Comme le conclut brutalement Stockton10, *« SuiteScript 2.1 [n'est] pas significativement [plus rapide] que le 2.0 »* (Source: www.stockton10.com).

Une autre façon de voir les données est d'examiner les limites de gouvernance par étape. Les scripts Map/Reduce disposent de pools séparés de 10 000 unités pour *chaque* étape (GetInput, Map, Reduce, Summarize) (Source: www.stockton10.com). En théorie, un travail de 5 000 enregistrements pourrait consommer jusqu'à 40 000 unités au total sur toutes les étapes map, contre 10 000 au total pour un script planifié unique (Source: www.stockton10.com). Mais en pratique, des consommations comme 2 unités par enregistrement (pour un traitement map minimal) déclencheraient une auto-interruption (auto-yield) après environ 5 000 maps, moment auquel NetSuite marque une pause puis reprend. La combinaison de l'auto-interruption et de l'utilisation explicite de `await` signifie que les appels de longue durée sont généralement découpés automatiquement en morceaux. (Par exemple, si un appel HTTP est lent, la fonction map restera en attente, mais NetSuite pourra toujours créer un point de contrôle et reprendre cette tâche map si nécessaire.)

Le *Tableau 2* ci-dessous compare SuiteScript 2.0 et 2.1 sur les fonctionnalités et performances pertinentes discutées :

ASPECT	SUITESCRIPT 2.0	SUITESCRIPT 2.1	SOURCE
Version JavaScript	ES5 (pas de syntaxe moderne ; callbacks uniquement)	ES6+/ES2019+ : `let/const`, fonctions fléchées, modules, mode strict, etc.	[70]; [68]
Support asynchrone	Aucun (pas de Promise/await natif)	Support complet des Promises/ `async` pour certains modules	[24]; [48]
API `N/https`	Méthodes synchrones uniquement (ex: appels `https.get()`)	Ajoute des variantes asynchrones (ex: `https.get.promise()`)	[62]; [61]
Lisibilité du code	Riche en callbacks, gestion des erreurs dans les callbacks	Flux `async/await` linéaires, `try/catch` unique améliorant la clarté	[49]; [52]
Performance (débit)	Référence – limitée par la surcharge de NetSuite	*Similaire* – « pas significativement » plus rapide	[68]
Unités par appel HTTP	10 unités (`https.get`)	10 unités (`https.get.promise`)	[31]
Adapté aux tâches de masse	Conçu pour le traitement par lots (Map/Reduce disponible)	Toujours conçu pour le traitement par lots, mais les promesses « ne sont pas pour le traitement de masse »	[20]; [24]

Tableau 2. Comparaison des fonctionnalités clés de SuiteScript 2.0 vs 2.1 liées à la programmation asynchrone. Sources : Documentation Oracle et blogs de développeurs.

Comme le montre le Tableau 2, SuiteScript 2.1 ajoute principalement des fonctionnalités JS modernes et le support des Promises ; il ne change **pas** le modèle d'exécution sous-jacent ni les plafonds de gouvernance. Par conséquent, tout effet sur les performances doit provenir d'une structure de code plus efficace plutôt que d'une vitesse système accrue.

Études de cas et exemples

Bien que les études de cas formelles sur les modèles asynchrones de SuiteScript soient rares, nous pouvons nous appuyer sur des exemples de la communauté et des guides de développeurs pour illustrer l'utilisation réelle :

- **Exemple de mise à jour d'inventaire :** Un blog Heliverse décrit l'utilisation d'un script Map/Reduce en 2.1 pour mettre à jour les quantités d'articles d'inventaire (Source: medium.com). Bien que cet exemple soit purement SuiteScript (sans HTTP externe), il montre le flux de travail typique Map/Reduce. Dans *getInputData*, une recherche enregistrée trouve les articles avec une quantité > 0 ; dans *map*, il met à jour la quantité de chaque article via `record.submitFields` ; dans *summarize*, il rapporte les succès. Ils soulignent la gestion sécurisée de plus de 10 000 enregistrements en parallèle (traitement par lots de 1 000 à la fois) (Source: medium.com). On peut imaginer étendre un tel script pour appeler un service externe : par exemple, pour chaque article dans le map, nous pourrions utiliser `await https.get.promise` pour récupérer les prix ou les stocks mis à jour depuis une API fournisseur avant d'enregistrer la modification.

