BridgeLink Lookup Manager User Guide

Table of Contents

About This Guide 3
Requirements 4
Key Concepts 5
Accessing the Lookup Manager 6
Managing Lookup Groups 8
Managing Values 11
Importing and Exporting 13
Searching Values 15
Using Lookups in Channels 18
Cache Status and Performance 24
Viewing History 26
Storage and External Database 28
Access Control 30
Code Template Reference 31
REST API Reference 33
Import File Format Reference 35
Troubleshooting 38
Support 40

About This Guide

Lookup Manager centralizes the key-value reference data your channels depend on: billing codes, provider directories, code crosswalks between two systems, and similar lookup tables. Instead of hardcoding these mappings in JavaScript or scattering them across ad hoc SQL, you manage them in one place in WebAdmin, and your channels read and write them at runtime through a JavaScript helper or the REST API. This guide is for BridgeLink administrators and integration developers who create and maintain lookup groups, and for anyone writing a channel script that reads or writes lookup data. By the end of this guide, you will be able to create and manage lookup groups and their values, import and export them, search them (including structured search inside JSON values), call them from a channel script, and monitor their cache performance.

Key Features

Feature	Description
Centralized value management	Replace inline scripts and scattered SQL with one managed lookup group per reference data set.
TEXT and JSON values	Store plain strings, or JSON documents with optional indexed-field search.
Real-time channel access	Read and write lookup data from a transformer or code template through the `LookupHelper` namespace, or from any system through the REST API.
Per-group caching	Configurable in-memory caching (**LRU** or **FIFO**) with hit and miss statistics.
Full audit trail	Every value change is recorded and searchable on the group's **History** tab.
CSV and JSON import and export	Bulk-load or back up a group's values as CSV, or a complete group as JSON.

Requirements

- BridgeLink 26.6.0 or later.
- If your organization uses the Advanced Access Control plugin: a role with the **Lookup Manager** permission, assigned by your administrator (see **Access Control**). Without that plugin, no permission setup is needed.
- An external database (PostgreSQL 9.4 or later, MySQL 8.0 or later, Microsoft SQL Server, or Oracle 12.2 or later) if you plan to store values as JSON. The default Derby database does not support the JSON value type. See **Storage and External Database**.

Key Concepts

- **Group**: a named collection of related key-value pairs, similar to a table. Every group has its own cache settings, value type, and, for JSON groups, indexing configuration.
- **Key**: a unique identifier within a group, up to 255 characters.
- **Value**: the data stored for a key. Every value is stored and returned as a string, even for JSON groups (see below).
- **Value Type**: set when you create a group and cannot be changed afterward. **TEXT** (default) accepts any string and works with any database. **JSON** validates each value as well-formed JSON when it is written and can index named fields inside it; it requires an external database.
- **FIELD indexing**: for JSON groups, you can index one or more field paths inside the JSON value, for example `npi` or `address.city`. An indexed field can be searched directly, both from the **Advanced Search** dialog and from a channel script, without scanning every value in the group.
- **Cache**: each group keeps recently used values in memory for fast repeated lookups. Cache size and eviction policy are configured per group. See **Cache Status and Performance**.

A note on JSON values: Lookup Manager always stores and returns values as strings, even for JSON-typed groups. A channel script that reads a JSON value gets the raw JSON text back, not a parsed object, and must call `JSON.parse()` on it before reading a field. This is covered in detail in **Using Lookups in Channels**.

Accessing the Lookup Manager

On servers using the Advanced Access Control plugin, whether the **Lookup Manager** navigation item appears for a given user depends on the permissions assigned to that user's role (see **Access Control**); without that plugin, it appears for every signed-in user. If it does not appear at all, confirm that the underlying extension, listed on the **Settings > Extensions** page as **Lookup Table Management System**, is **Enabled**.

