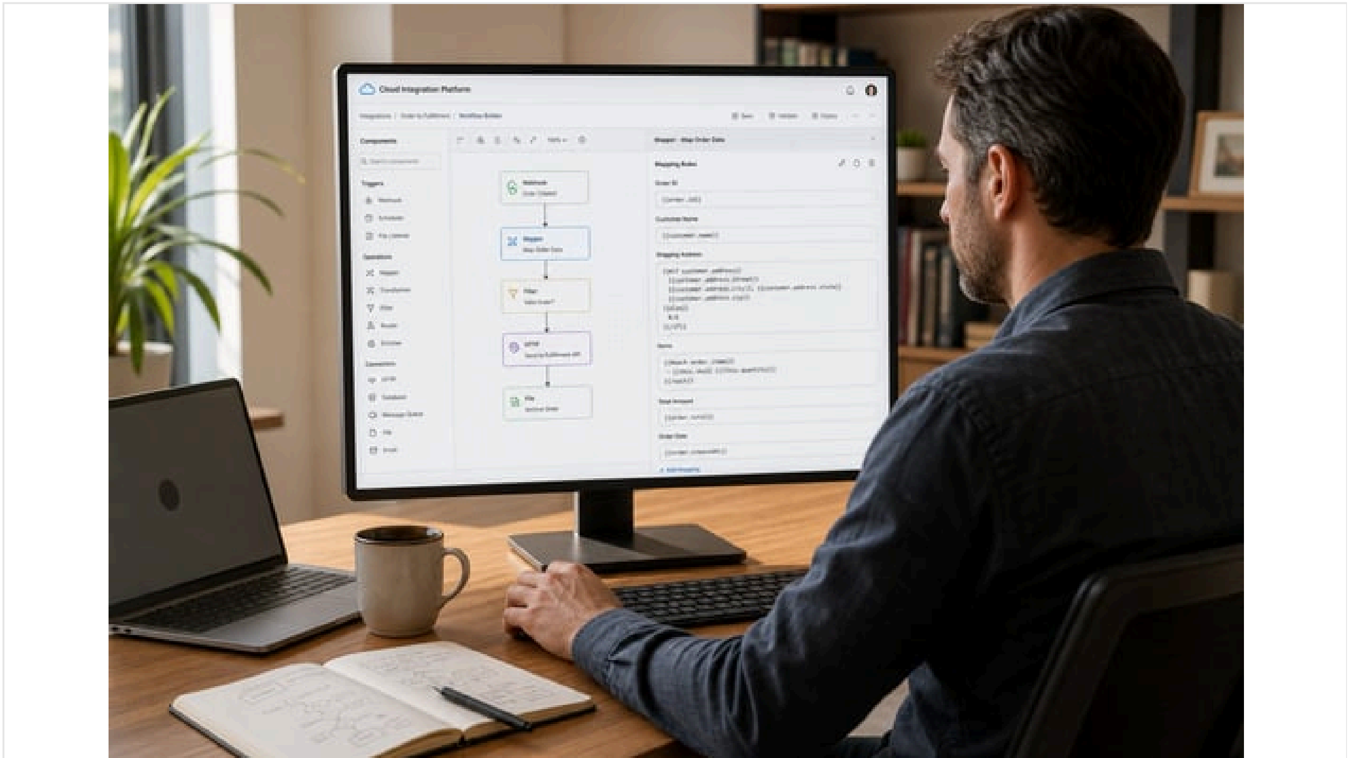
Celigo Handlebars Syntax & Helpers for NetSuite Flows

Published May 6, 2026 36 min read



Executive Summary

This report provides an exhaustive, detailed guide to using Handlebars syntax within Celigo's **integrator.io** platform – specifically focusing on helpers, conditionals, and data transformation patterns as they apply to NetSuite integration flows. It synthesizes official Celigo documentation, technical best practices, community knowledge, and industry trends to explain how Handlebars templating is employed to map, manipulate, and route data in complex integration scenarios. Handlebars is presented as the key templating language in Celigo flows, enabling dynamic field mapping, built-in arithmetic, logical branching, string and date formatting, and powerful list/array operations (Source: docs.celigo.com) (Source: www.celigo.com).

We cover the history and context of Celigo and Handlebars, the syntax conventions, and a taxonomy of built-in helpers (string, numeric, block, etc.), including numerous illustrative examples. Data variables (like `@index`, `@first`, `@root`) and control structures (#if/#else, each, compare, contains) are explained in depth. We then examine common data transformation patterns used in NetSuite flows – such as flattening nested records, aggregating and summing values, conditional defaults, and array building – showing how Handlebars helpers and Celigo's transformation engines (Rules/Mapper 2.0) support these tasks (Source: docs.celigo.com) (Source: connective.celigo.com). Case studies and real-world examples highlight how organizations leverage these features to solve integration challenges (e.g. mapping Shopify metafields to NetSuite fields (Source: neosalpha.com) or automating order workflows across platforms (Source: connective.celigo.com) (Source: integscloud.com). The report also incorporates perspectives from experts and analysts on the role of iPaaS and automation (including Celigo's recognition in Gartner's MQ) (Source: www.celigo.com) (Source: www.celigo.com). Finally, we discuss implications and future directions: the rise of AI-driven mapping and error handling (Source: teknuro.com) (Source: www.celigo.com), growing integration complexity (100+ apps per enterprise) (Source: teknuro.com), and how Celigo's platform (with its AI-assisted features like auto-mapping) positions users to meet these trends (Source: teknuro.com) (Source: www.celigo.com). Overall, the report serves as a comprehensive technical reference for any integrator designer or developer using Celigo Handlebars in NetSuite or other flows, with all claims backed by authoritative sources.

