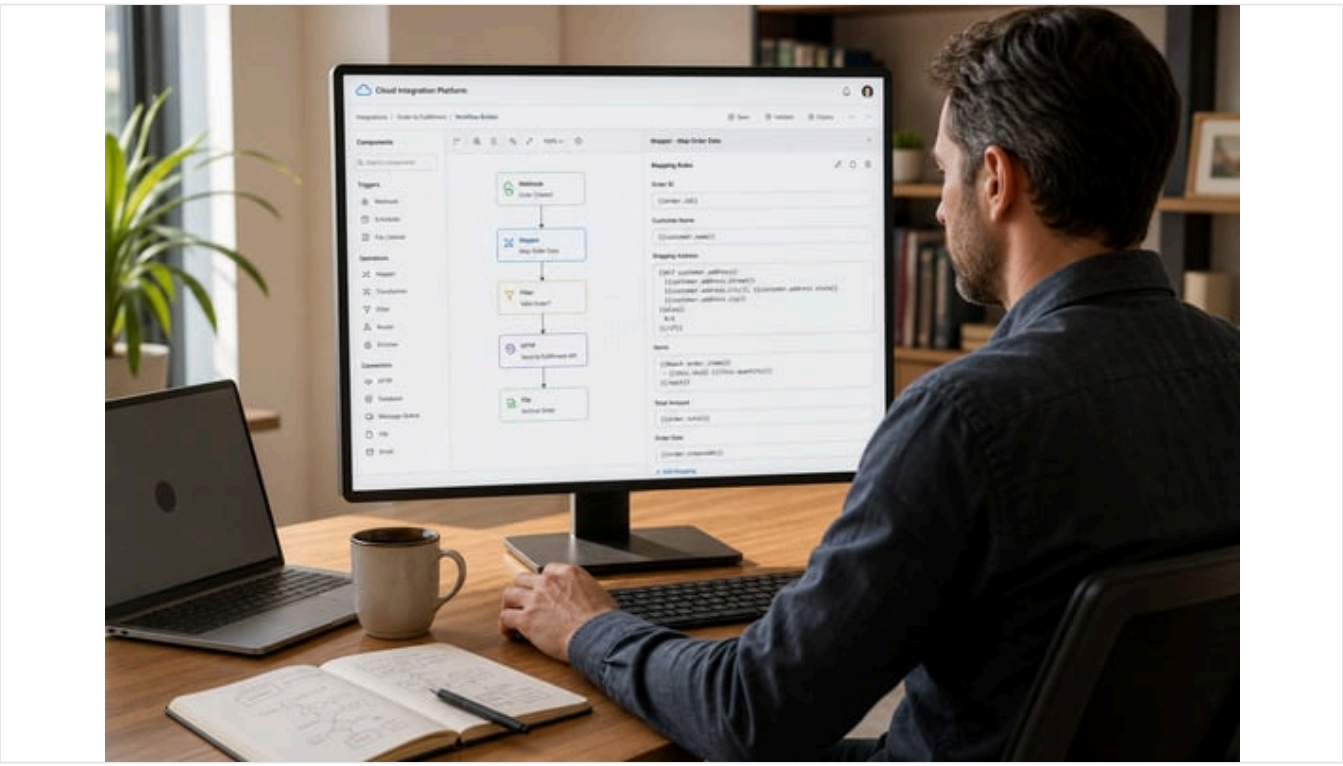
Syntaxe et assistants Handlebars de Celigo pour les flux NetSuite

Publié le 6 mai 2026 36 min de lecture



Résumé analytique

Ce rapport fournit un guide exhaustif et détaillé sur l'utilisation de la syntaxe Handlebars au sein de la plateforme **integrator.io** de Celigo, en se concentrant spécifiquement sur les assistants (helpers), les conditions et les modèles de transformation de données appliqués aux flux d'intégration NetSuite. Il synthétise la documentation officielle de Celigo, les meilleures pratiques techniques, les connaissances de la communauté et les tendances du secteur pour expliquer comment le templating Handlebars est utilisé pour mapper, manipuler et acheminer les données dans des scénarios d'intégration complexes. Handlebars est présenté comme le langage de templating clé dans les flux Celigo, permettant le mappage dynamique de champs, l'arithmétique intégrée, le branchement logique, le formatage de chaînes et de dates, ainsi que de puissantes opérations sur les listes/tableaux (Source: docs.celigo.com) (Source: www.celigo.com).

Nous couvrons l'historique et le contexte de Celigo et Handlebars, les conventions de syntaxe et une taxonomie des assistants intégrés (chaîne, numérique, bloc, etc.), incluant de nombreux exemples illustratifs. Les variables de données (telles que `@index`, `@first`, `@root`) et les structures de contrôle (#if/#else, each, compare, contains) sont expliquées en profondeur. Nous examinons ensuite les modèles de transformation de données courants utilisés dans les flux NetSuite – tels que l'aplatissement d'enregistrements imbriqués, l'agrégation et la sommation de valeurs, les valeurs par défaut conditionnelles et la création de tableaux – en montrant comment les assistants Handlebars et les moteurs de transformation de Celigo (Rules/Mapper 2.0) prennent en charge ces tâches (Source: docs.celigo.com) (Source: connective.celigo.com). Des études de cas et des exemples concrets soulignent comment les organisations exploitent ces fonctionnalités pour résoudre des défis d'intégration (par exemple, le mappage de méta-champs Shopify vers des champs NetSuite (Source: neosalpha.com) ou l'automatisation des flux de commandes entre plateformes (Source: connective.celigo.com) (Source: integscloud.com). Le rapport intègre également les perspectives d'experts et d'analystes sur le rôle de l'iPaaS et de l'automatisation (y compris la reconnaissance de Celigo dans le MQ de Gartner) (Source: www.celigo.com) (Source: www.celigo.com). Enfin, nous discutons des implications et des orientations futures : l'essor du mappage piloté par l'IA et de la gestion des erreurs (Source: teknuro.com) (Source: www.celigo.com), la complexité croissante des intégrations (plus de 100 applications par entreprise) (Source: teknuro.com), et comment la plateforme

de Celigo (avec ses fonctionnalités assistées par IA comme le mappage automatique) permet aux utilisateurs de répondre à ces tendances (Source: teknuro.com) (Source: www.celigo.com). Dans l'ensemble, le rapport sert de référence technique complète pour tout concepteur ou développeur intégrateur utilisant Celigo Handlebars dans NetSuite ou d'autres flux, toutes les affirmations étant étayées par des sources faisant autorité.

Introduction et contexte

Les entreprises modernes s'appuient sur des dizaines d'applications disparates, exigeant des plateformes d'intégration robustes pour automatiser les flux de données. **Celigo integrator.io** est une plateforme d'intégration en tant que service (iPaaS cloud de premier plan, spécialisée dans la connexion de systèmes tels que NetSuite (l'ERP cloud populaire) avec diverses autres applications. En fournissant des centaines d'**applications d'intégration** pré-construites et une interface de création de flux visuelle, Celigo permet aux organisations de lier NetSuite de manière bidirectionnelle avec le commerce électronique, le CRM et d'autres systèmes sans écrire de code personnalisé (Source: neosalpha.com) (Source: www.celigo.com). Celigo rapporte plus de *5 000* clients NetSuite dans le monde et est souvent cité comme le « leader mondial n°1 de l'intégration NetSuite », soulignant sa compréhension native approfondie des structures NetSuite (recherches enregistrées, hooks SuiteScript, types d'enregistrements, etc.) (Source: neosalpha.com) (Source: www.celigo.com).

Au sein de Celigo, les flux sont composés d'étapes d'exportation (source) et d'étapes d'importation/recherche (destination). Les données circulent séquentiellement à travers ces étapes (souvent avec des étapes middleware optionnelles) (Source: www.houseblend.io). Pour contrôler et transformer ces données en cours de route, Celigo propose des outils de filtrage et de mappage. Au cœur du **mappage et de la transformation de champs** se trouve le langage de templating Handlebars. Handlebars (à l'origine un moteur de templating JavaScript compatible avec Mustache) est utilisé par Celigo pour permettre le référencement dynamique des champs et la transformation des données via des expressions et des assistants (Source: docs.celigo.com) (Source: docs.celigo.com). Essentiellement, plutôt que de coder en dur des valeurs statiques, les mappages dans Celigo utilisent des expressions `{{...}}` pour extraire des données du contexte JSON et appliquer une logique. Par exemple, le centre d'aide Celigo explique que « Handlebars est un langage de templating simple » où les expressions entre `{{doubles accolades}}` sont évaluées au moment de l'exécution par rapport au contexte JSON entrant (Source: docs.celigo.com). Ces expressions peuvent effectuer diverses tâches – du mappage des champs d'exportation/importation à l'arithmétique ou à l'encodage – rendant les intégrations à la fois puissantes et flexibles (Source: docs.celigo.com) (Source: docs.celigo.com).