1. Sign in to **WebAdmin**.
2. Click **Build** in the left sidebar.
3. Click **Lookup Manager**.

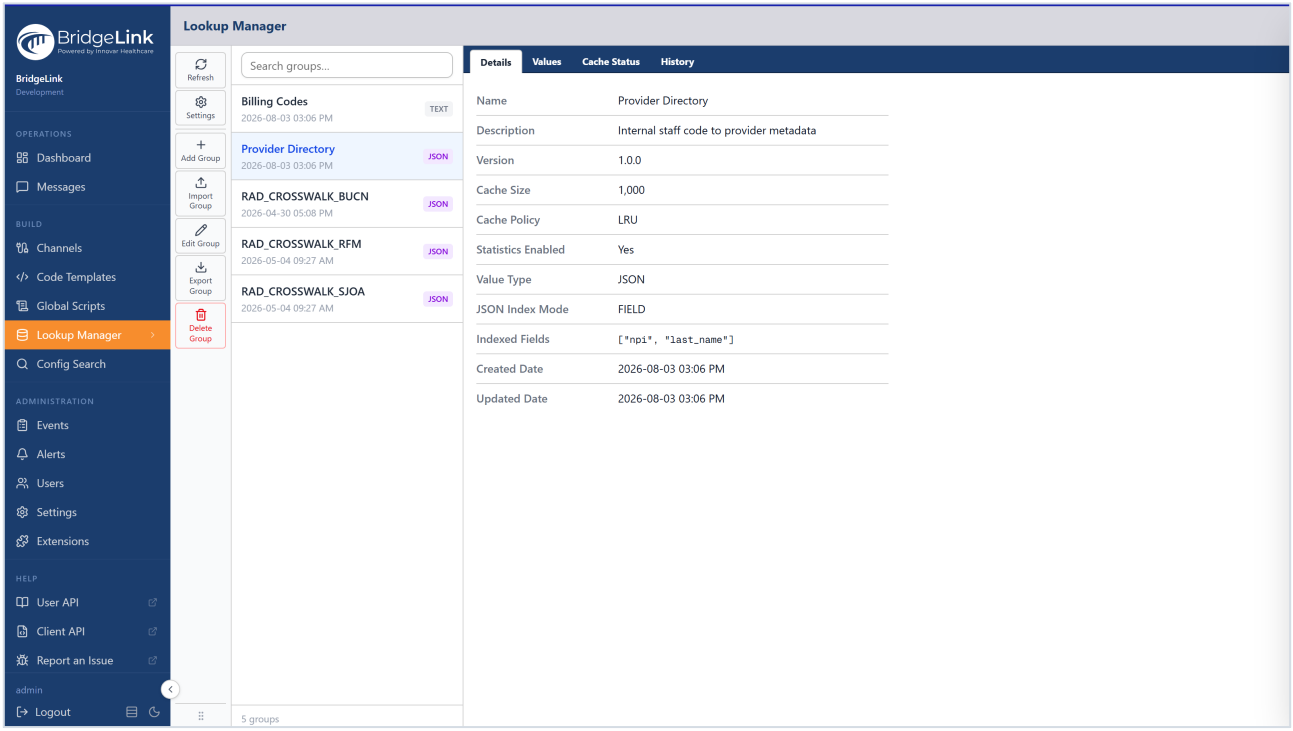

Figure 1: The Lookup Manager page under Build in the left sidebar, showing the group list on the left and the Details tab for a selected group on the right.

The page has two panels. The left panel lists every lookup group on your server, with a search box above it and a group count below it. Each row shows the group's name, when it was last updated, and a badge for its value type (**TEXT** or **JSON**). Click a group to select it.

The right panel shows four tabs for the selected group:

- **Details** shows the group's full configuration: **Name**, **Description**, **Version**, **Cache Size**, **Cache Policy**, **Statistics Enabled**, **Value Type**, and, for JSON groups, **JSON Index Mode** and **Indexed Fields**, plus **Created Date** and **Updated Date**.
- **Values** is where you view, search, add, edit, delete, import, and export the group's key-value pairs. Covered in **Managing Values**.

- **Cache Status** shows cache and lookup statistics for the group. Covered in **Cache Status and Performance**.
- **History** shows an audit trail of changes to the group's values. Covered in **Viewing History**.

Right-click a group in the left panel for a shortcut menu with **Edit**, **Export JSON**, and **Delete**.

Managing Lookup Groups

Creating a Group

1. Click **Add Group** in the toolbar.
2. In the **Add Group** dialog, type a **Name** and, optionally, a **Description**.
3. Type a **Version** label, for example `1.0.0`. This is a free-text label for your own tracking; BridgeLink does not enforce a format or increment it automatically.
4. Set **Cache Size**, the maximum number of entries BridgeLink keeps in memory for this group. Type `0` to disable caching. Default: `1000`.
5. Choose a **Cache Policy**: **LRU** (evicts the least recently used entry first) or **FIFO** (evicts the oldest entry by insertion order first). Default: **LRU**.
6. Leave **Statistics Enabled** checked to track lookup and cache-hit counts for this group, or clear it to skip that tracking. Default: checked.
7. Choose a **Value Type**: **TEXT** (default) or **JSON**. You cannot change this after the group is created.
8. If you chose **JSON**, choose a **JSON Index Mode**. Choose **NONE** to store and validate JSON values without indexing them; searches against a JSON field still work, from the **Advanced Search** dialog or from a script, but scan every value in the group. Choose **FIELD** to have BridgeLink build a database index for one or more field paths, entered next under **Indexed JSON Fields**: type a path, for example `npi` or `address.city` for a nested field, and click **Add**, or press **Enter**. You cannot change the **JSON Index Mode** or the indexed fields after the group is created.
9. Click **Save**.