- **Exemple d'intégration API :** L'exemple de Houseblend (initialement dans un script planifié) montre comment intégrer un appel HTTPS et une recherche enregistrée ensemble (Source: www.houseblend.io). Bien qu'il ne s'agisse pas d'un Map/Reduce, le modèle est identique : une fonction `async execute` effectue `const resp = await https.get.promise(...)` pour récupérer des données externes, puis `await search.load.promise(...)` pour exécuter une recherche enregistrée (Source: www.houseblend.io). La même approche peut être utilisée dans une fonction Map. Cela confirme que SuiteScript prend en charge l'attente simultanée d'appels HTTP et de recherches. Un cas d'utilisation pratique pourrait être : pour chaque enregistrement dans le map, appeler une API REST externe pour obtenir des données d'enrichissement *et* interroger une SuiteQL, puis combiner les résultats.

- **Planification et polling de tâches :** L'exemple de Suitelet de polling (Source: www.houseblend.io) est une réutilisation concrète de modèle. Il montre le déploiement d'un script Map/Reduce (ou toute tâche de fond longue) et l'attente asynchrone de celle-ci dans un code linéaire. Un modèle similaire est utilisé dans certains processus métier : par exemple, un Suitelet qui, au clic sur un bouton, lance une énorme exécution de facturation et demande à l'utilisateur d'attendre jusqu'à ce qu'une page de confirmation affiche "Terminé". Utiliser `await Task.checkStatus()` avec une pause est souvent plus facile à coder que de gérer des callbacks reprenables. La note du développeur dans [53] reconnaît même la mise en garde : le Suitelet « attendra effectivement jusqu'à la fin », donc pour des travaux vraiment très longs, on pourrait plutôt demander au client d'interroger un point de terminaison de statut (Source: www.houseblend.io).

- **Rapports de la communauté :** Sur les forums communautaires (ex: NetSuite Professionals), les développeurs ont discuté des performances. Un utilisateur a exécuté un Map/Reduce 2.1 sur 2 000 transactions de vente et a constaté un débit d'environ **2 enregistrements/seconde** avec une concurrence de 7 (Source: archive.netsuiteprofessionals.com). Un autre a commenté que même une sauvegarde manuelle via l'interface d'une facture de 2 000 lignes prenait ~5 minutes, suggérant que le travail de base de données sous-jacent est assez lourd. Ces anecdotes soulignent que les appels asynchrones n'ont pas magiquement accéléré le travail : sans aucun script ou workflow personnalisé, le débit était d'environ 2,2 enr/s (Source: archive.netsuiteprofessionals.com). Ces chiffres réels nous rappellent que les API externes ne sont qu'une partie de la charge de travail ; le chargement et l'écriture d'enregistrements dans NetSuite ajoutent également une latence et une consommation d'unités significatives.

- **Exemple de gouvernance :** En utilisant les tableaux de gouvernance [31][43][44], on peut calculer la consommation d'unités attendue. Par exemple, si une étape Map effectue 50 GET HTTPS et 50 chargements d'enregistrements par exécution ; cela représente à lui seul 50×10 + 50×10 = 1000 unités. Même si chaque exécution individuelle est rapide, consommer 1 000 unités signifie qu'une étape de 10 000 unités ne pourrait traiter qu'environ 10 exécutions de ce type en séquence avant de s'interrompre. Empiriquement, le Map/Reduce crée automatiquement des points de contrôle toutes les quelques exécutions. Un rapport de développeur suggère que malgré le code 2.1, « les scripts attendent toujours la promesse avant de s'arrêter » (Source: archive.netsuiteprofessionals.com), ce qui signifie que chaque appel attendu occupe toujours pleinement la gouvernance.

En résumé, les exemples réels confirment notre analyse : les modèles asynchrones de SuiteScript 2.1 facilitent le codage de flux complexes, mais ne contournent pas le modèle de performance de NetSuite. Les travaux Map/Reduce réels impliquant des appels HTTP doivent prévoir des coûts de 10 unités par appel et intégrer un traitement par lots approprié, une gestion des erreurs et éventuellement une logique de nouvelle tentative pour les API instables.

Implications et orientations futures

L'avènement des Promises dans SuiteScript a plusieurs implications importantes :

- **Maintenabilité :** L'async/await améliore considérablement la clarté du code. Les flux complexes qui nécessitaient auparavant des callbacks imbriqués ou plusieurs déploiements de scripts peuvent désormais être écrits plus naturellement. Comme le soutiennent Stockton10 et d'autres, le principal avantage pratique de la version 2.1 est une meilleure lisibilité et une gestion des erreurs plus facile, et non la vitesse (Source:

www.stockton10.com) (Source: www.stockton10.com). Cela devrait réduire le temps de développement et de débogage pour les personnalisations NetSuite (une préoccupation majeure étant donné les mises à jour fréquentes de la plateforme et le roulement des développeurs).