Bien que les scripts Handlebars gèrent de nombreux besoins de transformation, les intégrateurs peuvent également utiliser des hooks JavaScript dans Celigo pour une logique plus complexe. La documentation de la plateforme conseille d'utiliser Handlebars pour la manipulation de données simple et de recourir au JavaScript uniquement lorsque cela est nécessaire pour des calculs complexes ou une logique conditionnelle (Source: www.celigo.com) (Source: connective.celigo.com). En pratique, les flux mélangent souvent les deux : des modèles de mappage utilisant Handlebars dans les règles du mappeur de champs, et des hooks JavaScript personnalisés dans la logique d'exportation/importation si nécessaire. Ce rapport se concentre sur l'aspect Handlebars – détaillant sa syntaxe, ses assistants intégrés et ses modèles de transformation typiques – en particulier dans le contexte des flux centrés sur NetSuite où la conversion et l'assainissement des données sont des tâches courantes.

Utilisation de Handlebars par Celigo dans les flux d'intégration

Handlebars joue un rôle central dans la **transformation des données** au sein des flux Celigo. Selon la documentation de Celigo, les expressions Handlebars peuvent être placées partout où cela est nécessaire dans les mappages d'importation ou d'exportation pour **« boucler, évaluer, étendre et modifier des enregistrements à de nombreux points d'un flux »**. Cette approche low-code permet aux intégrateurs d'implémenter une logique conditionnelle, un traitement itératif et un formatage directement dans le mappeur, sans script personnalisé (Source: docs.celigo.com) (Source: www.celigo.com). Dans la terminologie de Celigo, les données JSON entrant dans un flux sont appelées le *contexte*, et les expressions Handlebars sont évaluées par rapport à ce contexte pour produire des valeurs de champ de sortie (Source: docs.celigo.com) (Source: docs.celigo.com). Par exemple, dans une formule de mappage, on pourrait écrire `{{customer.name}}` pour récupérer le champ `name` d'un objet `customer` imbriqué. La documentation de Celigo souligne que les expressions Handlebars utilisent la notation par points pour accéder aux propriétés d'objets imbriqués, similaire à JSONPath (Source: docs.celigo.com) (Source: neosalpha.com). La syntaxe des accolades elle-même est clé : les doubles accolades `{{field}}` encoderont les caractères spéciaux en URL dans la sortie, tandis que les **triple accolades** `{{{field}}}` produisent un contenu brut sans encodage (Source: docs.celigo.com). En substance, le modèle Handlebars est comme un mini-programme à l'intérieur de la feuille de mappage. Lorsque le flux s'exécute, le modèle est compilé et exécuté : l'intégrateur transmet l'objet de contexte JSON à la fonction compilée, qui remplace ensuite chaque expression Handlebars par sa valeur calculée (Source: docs.celigo.com) (Source: docs.celigo.com).

Celigo fournit un aperçu notant que de telles expressions « mappent les champs d'application d'exportation et d'importation, effectuent des calculs arithmétiques dynamiques et encodent et décodent dynamiquement les données pendant l'intégration » (Source: docs.celigo.com) (Source: docs.celigo.com). Cela signifie que vous pouvez non seulement copier un champ de la source vers la cible, mais aussi le manipuler à la volée (par exemple, sommer des valeurs, formater des dates, concaténer des chaînes) dans le mappage lui-même. Par exemple, on pourrait utiliser `{{amount`

`* 100}}` pour convertir des devises, ou encoder un paramètre d'URL via `{{encodeURI urlField}}`. La combinaison de Handlebars avec le moteur **Transformation 2.0** de Celigo (l'interface de mappage rules 2.0) signifie que vous pouvez utiliser JSONPath pour sélectionner des champs et également insérer Handlebars là où nécessaire pour des calculs de valeur (Source: docs.celigo.com) (Source: docs.celigo.com).

En somme, Handlebars dans les flux Celigo offre un mélange puissant de lisibilité et de capacité. Il abstrait une grande partie de la « plomberie » d'intégration en modèles de haut niveau. Comme le note un blog de Celigo, « Handlebars simplifie la manipulation JSON, la rendant accessible aux utilisateurs sans expérience de programmation approfondie ». Les structures JSON complexes peuvent être parcourues avec des expressions comme `{{myObj.child.array[0].value}}`, tandis que les assistants automatisent les tâches répétitives ou complexes (Source: www.celigo.com) (Source: neosalpha.com). Cependant, comme avec tout système basé sur des modèles, une conception minutieuse est nécessaire : une logique complexe peut devenir difficile à maintenir si elle est entassée dans un bloc `{{#if ...}}` ridiculement long. Les meilleures pratiques soulignent donc l'importance de commencer par une architecture de flux claire, une conception modulaire et la documentation de chaque modèle (Source: www.celigo.com) (Source: www.celigo.com).

Syntaxe et conventions Handlebars

La syntaxe Handlebars dans Celigo est simple mais inclut certaines conventions spéciales. Comme le **résument les documents de Celigo**, les modèles Handlebars ressemblent à du JSON ou du texte régulier avec des expressions intégrées entre doubles accolades (`{{ }}`) (Source: docs.celigo.com) (Source: docs.celigo.com). Entre ces accolades, vous spécifiez soit un chemin vers un champ, soit une fonction d'assistance. Par exemple, `{{record.id}}` fait référence au champ `id` dans le contexte d'enregistrement actuel, tandis que `{{uppercase name}}` appellerait l'assistant `uppercase` sur une variable `name`.

Les conventions de syntaxe clés incluent :

- **Doubles vs Triple accolades** : `{{value}}` encode automatiquement les caractères spéciaux (`>?` etc.) en URL dans la sortie. Utilisez les triple accolades `{{{value}}}` pour produire des données brutes, non encodées (Source: docs.celigo.com).
- **Accolades littérales** : Faire précéder une expression Handlebars d'une barre oblique inverse, comme dans `\{{escaped}}`, produira le texte littéral `{{escaped}}` (accolades incluses) au lieu de l'évaluer (Source: docs.celigo.com).
- **Gestion des espaces** : Les espaces entre les accolades et l'expression (`{{ field }}` vs `{{field}}`) n'ont généralement pas d'importance, sauf dans les chaînes entre guillemets. Les accolades ne doivent pas contenir d'espaces non autorisés (par exemple, pas d'espaces entre `{{` et le contenu de l'expression, comme noté dans l'utilisation de #if) (Source: docs.celigo.com).
- **Indexation de tableau** : Dans une boucle `each`, les éléments peuvent être référencés par index. Par exemple, si vous itérez à travers `data`, `{{0.id}}` fait référence à l' `id` du premier élément ; comme le montre l'**exemple de mappage JSON** de Celigo pour une importation de recherche enregistrée NetSuite, ils préfixent les champs avec `0.` pour désigner le premier enregistrement à chaque itération (Source: docs.celigo.com).
- **Blocs bruts** : Handlebars prend également en charge la syntaxe de bloc brut `{{{{ }}}}` pour désactiver le traitement des sections, mais cela est moins couramment utilisé dans les mappages Celigo.
- **Notation entre crochets** : Si un nom de champ contient des espaces ou des caractères spéciaux, utilisez des crochets. Exemple : `{{this.[Shipping Address]}}` pour accéder à un champ littéralement nommé « Shipping Address » (Source: docs.celigo.com).

Un « modèle » ou « expression » complet est généralement juste la séquence de champs, de littéraux et d'appels d'assistance entre accolades. Par souci de clarté, les documents de Celigo désignent les règles de mappage globales ou la logique de transformation comme un **modèle**, qui est évalué avec le *contexte* JSON pour produire la sortie. Un exemple des documents Celigo montre un modèle simple et un contexte côte à côte :

```
Modèle                 Contexte                                  Sortie
---------              --------                                  ------
{{library.title}}      {                                         Danube Incident
                        "library": {
                          "album": "The Sound",
                          "title": "Danube Incident",
                          "artist": "Lalo Schifrin"
                        }
                       }
```

Dans cet exemple, `{{library.title}}` a recherché le champ title dans l'objet « library », produisant **Danube Incident** (Source: docs.celigo.com). Un tel rendu de modèle est l'opération de base de Handlebars au sein d'un flux d'intégration.