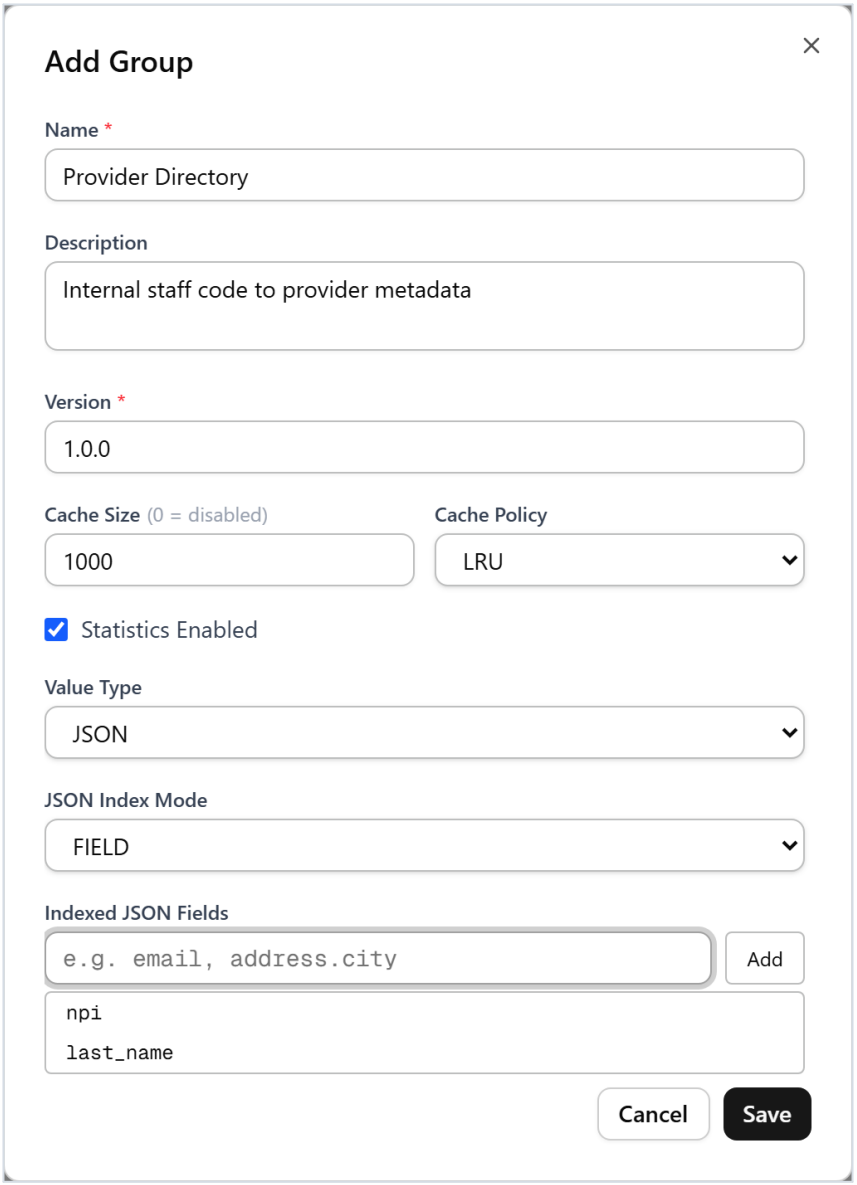

Figure 2: The Add Group dialog, showing Name, Description, Version, Cache Size, Cache Policy, Statistics Enabled, Value Type, and the JSON Index Mode and Indexed JSON Fields controls shown for a JSON-typed group.

Editing a Group

1. Select the group in the left panel.
2. Click **Edit Group** in the toolbar, or right-click the group and select **Edit**.
3. Update the fields you want to change. **Value Type**, **JSON Index Mode**, and **Indexed JSON Fields** cannot be changed here; delete and recreate the group if you need to change any of them.
4. Click **Save**.

Deleting a Group

Deleting a group permanently removes it and all of its values.

1. Select the group in the left panel.
2. Click **Delete Group** in the toolbar, or right-click the group and select **Delete**.
3. In the confirmation dialog, type `REMOVEGROUP` and click **OK**.

Managing Values

Select a group and click its **Values** tab.

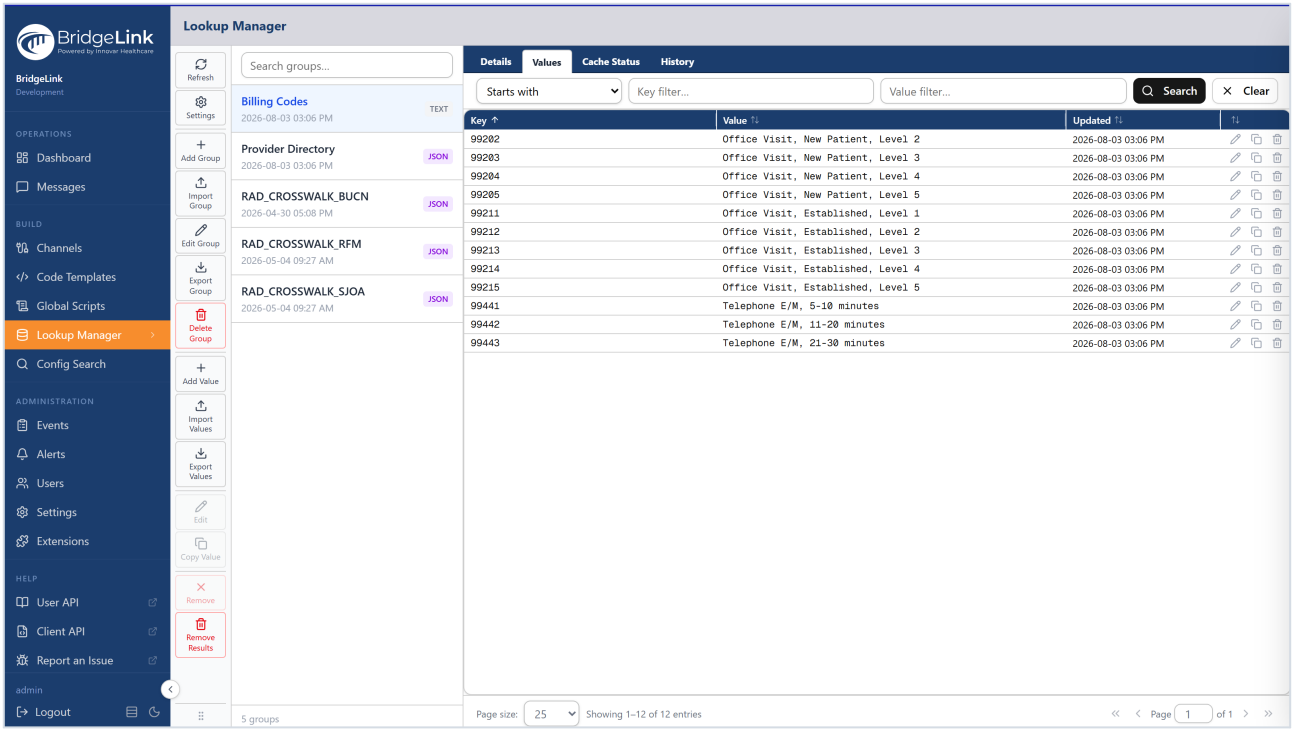

Figure 3: The Values tab for a selected group, showing the simple search bar, the values table, and the pagination bar.

