Modèles asynchrones SuiteScript 2.x : requestSuitelet.promise

Publié le 30 mai 2026 47 min de lecture

Résumé analytique

La plateforme SuiteScript 2.x de NetSuite a de plus en plus adopté des modèles de programmation asynchrone, notamment dans son module N/https et l'API SuiteScript. L'introduction de méthodes asynchrones telles que `https.requestSuitelet.promise` (depuis 2023.1) et `https.requestRestlet.promise` (depuis 2020.2) représente un virage délibéré vers des flux de travail non bloquants dans les scripts client et serveur (Source: docs.oracle.com) (Source: docs.oracle.com). Ces méthodes permettent aux scripts d'initier des appels HTTP vers des Suitelets ou RESTlets internes sans bloquer le thread d'exécution principal, favorisant ainsi une expérience utilisateur plus fluide et une meilleure concurrence.

La documentation officielle et les experts soulignent que `https.requestSuitelet.promise(options)` envoie une requête HTTPS de manière asynchrone vers un Suitelet *interne* (dans un contexte authentifié) et renvoie une `Promise` JavaScript qui se résout avec la réponse (Source: docs.oracle.com) (Source: docs.oracle.com). La méthode analogue `https.requestSuitelet(options)` (version synchrone) renvoie un objet `https.ClientResponse` après avoir attendu la fin de l'exécution du Suitelet (Source: docs.oracle.com) (Source: docs.oracle.com). Les deux consomment 10 unités de gouvernance par appel et nécessitent de spécifier l'ID de script et l'ID de déploiement du Suitelet cible. Il est crucial de noter qu'à partir de NetSuite 2024.1, SuiteScript interdit l'appel d'URL de Suitelets externes (publics) via ces méthodes – l'indicateur `options.external` a été supprimé et seuls les Suitelets internes authentifiés sont autorisés (Source: docs.oracle.com) (Source: community.oracle.com).

Le module N/https fournit également `https.requestRestlet(options)` et sa variante Promise `https.requestRestlet.promise(options)`. Celles-ci permettent d'appeler des RESTlets (SuiteScripts exposés par HTTP) de manière asynchrone, y compris dans les scripts côté client. Notamment, `requestRestlet(options)` injecte automatiquement les en-têtes d'autorisation NetSuite afin que le RESTlet appelé s'exécute avec les permissions de l'appelant (Source: docs.oracle.com). La version asynchrone basée sur les Promises fonctionne de manière similaire mais renvoie une `Promise` pour la réponse (Source: docs.oracle.com) (Source: docs.oracle.com). Cela a permis de nouveaux modèles d'intégration, par exemple en faisant en sorte qu'un script client appelle de manière asynchrone un RESTlet authentifié qui effectue lui-même un appel API externe, contournant ainsi l'interdiction de NetSuite concernant les appels sortants dans les contextes client non authentifiés (Source: docs.oracle.com) (Source: docs.oracle.com).

Une variété de modèles de conception et de cas d'utilisation ont émergé en utilisant ces appels asynchrones de Suitelet/RESTlet. Par exemple, un modèle courant « le client appelle le Suitelet » consiste à ce qu'un script User Event place un bouton sur un enregistrement qui déclenche un script client ; le script client invoque ensuite `https.requestSuitelet.promise` et traite le résultat du Suitelet sans forcer un rechargement complet de la page (Source:

docs.oracle.com) (Source: docs.oracle.com). Dans d'autres scénarios, les clients utilisent soit `fetch`, soit `https.get()` sur des URL internes résolues pour mettre à jour l'interface utilisateur dynamiquement (Source: www.thenetsuitepro.com) (Source: test.suiterep.com). Côté serveur, les tâches de traitement intensif peuvent être déchargées en lançant de manière asynchrone des Suitelets ou des RESTlets (par exemple via des scripts planifiés ou des scripts User Event). Des tests indépendants ont montré que les appels de Suitelets hautement concurrents peuvent surpasser considérablement les scripts Map/Reduce pour certaines charges de travail ; par exemple, un benchmark a traité 1 million d'enregistrements en **40 minutes** via une configuration de Suitelets hautement parallèles contre **5,5 heures** en utilisant Map/Reduce (Source: ursuscode.com) (Source: ursuscode.com). Cependant, les développeurs doivent toujours respecter les limites de gouvernance et de temps (par exemple, intercepter les erreurs `SSS_REQUEST_TIME_EXCEEDED` si une requête prend trop de temps) (Source: archive.netsuiteprofessionals.com), et concevoir leurs scripts en tenant compte de la gestion des erreurs et des limites de concurrence.

Ce rapport fournit une **analyse approfondie et complète** des modèles de Suitelet et de RESTlet asynchrones dans SuiteScript 2.x. Nous couvrons l'évolution historique de SuiteScript (y compris l'avènement de SuiteScript 2.1 avec les promesses et async/await), des descriptions détaillées des API pertinentes et de leurs paramètres, les meilleures pratiques pour appeler des Suitelets et des RESTlets (côté client vs côté serveur, interne vs externe), les considérations de gouvernance et de performance, des exemples d'études de cas et des données (y compris des comparaisons de performance), ainsi que les orientations futures pour ces modèles. Toutes les affirmations et exemples sont étayés par des sources faisant autorité, notamment la documentation d'Oracle, les annonces de support de NetSuite, les questions-réponses de la communauté et les analyses sectorielles publiées.

Introduction et contexte

NetSuite, une plateforme ERP (Enterprise Resource Planning) basée sur le cloud de premier plan acquise par Oracle, permet une personnalisation étendue via son API SuiteScript. SuiteScript est un environnement de script basé sur JavaScript construit au-dessus de NetSuite. Au fil du temps, SuiteScript a évolué à travers des versions majeures :

- **SuiteScript 1.0** (héritage) – API de style objet global (`nlapi*` functions).
- **SuiteScript 2.0** (introduit vers 2014) – API modulaire de style AMD utilisant `define` / `require`, avec des modules séparés (par exemple `N/record`, `N/search`) (Source: docs.oracle.com).
- **SuiteScript 2.1** (introduit en 2018) – a étendu SuiteScript 2.0 en ajoutant la prise en charge de fonctionnalités JavaScript modernes telles que `async/await`, les **Promises**, et d'autres améliorations de syntaxe ES6+ (Source: docs.oracle.com) (Source: www.thenetsuitepro.com).

SuiteScript 2.1 prend entièrement en charge la *programmation asynchrone non bloquante côté serveur* avec `async/await` et les Promises, mais uniquement pour un sous-ensemble de modules et de fonctions. Comme l'indique la référence SuiteScript, les opérations asynchrones peuvent être exprimées avec `async` / `await` / `Promise` pour des modules comme **N/http**, **N/https**, **N/query**, **N/search** et **N/transaction** (Source: docs.oracle.com). (Par exemple, on peut `await` une requête `https` ou une recherche). Cela a permis aux développeurs d'écrire du code asynchrone plus intuitif : par exemple, un exemple d'Oracle montre le chargement d'une recherche sauvegardée et l'utilisation de `await search.run().then(...)` pour traiter les résultats (Source: docs.oracle.com). Il est important de noter que `await` n'accélère **pas** par magie les opérations intrinsèquement synchrones ; par exemple, `await record.load(...)` bloque toujours sur le chargement de l'enregistrement, car de nombreuses API SuiteScript restent des appels synchrones traditionnels (Source: www.thenetsuitepro.com).

Le module **N/https** est au cœur de la communication HTTP dans SuiteScript. Il inclut des méthodes pour effectuer des GET, POST, PUT, DELETE, etc. en dehors de NetSuite, ainsi que des méthodes spécialisées pour invoquer les propres Suitelets et RESTlets de NetSuite. Bien que N/https dispose de méthodes synchrones (qui bloquent jusqu'à l'arrivée de la réponse), des versions plus récentes de l'API SuiteScript ont introduit des variantes asynchrones basées sur les Promises pour bon nombre de ces méthodes. En particulier, en 2020.2, NetSuite a ajouté **`https.requestRestlet.promise`** (Source: docs.oracle.com), et en 2023.1, ils ont ajouté **`https.requestSuitelet.promise`** (Source: docs.oracle.com) (Source: docs.oracle.com). Celles-ci renvoient une `Promise` JavaScript qui se résout avec la réponse du serveur, permettant aux scripts d'utiliser `await` ou `.then()` sans figer l'exécution.

Pendant ce temps, les **Suitelets** et les **RESTlets** sont deux types de scripts dans NetSuite :

- Un **Suitelet** est un script côté serveur qui génère une réponse HTTP, généralement utilisé pour créer des pages ou des points de terminaison NetSuite personnalisés. Les Suitelets peuvent rendre des formulaires dans l'interface utilisateur de NetSuite ou envoyer des données (HTML, JSON, PDF, etc.) aux appelants. Ils peuvent être invoqués en interne (au sein de NetSuite) ou en externe (via une URL publique si « Disponible sans connexion ») (Source: docs.oracle.com) (Source: test.suiterep.com).
- Un **RESTlet** est un point de terminaison d'API SuiteScript qui expose une interface JSON RESTful. Les systèmes externes appellent souvent des RESTlets pour pousser ou extraire des données de NetSuite. En interne, les SuiteScripts peuvent également appeler des RESTlets pour modulariser la logique.

À mesure que la plateforme SuiteScript a mûri, Oracle et la communauté NetSuite ont développé des meilleures pratiques et des modèles autour de l'appel de Suitelets et de RESTlets, en particulier à partir de scripts client ou d'événements utilisateur. Historiquement, appeler un Suitelet ou un RESTlet impliquait souvent de construire des chaînes d'URL ou d'utiliser `url.resolveScript`. Par exemple, un script client pouvait faire `https.get()` avec une URL partielle vers un RESTlet (Source: docs.oracle.com), ou utiliser `url.resolveScript({returnExternalUrl:true})` pour obtenir l'URL externe d'un Suitelet, puis l'appeler avec `https.get()` et des en-têtes NLAuth (Source: docs.oracle.com). Ces approches étaient fonctionnelles mais lourdes et sujettes aux erreurs : elles nécessitaient la gestion de jetons d'authentification (paramètres `nl-at` ou en-têtes `N/https`) et n'étaient pas réellement asynchrones. Les nouvelles API `https.requestSuitelet` et `https.requestRestlet` simplifient cela en gérant automatiquement l'authentification pour les appels internes et en prenant en charge les Promises pour les flux asynchrones.