Dans l'ensemble, le Handlebars de Celigo suit la sémantique standard : notation par points pour parcourir les objets, chaînes entre guillemets pour les littéraux, `#` et `/` pour désigner les assistants de bloc, et préfixes `@` pour les variables de données (discutées ci-dessous). Lors de l'écriture d'un modèle, vous pouvez enchaîner plusieurs expressions ou les mélanger avec du texte JSON littéral. Par exemple, pour produire du JSON avec à la fois du texte statique et des valeurs dynamiques, on pourrait écrire :

```
{"rocketID": "{{getValue 'record.rocket' 'defaultValue'}}"}
```

Ici, le texte extérieur produit un objet JSON, tandis que `{{getValue 'record.rocket' 'defaultValue'}}` insère une valeur au moment de l'exécution (en utilisant l'assistant `getValue`) (Source: docs.celigo.com). Remarquez comment les guillemets changent à l'intérieur d'un assistant – les documents de Celigo vous rappellent d'utiliser des guillemets simples à l'intérieur d'une chaîne entre guillemets doubles afin que l'analyseur de modèle ne soit pas confus (Source: docs.celigo.com).

Variables de données et contexte

Les modèles Handlebars gagnent en puissance grâce à des **variables de données** intégrées qui exposent des métadonnées contextuelles. Ces variables sont préfixées par `@` et sont valides à l'intérieur des helpers de bloc. Celigo prend en charge plusieurs de ces variables, documentées en tant que *helpers* : `@first`, `@last`, `@index`, `@key`, `@length`, `@root` et `@this` (Source: docs.celigo.com). Chacune vous donne un aperçu de l'itération ou du contexte actuel :

- `@first` : **True** s'il s'agit de la première itération d'une boucle. Par exemple, au sein de `{{#each array}}...{{/each}}`, le premier élément renvoie `@first = true`, sinon false (Source: docs.celigo.com). La documentation illustre l'utilisation de `{{#if @first}}` à l'intérieur d'un `each` pour afficher quelque chose uniquement lors du premier passage (par exemple, un en-tête) (Source: docs.celigo.com).
- `@last` : **True** pour le dernier élément d'une boucle. Couramment utilisé pour éviter les délimiteurs en fin de chaîne (par exemple, ajouter des virgules entre les éléments, sauf après le dernier) (Source: docs.celigo.com) (Source: connective.celigo.com). Par exemple, `{{#if @last}} {{else}},{{/if}}` n'imprime une virgule que lorsqu'il ne s'agit *pas* du dernier élément (Source: docs.celigo.com) (Source: connective.celigo.com).
- `@index` : L'index numérique (commençant à zéro) de l'élément actuel dans une boucle de tableau. Pour chaque élément d'un tableau, `@index` commence à 0 et s'incrémente (Source: docs.celigo.com). L'exemple du helper `index` montre comment itérer sur un tableau et imprimer `0 1 2 ...` pour chaque élément (Source: docs.celigo.com).
- `@key` : Dans une boucle d'objet (ou de tableau), `@key` fournit le nom de la propriété actuelle ou l'index sous forme de chaîne (Source: docs.celigo.com). En pratique, avec les tableaux, `@key` est essentiellement identique à `@index`. Dans un objet comme `{"a":1,"b":2}`, itérer `#each this` donnerait `@key = "a"` puis `"b"`.
- `@length` : Renvoie la longueur totale d'une valeur donnée (généralement une chaîne ou un tableau). Par exemple, `{{#compare state.length "===" 2}}` est utilisé dans la documentation pour vérifier si une chaîne de code d'état a une longueur de 2 (Source: docs.celigo.com).
- `@root` : Fait toujours référence à l'objet de contexte de niveau supérieur, quelle que soit la profondeur de l'imbrication. Dans un `each` profondément imbriqué, `{{@root.title}}` renverra toujours la propriété `title` de niveau supérieur (Source: docs.celigo.com) (Source: docs.celigo.com). Ceci est utile si vous devez récupérer des champs externes tout en effectuant une boucle.
- `this` : Référence l'objet de contexte actuel. À l'intérieur de `{{#each array}}`, `this` est l'élément actuel du tableau. Si vous itérez sur un objet, `this` est la valeur actuelle. Par exemple, la communauté montre l'utilisation de `{{this.Name}}` pour accéder à un champ de l'objet actuel à l'intérieur de `{{#each myData.N1}}` (Source: connective.celigo.com). L'aperçu du helper `this` confirme qu'au sein d'un bloc, `this` pointe vers l'élément ou l'objet actuel (Source: docs.celigo.com).

En effet, ces variables vous permettent d'inspecter dynamiquement les boucles et les objets. Un modèle de transformation courant consiste à utiliser `@last` ou `@first` pour formater correctement la sortie JSON (par exemple, en ajoutant des virgules). Par exemple, pour créer un tableau JSON d'éléments, on pourrait écrire :

```
[
{{#each items}}
  {"Name": "{{this.name}}", "Qty": {{this.qty}}}{{#if @last}}{{else}},{{/if}}
{{/each}}
]
```

Cela génère une liste JSON correcte, séparée par des virgules, sauf après le dernier élément (Source: connective.celigo.com) (Source: docs.celigo.com). Sans les vérifications `@last`, les virgules résiduelles rendraient le JSON invalide. De même, `@index` et `@key` peuvent être utilisés pour des opérations de numérotation ou de création de clés si nécessaire. Comme le note la documentation de Celigo, `@first` et `@last` renvoient simplement true/false pour les analyses de tableaux (en commençant à l'index 0) (Source: docs.celigo.com) (Source: docs.celigo.com).

Ensemble, ces variables de données offrent aux développeurs un contrôle précis sur la manière dont les modèles effectuent leurs itérations. Elles sont particulièrement précieuses dans les flux NetSuite où la construction du format JSON ou CSV correct est cruciale. Par exemple, dans l'exemple de recherche enregistrée NetSuite (ci-dessous), `@last` est utilisé pour garantir que les virgules séparent les éléments, tandis que `this` et les préfixes numériques (`0.`) extraient les champs appropriés pour chaque enregistrement et ses lignes (Source: docs.celigo.com).

Helpers Handlebars de Celigo et logique conditionnelle

Celigo étend le langage Handlebars de base avec des dizaines de **helpers** intégrés pour gérer les tâches courantes. Ceux-ci couvrent des catégories telles que la **logique/les conditions (helpers de bloc)**, la **manipulation de chaînes**, les **opérations numériques**, le **formatage de date/heure**, le **traitement de listes/tableaux** et des **utilitaires divers** (encodage, recherches, etc.). Ci-dessous, nous passons en revue ces éléments, organisés par fonction. (Notez que la *Référence des helpers Handlebars* officielle de Celigo répertorie tous les helpers ; nous mettons ici en évidence ceux qui sont pertinents pour les flux NetSuite.)

Helpers conditionnels et de bloc

Les helpers de bloc contrôlent la logique de flux et l'itération. Tous commencent par `#` et se terminent par une balise de bloc `/` correspondante. Les principaux helpers de bloc incluent :

- **`{{#each}}`** : Effectue une boucle sur un tableau ou un objet. Pour chaque élément, il fait de cet élément le contexte actuel. Syntaxe : `{{#each array}} ... {{/each}}`. À l'intérieur, utilisez `this` ou `@index/@first/@last` pour faire référence à chaque élément. L'exemple de la communauté montre `{{#each myData.N1}}...{{/each}}` itérant sur les lignes de commande Shopify et utilisant `{{this.[Entity Identifier Code]}}` dans des comparaisons imbriquées (Source: connective.celigo.com). Utilisez `{{/each}}` pour fermer. En tant que modèle, `#each` est utilisé pour créer une sortie répétitive (par exemple, plusieurs commandes client ou lignes d'articles dans NetSuite).

- **`{{#if}}` / `{{else}}`** : Conditionnelle standard. Si l'expression ou le champ donné est *truthy* (non vide, non nul/faux/zéro), le bloc est rendu ; sinon, la partie `{{else}}` est rendue. Exemple : `{{#if companyName}}{{companyName}}{{else}}{{firstName}} {{lastName}}{{/if}}` (issu d'un mappage BigCommerce–NetSuite) affiche l'entreprise si elle est présente, sinon le prénom+nom du contact (Source: connective.celigo.com). Celigo précise que les conditions "false" incluent undefined, null, chaîne vide, zéro ou tableau vide (Source: docs.celigo.com). Crucial : ne mettez pas d'espaces à l'intérieur des accolades `{{#if }}` (pas de `{{#if field }}`) sinon cela ne sera pas analysé (Source: docs.celigo.com).