Adding a Value

1. Click **Add Value** in the toolbar, or right-click in the values table and select **Add Value**.
2. Type a **Key**.
3. Type a **Value**. For a **TEXT** group, this is a plain multi-line text field. For a **JSON** group, it is a JSON editor with **Raw** and **Tree** tabs: edit the JSON as text on **Raw**, or browse it as a collapsible tree on **Tree**. If the JSON has a syntax error, switch back to **Raw** to fix it; **Tree** cannot display invalid JSON.
4. Click **Save**.

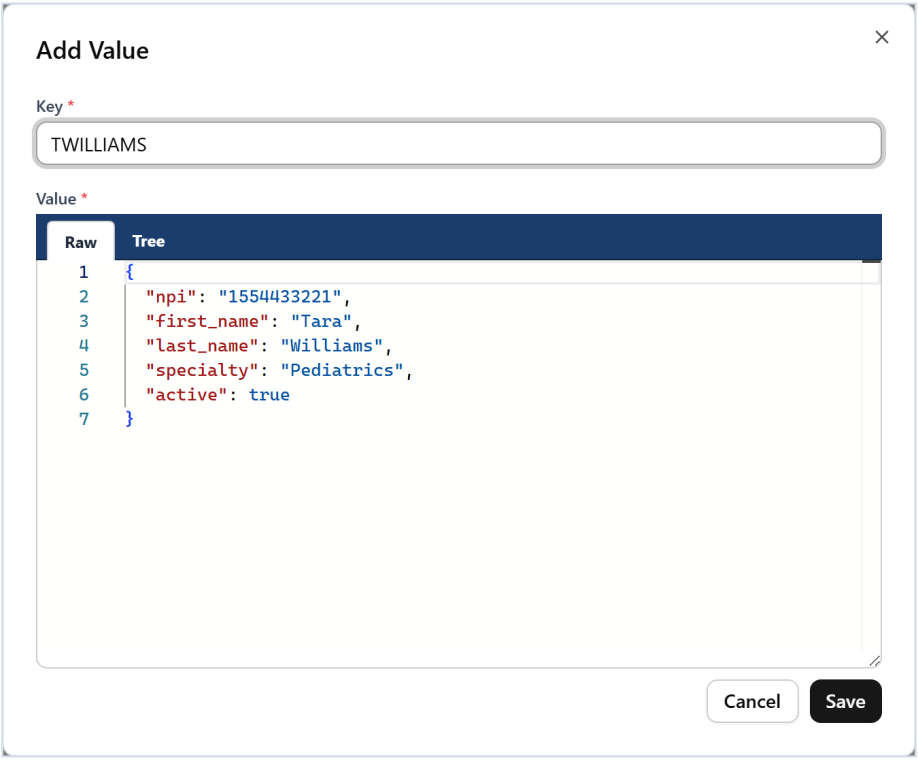

Figure 4: The Add Value dialog for a JSON-typed group, showing the Raw and Tree tabs.

If the key you typed already exists, a warning appears asking whether to overwrite it. Click **Yes, Overwrite** to replace the existing value, or **Cancel** to go back.

Editing or Deleting a Value

1. Double-click a row in the values table, or select it and click **Edit** in the toolbar or the pencil icon in the row.
2. Update the **Value**. The **Key** field is read-only here; delete the value and add a new one if you need to change a key.
3. Click **Save**.

To delete one or more values:

1. Select one or more rows. Ctrl-click, or Cmd-click on macOS, to select more than one.
2. Click **Remove** in the toolbar, or the trash icon in a single row.
3. Confirm the deletion.

Remove Results, also in the toolbar, deletes every value currently matching your search or filter, not just the selected rows. Because this can affect a large number of values at once, BridgeLink asks you to type `REMOVEALL` to confirm.

Use the pagination bar below the table to change the page size (`10` to `1000` rows) or jump to a specific page.

Importing and Exporting

Lookup Manager supports two kinds of import and export: CSV for a single group's values, and JSON for a complete group, meaning its configuration and every value together.

Importing Values from CSV

1. Select a group and open its **Values** tab.
2. Click **Import Values** in the toolbar.
3. Choose a CSV file with two columns, `key` and `value`. BridgeLink treats the first row as a header and skips it regardless of its exact text. Quote any field that contains a comma or a quote, and double any quote characters inside a quoted field.
4. Choose **Append** to add the new rows to the group's existing values, or **Replace existing** to clear the group first.