Introduction and Background

Modern enterprises rely on dozens of disparate applications, demanding robust integration platforms to automate data flows. **Celigo integrator.io** is a leading cloud-based Integration Platform as a Service (iPaaS) that specializes in connecting systems like NetSuite (the popular cloud ERP) with various other applications. By providing hundreds of pre-built **Integration Apps** and a visual flow-builder interface, Celigo enables organizations to link NetSuite bi-directionally with e-commerce, CRM, and other systems without writing custom code (Source: neosalpha.com) (Source: www.celigo.com). Celigo reports over *5,000* NetSuite customers worldwide and is often cited as the "#1 global leader in NetSuite integration," emphasizing its deep native understanding of NetSuite constructs (saved searches, SuiteScript hooks, record types, etc.) (Source: neosalpha.com) (Source: www.celigo.com).

Within Celigo, flows are composed of export (source) steps and import/lookup (destination) steps. Data flows through these steps sequentially (often with optional middleware steps) (Source: www.houseblend.io). To control and transform this data en route, Celigo offers filtering and mapping tools. At the heart of **field mapping and transformation** is the Handlebars templating language. Handlebars (originally a JavaScript templating engine compatible with Mustache) is used by Celigo to allow dynamic referencing of fields and transformation of data via expressions and helpers (Source: docs.celigo.com) (Source: docs.celigo.com). Essentially, rather than hard-coding static values, mappings in Celigo use `{{...}}` expressions to pull data from the JSON context and apply logic. For example, the Celigo Help Center explains that "Handlebars is a simple templating language" where expressions inside `{{double braces}}` are evaluated at runtime against the incoming JSON context (Source: docs.celigo.com). These expressions can perform a variety of tasks – from mapping export/import fields to doing arithmetic or encoding – making integrations both powerful and flexible (Source: docs.celigo.com) (Source: docs.celigo.com).

While Handlebars scripts handle many transformation needs, integrators can also use JavaScript hooks in Celigo for more complex logic. The platform documentation advises using Handlebars for straightforward data manipulation and falling back to JavaScript only when necessary for complex calculations or conditional logic (Source: www.celigo.com) (Source: connective.celigo.com). In practice, flows often intermix both: mapping templates using Handlebars in the field mapper rules, and custom JavaScript hooks in export/import logic when needed. This report focuses on the Handlebars aspect – detailing its syntax, built-in helpers, and typical transformation patterns – especially in the context of NetSuite-centric flows where converting and sanitizing data is a common task.

Celigo's Use of Handlebars in Integration Flows

Handlebars plays a central role in **data transformation** within Celigo flows. According to Celigo documentation, Handlebars expressions may be placed anywhere needed in import or export mappings to **"loop through, evaluate, extend, and modify records at many points in a flow."** This low-code approach empowers integrators to implement conditional logic, iterative processing, and formatting directly in the mapper, without custom scripting (Source: docs.celigo.com) (Source: www.celigo.com). In Celigo's terminology, the JSON data coming into a flow is referred to as the *context*, and Handlebars expressions evaluate against that context to produce output field values (Source: docs.celigo.com) (Source: docs.celigo.com).For example, in a mapping formula one might write `{{customer.name}}` to retrieve the `name` field of a nested `customer` object. Celigo's documentation highlights that Handlebars expressions use dot-notation to access nested object properties, similar to JSONPath (Source: docs.celigo.com) (Source: neosalpha.com). The curly-brace syntax itself is key: double braces `{{field}}` will URL-encode special characters in the output, while **triple-braces** `{{{field}}}` output raw content without encoding (Source: docs.celigo.com). In essence, the Handlebars template is like a mini-program inside the mapping sheet. When the flow runs, the template is compiled and executed: the integrator passes the JSON context object to the compiled function, which then replaces each Handlebars expression with its calculated value (Source: docs.celigo.com) (Source: docs.celigo.com).

Celigo provides an overview noting that such expressions "map Export and Import application fields, perform dynamic arithmetic calculations, and dynamically encode and decode data during integration" (Source: docs.celigo.com) (Source: docs.celigo.com). This means you can not only copy a field from source to target, but also manipulate it on the fly (for instance, summing values, formatting dates, concatenating strings) within the mapping itself. For example, one might use `{{amount * 100}}` to convert currencies, or encode a URL parameter via `{{encodeURI urlField}}`. The combination of Handlebars with Celigo's **Transformation 2.0** engine (the rules 2.0 mapping interface) means you can use JSONPath to select fields and also insert Handlebars where needed for value computations (Source: docs.celigo.com) (Source: docs.celigo.com).

All told, Handlebars in Celigo flows offers a powerful blend of readability and capability. It abstracts much of the integration "plumbing" into high-level templates. As a Celigo blog notes, "Handlebars simplify JSON manipulation, making it accessible to users without extensive programming experience." Complex JSON structures can be navigated with expressions like `{{myObj.child.array[0].value}}`, while helpers automate repetitive or complex tasks (Source: www.celigo.com) (Source: neosalpha.com). However, as with any template-based system, careful design is needed: complex logic can become hard to maintain if crammed into one ridiculously long `{{#if ...}}` block. Best practices therefore emphasize starting with a clear flow architecture, modular design, and documenting each template (Source: www.celigo.com) (Source: www.celigo.com).