- **`{{#if...else}}`** : Parfois écrit comme un seul helper (comme dans la documentation de Celigo) ou simplement comme `#if` avec un `{{else}}`. Cela garantit que seule la branche appropriée est exécutée (Source: docs.celigo.com). Il existe également des variantes `{{else if}}` prises en charge. Celigo répertorie des opérateurs de type JavaScript : `<`, `>`, `===`, etc., qui peuvent être utilisés de manière similaire à #compare (ci-dessous).

- **`{{#compare}}`** : Compare deux valeurs à l'aide d'un opérateur. Syntaxe : `{{#compare value1 "operator" value2}}trueBlock{{else}}falseBlock{{/compare}}` (Source: docs.celigo.com). Par exemple, la référence du helper montre : `{{#compare details.fromState "===" "NE"}}+{{details.qty}}{{else}}{{details.qty}}{{/compare}}` pour préfixer un signe plus uniquement si `fromState == "NE"` (Source: docs.celigo.com). Le helper compare encapsule essentiellement `===`, `!==`, `<`, `>`, etc. Il est utile pour les comparaisons numériques ou de chaînes qui ne sont pas simplement des tests "est vide".

- **`{{#contains}}`** : Vérifie si un tableau ou une chaîne contient une sous-chaîne/valeur donnée. Par exemple, `{{#contains items "Special"}}Yes{{else}}No{{/contains}}`. Utile pour la logique conditionnelle basée sur l'appartenance ou les vérifications de sous-chaînes. (Il s'agit d'un helper fourni par Celigo qui ne figure pas dans le Handlebars standard.)

- **`{{#and}}`**, **`{{#or}}`**, **`{{#not}}`** : Helpers logiques. `#and` et `#or` prennent plusieurs conditions ; ils exécutent le bloc si toutes (and) ou l'une quelconque (or) sont truthy. `#not` nie une condition. Par exemple, `{{#or (contains names "Alice") (contains names "Bob")}}Hi! {{/or}}` salue si l'un ou l'autre nom est présent. Ceux-ci prennent en charge une logique booléenne plus complexe sans quitter le modèle.

- **`{{#unless}}`** : L'inverse de `#if` ; il est rendu lorsque la condition est *false*. Syntaxe : `{{#unless field}} (field is missing or false) {{/unless}}`. C'est un raccourci pour `{{#if}}` avec un sens inversé.

- **`{{#with}}`** : Helper de contexte qui change le contexte en un sous-objet, `{{#with object}}...{{/with}}`. À l'intérieur, les champs peuvent être consultés directement. C'est similaire à `#each` mais pour des objets uniques.

En résumé, les helpers de bloc vous permettent d'effectuer des boucles (`#each`), des branchements (`#if`, `#compare`, `#contains`, etc.) et de gérer le formatage de sortie (`#with`, `#unless`). Leur utilisation est largement documentée : par exemple, dans les articles communautaires "How-to" de Celigo, la combinaison de `#each` avec `#compare` est montrée pour le traitement des lignes de commande Shopify (en utilisant `this.[Entity Identifier Code]`) (Source: connective.celigo.com). De même, l'utilisation de `{{#if @last}}...{{else}},{{/if}}` est démontrée dans des exemples officiels pour émettre conditionnellement des virgules entre les objets JSON (Source: docs.celigo.com). Ces modèles (boucle + conditionnel sur `@last`) sont courants dans la construction de tableaux JSON pour les API REST ou les recherches enregistrées de NetSuite.

Helpers en ligne et utilitaires

Outre le contrôle de bloc, Celigo fournit de nombreux helpers en ligne (sans bloc `#`) pour la manipulation de données :

- **Helpers de manipulation de chaînes** : Ceux-ci transforment le texte. Les exemples incluent `uppercase`, `lowercase`, `capitalize`, `capitalizeAll` (met en majuscule chaque mot), `dash-case`, `snake_case`, `PascalCase`, `camelCase`, `sentence` (met en majuscule le premier mot), `trim`, `trimLeft`, `trimRight`, `replace`, `replaceFirst`, `substring`, `split`, `join`, `encodeURI`, `decodeURI`, `htmlEncode`, `htmlDecode`, `hash`, etc. Par exemple, `{{uppercase status}}` convertira une chaîne de statut en majuscules ; `{{replace text " " "-" }}` pourrait ajouter des traits d'union à un champ ; `{{join array ", "}}` concatène les éléments avec une virgule et un espace (Source: docs.celigo.com) (Source: docs.celigo.com). Ceux-ci sont essentiels pour les tâches de formatage de texte – par exemple, transformer une liste de balises séparées par des virgules en un tableau de chaînes, supprimer le HTML du texte enrichi ou garantir que les données correspondent au format attendu par NetSuite.

- **Helpers numériques** : Pour l'arithmétique et le formatage numérique, Celigo propose `add`, `subtract` (ou `minus`), `multiply`, `divide`, `modulo`, ainsi que `sum`, `avg` (pour agréger les valeurs d'un tableau), `abs`, `ceil`, `floor`, `round`, `toFixed`, `toPrecision`, etc. Par exemple, `{{sum thisLinePrices}}` afficherait le total si `thisLinePrices` est un tableau de nombres (Source: docs.celigo.com) (Source: docs.celigo.com). Ou `{{divide total 100}}` pourrait convertir des centimes en dollars. Ceux-ci vous permettent d'effectuer des calculs à la volée. Par exemple, les exemples de gestion des décimales de Celigo montrent l'utilisation de `{{divide $value 100}}` pour corriger la précision monétaire. Le blog note que JavaScript est mieux adapté aux *mathématiques complexes*, mais pour des sommes simples ou des mises à l'échelle, les helpers Handlebars suffisent (Source: www.celigo.com).

- **Helpers de date/heure** : Celigo inclut des helpers comme `dateAdd`, `dateFormat`, `timestamp` (heure actuelle). Ceux-ci sont essentiels lors de la manipulation des champs de date NetSuite ou de la conversion entre les fuseaux horaires. Par exemple, `{{dateAdd createdDate 1 "days"}}` pourrait décaler une date d'un jour vers l'avant, et `{{dateFormat createdDate "YYYY-MM-DDTHH:mm:ssZ"}}` la formate au format ISO8601. Le guide Houseblend sur les filtres (bien qu'il concerne l'opérateur de correspondance) et les blogs Celigo suggèrent d'utiliser des helpers de date/heure lors du mappage des champs de date (Source: www.houseblend.io) (Source: www.celigo.com). Il est également fait mention de la prise en charge des fuseaux horaires dans la section des exemples Handlebars.

- **Helpers de liste/tableau** : Au-delà de `each`, Celigo propose des helpers comme `arrayify` (garantit qu'une valeur est un tableau), `unique` (supprime les doublons), `sort`, `pluck` (extrait un tableau d'un champ donné à partir d'une liste d'objets) et `hash` (crée des paires clé/valeur). Par exemple, si vous avez un tableau d'objets de commande, `{{pluck orders "id"}}` renverrait un tableau de tous les ID de commande. Le helper `filter` (bloc) peut filtrer un tableau par un prédicat (par exemple, `{{#filter items "category" "Electronics"}}...{{/filter}}`). Ceux-ci sont utiles dans les flux de transformation : par exemple, extraire des e-mails clients uniques d'un lot de ventes ou créer une table de recherche à partir d'une partie des données.

- **Helpers d'encodage et divers** : Des helpers comme `encodeURI`, `decodeURI`, `htmlEncode`, `htmlDecode` gèrent les caractères spéciaux. `base64Encode` / `Decode` gèrent les opérations Base64. Il existe même des helpers liés à AWS (`aws4` pour la signature de requêtes, `hmac` pour le hachage) pour les appels API personnalisés. Celigo fournit également des utilitaires JSON comme `jsonParse`, `jsonSerialize`, `jsonEncode` pour traiter les chaînes JSON brutes dans les mappages. Le helper `getValue` (discuté ci-dessous) récupère en toute sécurité les champs imbriqués, et `typeOf` renvoie le type JavaScript d'une valeur (pour le débogage). Enfin, les **helpers de recherche** (catégorie `lookup`) méritent une mention spéciale : `{{lookup}}` récupère des données à partir des tables de "Recherche" Celigo définies dans le flux (voir ci-dessous), et le contexte `$` (issu des applications d'intégration) fait référence à l'ensemble du contexte de l'enregistrement (Source: connective.celigo.com) (Source: docs.celigo.com).