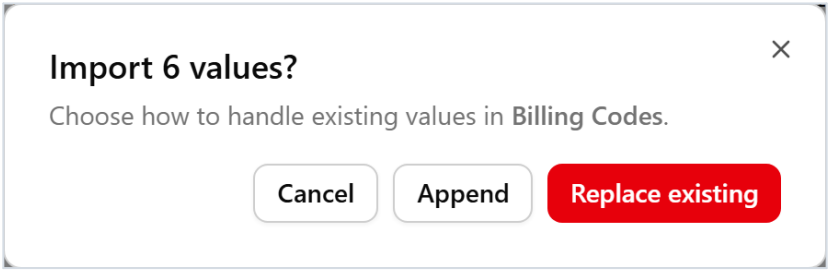

Figure 5: The dialog shown after choosing a CSV file, offering Cancel, Append, and Replace existing.

BridgeLink imports large files in batches of 100 rows at a time, with a progress dialog and a **Cancel** button. If you cancel partway through, the rows already imported stay in the group.

Exporting Values to CSV

1. Select a group and open its **Values** tab.
2. Click **Export Values** in the toolbar.

BridgeLink downloads every value in the group as a CSV file named `all_values_YYYY_MM_DD_HH_mm.csv`.

Importing a Group from JSON

1. Click **Import Group** in the toolbar.
2. Choose a source. Choose **System** for one of BridgeLink's bundled default groups: **Race**, **Ethnicity**, **Administrative Gender**, **Marital Status**, and **Religion**, each a ready-made value set for a common HL7-style demographic field. Choose **File** to import your own JSON file, matching the shape produced by **Export Group** (see **Import File Format Reference**).

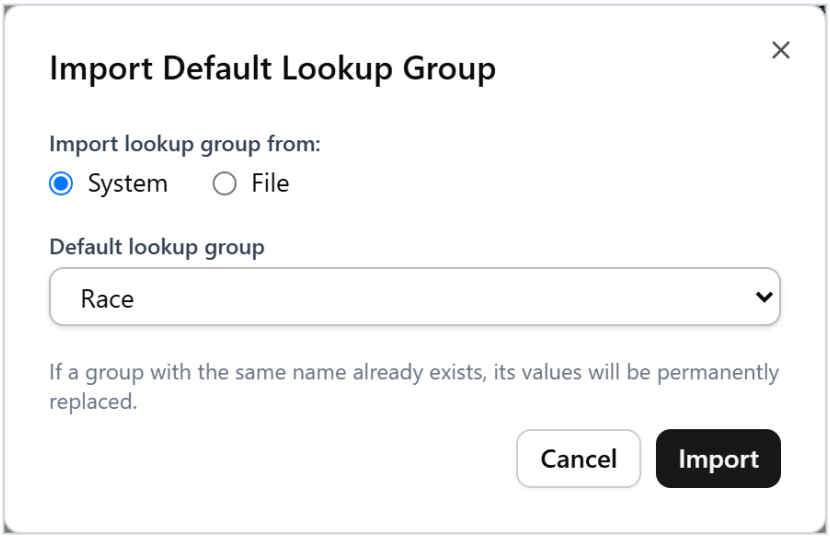

Figure 6: The Import Default Lookup Group dialog, showing the System and File source options.

3. Click **Import**.
4. Confirm the import. If a group with the same name already exists, its configuration is updated and all of its existing values are permanently replaced.

Exporting a Group to JSON

1. Select the group in the left panel.
2. Click **Export Group** in the toolbar, or right-click the group and select **Export JSON**.

BridgeLink downloads the group's configuration and every value as a single JSON file, named after the group.

Searching Values

The **Values** tab supports two search modes. **TEXT** groups always use **Simple Search**; **JSON** groups can switch between **Simple** and **Advanced**.

Simple Search

Choose a mode from the dropdown next to the **Key filter** and **Value filter** fields, then click **Search**:

Mode	What it matches
Starts with	Keys or values that begin with the text you typed. This is the default mode.
Contains	Keys or values that contain the text anywhere.
Exact match	Keys or values that match the text exactly.
Pattern (SQL wildcards)	A pattern you type yourself, using % for any sequence of characters and _ for any single character.

Click **Clear** to reset the filters and show every value again.

Advanced Search (JSON groups)

Click **Advanced** in the search bar, then **Advanced Search...** to open the dialog.

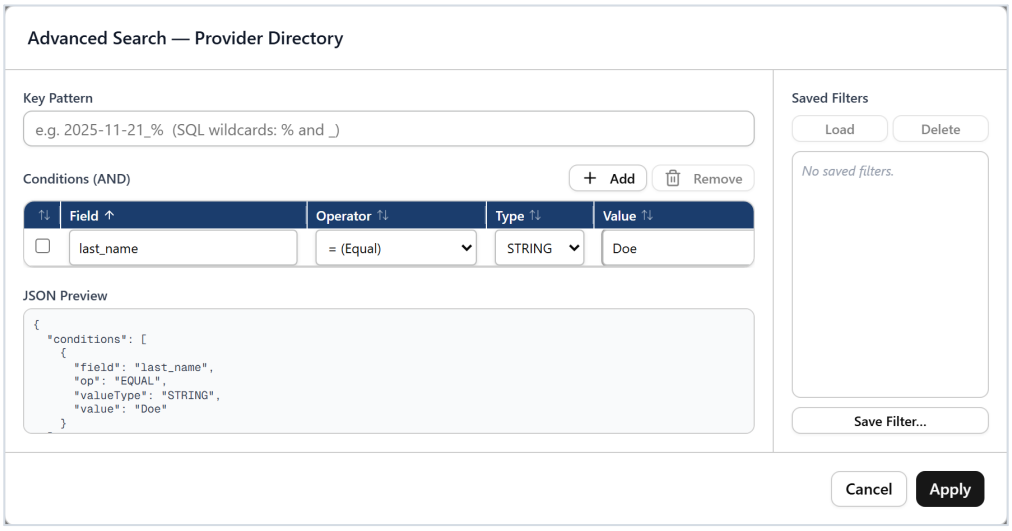

Figure 7: The Advanced Search dialog for a JSON-typed group, showing the key pattern field, the conditions table, and the JSON preview.