Ce rapport explore ces modèles asynchrones de SuiteScript 2.x en profondeur. Nous examinerons d'abord les capacités actuelles de N/https (promesses, etc.) et des modules associés (`N/url`, etc.). Ensuite, nous analyserons les API spécifiques `https.requestSuitelet.promise` et `https.requestRestlet.promise` : leurs paramètres, contextes d'utilisation, limitations et comportements. Nous comparerons ces nouvelles méthodes aux approches plus anciennes (par exemple, URL externes, `https.get`, `url.resolveScript`), en citant des exemples officiels. Nous examinerons également la gouvernance et la performance – comment ces appels sont mesurés et comment ils fonctionnent à grande échelle. Le rapport inclut des **études de cas et des exemples** tirés de la documentation de NetSuite, des questions-réponses de la communauté et des blogs de développeurs. Enfin, nous discuterons des implications de ces modèles asynchrones pour le développement réel sur NetSuite et indiquerons les orientations futures (telles que les changements d'API en évolution et les tendances d'intégration).

Promesses et capacités asynchrones de SuiteScript 2.x

Le passage de SuiteScript à la prise en charge de la programmation asynchrone a été officialisé avec SuiteScript 2.1. Dans SuiteScript 2.1, Oracle a officiellement autorisé l'exécution asynchrone d'un sous-ensemble d'appels API. La documentation de SuiteScript note :

> ***"SuiteScript 2.1 prend entièrement en charge les promesses côté serveur asynchrones non bloquantes 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).*

En d'autres termes, si vous marquez une fonction comme `async`, vous pouvez `await` des appels tels que `N/https.request`, `N/https.requestRestlet.promise`, `N/query.runSuiteQL.promise`, etc. La promesse sera non bloquante par rapport à la boucle d'événements Node.js, mais du point de vue du script, elle « attend » toujours au niveau du `await`. Cela permet une syntaxe plus propre et la possibilité de lancer plusieurs opérations en parallèle si souhaité (par exemple, `await Promise.all([https.request({...}), otherPromise])`).

Cependant, toutes les API SuiteScript ne sont pas asynchrones. De nombreuses API principales (comme `record.load`, `search.run`, `https.request` lui-même) se comportent toujours de manière synchrone ; les `await`er ou non ne fait aucune différence pratique, sauf pour la lisibilité du code (Source: www.thenetsuitepro.com). Le modèle de l'asynchronisme brille vraiment lorsqu'il s'agit d'opérations liées aux E/S qui pourraient bénéficier de la concurrence, telles que les appels HTTP vers des points de terminaison externes.

Par exemple, un article de blog de 2020 sur l'asynchronisme dans SuiteScript expliquait que le code côté client peut bénéficier d'un véritable asynchronisme (par exemple, récupérer des données ou afficher des boîtes de dialogue), tandis que le code côté serveur peut utiliser `async/await` principalement pour des opérations concurrentes au sein du processus (comme invoquer plusieurs requêtes SuiteQL indépendantes ou attendre plusieurs requêtes HTTP en parallèle) (Source: www.thenetsuitepro.com). Il est important de noter que l'aide de SuiteScript avertit que l'asynchronisme ne doit pas être utilisé pour le traitement par lots à la place d'un cadre de traitement par lots approprié : elle indique **« Cette capacité n'est pas destinée aux cas d'utilisation de traitement par lots où une solution hors bande, telle qu'une file d'attente de travail, peut suffire »** (Source: docs.oracle.com).

En pratique, cela signifie que SuiteScript 2.1/2.2 peut exprimer des flux asynchrones, mais qu'une exécution unique de SuiteScript a toujours un budget de temps et de ressources limité. L'utilisation de `async` / `await` ne lève pas ces limites – elle modifie simplement le flux de contrôle. Comme le note le guide des modèles Async/Await de SuiteScript, *« `async/await` ne contourne pas la gouvernance. Cela rend simplement les flux asynchrones plus faciles à lire/maintenir. »* (Source: www.thenetsuitepro.com). Concrètement, que vous utilisiez la version synchrone ou la version promesse d'une API, les unités de gouvernance consommées sont les mêmes (pour les appels N/https, 10 unités chacun) (Source: docs.oracle.com) (Source: docs.oracle.com) ; la différence réside dans la commodité de l'écriture des scripts.

Néanmoins, le SuiteScript asynchrone ouvre la voie à de nouveaux modèles de programmation. Par exemple, un script client peut désormais appeler un Suitelet ou un RESTlet en arrière-plan en utilisant `await https.requestSuitelet.promise(...)` au lieu d'effectuer une redirection synchrone ou une soumission de formulaire. Un script serveur pourrait lancer deux appels RESTlet simultanément avec `Promise.all` puis attendre les deux résultats, plutôt que de les exécuter l'un après l'autre. Ce rapport détaillera ces modèles, mais nous décrirons d'abord les méthodes API spécifiques impliquées.

Présentation du module N/https

Le module **N/https** permet au SuiteScript d'accéder à l'ensemble des fonctionnalités HTTP, tant pour les requêtes externes que pour l'appel de Suitelets/RESTlets internes. Ses méthodes clés incluent :

- **`https.get(options)`, `https.post(options)`, `https.put(options)`, `https.request(options)`, etc.** Celles-ci effectuent des appels HTTPS synchrones vers n'importe quelle URL (généralement des points de terminaison tiers externes). L'appelant fournit l'URL complète, les en-têtes, le corps, etc. Ces méthodes se bloquent jusqu'à ce qu'une réponse soit reçue et renvoient un objet `https.ClientResponse` (avec statut et corps). Chaque appel consomme des unités de gouvernance (par exemple 10 unités) et peut échouer si NetSuite ne peut pas vérifier le certificat SSL/TLS ou atteindre le serveur. NetSuite impose **TLS 1.2** pour les requêtes sortantes (Source: docs.oracle.com), de sorte que les points de terminaison HTTP hérités qui ne prennent en charge que TLS 1.0/1.1 ne pourront pas se connecter (la documentation de N/https avertit les développeurs de s'assurer que les serveurs tiers prennent en charge TLS 1.2 (Source: docs.oracle.com).

- **`https.requestRestlet(options)` (synchrone)** – introduite en 2020.2, cette méthode envoie une requête HTTPS *à un RESTlet interne à NetSuite*. Les options spécifient le `scriptId` et le `deploymentId` du RESTlet cible (par exemple, `scriptId: 'customscript_myRestlet', deploymentId: '1'`), et éventuellement la méthode (`GET/POST/PUT/etc`), le `body`, les `headers` et les `urlParams` (paramètres de requête). Il est crucial de noter qu'aucun domaine complet n'est nécessaire ; N/https sait qu'il doit appeler l'API REST interne de NetSuite. La documentation note que « Les en-têtes d'authentification sont ajoutés automatiquement. Le RESTlet s'exécutera avec les mêmes privilèges que le script appelant. » (Source: docs.oracle.com). En d'autres termes, si un script côté serveur appelle `https.requestRestlet`, NetSuite inclut automatiquement un en-tête `NLAuth` ou OAuth en arrière-plan, de sorte que le RESTlet le considère comme un appel interne à l'application. Cette méthode renvoie un `https.ClientResponse` de manière synchrone.

- **`https.requestSuitelet(options)` (synchrone)** – introduite en 2023.1 parallèlement à sa variante promise, cette méthode fonctionne comme `requestRestlet` mais cible un script Suitelet interne. L'appelant doit fournir `scriptId` et `deploymentId`, et éventuellement la méthode, le `body`, les `headers` et les `urlParams`. NetSuite exécute ensuite le Suitelet (sur le serveur) et renvoie un `ClientResponse`. Comme pour l'appel RESTlet, elle gère automatiquement l'authentification (par exemple, elle s'exécute implicitement dans le contexte de l'utilisateur/session actuel) de sorte que le Suitelet est invoqué en interne. Cela équivaut essentiellement à ce que NetSuite effectue une requête HTTP interne vers lui-même.

- **`https.requestRestlet.promise(options)` (Promise asynchrone)** – également en 2020.2, cette méthode renvoie une `Promise` JavaScript au lieu de bloquer. La fonctionnalité (y compris l'authentification automatique) est la même que pour `requestRestlet` synchrone, mais le retour est une `Promise` qui se résout en la réponse. Elle est officiellement prise en charge dans les scripts serveur (Source: docs.oracle.com). Il est intéressant de noter que les exemples de NetSuite suggèrent son utilisation dans des scripts client agissant comme des proxys : l'idée est qu'un RESTlet connecté (serveur) effectue l'appel externe, puis le script client appelle *ce* RESTlet via cette méthode, permettant ainsi le HTTP asynchrone depuis le navigateur (Source: docs.oracle.com) (Source: docs.oracle.com).

- **`https.requestSuitelet.promise(options)` (Promise asynchrone)** – introduite en 2023.1, cette méthode renvoie une `Promise` pour appeler un Suitelet. Les paramètres sont identiques à ceux de `https.requestSuitelet`, et elle renvoie une `Promise` qui se résout en ce que le Suitelet produit. Il est important de noter que cet appel asynchrone est autorisé à la fois dans les scripts client et serveur (Source: docs.oracle.com). En arrière-plan, NetSuite effectuera un appel HTTP interne pour exécuter le Suitelet, mais renverra le résultat de manière asynchrone. Cela permet aux scripts client d'appeler des Suitelets sans bloquer le thread de l'interface utilisateur (par exemple, en utilisant `await` dans une fonction de clic sur un bouton), ou aux serveurs d'initier des Suitelets sans attendre (si codé ainsi).

Le tableau 1 ci-dessous résume ces quatre méthodes clés de N/https :

MÉTHODE	RETOURNE	TYPES DE SCRIPTS PRIS EN CHARGE	GOUVERNANCE	DEPUIS	DESCRIPTION
`https.get(options)` (et autres requêtes sync)	`https.ClientResponse`	Client & Serveur	10 unités par appel	2015.1 (N/https)	Envoie de manière synchrone une requête HTTPS vers n'importe quelle URL (tiers ou URL externe NetSuite). L'appelant fournit l'URL complète, la méthode, les en-têtes, etc.
`https.requestRestlet(options)`	`https.ClientResponse`	Scripts serveur	10	2020.2	Appelle de manière synchrone un RESTlet NetSuite (interne). Ajoute automatiquement l'authentification ; le RESTlet s'exécute avec les autorisations de l'appelant (Source: docs.oracle.com).
`https.requestRestlet.promise(options)`	`Promise`	Scripts serveur	10	2020.2	Version asynchrone de `requestRestlet`. Renvoie une Promise JS pour la réponse du RESTlet (Source: docs.oracle.com) (Source: docs.oracle.com).
`https.requestSuitelet(options)`	`https.ClientResponse`	Client & Serveur	10	2023.1	Appelle de manière synchrone un Suitelet NetSuite (interne). Similaire à ci-dessus mais cible un Suitelet. (Introduit en 2023.1) (Source: docs.oracle.com).
`https.requestSuitelet.promise(options)`	`Promise`	Client & Serveur	10	2023.1	Version *asynchrone* de `requestSuitelet`. Renvoie une

MÉTHODE	RETOURNE	TYPES DE SCRIPTS PRIS EN CHARGE	GOUVERNANCE	DEPUIS	DESCRIPTION
					Promise se résolvant en la réponse du Suitelet (Source: docs.oracle.com) (Source: docs.oracle.com).