Pour illustrer l'utilisation : on peut écrire

```
{{uppercase customer.name}} - {{formatTimestamp createdDate "YYYY-MM-DD"}}
```

pour afficher un nom en majuscules et une date formatée. Ou

```
{{#if hasDiscount}}Discount: {{discountAmount}}{{else}}No Discount{{/if}}
```

pour mentionner conditionnellement une remise. La **section des exemples officiels** dans la documentation de Celigo fournit de nombreux exemples de ce type : par exemple, l'utilisation de codes de format de date, hash/hmac pour les appels API, ou des remplacements regex pour nettoyer les numéros de téléphone (Source: connective.celigo.com) (Source: connective.celigo.com). De même, les FAQ de la communauté présentent des helpers regex (regexReplace, regexMatch) utilisés pour le nettoyage des données. Dans l'ensemble, ces helpers permettent des transformations de données puissantes au sein de la couche de mappage – réduisant le besoin de scripts externes, tout en offrant un contrôle précis.

Tableau 1 (ci-dessous) résume certaines catégories courantes de helpers Handlebars dans Celigo :

CATÉGORIE	OBJECTIF	EXEMPLES DE HELPERS / UTILISATION
Logique / Conditionnel	Branchement et tests booléens	`{{#if}}`, `{{#if ... else}}`, `{{#unless}}`, `{{#compare}}`, `{{#contains}}`, `{{#and}}`, `{{#or}}` *Exemple :* `{{#compare status "==" "Closed"}}Closed{{/compare}}`
Itération / Boucle	Boucle sur des listes ou des objets	`{{#each}}`, `{{@index}}`, `{{@first}}`, `{{@last}}`, `{{#with}}` *Exemple :* construction de tableau : voir ci-dessous.
Chaîne / Texte	Manipuler des valeurs textuelles	`uppercase`, `lowercase`, `capitalize`, `dash-case`, `snake_case`, `replace`, `split`, `join`, `encodeURI`, `htmlEncode`, `hash`, `jsonEncode` *Exemple :* `{{removeProtocol url}}` (supprime "http://"), `{{join tags ", "}}`

| **Numérique / Math** | Arithmétique sur les nombres | `add`, `subtract`, `multiply`, `divide`, `modulo`, `sum`, `avg`, `ceil`, `floor`, `round`, `toFixed`, etc.
Exemple : `{{round price 2}}` arrondit un nombre. | | **Date / Heure** | Formater ou ajuster les dates | `dateAdd`, `dateFormat`, `timestamp`
Exemple : `{{dateFormat orderDate "YYYY-MM-DD"}}` | | **Tableau / Liste** | Travailler avec des tableaux et des tableaux d'objets | `unique`, `sort`, `pluck`, `arrayify`, `filter`
Exemple : `{{unique emails}}` pour dédoublonner un tableau | | **Encodage / Divers** | Opérations d'encodage/recherche, hachage | `getValue`, `lookup`, `base64Encode/base64Decode`, `htmlDecode`, `regexMatch/regexReplace`, `aws4`, `hmac`, `typeOf`
Exemple : `{{lookup "stateConfig" record.state}}` |

Tableau 1 : Catégories courantes d'helpers Handlebars pris en charge par Celigo. Chaque catégorie contient plusieurs helpers pour des tâches spécifiques (selon la référence Handlebars de Celigo (Source: docs.celigo.com) (Source: docs.celigo.com).

Recherches de données et `getValue`

Deux helpers particulièrement utiles, souvent utilisés dans les flux NetSuite, sont **`getValue`** et **`lookup`**, qui récupèrent des données depuis le contexte ou depuis les tables de recherche (lookups) de Celigo.

L'helper `{{getValue}}` permet une récupération sécurisée d'un champ par son chemin, avec une valeur par défaut optionnelle si le champ est manquant (Source: docs.celigo.com). Sa syntaxe est `{{getValue "record.path.to.field" "defaultVal"}}`. Si le champ spécifié est nul ou indéfini, la chaîne par défaut est renvoyée. Par exemple, `{{getValue "record.email" "none@none.com"}}` affichera l'e-mail s'il est présent, ou `"none@none.com"` sinon (Source: docs.celigo.com) (Source: docs.celigo.com). Cela évite les erreurs lorsque certains enregistrements ne possèdent pas un champ. C'est particulièrement pratique lorsque les noms de champs sont dynamiques ou incertains : au lieu de risquer une erreur, vous pouvez utiliser une valeur par défaut. La documentation de Celigo souligne qu'il s'agit d'un modèle pour un accès « sécurisé » aux champs dans les modèles (Source: docs.celigo.com).

Les helpers `{{lookup}}` récupèrent des valeurs à partir des tables de *Lookup* définies dans l'interface du flux (Source: docs.celigo.com). Il existe deux types de recherches Celigo : **recherche dynamique** (mappage de valeurs basé sur une recherche enregistrée ou une requête API) et **valeur statique** (dictionnaire codé en dur). Une recherche dynamique peut, par exemple, traduire un code régional en un identifiant interne en interrogeant NetSuite. Dans un modèle, vous utilisez `{{lookup.lookupName}}` pour une recherche dynamique, ou `{{lookup "lookupName" record.field}}` pour des recherches statiques (Source: docs.celigo.com). La documentation de Celigo montre :

- `{{lookup.shippingLookup}}` utiliserait une recherche dynamique nommée « shippingLookup » pour récupérer, par exemple, une méthode d'expédition depuis le système de destination (Source: docs.celigo.com).
- `{{lookup "myStaticLookup" orderType}}` utiliserait un mappage statique basé sur le champ `orderType`.

Ces helpers de recherche sont inestimables pour les modèles de transformation de données comme le mappage de codes de taxe ou de SKU : vous pouvez définir à un seul endroit comment convertir les valeurs, puis les appeler en ligne. Par exemple, un flux de commande NetSuite pourrait utiliser une recherche nommée `taxLookup`, et dans le mappage, utiliser `{{lookup "taxLookup" order.taxCode}}` pour obtenir l'ID de taxe NS correct. Les documents Celigo soulignent que les noms de recherche doivent correspondre exactement à la recherche configurée dans le flux.

`getValue` et `lookup` étendent tous deux la puissance de Handlebars dans les flux en accédant à des données situées en dehors des paramètres immédiats du modèle. En combinaison avec JSONPath (par exemple `$.queryParams.id` dans la transformation) (Source: docs.celigo.com), on peut récupérer pratiquement n'importe quelle information de contexte ou paramètre externe.

Logique conditionnelle en pratique

Les conditions sont essentielles lors de l'intégration de systèmes ayant des règles de données variables. Quelques modèles courants incluent :

- **Vérification d'existence / Valeurs par défaut** : Utilisez `#if` pour vérifier si un champ existe ou possède une valeur, et fournissez une valeur par défaut sinon. Ex. : `{{#if account.default}}Yes{{else}}No{{/if}}`. La documentation officielle illustre cela comme un choix entre le nom de l'entreprise ou le prénom/nom (Source: connective.celigo.com). On peut fournir une solution de repli avec `getValue` de manière similaire.
- **Logique multi-branches** : Imbriquez `#if` ou utilisez `#compare` / `#contains` pour différents cas. L'exemple communautaire Shopify-vers-NetSuite (Tableau 2 ci-dessous) montre le découpage par type d'entité en utilisant des comparaisons imbriquées à l'intérieur de `{{#each}}`. En pratique, vous pourriez voir plusieurs chaînes `{{else if}}` pour gérer diverses conditions sources.
- **Conditions agrégées** : Utilisez `#and` / `#or` pour combiner des vérifications. Par exemple, `{{#and (contains comments "urgent") (eq priority "high")}}` pour détecter les tickets urgents de haute priorité.
- **@first/@last dans les listes de données** : Au sein des boucles, on écrit souvent `{{#if @first}}...{{/if}}` pour préfixer des éléments spéciaux, ou `{{#if @last}}...{{else}},{{/if}}` comme illustré, pour séparer correctement les tableaux JSON par des virgules (Source: docs.celigo.com) (Source: connective.celigo.com). Par exemple, la construction d'un tableau JSON pour une API NetSuite pourrait ressembler à ceci :

```
{ "items": [
    {{#each lines}}
      {"itemId": "{{this.item}}", "qty": {{this.quantity}}}{{#if @last}}{{else}},{{/if}}
    {{/each}}
]}
```

Cela garantit que le JSON est valide en contrôlant le placement de la virgule (Source: docs.celigo.com) (Source: connective.celigo.com).