1. Optionally type a **Key Pattern**, using the same % and _ wildcards as **Pattern** simple search.
2. Add one or more conditions: click **Add** for a new row, then set its **Field**, **Operator**, **Type**, and **Value**.

3. Click **Apply**.

Conditions combine with AND: a value must match every condition to appear in the results. The available operators depend on the condition's **Type**:

Type	Available operators
STRING	**Equal**, **Not Equal**.
NUMBER	**Equal**, **Not Equal**, **Greater Than**, **Less Than**, **Greater or Equal**, **Less or Equal**.
BOOLEAN	**Equal**, **Not Equal**. The **Value** field becomes a **true** / **false** dropdown rather than free text.

The **Operator** dropdown for a **STRING** field also lists **Contains** and **Not Contains**, but BridgeLink does not currently support either one for any value type: choosing one and running the search returns a validation error. Use **Equal** or **Not Equal** for STRING fields.

Only fields configured with **FIELD** indexing on the group are indexed; a condition against a non-indexed field still works, but scans every value in the group.

Click **Save Filter...** to save the current key pattern and conditions under a name for later reuse. Saved filters are shared across every lookup group, not only the one you saved them from. Select one from the **Saved Filters** list on the right, then click **Load** to reuse it or **Delete** to remove it.

Generating a Channel Script Snippet

Once you have an **Advanced Search** filter you like, click **Snippet** to open the **Channel Script Snippet** dialog.

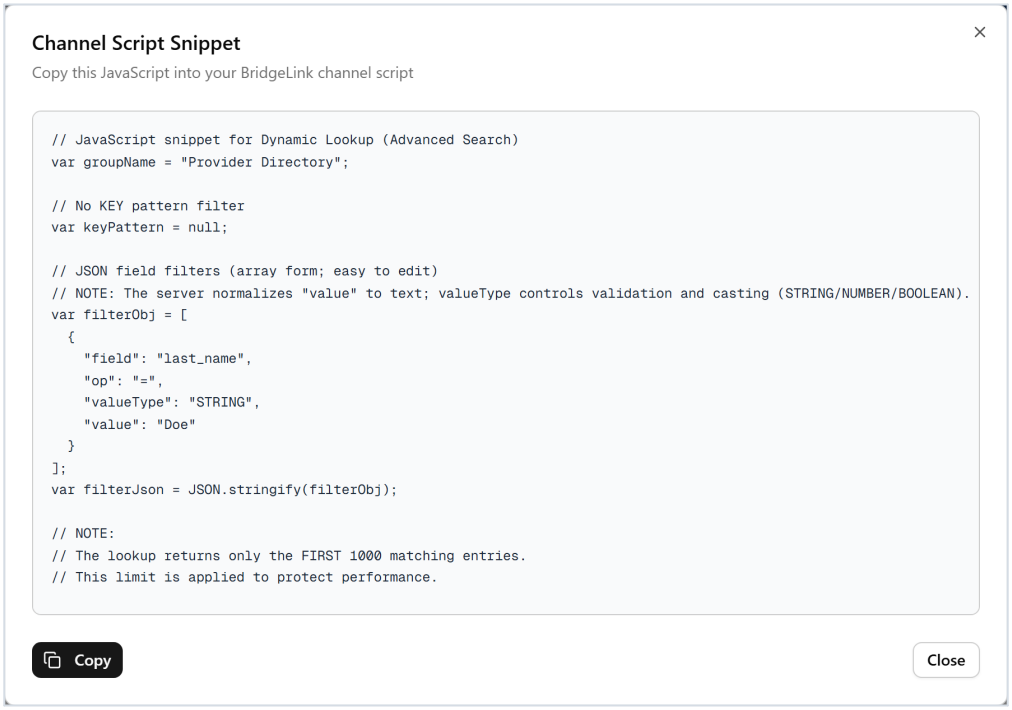

Figure 8: The Channel Script Snippet dialog, showing a generated JavaScript snippet for the current Advanced Search filter.

Click **Copy** to copy the generated JavaScript to your clipboard, then paste it into a transformer or code template. The snippet always calls `LookupHelper.searchValuesByJsonFields()` with the correct argument order, so it is the safest starting point for a script that searches by JSON field. If your filter includes a **Contains** or **Not Contains** condition, the snippet drops it and adds a comment explaining why: a channel script's lookup only supports **Equal**, **Not Equal**, **Greater Than**, **Less Than**, **Greater or Equal**, and **Less or Equal**.

Using Lookups in Channels

All script access to Lookup Manager goes through one global JavaScript namespace, `LookupHelper`, which is available in every JavaScript context BridgeLink runs: filters, transformers, response transformers, code templates, channel scripts (deploy, undeploy, preprocessor, postprocessor), global scripts, JavaScript Reader and JavaScript Writer connectors, and attachment handlers. Every method is also available as a ready-made snippet in BridgeLink's Code Templates editor, under **Lookup Table Functions** (see **Code Template Reference**). The examples below use transformer steps, but the same calls work unchanged in any of these contexts.