Handlebars Syntax and Conventions

Handlebars syntax in Celigo is straightforward but includes some special conventions. As **Celigo's docs summarize**, Handlebars templates look like regular JSON or text with embedded expressions in double braces (`{{ }}`) (Source: docs.celigo.com) (Source: docs.celigo.com). Within these braces, you specify either a path to a field or a helper function. For example, `{{record.id}}` refers to the `id` field in the current record context, while `{{uppercase name}}` would call the `uppercase` helper on a `name` variable.

Key syntax conventions include:

- **Double vs Triple Braces**: `{{value}}` automatically URL-encodes special characters (`>?` etc.) in the output. Use triple braces `{{{value}}}` to output raw, unencoded data (Source: docs.celigo.com).
- **Literal Braces**: Prefixing a Handlebars expression with a backslash, as in `\{{escaped}}`, will output the literal text `{{escaped}}` (Braces included) instead of evaluating it (Source: docs.celigo.com).
- **Whitespace Handling**: Spaces between braces and expression (`{{ field }}` vs `{{field}}`) generally do not matter unless in quoted strings. Curly braces must not contain disallowed whitespace (e.g. no spaces between `{{` and the expression content, as noted in #if usage) (Source: docs.celigo.com).
- **Array Indexing**: Within an `each` loop, elements can be referenced by index. For instance, if iterating through `data`, `{{0.id}}` refers to the `id` of the first element; as shown in Celigo's **JSON mapping example** for a NetSuite saved search import, they prefix fields with `0.` to denote the first record in each iteration (Source: docs.celigo.com).
- **Raw Blocks**: Handlebars also supports raw block syntax `{{{{ }}}}` to disable processing for sections, but this is less commonly used in Celigo mappings.
- **Bracket Notation**: If a field name contains spaces or special chars, use square brackets. Example: `{{this.[Shipping Address]}}` to access a field literally named "Shipping Address" (Source: docs.celigo.com).

A complete "expressions" or "template" is typically just the sequence of fields, literals, and helper calls between braces. For clarity, Celigo's docs refer to the overall mapping rules or transformation logic as a **template**, which is evaluated with the JSON *context* to produce the output. An example from the Celigo docs shows a simple template and context side-by-side:

```
Template                Context                                 Output
---------               --------                                ------
{{library.title}}       {                                       Danube Incident
                         "library": {
                           "album": "The Sound",
                           "title": "Danube Incident",
                           "artist": "Lalo Schifrin"
                         }
                       }
```

In this example, `{{library.title}}` looked up the title field in the "library" object, yielding **Danube Incident** (Source: docs.celigo.com). Such template rendering is the basic operation of Handlebars within an integrator flow.

Overall, Celigo's Handlebars follows the standard semantics: dot notation to traverse objects, strings in quotes for literals, `#` and `/` to denote block helpers, and `@` prefixes for data variables (discussed below). When writing a template, you can chain multiple expressions or mix them with literal JSON text. For instance, to output JSON with both static text and dynamic values, one might write:

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

Here the outer text produces a JSON object, while `{{getValue 'record.rocket' 'defaultValue'}}` inserts a run-time value (using the `getValue` helper) (Source: docs.celigo.com). Notice how quotes change inside a helper – Celigo's docs remind you to use single quotes inside a double-quoted string so the template parser doesn't get confused (Source: docs.celigo.com).

Data Variables and Context

Handlebars templates gain additional power through built-in **data variables** that expose contextual metadata. These variables are prefixed with `@` and are valid inside block helpers. Celigo supports several such variables, documented as *helpers*: `@first`, `@last`, `@index`, `@key`, `@length`, `@root`, and `@this` (Source: docs.celigo.com). Each gives you insight into the current iteration or context:

- `@first`: **True** if this is the first iteration of a loop. For example, within `{{#each array}}...{{/each}}`, the first element yields `@first = true`, otherwise false (Source: docs.celigo.com). The docs illustrate using `{{#if @first}}` inside an `each` to output something only on the first pass (e.g. a header) (Source: docs.celigo.com).
- `@last`: **True** for the last element in a loop. Commonly used to avoid trailing delimiters (e.g. adding commas between items except after the last) (Source: docs.celigo.com) (Source: connective.celigo.com). For instance, `{{#if @last}}{{else}},{{/if}}` prints a comma only when *not* on the last element (Source: docs.celigo.com) (Source: connective.celigo.com).
- `@index`: The zero-based numeric index of the current element in an array loop. For each item in an array, `@index` starts at 0 and increments (Source: docs.celigo.com). The `index` helper example shows iterating over an array and printing `0 1 2 ...` for each element (Source: docs.celigo.com).
- `@key`: In an object loop (or array loop), `@key` provides the current property name or index as a string (Source: docs.celigo.com). In practice with arrays, `@key` is effectively the same as `@index`. In an object like `{"a":1,"b":2}`, iterating `#each this` would yield `@key = "a"` then `"b"`.
- `@length`: Returns the total length of a given value (typically a string or array). For example, `{{#compare state.length "===" 2}}` is used in docs to check if a state code string has length 2 (Source: docs.celigo.com).
- `@root`: Always refers to the top-level context object, no matter how deeply nested you are. In a deeply nested `each`, `{{@root.title}}` will still return the top-level `title` property (Source: docs.celigo.com) (Source: docs.celigo.com). This is useful if you need to pull in external fields while looping.
- `this`: References the current context object. Inside `{{#each array}}`, `this` is the current array element. If iterating an object, `this` is the current value. For example, the community shows using `{{this.Name}}` to access a field of the current object inside `{{#each myData.N1}}` (Source: connective.celigo.com). The `this` helper overview confirms that within a block, `this` points at the current element or object (Source: docs.celigo.com).