- **Gestion de la concurrence :** Les développeurs doivent désormais réfléchir explicitement à la concurrence à deux niveaux : l'exécution des tâches parallèles de NetSuite et le parallélisme au sein de chaque tâche via les Promises. Les directives suggèrent de traiter les tâches SuiteScript comme du code Node moderne : utilisez `await` de manière réfléchie, utilisez `Promise.all` ou des bibliothèques pour gérer les appels simultanés, et interceptez toujours les erreurs. Ne pas le faire peut entraîner des délais d'attente « Unexpected Error » ou des exceptions de gouvernance. La base de connaissances de NetSuite souligne des bonnes pratiques similaires (voir [52] citant « promise.map » et la gestion des erreurs) (Source: www.houseblend.io). Une conclusion à retenir est : **ne lancez pas une tempête illimitée de requêtes XHR**.

- **Tests et diagnostics :** Avec le code asynchrone, le débogage diffère. Côté client, on peut utiliser les outils de développement du navigateur pour les Suitelets. Côté serveur (Map/Reduce), les développeurs doivent s'appuyer sur `log.debug/error` et l'étape Summarize pour capturer les erreurs. Il est désormais plus facile d'envelopper un appel `await` avec un `try/catch` et de journaliser l'erreur, plutôt que d'espérer qu'un callback journalise quelque chose. Les équipes devraient mettre à jour leurs approches de débogage pour tenir compte de la possibilité de rejets de promesses non interceptés (en utilisant `.catch()` avec diligence).

- **Stratégie de gouvernance :** Comme le code asynchrone peut plus facilement générer de multiples requêtes, les architectes doivent réévaluer la manière dont ils conçoivent les contrôles d'autorisation et les budgets de gouvernance. Par exemple, un Suitelet lançant plusieurs tâches Map/Reduce (via `N/task`) pourrait facilement atteindre la limite de 10 000 unités d'un script planifié. Le modèle décrit en [53], qui consiste à interroger (polling) une tâche, est simple mais peut ne pas passer à l'échelle pour des tâches très volumineuses. La position officielle d'Oracle est que des systèmes tels que les files d'attente de messages (message queues) ou la propre file d'attente de travail (Work Queue) de NetSuite sont appropriés pour le traitement par lots ou infini ; les promesses asynchrones sont destinées à être utilisées *au sein* de la logique d'un script (Source: docs.oracle.com).

- **Intégration avec des bibliothèques tierces :** Le rapport Houseblend [19] et les blogs de développement d'Oracle (par exemple, « Navigating Third-Party Library Compatibility in SuiteScript 2.1 » (Source: blogs.oracle.com) indiquent une évolution plus large : SuiteScript 2.1 est de plus en plus traité comme un environnement JS standard. Les développeurs peuvent désormais intégrer et inclure des bibliothèques tierces (via AMD/UMD ou le regroupement Webpack) pour utiliser des outils modernes (par exemple, TypeScript, Axios, Lodash, etc.). Par exemple, un blog d'Oracle montre comment inclure des modules Node en utilisant Webpack/Zod dans SuiteScript 2.1. À l'avenir, la mention par Oracle de « polyfills Node.js » (Source: www.houseblend.io) suggère que davantage de bibliothèques standard Node pourraient devenir disponibles dans les scripts SuiteScript (par exemple, buffer, et peut-être des améliorations liées au chiffrement). Cela ouvre la porte à une logique client et serveur complexe auparavant inaccessible.

- **Mises à jour continues :** L'introduction de la version 2.1 souligne également la stratégie de versioning de NetSuite. Auparavant, la mise à niveau d'un script vers la version 2.1 était un choix manuel. Stockton10 souligne une nouvelle option : l'utilisation de `@NApiVersion 2.x`, qui met automatiquement à jour votre code vers la dernière version mineure prise en charge (par exemple, 2.2, 2.3...) (Source: www.stockton10.com). Bien que cela soit quelque peu indépendant de HTTP/Promises, cela signifie que tout ajout futur à ECMAScript sera disponible sans avoir à modifier l'en-tête. Les développeurs doivent en être conscients pour la maintenance à long terme. (Cependant, une mise à niveau automatique peut casser le code – un exemple donné est que la version 2.2 a modifié la gestion des valeurs nulles dans `N/search`, de sorte qu'un script 2.x existant s'est comporté différemment de manière inattendue (Source: www.stockton10.com).)