- **Regex et conditions de motif** : Celigo inclut des helpers regex. Par exemple, `{{#regexSearch phone "(\d{3})-(\d{3})-(\d{4})"}}` pourrait vérifier si un numéro de téléphone correspond à un motif. Les FAQ communautaires montrent l'utilisation de `regexReplace` pour formater les numéros de téléphone (supprimer les caractères non numériques, etc.) (Source: connective.celigo.com). Un motif pourrait être la suppression des caractères ASCII ou de la ponctuation : `{{regexReplace description "[^\\x20-\\x7E]" ""}}` pour supprimer l'ASCII étendu.

- **Vérifications de listes avancées** : Si un enregistrement possède des champs dans plusieurs emplacements possibles, on pourrait voir quelque chose comme `{{#if array[*].sku}}...{{else}}...{{/if}}` où `[*]` indique une vérification sur n'importe quel élément. (Cela a été vu dans une question communautaire problématique [22], bien que celle-ci tentait un indexage par caractère générique qui n'a pas fonctionné comme prévu.) En général, il est plus sûr d'utiliser `#each` avec un `#contains` ou `#compare` interne pour vérifier si *n'importe quel* élément remplit une condition.

Le blog des meilleures pratiques de Celigo insiste sur le fait de garder la logique conditionnelle **claire et maintenable** : par exemple, utilisez des helpers de bloc intégrés plutôt que d'enfouir trop de code dans des expressions `{{}}`, et testez minutieusement les modèles pour tenir compte de tous les scénarios de données (Source: www.celigo.com) (Source: docs.celigo.com). Les branchements complexes pourraient être mieux gérés dans une étape de pré-traitement ou un hook JavaScript pour la lisibilité, en réservant Handlebars aux conditions plus simples.

Modèles de transformation de données dans les flux

En pratique, Celigo gère une variété de formes et de formats de données. Les flux d'intégration NetSuite ont souvent besoin de **remodeler les données** pour correspondre aux schémas d'enregistrement. Voici des modèles de transformation courants et la façon dont Handlebars s'y intègre :

- **Aplatissement de structures imbriquées** : Les systèmes sources peuvent générer du JSON imbriqué (par exemple, une commande avec des articles de ligne imbriqués). Les transformations Celigo peuvent aplatir ces structures dans le format tabulaire requis par les enregistrements NetSuite. Un modèle consiste à utiliser plusieurs boucles `#each` : une pour l'en-tête de commande (ex. : ID externe, date) et un `#each` interne pour les articles de ligne (chacun créant un sous-enregistrement). Dans l'exemple JSON de recherche enregistrée, un `{{#each data}}` externe boucle à travers les commandes, puis un `{{#each this}}` interne boucle à travers les articles de ligne (Source: docs.celigo.com). À l'intérieur de la boucle, les champs sont extraits en utilisant des références indexées (ex. : `{{0.id}}` pour les champs d'en-tête et `{{Item}}` pour les champs de ligne). Ainsi, l'entrée imbriquée est sérialisée en tableaux d'objets JSON par ligne. La logique de virgule (`{{#if @last}}…{{else}},{{/if}}`) que nous avons vue assure également une syntaxe JSON correcte (Source: docs.celigo.com). Une autre approche consiste à utiliser le moteur **Transformation 2.0 (Rules 2.0)** de Celigo en mode « créer des lignes de sortie », conçu pour produire plusieurs lignes à partir d'une seule entrée. Ce mode effectue en interne une expansion similaire, mais Handlebars peut toujours être utilisé dans les champs de mappage pour les valeurs de champ (ex. : `{{lookup "itemLookup" this.sku}}`).

- **Agrégation ou sommation de valeurs** : Parfois, plusieurs enregistrements sources doivent être combinés en un seul cible. Par exemple, si plusieurs commandes Shopify doivent être mappées vers une seule commande combinée sur NetSuite avec des quantités additionnées. Ici, on pourrait utiliser `sum` sur un tableau : si les étapes précédentes ont collecté toutes les quantités dans un tableau `quantities`, utilisez `{{sum quantities}}` pour obtenir le total (Source: docs.celigo.com). Ou utilisez `avg` / `min` / `max` de la même manière. Avec les dates, on pourrait utiliser `dateAdd` pour calculer des durées entre les enregistrements. Dans des cas plus simples, on pourrait simplement faire `{{quantity * price}}` pour le total d'un seul enregistrement, mais pour les tableaux, les helpers sont pratiques.

- **Valeurs par défaut et solutions de repli** : Le modèle d'helper `getValue` apparaît souvent pour fournir des valeurs par défaut lorsque les données en amont sont manquantes, évitant ainsi les erreurs. Par exemple, `{{getValue "record.discount" "0"}}` garantit qu'un champ de remise renvoie toujours un nombre. Combiné avec `#if`, on peut aussi faire `{{#if record.discount}}{{record.discount}}{{else}}0{{/if}}` (Source: connective.celigo.com). De même, on pourrait fournir des chaînes par défaut pour un téléphone/e-mail manquant : `{{getValue "record.phone" "000-000-0000"}}`.

- **Tables de recherche et de traduction** : NetSuite nécessite souvent des identifiants internes au lieu de codes lisibles par l'homme. Un modèle de mappage consiste à utiliser une recherche Celigo (construite via une table de recherche dans le flux). Par exemple, traduire un code devise en un currencyId NetSuite peut utiliser une recherche statique : `{{lookup "currencyMap" record.currencyCode}}`. Ou une recherche dynamique pourrait récupérer un employeeID depuis NetSuite à partir d'un e-mail. Après avoir créé la recherche dans l'interface utilisateur, l'helper Handlebars `{{lookup}}` rend la traduction transparente (Source: docs.celigo.com).

- **Inclusion conditionnelle de champ** : Parfois, un champ ne doit être envoyé que si une condition est remplie. Dans le mappage de facture NetSuite, on pourrait avoir une ligne comme : `Remise : {{discountAmount}}` uniquement si une remise s'applique. Utiliser `#if` ou `#contains` autour de `{{discountAmount}}` peut le supprimer lorsqu'il est à zéro. L'étude de cas Headspace fait allusion à une telle logique en éliminant les entrées en double et les conditions d'erreur via les flux Celigo (Source: www.celigo.com) (bien qu'elle ne détaille pas le modèle, les flux réels ont probablement utilisé de telles conditions).

- **Assemblage JSON (Modèles de texte)** : Bien que Celigo fonctionne principalement avec des mappages de champs, on construit parfois manuellement de grandes charges utiles JSON. Dans ces cas, Handlebars est utilisé pour injecter des morceaux. Un modèle issu de la communauté montre la construction d'une chaîne de tableau JSON avec `#each` et des virgules (Source: connective.celigo.com). Un autre modèle consiste à construire du SQL dynamique (comme des requêtes Postgres) en utilisant des doubles `{{}}` pour encoder les valeurs. Les documents Celigo notent que les doubles accolades encodent automatiquement les URL, ce qui est important si vous incluez des paramètres de requête dans une charge utile HTTP GET ou POST (Source: docs.celigo.com).

- **Regex et nettoyage** : Les modèles impliquent souvent le nettoyage des champs de texte. Par exemple, pour supprimer les caractères non ASCII ou HTML d'une description, utilisez des helpers comme `regexReplace` ou `htmlDecode`. Exemple : `{{regexReplace description "[^\\x00-\\x7F]" ""}}` supprime les caractères étendus. Dans le formatage de téléphone, on pourrait faire `{{regexReplace phone "[^0-9]" ""}}` pour ne laisser que les chiffres, puis réinsérer le formatage. Les forums Celigo Connective ont des articles « comment faire » spécifiquement pour le formatage des numéros de téléphone et des devises en utilisant regexReplace et d'autres helpers.

- **Conversion Date/Heure** : NetSuite attend les dates dans un format spécifique. Un modèle courant consiste à convertir une date source (ex. : « MM/JJ/AAAA ») en ISO. Celigo fournit des codes de format de date pour les modèles de style Ruby/JavaScript. Par exemple, utilisez `{{dateFormat NSDate "yyyy-MM-dd'T'HH:mm:ssZ"}}` ou similaire. Il existe également un helper `timestamp` pour obtenir « maintenant ». Si la logique métier nécessite de décaler les dates (ex. : changer de fuseau horaire ou ajouter des jours ouvrables), utilisez `dateAdd` avec l'unité appropriée (jours, heures). Un blog Celigo sur les meilleures pratiques suggère de conserver les dates/heures en UTC sauf nécessité, mais les helpers de date Handlebars facilitent l'ajustement selon les besoins.