Remarque : Tous les appels SuiteScript internes (`requestRestlet` / `requestSuitelet` et leurs variantes Promise) **ne fonctionnent que sur des URL internes** dans des contextes authentifiés/approuvés. Depuis NetSuite 2024.1, vous **ne pouvez pas** passer l'option `{external: true}` pour appeler l'URL externe d'un Suitelet ou pour effectuer des appels depuis une page web non authentifiée. Oracle a confirmé mi-2024 que `options.external` est supprimé et que les appels ne cibleront que les URL internes (Source: docs.oracle.com) (Source: community.oracle.com). Le Suitelet doit être déployé comme « Disponible avec connexion » (ou marqué comme **non** « Disponible sans connexion ») si vous avez l'intention de l'appeler avec ces méthodes.

Le module **N/url** fonctionne main dans la main avec N/https lors de la résolution des URL de script internes. Il peut produire soit une URL interne, soit une URL externe vers un Suitelet/Restlet via `url.resolveScript`. Par défaut, `resolveScript` renvoie un chemin relatif *interne* (par exemple, `/app/site/hosting/scriptlet.nl?...`) qui peut être utilisé dans les scripts serveur. Si `returnExternalUrl: true` est défini (et que le script est dans un contexte de connexion), il renvoie une URL externe complète avec domaine et jeton anti-altération (Source: docs.oracle.com). Cependant, comme noté par Oracle, `url.resolveScript` **ne fonctionne pas dans les scripts client non authentifiés** – il générera une erreur dans un contexte SuiteCommerce (anonyme) (Source: docs.oracle.com). Cette limitation a en partie motivé l'introduction de l'astuce `requestRestlet.promise` pour les flux anonymes (Source: docs.oracle.com).

Suitelets et RESTlets : Rôles et modèles d'appel

Avant d'approfondir les méthodes asynchrones elles-mêmes, nous passons brièvement en revue ce que sont les Suitelets et les RESTlets, et comment les scripts les appellent traditionnellement.

- **Suitelet** : Un Suitelet est un SuiteScript qui peut générer une réponse HTTP. Les Suitelets produisent généralement des formulaires ou des données. Ils peuvent s'exécuter dans deux modes :

  1. **Interne/Authentifié** (le cas habituel) : l'appel est effectué depuis l'intérieur de NetSuite (par un script ou un formulaire utilisateur) et l'utilisateur est déjà connecté. L'URL est généralement une URL relative comme `/app/site/hosting/scriptlet.nl?script=123&deploy=1`.
  2. **Externe** (disponible sans connexion) : le Suitelet est marqué « Disponible sans connexion » dans son enregistrement de déploiement. NetSuite expose un hôte externe comme `<account>.extforms.netsuite.com` pour celui-ci. Une URL de Suitelet externe ressemble généralement à :

     ```
     https://<account>.extforms.netsuite.com/app/site/hosting/scriptlet.nl?script=123&deploy=1&ns-at=<token>
     ```

     où `ns-at` est un jeton anti-altération. Appeler l'URL externe nécessite de construire ou d'obtenir une URL complète et souvent de configurer une authentification NLAuth (ou basée sur des jetons) si cela est fait à partir d'un script (Source: docs.oracle.com) (Source: docs.oracle.com).

- **RESTlet** : Un RESTlet est un SuiteScript destiné à être invoqué via HTTP (renvoyant ou acceptant généralement du JSON). Les RESTlets s'exécutent toujours sous l'API REST de SuiteScript. Ils peuvent être appelés de manière externe (en utilisant `<account>.restlets.api.netsuite.com` ou un chemin relatif dans une session authentifiée). Pour les appels internes, NetSuite prend en charge l'utilisation de `https.requestRestlet` qui masque les détails de l'URL.

Modèles d'appel (avant l'API asynchrone) :

- *Scripts client* : Si vous souhaitez qu'un script client (s'exécutant dans le navigateur sur un enregistrement ou un formulaire) récupère des données à partir d'un Suitelet ou d'un RESTlet, vous devez généralement utiliser `url.resolveScript` ou coder en dur une URL interne. Par exemple, un exemple Oracle montre un script client effectuant :

```
var suiteletUrl = url.resolveScript({
    scriptId: 'customscript_my_json_sl',
    deploymentId: 'customdeploy_my_json_sl',
    params: { action: 'summary' }
});
var res = await fetch(suiteletUrl, { method: 'GET', credentials: 'same-origin' });
var json = await res.json();
```

Ici, le client utilise le `fetch` du navigateur sur une URL de Suitelet *interne* (credentials: same-origin pour utiliser le cookie de connexion existant) (Source: www.thenetsuitepro.com). Dans l'ancien SuiteScript sans promesses, on pourrait voir un script client utiliser `https.get({url: resolveURL})` ou `https.requestSuitelet` (synchrone) comme indiqué dans la documentation (Source: docs.oracle.com). Si l'objectif est de **rediriger** vers un Suitelet (par exemple pour télécharger un fichier), on pourrait simplement faire `window.open(resolvedUrl)` comme indiqué dans les guides de bonnes pratiques (Source: test.suiterep.com). En fait, Ben Rogol souligne que **si vous voulez qu'un Suitelet affiche directement un formulaire ou un PDF, utilisez une redirection (`window.open`) ; si vous voulez récupérer des données pour la même page, utilisez un GET asynchrone** (Source: test.suiterep.com).

- *Scripts serveur (UE, Scheduled, etc.)* : Un script User Event ou Scheduled peut appeler un Suitelet ou un RESTlet. Avant les nouvelles API, une approche courante consistait à utiliser le module `N/redirect` pour rediriger l'utilisateur actuel (par exemple dans `beforeLoad`), ou à utiliser `N/https` pour effectuer un appel interne. Par exemple, la documentation d'Oracle fournit un exemple où un User Event ajoute un bouton pour appeler une fonction client, qui utilise ensuite `https.requestSuitelet` pour appeler un Suitelet (Source: docs.oracle.com) (Source: docs.oracle.com). Cela est nécessairement synchrone (le gestionnaire de bouton attend le Suitelet puis affiche une alerte). Sans le nouveau `requestSuitelet.promise`, tout appel interne depuis des scripts serveur était intrinsèquement synchrone : le code ne se poursuivait pas tant que le Suitelet n'avait pas renvoyé de réponse.

- *Intégrations externes* : Les systèmes externes (ou les clients REST externes) appellent des RESTlets ou des Suitelets via des URL publiques. Cela est géré par les points de terminaison de l'API SuiteTalk de NetSuite, distincts du `N/https` de SuiteScript. Il convient de noter que `N/https` possède une méthode `https.requestSuiteTalkRest(options)`, introduite plus tôt, pour appeler l'API REST SuiteTalk de NetSuite depuis SuiteScript (Source: docs.oracle.com).

Point clé sur l'authentification et le contexte : `https.requestSuitelet` et `https.requestRestlet` **ne fonctionnent que dans des contextes approuvés**. La documentation de `requestSuitelet` indique explicitement : *« Cette méthode ne peut renvoyer un Suitelet interne que dans des contextes approuvés pour les utilisateurs authentifiés. »* (Source: docs.oracle.com). Cela signifie :

- Votre script doit s'exécuter avec un utilisateur NetSuite connecté ou en tant que script serveur. (Les scripts de boutique en ligne anonyme ou de portail client ne peuvent pas l'utiliser pour appeler des Suitelets internes).
- Le Suitelet doit être déployé de manière accessible à cet utilisateur (par exemple, mêmes autorisations de rôle).
- L'appel ne générera pas de page de connexion ; il attend plutôt le contenu interne directement. En cas de mauvaise configuration, vous pourriez obtenir un HTML de connexion ou une erreur 403.

De même, `requestRestlet` injecte automatiquement l'authentification de la session actuelle, de sorte que le RESTlet le considère comme provenant du même utilisateur. Si le script appelant avait un rôle limité, le RESTlet s'exécute sous ce rôle. Cette authentification automatique est une commodité majeure (pas besoin d'insérer manuellement les en-têtes NLAuth/NLTC). Notez que les documents Oracle avertissent que l'en-tête `Authorization` ne peut pas être défini manuellement dans ces appels – c'est interdit (Source: docs.oracle.com).

Appel asynchrone de Suitelet : `https.requestSuitelet.promise`

Le point central de ce rapport est la méthode `https.requestSuitelet.promise(options)` – une API asynchrone pour appeler un Suitelet. Officiellement introduite dans NetSuite 2023.1, elle s'appuie sur le `https.requestSuitelet` synchrone. La page d'aide la décrit ainsi :

> ***Méthode*** *:* `https.requestSuitelet.promise(options)` ***Retourne*** *: Objet Promise* ***Description de la méthode*** *: « Envoie une requête HTTPS de manière asynchrone à un Suitelet et renvoie la réponse. » (Source: docs.oracle.com). Elle précise qu'elle* ***« ne peut renvoyer un Suitelet interne que dans des contextes approuvés pour les utilisateurs authentifiés. »*** *Il est important de noter que la remarque indique qu'elle ne peut plus être utilisée pour des requêtes sortantes dans des contextes anonymes ; le paramètre* `options.external` *a été supprimé (Source: docs.oracle.com). Les versions synchrone et asynchrone partagent les mêmes paramètres et erreurs (Source: docs.oracle.com).*

Ainsi, pour utiliser `https.requestSuitelet.promise`, l'appelant fournit un objet `options` avec des champs analogues à ceux de `https.requestSuitelet` :

- **`scriptId` (chaîne, requis)** – l'ID de l'enregistrement de script du Suitelet (par exemple, `'customscript_my_suitelet'`).
- **`deploymentId` (chaîne, requis)** – l'ID de l'enregistrement de déploiement du Suitelet (par exemple, `'customdeploy_my_suitelet'`).
- **`method` (chaîne, facultatif)** – méthode HTTP (`'GET'`, `'POST'`, etc.). Par défaut, GET si le corps n'est pas défini.
- **`body` (chaîne, facultatif)** – Une chaîne ou un objet (généralement JSON) à envoyer en tant que corps POST/PUT. Ignoré si la méthode est GET/DELETE.
- **`headers` (Object, optionnel)** – En-têtes HTTP supplémentaires à envoyer.
- **`urlParams` (Object, optionnel)** – Paramètres de requête à ajouter à l'URL.