- **Tendances émergentes – IA :** Enfin, tant Houseblend [19] que les discussions du secteur font allusion au codage assisté par IA dans l'écosystème NetSuite. Bien qu'encore naissant, on peut imaginer utiliser des outils comme Copilot ou des chatbots personnalisés entraînés sur les bibliothèques SuiteScript pour aider à écrire du code asynchrone. C'est spéculatif, mais la mention par Oracle d'outils assistés par IA (Source: www.houseblend.io) montre que cela fait partie de la planification future. Pour l'instant, cependant, les tests systématiques et les revues de code restent cruciaux, surtout avec la complexité accrue du parallélisme et des flux asynchrones.

En résumé, les promesses HTTPS de SuiteScript 2.1 rapprochent NetSuite du développement JavaScript moderne. Les développeurs doivent s'adapter en utilisant de nouveaux idiomes linguistiques et en suivant les meilleures pratiques pour le code asynchrone, tout en respectant le modèle d'exécution unique de NetSuite. Utilisées correctement, ces fonctionnalités rendront les intégrations (appels d'API externes, pipelines de données) plus faciles à mettre en œuvre et à maintenir. Mal utilisées (par exemple, parallélisme illimité), elles peuvent entraîner des goulots d'étranglement au niveau de la gouvernance. Les compromis sont clairement documentés et relayés par la communauté : un code plus simple pour une enveloppe de performance globalement identique (Source: www.houseblend.io) (Source: www.stockton10.com).

Conclusion

L'introduction des promesses asynchrones dans SuiteScript 2.1 – particulièrement dans le module `N/https` – change la donne pour le développement sur NetSuite. Elle permet aux développeurs d'effectuer des appels HTTP externes et d'autres opérations d'E/S de manière plus propre et plus facile à maintenir en utilisant `async/await`. Au sein des scripts Map/Reduce, cela signifie que l'on peut écrire du code linéaire, comme la récupération de données REST, plutôt que d'enchaîner des rappels (callbacks) ou de lancer des processus externes. Cependant, les contraintes fondamentales de la plateforme NetSuite demeurent : chaque appel HTTP coûte 10 unités d'utilisation, et les étapes Map/Reduce elles-mêmes ont des limites de gouvernance (Source: docs.oracle.com) (Source: docs.oracle.com). En pratique, les modèles asynchrones améliorent la productivité des développeurs et réduisent les erreurs, mais **n'améliorent pas intrinsèquement le débit et ne réduisent pas la consommation de ressources** (« SuiteScript 2.1 n'est pas significativement plus rapide » (Source: www.stockton10.com).

Toutes les recommandations de ce rapport sont étayées par des sources faisant autorité. La documentation officielle de NetSuite confirme quels modules prennent en charge les promesses (Source: docs.oracle.com) et liste les nouvelles méthodes `.promise()` dans `N/https` (Source: docs.oracle.com). Les experts développeurs fournissent des exemples d'utilisation et des mises en garde (Source: www.houseblend.io) (Source: www.houseblend.io). Les expériences de la communauté fournissent des preuves empiriques sur les performances (Source: archive.netsuiteprofessionals.com) (Source: archive.netsuiteprofessionals.com). En synthétisant ces perspectives, nous concluons que les promesses HTTPS de SuiteScript 2.1 devraient être adoptées pour tout développement NetSuite 2.1, mais avec une attention particulière aux meilleures pratiques asynchrones et aux limites de gouvernance.

À l'avenir, alors que NetSuite continue de moderniser sa plateforme et d'adopter les standards JavaScript (y compris les polyfills et l'automatisation des versions de script), ces fonctionnalités asynchrones permettront des intégrations encore plus riches. Pour l'instant, une utilisation réfléchie de `https.get.promise()` et des modèles associés permettra aux organisations de créer des tâches Map/Reduce plus robustes et plus faciles à maintenir pour leurs processus critiques.

Sources : Aide officielle de NetSuite (documents Oracle sur SuiteScript et Map/Reduce) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com); Blogs et livres blancs des développeurs NetSuite (Source: www.houseblend.io) (Source: www.houseblend.io) (Source: www.houseblend.io); Articles de la base de connaissances et forums communautaires (Source: archive.netsuiteprofessionals.com) (Source: archive.netsuiteprofessionals.com); Articles de blog d'experts (Source: www.stockton10.com) (Source: www.stockton10.com). Toutes les affirmations sont étayées par ces références.

Étiquettes: suitescript-21, scripts-map-reduce, promesses-https, async-await, netsuite-nhttps, gouvernance-suitescript, promesses-javascript

---