Values Are Always Strings

Regardless of a group's value type, `LookupHelper` always stores and returns values as strings. For a **JSON** group, this means a script gets the JSON back as raw text, not a parsed object, and must call `JSON.parse()` before reading any field:

```
// Group "Provider Directory" has Value Type JSON and contains:
//   key = "JDOE", value = {"npi":"1987654321","first_name":"John","last_name":"Doe"}

var raw = LookupHelper.get("Provider Directory", "JDOE");
// raw is the literal string: {"npi":"1987654321","first_name":"John","last_name":"Doe"}

raw.npi          // undefined: raw is a string, not an object
raw["npi"]       // undefined: same reason

var provider = JSON.parse(raw);
provider.npi          // "1987654321"
provider.first_name   // "John"
```

BridgeLink's Rhino-based JavaScript runtime reports `typeof` on a Java string as `"object"`, which can look like the value is already parsed. It is not. Always call `JSON.parse()` on a JSON-typed value before reading its fields.

`searchValuesByJsonFields()` has one more wrinkle: it returns a Java map, not a JavaScript array or object. Iterate it with Java-style methods, and coerce each value with `String(...)` before parsing it:

```
var matches = LookupHelper.searchValuesByJsonFields("Provider Directory", "", filterJson);

if (matches == null || matches.isEmpty()) {
    logger.warn("No matching entries");
    return;
}

var entry = matches.entrySet().iterator().next();
var key   = String(entry.getKey());
var value = JSON.parse(String(entry.getValue()));
```


The LookupHelper Namespace

Read operations:

Method	Returns	Notes
`get(group, key)`	String or `null`	Plain lookup, no TTL.
`get(group, key, defaultValue)`	String	No TTL; returns `defaultValue` if not found.
`get(group, key, ttlHours)`	String or `null`	Returns `null` if the value is older than `ttlHours`.
`get(group, key, ttlHours, ttlMinutes)`	String or `null`	Same, with hours and minutes combined.
`get(group, key, ttlHours, defaultValue)`	String	TTL in hours, with a fallback default.
`get(group, key, ttlHours, ttlMinutes, defaultValue)`	String	TTL in hours and minutes, with a fallback default.
`exists(group, key)`	boolean	
`getMatching(group, keyPattern)`	Map	Keys matching a SQL `LIKE` pattern, up to 1000 entries.
`getMatching(group, keyPattern, limit)`	Map	Same, with a custom limit.
`getMatchingCount(group, keyPattern)`	long	Count only, no values retrieved.
`getBatch(group, keys)`	Map	Multiple keys in one call.
`getBatch(group, keys, ttlHours)`	Map	Same, with an hours-only TTL.
`getBatch(group, keys, ttlHours, ttlMinutes)`	Map	Same, with hours and minutes combined.
`getCacheStats(group)`	Map	Hit and miss counts, hit and miss ratio, eviction count, total lookups, and last accessed time.
`searchValuesByJsonFields(group, keyPattern, filterJson)`	Map	JSON-field search. See **Searching Values**. Up to 1000 entries.

Passing `0` for `ttlHours`, and for `ttlMinutes` where present, disables TTL checking entirely, the same as calling the plain two-argument `get()`. Neither argument can be negative.

`getMatching()` and `searchValuesByJsonFields()` preserve the order values were inserted in. `getBatch()` and `getCacheStats()` do not guarantee any particular order.

BridgeLink's Code Templates panel does not include a ready-made snippet for the hours-only `get(group, key, ttlHours)` and `get(group, key, ttlHours, defaultValue)` forms, or for `getBatch(group, keys, ttlHours)`. All three exist and behave exactly like their

hours-and-minutes equivalents with `ttlMinutes` set to `0`.

Write operations:

Method	Returns	Notes
`set(group, key, value)`	boolean	Inserts or overwrites.
`putIfAbsent(group, key, value)`	boolean	Inserts only if the key does not already exist. Returns `false` if the key exists or an error occurred; the two cases are not distinguished.
`compareAndSwap(group, key, expectedValue, newValue)`	boolean	Updates only if the current value equals `expectedValue`.
`updateValueByDelta(group, key, delta)`	boolean	Adds `delta` to a numeric TEXT value. Not supported for JSON groups, and not supported on the default Derby database.
`deleteValue(group, key)`	boolean	Deletes a single value.
`deleteAllValues(group)`	boolean	Deletes every value in the group.
`importValues(group, values, clearExisting)`	Map	Bulk insert. `values` is a flat map of key to string value. Returns `{ ok: 'true', groupId, importedCount }` on success, or `{ ok: 'false', errorCode, errorMessage }` on failure.
`createGroup(payload)`	Map	Creates a group. `payload` accepts `name`, `description`, `version`, `cacheSize`, `cachePolicy`, `statisticsEnabled`, `valueType`, and, for JSON groups, `jsonIndexMode` and `indexedJsonFields`. Returns `{ ok: 'true', group: {...} }` on success, or `{ ok: 'false', errorCode, errorMessage }` on failure.
`deleteGroup(group)`	boolean	Deletes a group and all of its values.

`ok` in the result of `importValues()` and `createGroup()` arrives as the string `'true'` or `'false'`, not a native boolean. Always check `String(result.ok) === 'true'` before trusting the success path.