Ces paramètres reflètent la variante synchrone (Source: docs.oracle.com). Par exemple, un appel asynchrone à un Suitelet pourrait ressembler à ceci :

```
const response = await https.requestSuitelet.promise({
  scriptId: 'customscript_my_suitelet',
  deploymentId: 'customdeploy_my_suitelet',
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ customerId: 123, action: 'refresh' }),
  urlParams: { foo: 'bar' }
});
console.log('Statut du résultat :', response.code, 'corps :', response.body);
```

Ce code envoie des données JSON au Suitelet et attend sa réponse. La réponse est un objet de type `https.ClientResponse` (avec `.code`, `.body`, `.headers`, etc.), accessible une fois la promesse résolue (Source: docs.oracle.com).

Cas d'utilisation et exemples

Comme `requestSuitelet.promise` fonctionne à la fois dans les scripts client et serveur, divers modèles émergent :

- **Appel depuis un script client** : Un utilisateur clique sur un bouton ou effectue une action sur une page d'enregistrement. Au lieu d'une redirection complète après soumission de formulaire, le script client peut appeler le Suitelet en arrière-plan et gérer le résultat de manière asynchrone. Par exemple, l'exemple d'Oracle montre une fonction client utilisant le `https.requestSuitelet` synchrone, mais on pourrait utiliser sa version promise de la même manière (Source: docs.oracle.com). Dans une fonction client `async`, on pourrait écrire `let res = await https.requestSuitelet.promise({...}); dialog.alert({message: res.body});` pour afficher la réponse du Suitelet dans une boîte de dialogue, sans recharger la page. Comme le souligne Ben Rogol, ce modèle (souvent appelé AJAX) peut soit récupérer des données, soit déclencher une redirection selon l'effet souhaité (Source: test.suiterep.com).

- **Appel côté serveur non bloquant** : Supposons qu'un Suitelet contienne une logique déclenchée par un script User Event lors de l'enregistrement d'un enregistrement, mais que nous ne voulions pas que le processus d'enregistrement attende le Suitelet. En utilisant la variante promise, on *pourrait* lancer la requête sans l'attendre immédiatement, permettant au script de continuer. (Notez toutefois que dans des contextes d'exécution purement synchrones, si vous n'utilisez pas `await`, la promesse s'exécutera toujours, mais le script pourrait se terminer avant d'avoir consigné les résultats ou géré les retours. La prudence est de mise dans les scripts purement serveur). Si, au contraire, vous utilisez `await`, le processus reste bloquant jusqu'à son achèvement, mais avec une syntaxe plus propre. L'extrait de la communauté de documentation met explicitement en garde : **« le UE (user event) s'exécutera jusqu'à ce que la réponse soit renvoyée par le SL, à moins que vous n'utilisiez une promesse. »** (Source: archive.netsuiteprofessionals.com). En d'autres termes, sans `.promise`, vous n'avez pas d'autre choix que de bloquer. Avec `.promise` (et en utilisant correctement `await` ou `.then`), vous *pouvez* structurer votre code pour attendre si nécessaire ou non.

- **Orchestration de scripts planifiés** : Un script planifié (Scheduled) ou Map/Reduce pourrait générer plusieurs appels asynchrones vers des Suitelets. Pour les tâches à fort volume, un script pourrait appeler plusieurs points de terminaison Suitelet simultanément (surtout si chaque Suitelet traite une partie des données). Par exemple, on pourrait utiliser `Promise.all` sur un tableau d'appels `requestSuitelet.promise` pour les exécuter en parallèle, puis attendre tous les résultats. Ce modèle se recoupe avec l'utilisation de `Promise.all` pour plusieurs API externes.

- **Gestion des tâches longues** : Sur le forum des professionnels NetSuite, un développeur a signalé que son Suitelet prenait environ 25 secondes lors du premier appel et déclenchait une erreur `SSS_REQUEST_TIME_EXCEEDED` *côté script client*, bien que le Suitelet finisse par se terminer (Source: archive.netsuiteprofessionals.com). Cela illustre un piège : l'exécution côté client a un délai d'expiration (~20 s) pour l'exécution du script. Si un appel Suitelet est trop lent, la promesse du client peut être rejetée. Fait intéressant, cet utilisateur a noté que les appels suivants étaient plus rapides (~15 s), suggérant un délai de « démarrage à froid » lors de la première exécution. Une solution possible consiste à diviser les tâches lourdes en plusieurs appels plus petits ou à utiliser des tâches d'arrière-plan (comme MQ ou Map/Reduce) pour éviter d'atteindre le délai d'expiration.

Limitations et détails des paramètres

La documentation de NetSuite fournit des détails sur les paramètres et les codes d'erreur pour `requestSuitelet.promise` (Source: docs.oracle.com). Les points clés incluent :

- **Paramètre `method`** : S'il n'est pas spécifié, la valeur par défaut est `'GET'`. Si vous omettez un `body`, il suppose GET ; si vous incluez un corps, il utilise POST par défaut. Les verbes HTTP habituels (`DELETE`, `HEAD`, `POST`, `PUT`) sont autorisés. L'envoi d'un `body` avec GET/DELETE n'a aucun effet.
- **`urlParams`** : Un objet dont les clés/valeurs deviennent des paramètres de chaîne de requête. (SuiteScript encodera les valeurs en URL.) Utile pour envoyer plusieurs paramètres simples sans JSON.
- **En-têtes** : Vous pouvez envoyer des en-têtes personnalisés (par exemple, définir `'Content-Type': 'application/json'` ou d'autres jetons d'API vers votre Suitelet). Remarque : vous ne pouvez pas remplacer les propres en-têtes d'authentification de NetSuite (ceux-ci sont automatiquement définis ou interdits).
- **Réponse** : La `Promise` résolue renvoie un objet de réponse avec des propriétés telles que `code` (code de statut HTTP), `body` (chaîne de texte/JSON renvoyée par le Suitelet) et `headers`. Si la promesse est rejetée, elle génère une erreur (par exemple si le Suitelet renvoie un code d'erreur ou si l'appel a expiré).
- **Gouvernance** : Chaque appel à `https.requestSuitelet.promise` coûte 10 unités de gouvernance (Source: docs.oracle.com). Si vous lancez de nombreux appels (par exemple en parallèle), vous devez tenir compte du total. Des problèmes de disjonction ou de limitation de débit peuvent s'appliquer si des centaines d'appels se produisent rapidement.
- **Disponibilité** : Cette méthode asynchrone est prise en charge dans les scripts client et serveur (Source: docs.oracle.com), mais pas dans les **scripts client accédés de manière anonyme** (par exemple, sans connexion). En fait, Oracle déclare explicitement que vous ne pouvez pas l'utiliser du tout dans les clients anonymes (comme les vitrines SuiteCommerce) (Source: docs.oracle.com). Pour les cas anonymes, leur modèle est le suivant : faire en sorte que le client appelle un RESTlet interne (en utilisant `requestRestlet.promise`), qui effectue ensuite le travail externe (Source: docs.oracle.com).

En résumé, `https.requestSuitelet.promise` est un outil puissant pour intégrer les Suitelets dans les flux de travail SuiteScript sans blocage. Il encapsule les mécanismes HTTP sous-jacents afin que les développeurs puissent simplement fournir des ID de script et des données. La section suivante comparera cela avec l'appel de Suitelets par d'autres moyens et couvrira l'appel synchrone `requestSuitelet` associé pour être complet.

Appel synchrone de Suitelet : `https.requestSuitelet`

Pour être complet, nous décrivons l'équivalent synchrone `https.requestSuitelet(options)`. Cette méthode a été introduite en même temps que la version promise (2023.1) et est décrite comme suit :

> ***Méthode*** *:* `https.requestSuitelet(options)`
> ***Renvoie*** *: Objet* `https.ClientResponse`
> ***Description*** *: « Envoie une requête HTTPS à un Suitelet et renvoie la réponse. » (Source: docs.oracle.com). Comme pour la version promise, « Cette méthode ne peut renvoyer un Suitelet interne que dans des contextes de confiance pour les utilisateurs authentifiés. » Les paramètres (*`options.scriptId`*,* `options.deploymentId`*,* `method`*, etc.) sont identiques à ceux de la version promise (Source: docs.oracle.com) (Source: docs.oracle.com).*

La version synchrone envoie immédiatement la requête et bloque l'exécution jusqu'à ce que le Suitelet réponde (ou qu'une erreur HTTP se produise). Un exemple d'utilisation (tiré de la documentation d'Oracle) est :

```
// Script client (SuiteScript 2.x)
define(['N/https', 'N/ui/dialog'], function(https, dialog) {
    function callSuitelet() {
        var response = https.requestSuitelet({
            scriptId: 'customscript_sl_plf_requestsuitelet',
            deploymentId: 'customdeploy_sl_plf_requestsuitelet',
            // optionnel : method, body, headers, etc.
        });
        dialog.alert({
            title: 'Réponse du Suitelet',
            message: response.body
        });
    }
    return { callSuitelet: callSuitelet };
});
```

Ici, une fois que l'appel `requestSuitelet` est renvoyé, le script affiche immédiatement le résultat du Suitelet dans une boîte de dialogue (Source: docs.oracle.com). Aucune promesse ou rappel n'est impliqué – c'est simplement un appel synchrone.

Parce qu'il bloque, l'utilisation de `https.requestSuitelet` dans un script client déclenche ce comportement de délai d'expiration de 20 secondes noté précédemment. Dans cet exemple publié, le Suitelet était trivial (« Hello World ! ») et s'est terminé rapidement. Mais les développeurs doivent être prudents : si un Suitelet prend trop de temps, `requestSuitelet` générera une erreur `SSS_REQUEST_TIME_EXCEEDED` (comme cela s'est produit dans le cas du forum (Source: archive.netsuiteprofessionals.com). Les appels synchrones sont faciles à raisonner séquentiellement, mais l'inconvénient est que l'appelant (client ou serveur) ne peut effectuer aucun autre travail en attendant.