In effect, these variables let you introspect loops and objects dynamically. A common transformation pattern is to use `@last` or `@first` to format JSON output correctly (e.g. adding commas). For instance, to build a JSON array of items, one might write:

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

This generates a proper JSON list, comma-separated except after the last element (Source: connective.celigo.com) (Source: docs.celigo.com). Without `@last` checks, residual commas would make invalid JSON. Similarly, `@index` and `@key` can be used for numbering or key-building operations if needed. As the Celigo docs note, `@first` and `@last` simply return true/false for array scans (starting at index 0) (Source: docs.celigo.com) (Source: docs.celigo.com).

Together, these data variables give developers fine-grained control over how templates iterate. They are especially valuable in NetSuite flows where constructing the correct JSON or CSV format is crucial. For example, in the NetSuite saved-search example (below) `@last` is used to ensure commas separate elements, while `this` and numeric prefixes (`0.`) extract the appropriate fields for each record and its lines (Source: docs.celigo.com).

Celigo Handlebars Helpers and Conditional Logic

Celigo extends the basic Handlebars language with dozens of built-in **helpers** to handle common tasks. These span categories such as **logic/conditionals (block helpers)**, **string manipulation**, **numeric operations**, **date/time formatting**, **list/array processing**, and **miscellaneous utilities** (encoding, lookups, etc.). Below we survey these, organized by function. (Note that Celigo's official *Handlebars Helper Reference* lists all helpers; here we highlight representative ones relevant to NetSuite flows.)

Conditional and Block Helpers

Block helpers control flow logic and iteration. All begin with `#` and end with a matching `/` block tag. The primary block helpers include:

- **`{{#each}}`**: Loops over an array or object. For each element, it makes that element the current context. Syntax: `{{#each array}} ... {{/each}}`. Inside, use `this` or `@index/@first/@last` to refer to each item. The community example shows `{{#each myData.N1}}... {{/each}}` iterating Shopify order lines and using `{{this.[Entity Identifier Code]}}` in nested compares (Source: connective.celigo.com). Use `{{/each}}` to close. As a pattern, `#each` is used to create repeating output (e.g. multiple sales orders or line items in NetSuite).

- **`{{#if}}` / `{{else}}`**: Standard conditional. If the given expression or field is *truthy* (non-empty, not null/false/zero), the block renders; otherwise the `{{else}}` part renders. Example: `{{#if companyName}}{{companyName}}{{else}}{{firstName}} {{lastName}}{{/if}}` (from a BigCommerce–NetSuite mapping) outputs the company if present, otherwise the contact's first+last name (Source: connective.celigo.com). Celigo clarifies that "false" conditions include undefined, null, empty string, zero, or empty array (Source: docs.celigo.com). Crucially, do not put spaces inside the `{{#if }}` braces (no `{{#if field }}`) or it will not parse (Source: docs.celigo.com).

- **`{{#if...else}}`**: Sometimes written as a single helper (as on Celigo docs) or simply as `#if` with an `{{else}}`. This ensures only the appropriate branch executes (Source: docs.celigo.com). There are also `{{else if}}` variants supported. Celigo lists JavaScript-like operators: `<`, `>`, `===`, etc., that can be used similarly to #compare (below).

- **`{{#compare}}`**: Compares two values using an operator. Syntax: `{{#compare value1 "operator" value2}}trueBlock{{else}}falseBlock{{/compare}}` (Source: docs.celigo.com). For example, the helper reference shows: `{{#compare details.fromState "===" "NE"}}+{{details.qty}}{{else}}{{details.qty}}{{/compare}}` to prefix a plus sign only if `fromState == "NE"` (Source: docs.celigo.com). The compare helper essentially wraps `===`, `!==`, `<`, `>`, etc. It's useful for numeric or string comparisons that aren't simply "is empty" tests.

- **`{{#contains}}`**: Checks if an array or string contains a given substring/value. E.g. `{{#contains items "Special"}}Yes{{else}}No{{/contains}}`. Useful for conditional logic based on membership or substring checks. (This is a Celigo-provided helper not in standard Handlebars.)

- **`{{#and}}`, `{{#or}}`, `{{#not}}`**: Logical helpers. `#and` and `#or` take multiple conditions; they execute the block if all (and) or any (or) are truthy. `#not` negates a condition. For example, `{{#or (contains names "Alice") (contains names "Bob")}}Hi!{{/or}}` greets if either name is present. These support more complex boolean logic without leaving the template.

- **`{{#unless}}`**: The inverse of `#if`; it renders when the condition is *false*. Syntax: `{{#unless field}} (field is missing or false) {{/unless}}`. It is shorthand for `{{#if}}` with inverted sense.

- **`{{#with}}`**: Context helper that changes the context to a sub-object, `{{#with object}}...{{/with}}`. Within, fields can be accessed directly. It's similar to `#each` but for single objects.