- **Construction de tableau** : Au-delà de l'aplatissement, vous devez parfois construire un nouveau tableau à partir de champs individuels. Un modèle est montré dans la communauté (voir Tableau 2) : après la boucle, vous pourriez utiliser `{{create}}` ou un assemblage via Handlebars (bien que Handlebars pur ne puisse construire que des chaînes, on peut le tromper en tant que texte JSON). Par exemple :

  ```
  [{{#each parts}}
    {"part": "{{this.name}}"}{{#if @last}}{{else}},{{/if}}
  {{/each}}]
  ```

  générera un tableau JSON d'objets de pièces. L'interface utilisateur de Celigo prend également en charge une fonctionnalité explicite de transformation « Tableau » de nos jours.

Ces modèles exploitent le moteur **Transformation 2.0** / **Mapper 2.0** de Celigo, qui traite le modèle Handlebars comme faisant partie d'un mappage déclaratif. Les documents Celigo notent que les Rules 2.0 (transformations) peuvent utiliser JSONPath pour sélectionner des champs et Handlebars **discuté ci-dessus** pour calculer des valeurs (Source: docs.celigo.com). On peut même mélanger JSONPath (`$.field`) avec Handlebars dans une seule expression. L'avantage est que Transformation 2.0 peut générer plusieurs lignes par entrée (mode de sortie tableau) et automatiser la gestion des tableaux imbriqués, rendant certains des modèles ci-dessus moins manuels. Néanmoins, comprendre l'approche Handlebars reste utile, car de nombreuses règles de transformation autorisent toujours le mode « expression Handlebars » pour des formules personnalisées (Source: docs.celigo.com) (Source: docs.celigo.com).

Le **Tableau 2** (ci-dessous) illustre quelques modèles de transformation Handlebars représentatifs avec des exemples d'expressions.

MODÈLE / CAS D'UTILISATION	EXEMPLE D'EXPRESSION HANDLEBARS	INTENTION / EXPLICATION
Valeur par défaut si manquant (fallback)	`{{getValue "record.discount" "0"}}`	Récupère en toute sécurité `record.discount`, par défaut à « 0 » si absent.
Sortie conditionnelle	`{{#if isActive}}Active{{else}}Inactive{{/if}}`	Affiche « Active » uniquement si `isActive` est vrai ; sinon « Inactive ».
Concaténer des champs	`{{record.firstName}} {{record.lastName}}`	Construit le nom complet en combinant prénom et nom.
Itérer et construire un tableau (séparé par des virgules)	`[{{#each items}} {"id":"{{this.id}}","qty":{{this.qty}}}{{#if @last}}{{else}},{{/if}} {{/each}}]`	Boucle à travers `items`, génère des objets JSON pour chacun, séparés par des virgules (ignore la virgule sur le dernier) (Source: docs.celigo.com) (Source: connective.celigo.com).
Somme des valeurs d'un tableau	`{{sum order.quantities}}`	Additionne toutes les quantités dans un tableau `order.quantities`.
Nettoyage Regex (ponctuation)	`{{regexReplace address "[^a-zA-Z0-9 ]" ""}}`	Supprime tous les caractères non alphanumériques de `address`.
Rechercher/Traduire une valeur	`{{lookup "taxMap" record.state}}`	Utilise une recherche nommée `taxMap` pour traduire le code `state` en ID NS.
Élément de tableau conditionnel	`{{#contains tags "urgent"}}HasUrgent{{else}}Normal{{/contains}}`	Vérifie si le tableau `tags` contient « urgent » ; affiche en conséquence.

Tableau 2 : Exemples de modèles de transformation de données exprimés avec Celigo Handlebars. Chaque ligne montre un besoin courant dans les flux NetSuite et un exemple d'extrait de modèle. Par exemple, la construction d'un tableau JSON à partir d'une liste `items` (ligne 3) utilise une boucle `each` avec une vérification `@last` pour assurer un placement correct des virgules (Source: docs.celigo.com) (Source: connective.celigo.com).

Études de cas et exemples concrets

Pour ancrer ces concepts, examinons quelques scénarios concrets et la manière dont Handlebars est appliqué :

- **Intégration des commandes e-commerce** : Un détaillant utilisant Shopify et NetSuite doit synchroniser ses commandes. Le JSON de commande de la boutique inclut des articles imbriqués et des méta-champs personnalisés. Lors du mappage du flux Shopify vers NetSuite, Celigo integrator utilise `{{#each order.lineItems}}` pour itérer sur le tableau des lignes, et à l'intérieur, utilise `{{#if this.sku}}...{{/if}}` et `{{compare this.price "*" ...}}` comme illustré dans un cas d'assistance communautaire (Source: connective.celigo.com). Un autre exemple, le mappage d'un méta-champ Shopify « CustomTag » vers une colonne de transaction personnalisée NetSuite, nécessite JSONPath pour sélectionner le méta-champ (`$.lineItems[*].properties` ou la syntaxe de Celigo), puis la spécification de l'ID de champ interne de NetSuite sur la destination. Les consultants notent que ce processus implique de spécifier le chemin JSONPath complet pour le champ Shopify et l'ID interne NS comme cible (Source: neosalpha.com). À ce titre, Celigo prend explicitement en charge la combinaison de JSONPath et de Handlebars : par exemple, l'utilisation d'une expression comme `{{lookup "productTags" record.product_id}}` pourrait appeler une recherche pour traduire les ID de produits en tags (à partir d'une table prédéfinie).
- **Importation de recherche enregistrée NetSuite** : Un système externe déclenche une recherche enregistrée NetSuite et envoie ses résultats à Celigo via HTTP(S). Le modèle de mappage doit transformer ce JSON en masse en enregistrements de transaction NetSuite valides. Le centre d'aide fournit un exemple : il enveloppe l'intégralité de la charge utile dans une structure JSON (par exemple `{"Order": [...]}`), utilise `{{#each data}}` pour chaque commande, et à l'intérieur, référence des champs comme `{{0.[Shipping City]}}` pour l'en-tête, et une autre

boucle imbriquée `{{#each this}}` pour les articles (Source: docs.celigo.com). En effet, le modèle Handlebars construit manuellement le JSON final à envoyer à NetSuite, incluant une logique pour séparer les éléments par des virgules et n'inclure que les champs non nuls. Cet exemple montre l'utilisation d'accolades triple-imbriquées (préfixe `0.`) et la logique `@last` pour aplatir le résultat complexe d'une recherche enregistrée.

- **Synchronisation CRM/ERP** : Une entreprise synchronisant Salesforce vers NetSuite pour des données d'opportunité peut souhaiter n'envoyer que les commandes dépassant un certain seuil. Elle peut utiliser une condition Handlebars dans un filtre d'entrée ou une transformation : `{{#if (gt Amount 1000)}}{{Amount}}{{else}}0{{/if}}`, garantissant que seules les transactions importantes sont traitées (Source: docs.celigo.com) (Source: www.celigo.com). Elle peut également formater les dates (`{{dateFormat CloseDate "YYYY-MM-DD"}}`) et mapper les valeurs de listes de sélection via des recherches (par exemple, mapper les étapes de vente aux statuts NetSuite).

- **Résultats concrets** : Dans les témoignages de réussite publiés, les entreprises soulignent l'impact de l'utilisation des flux Celigo. Par exemple, Headspace (application de santé/méditation) rapporte que Celigo a éliminé la saisie manuelle et économisé « 15 heures par mois » en synchronisant Salesforce et NetSuite (Source: www.celigo.com). Bien que le témoignage ne détaille pas le templating, il implique que des règles complexes ont été intégrées dans les mappages Celigo pour assurer la cohérence des données. Un autre cas décrit un détaillant de mode gérant **des millions de commandes Shopify** en période de pointe avec des flux personnalisés Celigo (Source: integscloud.com). Là, des flux spécialisés ont probablement utilisé Handlebars et des recherches pour gérer les remboursements et les multiples SKU, l'histoire notant des « flux personnalisés pour aborder l'impact complexe sur le grand livre » des remboursements (Source: integscloud.com). Ces exemples illustrent que, en coulisses, les développeurs Celigo s'appuient souvent sur les fonctionnalités Handlebars décrites ici pour implémenter la logique nécessaire.

- **Exemples de questions/réponses communautaires** : Les forums Celigo Connective montrent également divers problèmes résolus avec Handlebars. Par exemple, un utilisateur a dû utiliser `$.company` dans un mappage d'application d'intégration, ce que Celigo a clarifié comme étant le contexte d'enregistrement global (similaire à `this`) (Source: connective.celigo.com). Un autre fil de discussion fournit un exemple avancé de boucle et comparaison pour les commandes EDI (Source: connective.celigo.com) (Source: connective.celigo.com), démontrant que même un JSON multi-imbriqué peut être parcouru en combinant des boucles `each` et des helpers. Ces exemples d'utilisation réelle renforcent le fait que les modèles théoriques (itération, conditions, recherches) se traduisent directement dans le code que les utilisateurs écrivent dans les flux Celigo.