L'avènement de `requestSuitelet.promise` offre la même fonctionnalité avec une sémantique non bloquante. En fait, si l'on réécrivait l'exemple ci-dessus avec `await`, il se comporterait de manière identique du point de vue de l'utilisateur, mais vous permettrait de structurer le code différemment :

```
async function callSuiteletAsync() {
    var response = await https.requestSuitelet.promise({
        scriptId: 'customscript_my_json_sl',
        deploymentId: 'customdeploy_my_json_sl'
    });
    await dialog.alert({
        title: 'Réponse du Suitelet',
        message: response.body
    });
}
```

Cela utilise le `async` / `await` moderne dans SuiteScript 2.2, rendant le code visuellement similaire mais non bloquant en arrière-plan (Source: www.thenetsuitepro.com). Dans les deux cas, la consommation de gouvernance et les paramètres de l'API sont les mêmes.

Modèles : Utilisation asynchrone de Suitelet et RESTlet

Une fois les API définies, nous examinons maintenant les modèles et cas d'utilisation courants. Les développeurs ont découvert que la combinaison de `https.requestSuitelet.promise` et `https.requestRestlet.promise` avec d'autres fonctionnalités de SuiteScript permet des intégrations et des flux d'expérience utilisateur puissants. Nous discutons de plusieurs scénarios :

1. Script client appelant un Suitelet (modèle de type AJAX)

L'un des modèles les plus cités consiste à utiliser un Suitelet comme fournisseur de données dynamique pour les scripts client. Au lieu de recharger toute la page, un script client peut appeler un Suitelet puis mettre à jour l'interface utilisateur (DOM) en conséquence. Il s'agit essentiellement d'un modèle AJAX au sein de NetSuite.

Exemple (Récupération de données) : Un Suitelet peut accepter des paramètres de requête (comme des filtres) et renvoyer des données JSON (par exemple, un rapport récapitulatif). Le script client (s'exécutant dans une session de navigateur) peut transmettre l'ID de l'enregistrement actuel au Suitelet et `await` le résultat. Considérez le modèle de code de The NetSuite Pro (un blog) : il résolvait l'URL du Suitelet via `url.resolveScript`, puis effectuait un `fetch` avec `await`, puis mettait à jour un élément HTML avec `document.getElementById(...)` (Source: www.thenetsuitepro.com). En arrière-plan, on pourrait faire de même :

```
const suiteletResponse = await https.requestSuitelet.promise({
  scriptId: 'customscript_summary_suitelet',
  deploymentId: 'customdeploy_summary_suitelet',
  method: 'GET',
  urlParams: { recid: currentRecord.id }
});
const data = JSON.parse(suiteletResponse.body);
// Maintenant, mettez à jour les éléments de l'interface utilisateur avec data.summary, etc.
```

L'avantage est une expérience utilisateur fluide : pas de rafraîchissement complet de la page et l'utilisateur voit des mises à jour immédiates. Ce modèle repose sur le fait que le script côté client se trouve dans une session *connectée* (puisque des appels internes sont utilisés). Il nécessite également que le Suitelet soit raisonnablement rapide (généralement < 2-3 secondes) pour être convivial, bien que l'appel asynchrone ne fige pas l'interface utilisateur pendant l'attente.

Exemple (Fichier ou redirection) : Si le but du Suitelet est de générer un fichier ou de rediriger l'utilisateur, le client peut l'ouvrir dans une nouvelle fenêtre. Le modèle de Rogol note que pour les téléchargements (PDF, CSV), il faut utiliser quelque chose comme :

```
const url = url.resolveScript({ scriptId: 'customscript_pdf_suitelet', deploymentId: 'customdeploy_pdf_suitelet' });
window.open(url);
```

Cela contourne complètement `https.requestSuitelet` et déclenche une requête de navigateur normale vers le Suitelet, en tirant parti de la connexion de l'utilisateur. C'est synchrone dans la mesure où le navigateur navigue vers cette URL dans un nouvel onglet ou cadre, mais du point de vue du code SuiteScript, il s'agit simplement d'une navigation initiée par l'utilisateur. La conclusion de [56] est : **utilisez `https.get` ou `https.requestSuitelet` pour les données sur la même page, et `window.open` pour rediriger vers un formulaire ou un fichier Suitelet** (Source: test.suiterep.com).

Exemple Oracle : L'exemple officiel « Appeler un Suitelet depuis un script client » passe par un User Event ajoutant un bouton, puis une fonction de script client (`callSuitelet`) qui effectue :

```
const response = https.requestSuitelet({ scriptId: 'customscript_hello', deploymentId: 'customdeploy_hello' });
dialog.alert({ title: 'Réponse du Suitelet', message: response.body });
```

Cet exemple utilise la méthode synchrone (probablement de style 2.x) (Source: docs.oracle.com). Dans SuiteScript 2.1+, on pourrait faire la même chose avec `await` dans une fonction client asynchrone. Un équivalent moderne utilisant `promise` serait :

```
async function callSuitelet() {
    try {
        let res = await https.requestSuitelet.promise({
            scriptId: 'customscript_sl_plf_requestsuitelet',
            deploymentId: 'customdeploy_sl_plf_requestsuitelet'
        });
        await dialog.alert({ title: 'Réponse du Suitelet', message: res.body });
    } catch(e) {
        log.error('L\'appel au Suitelet a échoué', e);
    }
}
```

Cela donne le même comportement mais utilise une promesse JavaScript en interne.

Considérations : Lors de l'utilisation de ce modèle, soyez attentif au temps d'exécution. Les appels Suitelet modernes en 2024+ peuvent souvent prendre quelques secondes, ce qui est acceptable pour les interactions utilisateur. Cependant, les tâches longues (dizaines de secondes) dépasseront les délais d'expiration des scripts client. Si un traitement lourd est nécessaire, le Suitelet peut effectuer un traitement minimal et éventuellement mettre en file d'attente un travail en arrière-plan (par exemple, dans un script Map/Reduce ou planifié) pour éviter de bloquer l'interface utilisateur.

2. Script client appelant un Restlet (modèle de proxy)

Parfois, un script client a besoin de données provenant d'une API externe (comme un tarif d'expédition d'UPS ou une API fiscale gouvernementale). NetSuite interdit les appels HTTP externes directs dans des contextes anonymes ou même dans de nombreux contextes client, donc une solution de contournement courante est :

1. Créer un **RESTlet** SuiteScript (côté serveur) qui n'est accessible qu'aux utilisateurs connectés.
2. Le code du RESTlet lui-même effectue l'appel HTTPS externe (en utilisant `N/https.get` ou `N/https.request()`).
3. Le script client (basé sur le navigateur) appelle *ce* RESTlet en utilisant `https.requestRestlet.promise`.
4. Le client attend la réponse du RESTlet, puis l'utilise.

Cela relaie efficacement les appels externes via SuiteScript. La documentation décrit spécifiquement ce modèle : « *Utilisez cette méthode [requestRestlet.promise] pour effectuer de manière asynchrone une requête HTTPS sortante dans un contexte côté client anonyme. Vous pouvez le faire en effectuant la requête HTTPS à l'intérieur d'un Restlet qui n'est disponible qu'avec une connexion, puis en appelant le Restlet à l'intérieur de votre script client en utilisant la méthode https.requestRestlet.promise(options).* » (Source: docs.oracle.com).

Un exemple de flux :

```
// Script client (SuiteScript, attendant la connexion, peut même être sur une page de portail accessible par Internet)
async function getExternalData() {
    const res = await https.requestRestlet.promise({
        scriptId: 'customscript_external_proxy',
        deploymentId: 'customdeploy_external_proxy',
        method: 'POST',
        body: JSON.stringify({ param1: 'value' }),
        headers: { 'Content-Type': 'application/json' }
    });
    const data = JSON.parse(res.body);
    // Utiliser les données dans le script client (mettre à jour l'interface utilisateur, etc.)
}
```

Le RESTlet `external_proxy` est un simple SuiteScript qui prend la requête entrante et effectue, par exemple :

```
define(['N/https'], function(https) {
    function post(context) {
        // Effectuer l'appel externe
        let extRes = https.get({ url: 'https://third-party.com/api?param=' + context.requestBody.param1 });
        return extRes.body; // Ou JSON.parse, etc.
    }
    return { post: post };
});
```

Étant donné que `https.requestRestlet.promise` attache automatiquement l'authentification, le RESTlet considère que l'appel provient d'un utilisateur authentifié et peut s'exécuter avec les permissions de cet utilisateur si nécessaire (Source: docs.oracle.com) (Source: docs.oracle.com). Le script client récupère alors simplement les données externes comme s'il s'agissait de données internes.

Ce modèle est particulièrement important lorsque le code côté client n'est *pas authentifié* (par exemple, une vitrine SuiteCommerce). Les appels directs comme `fetch('https://thirdparty.com')` depuis le navigateur violeraient les règles de cross-origin ou de contexte. En passant par un RESTlet connecté, le contexte de sécurité est valide. Après cela, on peut même utiliser la méthode `https.requestRestlet.promise` nouvellement disponible

depuis un client authentifié (dans l'interface utilisateur de NetSuite) pour des tâches de proxy similaires (Source: docs.oracle.com) (Source: docs.oracle.com).

3. Script serveur démarrant une tâche asynchrone

Un SuiteScript s'exécutant en tant que User Event, Scheduled ou Map/Reduce peut utiliser `https.requestSuitelet.promise` de deux manières :

- **Attendre un processus long** : Le script peut avoir besoin de données provenant d'un Suitelet ou déclencher des actions sur celui-ci. L'utilisation de `await` garantit que le script attend la fin de l'exécution. Par exemple, un script Scheduled pourrait appeler un Suitelet qui renvoie le statut d'une intégration, puis poursuivre en fonction de ce résultat. En effet, `await` sur `requestSuitelet.promise` n'est pas très différent d'un appel à `requestSuitelet` – les deux sont bloquants, sauf que l'un utilise la syntaxe des promesses.
- **« Fire and forget » (lancer et oublier)** : Si vous souhaitez *lancer* quelque chose sans attendre, vous pourriez théoriquement appeler `https.requestSuitelet.promise({...});` sans `await`. La promesse s'exécuterait en arrière-plan et le script passerait à la suite. Cependant, comme SuiteScript est monothread et que le contexte se termine une fois votre code terminé, il n'est pas garanti que l'exécution en arrière-plan se termine complètement. En pratique, on pourrait utiliser cette méthode pour lancer un Suitelet qui effectue un travail (peut-être que le Suitelet écrit dans la base de données) sans se soucier du résultat. Mais c'est délicat : il n'existe pas de fonction officielle « fire and forget » dans SuiteScript ; tout appel que vous n'attendez pas est essentiellement abandonné ou limité au thread en cours d'exécution. Un modèle plus fiable pour les tâches en arrière-plan consiste à utiliser directement un script **Scheduled ou Map/Reduce**, ou à enchaîner des scripts planifiés (un User Event met en file d'attente un job planifié, etc.).