In summary, block helpers let you loop (`#each`), branch (`#if`, `#compare`, `#contains`, etc.), and manage output formatting (`#with`, `#unless`). Their usage is extensively documented: e.g. in Celigo's "How-to" community posts, combining `#each` with `#compare` is shown for processing Shopify line items (using `this.[Entity Identifier Code]`) (Source: connective.celigo.com). Similarly, using `{{#if @last}}...{{else}}, {{/if}}` is demonstrated in official examples to conditionally emit commas between JSON objects (Source: docs.celigo.com). These patterns (loop + conditional on `@last`) are common in building JSON arrays for NetSuite's REST APIs or saved searches.

Inline and Utility Helpers

Aside from block control, Celigo provides many inline helpers (no `#` block) for data manipulation:

- **String Manipulation Helpers**: These transform text. Examples include `uppercase`, `lowercase`, `capitalize`, `capitalizeAll` (capitalize every word), `dash-case`, `snake_case`, `PascalCase`, `camelCase`, `sentence` (capitalize first word), `trim`, `trimLeft`, `trimRight`, `replace`, `replaceFirst`, `substring`, `split`, `join`, `encodeURI`, `decodeURI`, `htmlEncode`, `htmlDecode`, `hash`, etc. For instance, `{{uppercase status}}` will convert a status string to all caps; `{{replace text " " "-" }}` could hyphenate a field; `{{join array ", "}}`

concatenates elements with a comma and space (Source: docs.celigo.com) (Source: docs.celigo.com). These are essential for text formatting tasks – e.g. turning a comma-separated list of tags into an array of strings, stripping HTML from rich-text, or ensuring data matches NetSuite's expected format.

- **Numeric Helpers**: For arithmetic and numeric formatting, Celigo has `add`, `subtract` (or `minus`), `multiply`, `divide`, `modulo`, as well as `sum`, `avg` (to aggregate array values), `abs`, `ceil`, `floor`, `round`, `toFixed`, `toPrecision`, etc. For example, `{{sum thisLinePrices}}` would output the total if `thisLinePrices` is an array of numbers (Source: docs.celigo.com) (Source: docs.celigo.com). Or `{{divide total 100}}` might convert cents to dollars. These allow you to do on-the-fly math. For instance, Celigo's decimal handling examples show using `{{divide $value 100}}` to fix currency precision. The blog notes that JavaScript is better suited for *complex maths*, but for simple sums or scaling, Handlebars helpers suffice (Source: www.celigo.com).

- **Date/Time Helpers**: Celigo includes helpers like `dateAdd`, `dateFormat`, `timestamp` (current time). These are essential when dealing with NetSuite date fields or converting between time zones. For example, `{{dateAdd createdDate 1 "days"}}` could shift a date forward by a day, and `{{dateFormat createdDate "YYYY-MM-DDTHH:mm:ssZ"}}` formats it in ISO8601. The Houseblend guide on filters (while about matches operator) and Celigo blogs suggest using date/time helpers when mapping date fields (Source: www.houseblend.io) (Source: www.celigo.com). There's also mention of time zone support in the Handlebars examples section.

- **List/Array Helpers**: Beyond `each`, Celigo offers helpers like `arrayify` (ensures a value is an array), `unique` (remove duplicates), `sort`, `pluck` (extract an array of a given field from a list of objects), and `hash` (create key/value pairs). For instance, if you have an array of order objects, `{{pluck orders "id"}}` would return an array of all the order IDs. The `filter` helper (block) can filter an array by a predicate (e.g. `{{#filter items "category" "Electronics"}}...{{/filter}}`). These are useful in transformation flows: for example, extracting unique customer emails from a sales batch, or building a lookup table from part of the data.

- **Encoding and Miscellaneous Helpers**: Helpers like `encodeURI`, `decodeURI`, `htmlEncode`, `htmlDecode` manage special characters. `base64Encode`/`Decode` handle Base64 operations. There are even AWS-related helpers (`aws4` for signing requests, `hmac` for hashing) for custom API calls. Celigo also provides JSON utilities like `jsonParse`, `jsonSerialize`, `jsonEncode` for dealing with raw JSON strings within mappings. The `getValue` helper (discussed below) safely fetches nested fields, and `typeOf` returns the JavaScript type of a value (for debugging). Finally, the **Lookup helpers** (`lookup` category) deserve special mention: `{{lookup}}` retrieves data from Celigo "Lookup" tables defined in the flow (see below), and the `$` context (from integration apps) refers to the entire record context (Source: connective.celigo.com) (Source: docs.celigo.com).

To illustrate usage: one can write

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

to output a name in caps and a formatted date. Or

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

to conditionally mention a discount. The **official examples section** in Celigo docs provides many such samples: e.g. using date format codes, hash/hmac for API calls, or regex replacements to sanitize phone numbers (Source: connective.celigo.com) (Source: connective.celigo.com). Likewise, community FAQs showcase regex helpers (regexReplace, regexMatch) used for data cleansing. Overall, these helpers allow powerful data transformations within the mapping layer – reducing the need for external scripts, while still giving precise control.