Implications et orientations futures

L'intégration de Handlebars par Celigo s'inscrit dans des tendances plus larges de l'intégration de données et de l'iPaaS. Les analystes du secteur notent que les entreprises utilisent désormais des *centaines* d'applications : une étude de 2025 rapporte que la **moyenne mondiale est d'environ 101 applications par entreprise** (Source: teknuro.com). Cette explosion des systèmes SaaS signifie que les tâches de mappage de données se multiplient. Parallèlement, l'IA et l'automatisation imprègnent l'informatique : environ 71 % des organisations utilisent l'IA générative dans certaines fonctions métier, et 75 % des travailleurs du savoir utilisent des outils d'IA quotidiennement (Source: teknuro.com). Gartner prévoit que la demande en API (et en intégration) va monter en flèche, une part importante étant portée par l'IA et les LLM.

Dans ce contexte, Handlebars reste un outil précieux pour le moment, mais nous observons les tendances et implications suivantes :

- **Mappage assisté par l'IA** : Celigo lui-même adopte l'IA. Par exemple, Celigo affirme que sa plateforme résout environ 95 % des erreurs d'intégration automatiquement via des algorithmes d'IA (Source: neosalpha.com), et sa feuille de route inclut des fonctionnalités de type « Copilot ». En effet, un blog technique note que les mises à jour 2025 de Celigo se concentrent sur les capacités d'« auto-mappage » et de « Copilot contextuel » (Source: teknuro.com). Plus largement, les fournisseurs d'iPaaS explorent le « mappage de données intelligent », où le système apprend des mappages précédents pour suggérer ou remplir automatiquement les relations entre les champs (Source: www.celigo.com). L'article sur les tendances de Celigo mentionne explicitement le **« mappage de données intelligent : mappage automatique des champs de données entre les systèmes, apprenant des mappages précédents pour améliorer la précision »**. Cela suggère qu'à l'avenir, le code Handlebars pourrait être partiellement généré ou assisté par l'IA, obligeant les intégrateurs à réviser plutôt qu'à écrire chaque transformation.

- **Autonomisation Low-Code** : Les plateformes d'intégration évoluent vers un développement plus visuel et low-code. Le générateur de flux de Celigo est déjà visuel, et Handlebars s'y adapte en utilisant un langage de modèle plutôt que du code brut. L'accent mis sur les intégrateurs citoyens signifie que davantage d'utilisateurs métier – et non des développeurs professionnels – concevront des flux. Comme le souligne un rapport, Celigo se concentre sur l'activation du « libre-service gouverné » pour les équipes métier (Source: www.celigo.com). Handlebars, étant plus proche du mappage naturel que certains scripts, s'aligne avec cela. Cependant, cela signifie également que les modèles doivent rester aussi simples que possible. Les futures améliorations pourraient inclure davantage de fonctions intégrées (par exemple, des widgets dédiés au formatage des dates) pour couvrir les besoins courants sans codage manuel.

- **Intégration hybride et multi-cloud** : De nombreuses entreprises utilisent un mélange de systèmes sur site et dans le cloud. Handlebars de Celigo doit fonctionner avec diverses sources de données. Heureusement, étant purement une couche de templating, il est agnostique quant à l'origine ; tant que les données arrivent au format JSON ou formulaire, les mêmes helpers s'appliquent. La tendance plus large est celle d'architectures d'intégration hybrides robustes, mais cela affecte principalement la connectivité et non le langage de mappage lui-même (Source: www.celigo.com). Une orientation future pourrait être des connecteurs intégrés à davantage de protocoles (par exemple, ODBC direct vers des bases de données sur site) avec Handlebars utilisé pour le mappage malgré tout.
- **Gouvernance et sécurité de l'intégration** : À mesure que les flux de données augmentent, la sécurité et la conformité gagnent en importance. L'iPaaS de Celigo investit dans le chiffrement et les contrôles de gouvernance (accès utilisateur, audit). Du point de vue de Handlebars, cela signifie que les futures versions pourraient restreindre certains helpers dans des contextes sécurisés, ou fournir davantage de journalisation autour des transformations de données. Mais conceptuellement, le templating lui-même n'absorbe pas les données, il les formate simplement. La principale préoccupation est de garantir que les données (comme les informations personnellement identifiables) sont traitées correctement. Par exemple, Handlebars de Celigo dispose de helpers comme `htmlEncode` / `jsonSerialize` qui peuvent aider à assainir les sorties, réduisant le risque d'erreurs d'injection. La plateforme pourrait ajouter davantage de helpers d'assainissement intégrés au fil du temps.
- **Connecteurs Serverless/Edge** : Une tendance à long terme est celle d'une intégration plus dynamique ou pilotée par les événements. Celigo prend déjà en charge les déclencheurs en temps réel (par exemple, SuiteEvents). Handlebars peut également être utilisé dans de tels flux, mais à mesure que l'intégration devient plus centrée sur les API, nous pourrions voir Handlebars utilisé pour modéliser des charges utiles d'API ou même répondre dans des webhooks. Les fondamentaux resteront les mêmes : remplir des modèles avec des données.

En somme, l'approche Handlebars illustre comment l'iPaaS moderne équilibre facilité et puissance. C'est plus simple que d'écrire du JavaScript complet ou un middleware externe, tout en étant hautement personnalisable. À mesure que la complexité de l'intégration et les capacités pilotées par l'IA augmentent, nous nous attendons à ce que Celigo enrichisse Handlebars avec davantage de logique générée automatiquement (par exemple, via des assistants de mappage IA) et des fonctionnalités de transformation de plus haut niveau. Néanmoins, comprendre la syntaxe des `{{}}`, les helpers intégrés et la manière de les combiner – comme détaillé dans ce rapport – restera crucial pour construire des flux corrects et efficaces. Les experts soulignent que « l'intégration est le système d'orchestration à travers l'entreprise » (Source: www.celigo.com) ; les langages de templating comme Handlebars sont les briques de cette orchestration.

Conclusion

Le templating Handlebars de Celigo offre un moyen riche et accessible d'implémenter des transformations de données complexes dans les flux d'intégration NetSuite. En offrant une grande variété de helpers – couvrant la logique, l'itération, l'arithmétique, le traitement des chaînes et des dates, ainsi que les recherches – Celigo permet aux non-développeurs de gérer de nombreux cas d'utilisation courants sans écrire de code. Nous avons exploré ses conventions de syntaxe, ses helpers et modèles clés, ainsi que des exemples pratiques. Ce faisant, nous avons vu comment construire des modèles qui parcourent des tableaux, appliquent une logique conditionnelle, formatent des données et intègrent des tables de recherche, autant d'éléments cruciaux lors du mappage de données vers des enregistrements NetSuite.

Notre enquête montre qu'une utilisation efficace de Handlebars nécessite une planification : les flux doivent être modulaires, testés avec des données variées et documentés pour la maintenabilité (Source: www.celigo.com) (Source: www.celigo.com). Bien que Handlebars simplifie grandement de nombreuses tâches, une logique très complexe peut toujours être mieux servie par des hooks JavaScript (Source: www.celigo.com). Néanmoins, dans la pratique, les clients Celigo intègrent régulièrement des modèles Handlebars pour personnaliser leurs intégrations, comme en témoignent les études de cas et les exemples communautaires. Nous avons également placé Handlebars dans le contexte des tendances plus larges de l'iPaaS – l'assistance IA imminente pour le mappage (Source: www.celigo.com) (Source: teknuro.com), la croissance explosive des applications connectées (Source: teknuro.com), et les meilleures pratiques du secteur.

Pour les équipes d'intégration NetSuite, maîtriser la syntaxe et les helpers Handlebars dans Celigo est essentiel. Cela libère tout le potentiel de la plateforme Celigo pour manipuler les données à la volée. À mesure que les systèmes évoluent, les modèles et les connaissances distillés ici fournissent une base qui s'adaptera : que les modèles soient écrits à la main ou générés par l'IA, la logique sous-jacente et les transformations restent guidées par ces mêmes principes. En résumé, une compréhension approfondie de Celigo Handlebars est une pierre angulaire pour les architectes d'automatisation visant à construire des intégrations NetSuite robustes, efficaces et prêtes pour l'avenir (Source: docs.celigo.com) (Source: www.celigo.com).

Étiquettes: handlebars-celigo, integratorio, integration-netsuite, transformation-de-donnees, mappage-de-champs, syntaxe-handlebars, contexte-json, assistants-handlebars