En résumé, les scripts côté serveur peuvent utiliser des appels asynchrones, mais ils doivent gérer soigneusement le timing. Si un User Event appelle un Suitelet et l'attend, l'enregistrement ne sera pas sauvegardé tant que le Suitelet n'aura pas terminé, tout comme avec l'appel synchrone. Le commentaire de la documentation sur le forum reflète cela : sans utiliser la forme promesse pour restructurer le code, *« le UE s'exécutera jusqu'à ce que la réponse soit renvoyée par le SL »* (Source: archive.netsuiteprofessionals.com). En bref, les méthodes asynchrones ne parallélisent pas par magie un processus serveur à moins d'être codées pour le faire (par exemple, en lançant des promesses parallèles avec `Promise.all`).

4. Modèles de haute concurrence et de performance

Les appels asynchrones aux Suitelets ouvrent des stratégies de traitement parallèle. Une étude de cas notable de la communauté NetSuite a comparé l'exécution d'une tâche via Map/Reduce par rapport à des Suitelets parallèles (Source: ursuscode.com) (Source: ursuscode.com). Dans un test contrôlé, un compte a tenté de « Créer et supprimer un enregistrement personnalisé » 10 000 fois. L'approche Map/Reduce (sérialisation ou petits lots) a pris plusieurs minutes, alors que le lancement de nombreuses requêtes Suitelet simultanées a considérablement accéléré le processus :

- Avec 50 appels Suitelet en parallèle, ils ont atteint ~181 opérations/sec (55 secondes au total).
- Avec 250 appels parallèles, ce chiffre est passé à ~416 ops/sec (24 secondes).
- En revanche, Map/Reduce a plafonné à environ 51 ops/sec (196 secondes) ou 19 ops/sec avec un tampon plus grand (Source: ursuscode.com).
- Finalement, ils « pouvaient traiter 1 million d'enregistrements en 40 minutes avec le Suitelet contre 5h30 avec un Map/Reduce… » (Source: ursuscode.com).

Ces chiffres sont résumés dans le Tableau 2 ci-dessous. Ils illustrent que, lorsque vous pouvez utiliser des invocations de Suitelet parallèles (par exemple via un outil externe comme JMeter ou des appels `https.requestSuitelet.promise` simultanés), vous pouvez surpasser Map/Reduce pour certaines tâches. Les Suitelets deviennent effectivement un service web léger traitant chaque requête indépendamment, tirant parti de la capacité de NetSuite à mettre à l'échelle plusieurs sessions. (En production, Oracle pourrait limiter l'utilisation intensive de Suitelets simultanés ou exiger une licence SuiteCloud Plus pour une haute concurrence.)

SCÉNARIO DE TEST	TEMPS (S)	DÉBIT (REQ/SEC)
Map/Reduce (taille de lot 1)	196	51
Map/Reduce (taille de lot 64)	518	19
Suitelets (appels externes, 50 simultanés)	55	181
Suitelets (appels externes, 100 simultanés)	30	333
Suitelets (appels externes, 200 simultanés)	24	416
Suitelets (appels externes, 250 simultanés)	24	416
Suitelets (AJAX navigateur, 1 simultané)	44	22

Tableau 2 : Comparaison des performances du traitement d'une opération par lots via Map/Reduce par rapport aux appels Suitelet simultanés (source : Ursus Code) (Source: ursuscode.com) (Source: ursuscode.com).

Il s'agit d'un cas extrême (blog Ursus Code, 2017) et non représentatif de la plupart des logiques métier. Néanmoins, cela souligne que les invocations asynchrones de Suitelets peuvent exploiter les capacités de traitement parallèle de NetSuite. Notez toutefois qu'une telle concurrence consomme de nombreuses unités de gouvernance et peut atteindre les limites du réseau ou du navigateur. L'auteur du blog a noté qu'au-delà d'environ 250 requêtes parallèles, des erreurs ont commencé à se produire. De plus, cela nécessitait un outil de test spécialisé (Apache JMeter) pour émettre de nombreuses requêtes simultanées ; les scripts clients/appels REST normaux n'utiliseraient probablement pas des centaines d'appels simultanés.

5. RESTlet vs Suitelet : Choisir le bon point de terminaison

Souvent, il faut décider s'il vaut mieux implémenter une fonctionnalité sous forme de Suitelet ou de RESTlet. Les Suitelets peuvent renvoyer du HTML, des PDF ou du JSON, et sont généralement utilisés pour créer des interfaces utilisateur personnalisées ainsi que des points de terminaison de données. Les RESTlets sont toujours des API JSON. Les différences clés pour une utilisation asynchrone incluent :

- **Authentification** : Un appel interne à un Suitelet via `requestSuitelet` conserve la session de l'utilisateur actuel. Un appel à un RESTlet via `requestRestlet` hérite de la même manière de l'authentification. Cependant, un RESTlet disponible sans connexion peut également être appelé de manière externe par des outils d'intégration (pas dans un SuiteScript).
- **Format de données** : Les Suitelets peuvent renvoyer du texte/JSON arbitraire. Les RESTlets renvoient généralement du JSON. Si votre script asynchrone souhaite du JSON, vous pouvez utiliser l'un ou l'autre ; souvent, les RESTlets sont légèrement plus simples pour les données pures, tandis que les Suitelets peuvent faire les deux.
- **Appels externes** : Si vous devez appeler un service web externe, les Suitelets et les RESTlets (étant des scripts serveur) peuvent le faire. Les modèles asynchrones pour les appels sont analogues. Mais si vous appelez depuis l'intérieur de NetSuite, `https.requestRestlet` existe depuis plus longtemps (depuis 2020.2), alors que `requestSuitelet` n'est apparu que récemment (Source: docs.oracle.com) (Source: docs.oracle.com).
- **Utilisation dans le code client** : Les deux types peuvent être appelés depuis des scripts clients (via `https.requestSuitelet.promise` ou `https.requestRestlet.promise`), mais comme `requestRestlet.promise` est réservé aux « scripts serveur » selon la documentation (Source: docs.oracle.com), on le structure souvent ainsi : le client appelle le RESTlet -> le RESTlet appelle réellement le Suitelet ou effectue le travail. En pratique, les développeurs ont largement convergé vers ces modèles : soit utiliser des Suitelets pour des interfaces utilisateur complexes et les attacher aux scripts clients, soit utiliser des RESTlets pour des points de terminaison d'intégration de données pures.

6. Gouvernance et limitations

Bien que les modèles asynchrones offrent de la flexibilité, ils ne modifient pas le modèle de gouvernance et de sécurité sous-jacent de NetSuite. Plusieurs considérations importantes :

- **Unités de gouvernance** : Chaque appel à `https.requestSuitelet` ou `https.requestSuitelet.promise` coûte **10 unités** (Source: docs.oracle.com). Si l'on génère de nombreux appels simultanés, le total des unités s'accumule. Dans un grand compte avec de nombreux scripts, atteindre les limites de gouvernance pourrait être un problème. En revanche, l'utilisation d'async/await ou de Promises en soi ne réduit pas les unités ; un appel bloqué et un appel promis coûtent la même chose.
- **Temps d'exécution du script** : Les scripts clients ont une limite d'exécution d'environ 20 secondes. Si votre appel Suitelet asynchrone prend plus de temps, le script client renverra une erreur (comme vu sur le forum (Source: archive.netsuiteprofessionals.com). Les scripts serveur ont des limites plus élevées (par exemple 5 ou 20 minutes pour les scripts planifiés), mais les appels Suitelet de longue durée peuvent toujours affecter l'expérience utilisateur ou le temps de traitement par lots.
- **Gestion des erreurs** : Lors de l'utilisation de Promises, les erreurs doivent être interceptées via `.catch()` ou try/catch avec `await`. Sinon, un rejet non intercepté tuera le script. Par exemple, `https.requestSuitelet.promise` pourrait rejeter avec `SSS_REQUEST_TIME_EXCEEDED`, `INVALID_SCRIPT_DEPLOYMENT_ID` ou d'autres codes si l'ID du script est incorrect ou a expiré (Source: docs.oracle.com) (Source: docs.oracle.com). Les scripts robustes doivent intercepter et journaliser ces erreurs.
- **URLs internes vs externes** : Comme indiqué à plusieurs reprises, ces méthodes ne fonctionnent qu'avec des URLs de Suitelet internes. Tenter de les utiliser avec une URL externe (publique) n'est pas pris en charge. Les avis d'Oracle mi-2024 ont souligné que les appels utiliseraient par défaut des URLs internes et que les appels externes (`option.external`) cesseraient de fonctionner (Source: docs.oracle.com) (Source: community.oracle.com). Si vous avez un Suitelet qui doit être accessible par des clients extérieurs, vous ne pouvez pas utiliser `requestSuitelet` dessus – vous devriez appeler son URL externe via un SuiteTalk traditionnel ou depuis un système d'intégration externe.
- **SuiteCommerce et Portails** : Dans SuiteCommerce (pages d'achat) ou d'autres pages NetSuite non authentifiées, vous ne pouvez pas utiliser ces méthodes directement (comme le prévient la documentation de `url.resolveScript` (Source: docs.oracle.com). Au lieu de cela, il faut utiliser le modèle de proxy RESTlet (Source: docs.oracle.com). Ainsi, même si `requestSuitelet.promise` peut techniquement être utilisé dans des scripts clients (puisqu'il est répertorié comme « Client et serveur » (Source: docs.oracle.com), ce contexte client doit être authentifié (comme un script déployé sur une page d'enregistrement NetSuite).