Table 1 (below) summarizes some common Handlebars helper categories in Celigo:

CATEGORY	PURPOSE	EXAMPLE HELPERS / USAGE
Logic / Conditional	Branching and boolean tests	`{{#if}}, {{#if ... else}}, {{#unless}}, {{#compare}}, {{#contains}}, {{#and}}, {{#or}}` *Example:* `{{#compare status "==" "Closed"}}Closed{{/compare}}`
Iteration / Looping	Loop over lists or objects	`{{#each}}, {{@index}}, {{@first}}, {{@last}}, {{#with}}` *Example:* building array: see below.
String / Text	Manipulate text values	`uppercase, lowercase, capitalize, dash-case, snake_case, replace, split, join, encodeURI, htmlEncode, hash, jsonEncode` *Example:* `{{removeProtocol url}}` (strips "http://"), `{{join tags ", "}}`
Numeric / Math	Arithmetic on numbers	`add, subtract, multiply, divide, modulo, sum, avg, ceil, floor, round, toFixed`, etc. *Example:* `{{round price 2}}` rounds a number.
Date / Time	Format or adjust dates	`dateAdd, dateFormat, timestamp` *Example:* `{{dateFormat orderDate "YYYY-MM-DD"}}`
Array / List	Work with arrays & arrays of objects	`unique, sort, pluck, arrayify, filter` *Example:* `{{unique emails}}` to dedupe an array
Encoding / Misc	Encode/lookup operations, hashing	`getValue, lookup, base64Encode/base64Decode, htmlDecode, regexMatch/regexReplace, aws4, hmac, typeOf` *Example:* `{{lookup "stateConfig" record.state}}`

Table 1: Common categories of Handlebars helpers supported by Celigo. Each category contains multiple helpers for specific tasks (per the Celigo Handlebars reference (Source: docs.celigo.com) (Source: docs.celigo.com).

Data Lookups and `getValue`

Two particularly useful helpers often used in NetSuite flows are **`getValue`** and **`lookup`**, which retrieve data from context or from Celigo's lookup tables.

The `{{getValue}}` helper allows safe retrieval of a field by path, with an optional default if missing (Source: docs.celigo.com). Its syntax is `{{getValue "record.path.to.field" "defaultVal"}}`. If the specified field is null/undefined, the default string is returned. For example, `{{getValue "record.email" "none@none.com"}}` will output the email if present, or `"none@none.com"` otherwise (Source: docs.celigo.com) (Source: docs.celigo.com). This prevents breakage when some records lack a field. It's especially handy when field names are dynamic or unsure – instead of risking an error, you can use a default. Celigo's docs underscore this as a pattern for "safe" field access in templates (Source: docs.celigo.com).

The `{{lookup}}` helpers fetch values from *Lookup* tables defined within the flow's interface (Source: docs.celigo.com). There are two types of Celigo lookups: **dynamic search** (mapping values based on a saved search or API query) and **static value-to-value** (hardcoded dictionary). A dynamic lookup might, for example, translate a regional code into an internal ID by searching NetSuite. In a template, you use `{{lookup.lookupName}}` for dynamic, or `{{lookup "lookupName" record.field}}` for static lookups (Source: docs.celigo.com). Celigo's documentation shows:

- `{{lookup.shippingLookup}}` would use a dynamic search called "shippingLookup" to retrieve, say, a shipping method from the destination system (Source: docs.celigo.com).
- `{{lookup "myStaticLookup" orderType}}` would use a static mapping based on the `orderType` field.

These lookup helpers are invaluable for data transformation patterns like tax-code or SKU mapping: you can define in one place how to convert values, then call them inline. For instance, a NetSuite order flow might use a lookup named `taxLookup` so in mapping use `{{lookup "taxLookup" order.taxCode}}` to get the correct NS tax ID. The Celigo docs underline that lookup names must match exactly the flow's configured lookup.

Both `getValue` and `lookup` expand the power of Handlebars in flows by accessing data outside the immediate template parameters. In combination with JSONPath (e.g. `$.queryParams.id` in transformation) (Source: docs.celigo.com), one can fetch virtually any context information or external setting.

Conditional Logic in Practice

Conditionals are critical when integrating systems with varying data rules. Some common patterns include:

- **Existence Check / Defaults**: Use `#if` to check if a field exists or has a value and provide a default otherwise. E.g. `{{#if account.default}}Yes{{else}}No{{/if}}`. Official docs illustrate this as choosing between company versus first/last name (Source: connective.celigo.com). One can provide a fallback with `getValue` similarly.
- **Multi-branch Logic**: Nest `#if` or use `#compare` / `#contains` for different cases. The Shopify-to-NetSuite community example (Table 2 below) shows cropping by entity type using nested compares inside `{{#each}}`. In practice, you might see multiple `{{else if}}` chains to handle various source conditions.
- **Aggregate Conditions**: Use `#and` / `#or` to combine checks. For example, `{{#and (contains comments "urgent") (eq priority "high")}}` to detect high-priority urgent tickets.
- **@first/@last in Data Lists**: Within loops, one often writes `{{#if @first}}...{{/if}}` to prefix special elements, or `{{#if @last}}...{{else}},{{/if}}` as shown, to properly comma-separate JSON arrays (Source: docs.celigo.com) (Source: connective.celigo.com). For example, building a JSON array for a NetSuite API might look like:

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

  This ensures the JSON is valid by controlling the comma placement (Source: docs.celigo.com) (Source: connective.celigo.com).
- **Regex and Pattern Conditions**: Celigo includes regex helpers. For instance, `{{#regexSearch phone "(\d{3})-(\d{3})-(\d{4})"}}` could check if a phone matches a pattern. Community FAQs demonstrate using `regexReplace` to format phone numbers (remove non-digits, etc.) (Source: connective.celigo.com). One pattern might be stripping ASCII or punctuation: `{{regexReplace description "[^\\x20-\\x7E]" ""}}` to remove extended ASCII.
- **Advanced List Checks**: If a record has fields in multiple possible locations, one might see something like `{{#if array[*].sku}}...{{else}}...{{/if}}` where `[*]` indicates checking any element. (This was seen in the problematic community question [22], though that attempted wildcard indexing which did not work as intended.) Generally, it's safer to use `#each` plus an inner `#contains` or `#compare` to check if *any* element meets a condition.

Celigo's best-practice blog emphasizes keeping conditional logic **clear and maintainable**: e.g. use built-in block helpers rather than burying too much code in `{{}}` expressions, and test templates thoroughly to account for all data scenarios (Source: www.celigo.com) (Source: docs.celigo.com). Complex branching might better be handled in a pre-processing step or JavaScript hook for readability, reserving Handlebars for simpler conditions.

Data Transformation Patterns in Flows

In practice, Celigo handles a variety of data shapes and formats. NetSuite integration flows often need to **reshape data** to match record schemas. Here are common transformation patterns and how Handlebars fits in:

- **Flattening Nested Structures**: Source systems may output nested JSON (e.g., an order with nested line items). Celigo transforms can flatten such structures into the tabular format required by NetSuite records. A pattern is to use multiple `#each` loops: one for the order header (e.g., external ID, date) and an inner `#each` for line items (each creating a sub-record). In the saved-search JSON example, an outer `{{#each data}}` loops through orders, then an inner `{{#each this}}` loops through line items (Source: docs.celigo.com). Inside the loop, fields are extracted using indexed references (e.g., `{{0.id}}` for header fields and `{{Item}}` for line fields). Thus, nested input is serialized into arrays of per-line JSON objects. The comma logic (`{{#if @last}}…{{else}},{{/if}}`) we saw also ensures proper JSON syntax (Source: docs.celigo.com). Another approach is to use Celigo's **Transformation 2.0 (Rules 2.0)** engine in "create output rows" mode, which is designed for producing multiple rows from one input. This mode internally performs a similar expansion, but Handlebars within mapper fields can still be used for field values (e.g. `{{lookup "itemLookup" this.sku}}`).

- **Aggregating or Summing Values**: Sometimes multiple source records must combine into one target. For example, if multiple sales orders from Shopify should map to a single combined order on NetSuite with summed quantities. Here one might use `sum` on an array: if prior steps collected all quantities into an array `quantities`, use `{{sum quantities}}` to get the total (Source: docs.celigo.com). Or use `avg` / `min` / `max` similarly. With dates, one could use `dateAdd` to compute durations across records. In simpler cases, one could just do `{{quantity * price}}` for a single record's total, but for arrays, helpers are handy.

- **Defaulting and Fallbacks**: The `getValue` helper pattern often appears to supply default values when upstream data is missing, avoiding errors. For instance, `{{getValue "record.discount" "0"}}` ensures a discount field always outputs a number. Combined with `#if`, one can also do `{{#if record.discount}}{{record.discount}}{{else}}0{{/if}}` (Source: connective.celigo.com). Similarly, one might provide default strings for missing phone/email: `{{getValue "record.phone" "000-000-0000"}}`.

- **Lookup and Translation Tables**: NetSuite often requires internal IDs instead of human-readable codes. A mapping pattern is to use a Celigo lookup (built via a lookup table in the flow). For example, translating a currency code to a NetSuite currencyId can use a static lookup: `{{lookup "currencyMap" record.currencyCode}}`. Or a dynamic lookup might fetch an employeeID from NetSuite given an email. After creating the lookup in the UI, the Handlebars `{{lookup}}` helper makes the translation seamless (Source: docs.celigo.com).

- **Conditional Field Inclusion**: Sometimes a field should only be sent if a condition holds. In the NetSuite invoice mapping, one might have a line like: `Discount: {{discountAmount}}` only if a discount applies. Using `#if` or `#contains` around `{{discountAmount}}` can suppress it when zero. The Headspace case study hints at such logic by eliminating duplicate entries and error conditions through Celigo flows (Source: www.celigo.com) (though it does not detail the template, real flows likely used such conditionals).

- **JSON Assembly (Text Templates)**: Although Celigo mostly works with field mappings, sometimes one builds large JSON payloads manually. In these cases, Handlebars is used to inject pieces. A pattern from the community shows building a JSON array string with `#each` and commas (Source: connective.celigo.com). Another pattern is constructing dynamic SQL (such as Postgres queries) using double `{{}}` to encode values. Celigo docs note that double curly braces automatically URL-encode, which is important if including query parameters in an HTTP GET or POST payload (Source: docs.celigo.com).

- **Regex and Cleanup**: Patterns often involve cleaning text fields. For instance, to remove non-ASCII or HTML from a description, use helpers like `regexReplace` or `htmlDecode`. Example: `{{regexReplace description "[^\\x00-\\x7F]" ""}}` strips extended characters. In phone formatting, one might do `{{regexReplace phone "[^0-9]" ""}}` to leave only digits, then re-insert formatting. Celigo Connective forums have "how-to" posts specifically for phone number and currency formatting using regexReplace and other helpers.

- **Date/Time Conversion**: NetSuite expects dates in a specific format. A common pattern is converting a source date (e.g. "MM/DD/YYYY") into ISO. Celigo provides date format codes for Ruby/JavaScript style templates. For example, use `{{dateFormat NSDate "yyyy-MM-dd'T'HH:mm:ssZ"}}` or similar. There is also a `timestamp` helper to get "now". If business logic requires offsetting dates (e.g. switch time zones or add business days), use `dateAdd` with the appropriate unit (days, hours). A Celigo blog on best practices suggests keeping date/times in UTC unless required, but the Handlebars date helpers make it easy to adjust as needed.