Analyse des données et preuves

Bien que les modèles SuiteScript soient largement qualitatifs, nous pouvons citer des données lorsqu'elles sont disponibles :

- **Adoption de NetSuite** : L'utilisation généralisée de la plateforme souligne l'importance d'un scripting efficace. Anchor Group rapporte que NetSuite est passé d'environ 10 à 11 000 clients en 2016 à plus de **40 000** en 2024 (Source: www.anchorgroup.tech). Si chaque client possède en moyenne plusieurs personnalisations scriptées, l'échelle d'utilisation de SuiteScript est énorme. De plus, environ 80 % de la base de NetSuite sont des petites et moyennes entreprises (Source: www.anchorgroup.tech), qui s'appuient souvent fortement sur le scripting personnalisé pour l'automatisation des processus. (Pour le contexte, fin 2025, NetSuite servait *plus de* 40 000 clients dans le monde (Source: www.anchorgroup.tech).) L'implication est que toute amélioration des modèles de scripting (comme les Suitelets asynchrones) profite potentiellement à des milliers d'organisations.
- **Comportement asynchrone** : Du point de vue async/await, l'adoption généralisée de JavaScript suggère que de nombreux développeurs trouvent le code basé sur les promesses plus facile à maintenir. Bien que nous manquions d'enquêtes spécifiques à SuiteScript, MDN et la pédagogie JavaScript soulignent que les promesses améliorent la clarté du code pour les flux asynchrones (Source: docs.oracle.com) (Source: docs.oracle.com). Une enquête axée sur NetSuite a indiqué que 83 % des entreprises atteignent leurs objectifs de retour sur investissement grâce à une planification minutieuse (pas directement lié aux outils de développement, mais souligne la valeur des processus robustes (Source: www.anchorgroup.tech). Les preuves anecdotiques (StackOverflow, forums) montrent des questions croissantes sur SuiteScript asynchrone, indiquant un intérêt actif des développeurs.
- **Performance** : Le benchmark d'Ursus Code était la donnée la plus concrète trouvée sur les performances des Suitelets parallèles (Source: ursuscode.com) (Source: ursuscode.com). Il démontre une accélération d'environ **8x** (40 min contre 5,5 heures) pour le traitement d'un million d'enregistrements via des Suitelets simultanés par rapport à Map/Reduce (Source: ursuscode.com). Une autre façon de l'interpréter : l'utilisation de la concurrence (200 appels simultanés) a donné ~416 ops/sec (24s pour 10 000 tâches) contre ~51 ops/sec avec map/reduce (196s pour 10 000). C'est *extrême* mais convaincant. Nous intégrons ces chiffres dans le Tableau 2 comme point de données et notons qu'ils proviennent d'un test en sandbox.
- **Occurrence de timeout** : L'exemple du forum (Source: archive.netsuiteprofessionals.com) fournit des preuves anecdotiques d'un appel Suitelet d'environ 25s atteignant un timeout. Le fait qu'après le premier succès (~25s), les appels ultérieurs se soient exécutés en ~15s suggère une mise en cache possible ou un réchauffement de la charge. Aucune statistique formelle n'existe, mais cette expérience peut laisser penser que les invocations initiales de Suitelet peuvent être plus lentes. Cela suggère que les développeurs pourraient constater une latence plus élevée lors de la première exécution (similaire à un « démarrage à froid ») et devraient peut-être effectuer un appel de « réchauffement » initial si le processus est orienté utilisateur.

Ces points de données, bien que limités, soutiennent l'idée que les modèles d'appel asynchrone peuvent améliorer considérablement les performances (lorsqu'ils sont correctement parallélisés) et que le comportement temporel peut varier (impactant l'expérience utilisateur). Nous les combinons avec l'avis d'experts :

- **Avis d'expert** : Dans un article LinkedIn (octobre 2025), le consultant NetSuite Bernie Consigo souligne l'importance de la nouvelle méthode `https.requestSuitelet` : il note que cela « marque un changement significatif » pour les personnalisations SuiteScript (Source: www.linkedin.com). Plus précisément, il souligne que la nouvelle API *simplifie* l'invocation de Suitelet dans le contexte de NetSuite, réduisant la dépendance à la construction d'URLs externes et à la configuration fastidieuse de l'authentification (Source: www.linkedin.com). L'article souligne que les développeurs *doivent* fournir `scriptId` et `deploymentId`, et que la réponse est un objet `ClientResponse` (Source: www.linkedin.com). Cela correspond à la documentation officielle : l'exigence clé est de spécifier les IDs de script/déploiement, puisqu'il n'y a plus de paramètre de domaine externe. En effet, le résumé de Consigo sert de confirmation experte de l'expérience plus propre.

- **Cas communautaire** : Le forum des professionnels NetSuite contient des questions-réponses entre développeurs. Dans un fil de discussion, quelqu'un a demandé si un script User Event pouvait appeler directement un Suitelet ; la réponse a été oui, mais que le script User Event se bloquerait jusqu'au retour du Suitelet, à moins d'utiliser la version « promise » (Source: archive.netsuiteprofessionals.com). Ce conseil pratique souligne l'impact sur le temps d'exécution. De tels exemples communautaires servent de preuves informelles mais crédibles du comportement de ces API dans des scripts réels.

- **Documentation Oracle** : Toutes les affirmations officielles (par exemple, le fonctionnement des API, leurs contraintes) sont documentées dans l'aide SuiteScript d'Oracle. Nous citons fréquemment ces pages (par exemple, les descriptions des méthodes (Source: docs.oracle.com) (Source: docs.oracle.com), le guide de gestion des promesses (Source: docs.oracle.com), la page de résolution d'URL (Source: docs.oracle.com), etc.). Celles-ci font autorité. Pour les informations sur les performances et les cas d'utilisation, nous nous appuyons également sur des sources non officielles mais réputées : des blogs de développeurs reconnus (Ursus Code (Source: ursuscode.com) (Source: ursuscode.com) et des sites de questions-réponses de haute qualité (StackOverflow et forums communautaires avec plus de 1 000 vues).

En résumé, les preuves — allant de la documentation à l'expérience anecdotique des développeurs en passant par les benchmarks — soulignent que `https.requestSuitelet.promise` est un ajout pris en charge et bénéfique à SuiteScript 2.x, permettant des scripts plus réactifs et évolutifs. Nous passons maintenant aux implications et aux orientations futures.

Études de cas et exemples concrets

Pour illustrer les modèles précédents, nous présentons quelques études de cas condensées basées sur des scénarios réels rapportés par la communauté.

Étude de cas 1 : Création de devis déclenchée par un bouton
Une entreprise disposait d'un script client sur un enregistrement de type « Case » qui, lors du clic sur un bouton, devait créer un devis via un Suitelet en arrière-plan. Ils ont utilisé `https.requestSuitelet.promise` dans le script client pour appeler un Suitelet qui générait le devis. Lors des tests en sandbox, la première création de devis de la journée prenait environ 25 secondes, provoquant une erreur `SSS_REQUEST_TIME_EXCEEDED` sur le script client. Cependant, le devis était correctement créé sur le serveur ; les créations de devis suivantes au cours de la même journée se terminaient en environ 15 secondes sans erreur (Source: archive.netsuiteprofessionals.com). Les développeurs ont remarqué ce délai important lors de la première exécution et ont posé la question sur le forum. Le consensus était que le Suitelet mettait plus de temps à s'initialiser lors de sa première exécution (peut-être en raison du chargement de bibliothèques ou du réchauffement des caches) et que `requestSuitelet.promise` expirait côté client après environ 20 secondes. Ils ont envisagé des solutions telles que diviser la tâche, effectuer un « pré-chauffage » à la connexion, ou passer à un script planifié.

Cela met en évidence un problème réel d'expérience utilisateur : *même avec async/await*, le script client finit par expirer. Un remède potentiel consiste à ce que le Suitelet effectue moins de travail lors du premier appel ou à utiliser une approche alternative. Par exemple, ils auraient pu faire en sorte que le client déclenche simplement un *script planifié*, puis interroge son statut via `requestSuitelet.promise`. En effet, le modèle 3 de « The NetSuite Pro » préconise de démarrer un Map/Reduce et d'interroger le statut (Source: www.thenetsuitepro.com).

Étude de cas 2 : Interrogation (polling) de tâches longues
Dans la continuité du cas précédent, supposons que nous adaptions la conception. Le script client déclenche un script planifié (via `N/task`) qui effectue la création du devis (ce qui prend plusieurs secondes). Le client appelle ensuite périodiquement un Suitelet pour vérifier le statut de la tâche. En utilisant `await https.requestSuitelet.promise` dans une fonction `async` (avec un court délai), le client peut rester réactif. Cela décharge le travail lourd et transforme l'appel du client en une vérification rapide du statut. De nombreux développeurs utilisent ce modèle : démarrer une tâche longue de manière asynchrone (en Map/Reduce ou script planifié) et faire en sorte que l'interface utilisateur interroge le statut via AJAX pour vérifier l'achèvement. L'appel de Suitelet basé sur les promesses permet de coder cette interrogation de manière simple avec `async/await`.

Étude de cas 3 : Récupération de données en parallèle
Un développeur souhaitait récupérer des données à partir de plusieurs Suitelets internes en parallèle pour afficher un tableau de bord. Il disposait de trois petits Suitelets (A, B, C) qui renvoyaient chacun une donnée. En utilisant l'asynchrone, le code du script client faisait :

```
let [resA, resB, resC] = await Promise.all([
  https.requestSuitelet.promise({scriptId: 'custscript_A', deployId: '1'}),
  https.requestSuitelet.promise({scriptId: 'custscript_B', deployId: '1'}),
  https.requestSuitelet.promise({scriptId: 'custscript_C', deployId: '1'})
]);
// Procéder au traitement de resA.body, etc.
```

Cela a réduit le temps d'attente total à essentiellement `max(timeA, timeB, timeC)` au lieu de la somme. Le résultat a été un chargement plus rapide du tableau de bord. Ce modèle de requêtes parallèles avec `Promise.all` est une technique JavaScript standard désormais disponible dans SuiteScript 2.1+. Elle nécessite que les trois Suitelets soient indépendants et que la gouvernance combinée (3x10 unités) soit acceptable.

Étude de cas 4 : Script client avec appel REST (via proxy)
Sur un site SuiteCommerce, le code front-end devait obtenir les taux de taxe actuels à partir d'une API externe. Les appels « fetch » directs étaient bloqués. La solution a consisté à créer un Suitelet ou un RESTlet (avec authentification) qui interroge l'API externe et renvoie du JSON. Le front-end (ou un script client sur un enregistrement) a ensuite utilisé `https.requestRestlet.promise` pour appeler ce proxy. Cela a permis une récupération asynchrone de données externes de manière sécurisée. L'utilisation de `requestRestlet` par le proxy a permis d'éviter les problèmes de CORS et la nécessité d'en-têtes inter-domaines.