- **Array Construction**: Beyond flattening, sometimes you need to build a new array from individual fields. One pattern is shown in the community (see Table 2): after looping, you might use `{{create}}` or assembly via Handlebars (though pure Handlebars can only build strings, one can trick it as JSON text). For instance:

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

will output a JSON array of part objects. Celigo's UI also supports an explicit "Array" transformation feature nowadays.

These patterns leverage Celigo's **Transformation 2.0** / **Mapper 2.0** engine, which treats the Handlebars template as part of a declarative mapping. Celigo docs note that Rules 2.0 (transforms) can use JSONPath to select fields and Handlebars **discussed above** to compute values (Source: docs.celigo.com). One can even mix JSONPath (`$.field`) with Handlebars in one expression. The advantage is that Transformation 2.0 can output multiple rows per input (array output mode) and automate nested array handling, making some of the above patterns less manual. Nevertheless, understanding the Handlebars approach remains useful, as many transformation rules still allow "Handlebars expression" mode for custom formulas (Source: docs.celigo.com) (Source: docs.celigo.com).

Table 2 (below) illustrates a few representative Handlebars transformation patterns with example expressions.

PATTERN / USE-CASE	HANDLEBARS EXPRESSION EXAMPLE	INTENT / EXPLANATION
Default if missing (fallback)	`{{getValue "record.discount" "0"}}`	Safely retrieve `record.discount`, default to "0" if absent.
Conditional Output	`{{#if isActive}}Active{{else}}Inactive{{/if}}`	Output "Active" only if `isActive` is truthy; else "Inactive".
Concatenate Fields	`{{record.firstName}} {{record.lastName}}`	Build full name by combining first and last name.
Iterate & Build Array (comma-separated)		

~~~~
[{{#each items}}
 {"id":"{{this.id}}","qty":{{this.qty}}}{{#if @last}}{{else}},{{/if}}
{{/each}}]
``` | Loop through `items`, output JSON objects for each, separated by commas (skip comma on last) (Source: [docs.celigo.
| **Sum of Array Values**                       | `{{sum order.quantities}}`                                      | Sum all
| **Regex Cleanup (punctuation)**               | `{{regexReplace address "[^a-zA-Z0-9 ]" ""}}`                   | Remove
| **Lookup/Translate Value**                    | `{{lookup "taxMap" record.state}}`                              | Use a
| **Conditional Array Element**                 | `{{#contains tags "urgent"}}HasUrgent{{else}}Normal{{/contains}}` | Che

*Table 2: Example data transformation patterns expressed with Celigo Handlebars.* Each row shows a common need in NetSuit

# Case Studies and Real-World Examples

To ground these concepts, consider some illustrative real-world scenarios and how Handlebars is applied:

- **E-commerce Order Integration**: A retailer running Shopify and NetSuite needs to sync orders. The shop’s order JSON i

- **NetSuite Saved Search Import**: An external system triggers a NetSuite saved search and sends its results to Celigo v

- **CRM/ERP Sync**: A company syncing Salesforce→NetSuite for opportunity data may only want to send orders above a certa

- **Real-World Results**: In published success stories, companies highlight the effect of using Celigo flows. For instanc

- **Community Q&A Examples**: The Celigo Connective forums also show various problems solved with Handlebars. For example

# Implications and Future Directions

Celigo’s Handlebars integration is part of broader trends in data integration and iPaaS. Industry analysts note that ente

In this context, Handlebars remains a valuable tool for now, but we see the following trends and implications:

- **AI-Assisted Mapping**: Celigo itself is embracing AI. For example, Celigo claims its platform resolves ~95% of integr

- **Low-Code Empowerment**: Integration platforms are moving toward more visual, low-code development. Celigo’s flow buil

- **Hybrid & Multi-Cloud Integration**: Many businesses use a mix of on-premises and cloud systems. Celigo’s Handlebars m

- **Integration Governance and Security**: As more data flows, security and compliance rise in importance. Celigo iPaaS i

- **Serverless/Edge Connectors**: A long-term trend is more dynamic or event-driven integration. Celigo already supports

In sum, the Handlebars approach exemplifies how modern iPaaS balances ease and power. It is simpler than writing full Jav

# Conclusion

Celigo’s Handlebars templating provides a rich yet accessible means to implement complex data transformations in NetSuite

Our investigation shows that effective usage of Handlebars requires planning: flows should be modular, tested against var

For NetSuite integration teams, mastering the Handlebars syntax and helpers in Celigo is essential. It unlocks the full p
~~~~

Tags: celigo handlebars, integrator.io, netsuite integration, data transformation, field mapping, handlebars syntax, json context, handlebars helpers

DISCLAIMER

This document is provided for informational purposes only. No representations or warranties are made regarding the accuracy, completeness, or reliability of its contents. Any use of this information is at your own risk. Houseblend shall not be liable for any damages arising from the use of this document. This content may include material generated with assistance from artificial intelligence tools, which may contain errors or inaccuracies. Readers should verify critical information independently. All product names, trademarks, and registered trademarks mentioned are property of their respective owners and are used for identification purposes only. Use of these names does not imply endorsement. This document does not constitute professional or legal advice. For specific guidance related to your needs, please consult qualified professionals.