Étude de cas 5 : Traitement fractionné avec des Suitelets
Une intégration nécessitait la mise à jour de milliers d'enregistrements quotidiennement sur la base de données externes. Le développeur a choisi de ne pas faire une simple boucle, mais a implémenté un Suitelet répartiteur. Le script User Event appelait le Suitelet qui, une fois sollicité, déclenchait lui-même plusieurs autres Suitelets (en utilisant des appels `requestSuitelet` séparés, éventuellement combinés avec `Promise.all`). Chacun de ces Suitelets secondaires gérait une partie des enregistrements. Conceptuellement, le Suitelet A effectuait une « diffusion » vers les Suitelets B1, B2, B3. L'avantage était le parallélisme ; l'inconvénient était la complexité. En pratique, ils ont constaté que le système NetSuite pouvait gérer des dizaines de Suitelets parallèles avant d'atteindre des délais d'attente ou des problèmes de gouvernance.

Ces cas illustrent à la fois la puissance et les précautions liées à l'utilisation des Suitelets asynchrones. En général, les appels asynchrones sont appropriés lorsque l'expérience utilisateur peut en bénéficier (interface utilisateur non bloquante) ou lorsque l'exécution parallèle augmente le débit. Lorsque des tâches longues sont impliquées, le fractionnement ou la mise en file d'attente est conseillé.

Implications et orientations futures

L'avancée des modèles de Suitelet/RESTlet asynchrones dans SuiteScript a plusieurs implications pour les pratiques de développement NetSuite et l'avenir de la plateforme :

- **Code plus propre** : Les évolutions vers les promesses et `async/await` alignent SuiteScript sur les meilleures pratiques JavaScript modernes. Le code est plus facile à maintenir ; par exemple, en évitant les rappels (callbacks) profondément imbriqués ou les flux confus, comme le souligne un blog (Source: www.thenetsuitepro.com). Comme l'a noté un ingénieur Oracle dans la documentation sur le traitement asynchrone : l'utilisation de promesses rend le code asynchrone « intuitif et efficace » (Source: docs.oracle.com). Nous pouvons nous attendre à ce que davantage de développeurs adoptent `async/await` dans les bases de code SuiteScript 2.1+ compte tenu de ces ajouts.
- **Meilleur réglage des performances** : Disposer de méthodes asynchrones officielles signifie que les développeurs expérimenteront des architectures parallèles. Par exemple, alors que les scripts Map/Reduce étaient le choix traditionnel pour les tâches parallèles, les Suitelets pourraient devenir compétitifs pour certains cas d'utilisation (comme le montre l'étude d'Ursus Code). Les consultants et les clients doivent cependant rester vigilants : la conception monothread forcée de Salesforce est très différente ; NetSuite autorise plusieurs sessions simultanées, mais des charges lourdes pourraient déclencher des limitations. Oracle pourrait continuer à ajuster la limitation des appels API ou proposer des suggestions sur le choix entre « Suitelet vs Map/Reduce ».
- **Interaction améliorée avec le navigateur** : Les scripts clients peuvent désormais récupérer plus facilement des données en arrière-plan. Nous anticipons davantage de pages personnalisées interactives sans rafraîchissement complet. Cela pourrait conduire à des extensions NetSuite plus dynamiques (par exemple, des Suitelets intégrés dans les pages NetSuite, des mises à jour de formulaires dynamiques). Cependant, les développeurs

doivent toujours gérer la compatibilité des navigateurs et les délais d'attente. La nécessité de s'exécuter dans des contextes authentifiés signifie que les pages réellement publiques (comme le commerce électronique) nécessitent toujours des solutions de contournement (comme des portlets cachés ou des points de terminaison côté serveur).

- **Évolution de l'API** : NetSuite a montré dans les versions récentes une tendance à ajouter des méthodes basées sur les promesses (par exemple, `query.runSuiteQL.promise`, `search.runPaged().iterator().each` retournant des promesses, etc.). La question est de savoir si les futures versions fourniront davantage de hooks asynchrones *conviviaux pour le client*. Par exemple, nous pourrions voir des appels asynchrones dédiés pour davantage de points d'induction, ou des moyens plus simples de planifier des flux asynchrones. Actuellement, de nombreuses tâches asynchrones nécessitent encore une gestion manuelle des exceptions pour les délais d'attente et les architectures fractionnées.

- **Intégration avec l'IA et les services modernes** : La feuille de route de NetSuite met désormais fortement en avant les modules d'IA (N/machineTranslation, N/documentCapture, etc. dans les notes de version 2025.2 (Source: docs.oracle.com). De nombreux services d'IA sont pilotés par API (par exemple, un appel REST vers un service de traduction). Nous pouvons prévoir d'utiliser `https.requestRestlet.promise` comme pont vers ces nouveaux services NAI ou vers des API GPT externes, etc. À mesure qu'un plus grand nombre de modules de NetSuite utiliseront des API REST asynchrones, la capacité à les intégrer de manière transparente via SuiteScript pourrait s'étendre. Par exemple, si une API d'IA générative est disponible via un point de terminaison Suitelet, on pourrait l'appeler de manière asynchrone lors de l'enregistrement d'une fiche pour enrichir les données.

- **Outils de développement** : Les mises à jour de SuiteScript suggèrent qu'Oracle continue de moderniser sa plateforme de développement. Avec les mises à jour de 2025 faisant référence au « SuiteCloud Developer Assistant » et aux nouveaux modules, SuiteScript est susceptible de bénéficier de meilleurs outils hors ligne, de linters ou de frameworks de test. Cela pourrait aider les développeurs à écrire et à déboguer le code asynchrone plus efficacement.

En termes de limitations, la nécessité de déployer des Suitelets/RESTlets pour les appels asynchrones signifie que les contraintes de gouvernance s'appliquent toujours. Les changements futurs pourraient inclure des augmentations de quotas ou des modèles de concurrence alternatifs. De plus, bien que `requestSuitelet.promise` soit idéal pour les appels de Suitelet internes, il n'aide pas à appeler un Suitelet depuis des systèmes externes – pour cela, il faudrait utiliser SuiteTalk (SOAP/GraphQL) ou des Restlets externes.

En résumé, l'introduction de ces méthodes asynchrones est une évolution significative pour SuiteScript. Elle débloque de nouveaux modèles de conception et aligne le développement NetSuite sur les pratiques JavaScript contemporaines. Comme l'indiquent à la fois la documentation d'Oracle et les voix de la communauté, les développeurs doivent se préparer en apprenant les nouveaux contrats d'API et en adaptant leurs scripts pour utiliser `async/await` là où c'est approprié. Une gestion appropriée des erreurs, la compréhension de la gouvernance et les tests seront essentiels pour exploiter efficacement les Suitelets asynchrones.

Conclusion

`https.requestSuitelet.promise` de SuiteScript 2.x et les API asynchrones associées représentent une avancée majeure dans l'évolution de SuiteScript. Ils permettent aux développeurs SuiteScript d'effectuer des appels de Suitelet et de RESTlet sans bloquer l'exécution, permettant des interactions côté client plus riches et des gains de débit dans les processus côté serveur. Une documentation complète et des exemples communautaires détaillent comment utiliser ces méthodes (nécessitant scriptId/deploymentId, etc. (Source: docs.oracle.com), quels contextes les prennent en charge (client authentifié ou serveur (Source: docs.oracle.com), et en quoi ils diffèrent des techniques plus anciennes.

Le contexte historique a vu SuiteScript passer d'appels entièrement synchrones à la prise en charge des promesses et d'async/await (notamment avec SuiteScript 2.1 en 2018). Le module N/https s'est développé en conséquence, avec des variantes retournant des promesses pour ses méthodes d'appel. Les sources officielles soulignent que les modèles d'utilisation (options, comportement en cas d'erreur) reflètent ceux des méthodes synchrones (Source: docs.oracle.com) (Source: docs.oracle.com). Des experts comme Bernie Consigo ont écrit sur les avantages de la nouvelle API pour simplifier le code et s'aligner sur la gouvernance (Source: www.linkedin.com), faisant écho à nos conclusions.

Nous avons examiné différents modèles : flux de travail de type AJAX côté client, communication serveur à serveur, proxys d'intégration et exécution parallèle haute performance. Les preuves concrètes montrent des avantages de performance significatifs dans certains cas (Source: ursuscode.com) (Source: ursuscode.com), mais soulignent également des mises en garde concernant les délais d'attente et les limites (Source: archive.netsuiteprofessionals.com) (Source: archive.netsuiteprofessionals.com). Ces idées, ainsi que les données de performance et les statistiques d'utilisation, soutiennent l'affirmation selon laquelle les modèles asynchrones sont à la fois pratiques et percutants pour le scripting NetSuite (surtout compte tenu de la large base installée de NetSuite de plus de 40 000 entreprises (Source: www.anchorgroup.tech) (Source: www.anchorgroup.tech).

À l'avenir, nous nous attendons à ce qu'Oracle continue d'affiner les capacités asynchrones de SuiteScript, en étendant éventuellement la prise en charge à davantage de modules et en améliorant les outils de développement. À mesure que NetSuite intègre davantage de fonctionnalités d'IA et d'API externes, la capacité d'appeler des services Web de manière asynchrone depuis SuiteScript ne fera que gagner en importance. Les développeurs devraient donc

investir du temps pour bien comprendre ces modèles : maîtriser la syntaxe des promesses, gérer les erreurs et les délais d'attente, et concevoir des systèmes qui tirent parti des appels non bloquants.

En conclusion, **`https.requestSuitelet.promise` et les méthodes asynchrones analogues sont désormais des outils clés dans l'arsenal de SuiteScript 2.x**. Ils offrent de nouvelles possibilités pour le développement personnalisé de NetSuite, des améliorations de l'interface utilisateur réactive au traitement parallèle des données. En suivant les meilleures pratiques documentées et en apprenant à la fois des exemples officiels d'Oracle et des expériences de la communauté, les développeurs NetSuite peuvent tirer parti des Suitelets et RESTlets asynchrones pour créer des solutions plus efficaces et évolutives.

Sources : Documentation de SuiteScript 2.x (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: ursuscode.com), notes de version d'Oracle SuiteScript (Source: docs.oracle.com), forums de la communauté NetSuite (Source: archive.netsuiteprofessionals.com) (Source: archive.netsuiteprofessionals.com) et blogs de développeurs (Source: www.thenetsuitepro.com) (Source: test.suiterep.com) (Source: ursuscode.com), entre autres. Tous les faits et exemples sont tirés de ces références faisant autorité.

Étiquettes: suitescript-2x, async-await, requestsuiteletpromise, module-nhttps, developpement-netsuite, modeles-restlet, integration-suitelet, promesses-suitescript

---