Useful Script Patterns

Wrap early-exit guards in an IIFE (an immediately invoked function expression). A top-level `return` works in a source transformer but not in a filter or code template. Wrapping the script body in a function lets every guard clause use `return` safely:

```
(function performLookup() {
    var key = msg["PV1"]["PV1.7"]["PV1.7.1"].toString();
```



```
    if (!key) {
        logger.warn("missing identifier in message");
        return;   // exits the IIFE, not the transformer
    }
    // ...
})();
```

Stop a message BridgeLink cannot map. Calling `destinationSet.removeAll()` stops a message from being delivered while keeping it visible in the message browser for review:

```
if (raw === null) {
    logger.warn("No mapping; dumping message");
    channelMap.put("lookupStatus", "UNMAPPED");
    destinationSet.removeAll();
    return;
}
```

Annotate the channel map. Writing status flags and resolved values to the channel map makes message-by-message troubleshooting in the message browser possible without checking the server log:

```
channelMap.put("lookupStatus", "MAPPED");
channelMap.put("lookupKey", key);
channelMap.put("lookupGroup", "Provider Directory");
```

Iterate repeating segments with E4X. A regular JavaScript loop does not work on a repeating HL7 segment such as `NK1` or `OBX`; it is an XMLList, not an array:

```
for each (var nk1 in msg["NK1"]) {
    // read or modify each NK1 segment
}
```

Example: Translating Codes in Both Directions

A common pattern is translating codes in both directions between two systems: an inbound message's code needs to become the other system's code on the way in, and the other system's code needs to translate back on the way out.

A single group with **Value Type JSON** and **FIELD** indexing handles both directions without duplicating data:

- Group name: `A_to_B_Codes`
- Value Type: `JSON`
- JSON Index Mode: `FIELD`, indexing the field `target_code`
- Keys: System A's native code
- Values: JSON shaped like `{"target_code": "<System B code>", "description": "<label>"}`

Forward direction, when System A's code is the key you already have:

```
var GROUP_NAME = "A_to_B_Codes";
```

```
var raw = LookupHelper.get(GROUP_NAME, sourceCode);

if (raw === null) {
    logger.warn("No mapping for " + sourceCode);
    return;
}

var target = JSON.parse(raw);
// Use target.target_code and target.description in the outbound message.
```

Reverse direction, when System B's code needs to become System A's code, so the group is searched by the indexed field instead of the key:

```
var filter = [{ "field": "target_code", "op": "=", "valueType": "STRING", "value": targetCode }];
var matches = LookupHelper.searchValuesByJsonFields(GROUP_NAME, "", JSON.stringify(filter));

if (matches == null || matches.isEmpty()) {
    logger.warn("No reverse mapping for " + targetCode);
    return;
}

var entry = matches.entrySet().iterator().next();
var sourceCode = String(entry.getKey());
var value = JSON.parse(String(entry.getValue()));
// Use sourceCode and value.description in the outbound message.
```

Both directions hit the FIELD index, so there is one source of truth and updates apply to both directions immediately. Treat the JSON value's shape as a contract between the indexed field name and the search filter, and document it for whoever maintains the group. If more than one upstream system maps into the same downstream system, create one group per upstream system, for example `A_to_B_Codes` and `C_to_B_Codes`, rather than mixing every mapping into one group.

Registering Placeholders for Unmapped Codes

When a script encounters a key with no mapping, register a placeholder entry instead of silently dropping the message, so operations staff can find and resolve it later. Use `putIfAbsent()` so a second occurrence of the same unmapped code does not overwrite an already-registered placeholder:

```
if (raw === null) {
    var placeholderKey = "UNK-" + sourceCode;
    var placeholderValue = JSON.stringify({
        target_code: "UNK",
        description: sourceDesc || "<no description>",
        category: "UNMAPPED"
    });

    var inserted = LookupHelper.putIfAbsent(GROUP_NAME, placeholderKey, placeholderValue);
    if (inserted) {
```



```
        logger.warn("Registered placeholder " + placeholderKey);
    }

    channelMap.put("lookupStatus", "PLACEHOLDER_REGISTERED");
    destinationSet.removeAll();
    return;
}
```

Set an indexed field on the placeholder, such as `target_code` above, to a sentinel value like `"UNK"`, never to the unresolved input itself. Otherwise a later reverse-direction search on that field could accidentally match the placeholder and treat it as a real mapping.

To find and resolve pending placeholders:

1. Open the group and its **Values** tab.
2. Click **Advanced**, then **Advanced Search...**.
3. Add a condition: **Field** `category`, **Operator Equal**, **Type STRING**, **Value** `UNMAPPED`.
4. Click **Save Filter...** so you can reload the same search later, then **Apply**.
5. Each result is a placeholder awaiting review. Edit one to replace its value with the real mapping and rename the key to the real source code, or delete the placeholder and add a new entry with the correct key.

Cache Status and Performance

Cache Configuration

Each group's cache is configured when you create or edit the group:

Setting	What it does	Default
Cache Size	Maximum number of entries held in memory. `0` disables caching for the group.	`1000`
Cache Policy	**LRU** evicts the least recently used entry first. **FIFO** evicts the oldest entry by insertion order first.	`LRU`
Statistics Enabled	Tracks total lookups and cache hits for the group, shown on the **Cache Status** tab's **DB Statistics** card.	`Checked`

TTL, meaning how long a cached or stored value stays valid before a read is considered stale, is not a group setting. It is passed in on each call from a script, using the `ttlHours` and `ttlMinutes` arguments of `LookupHelper.get()` and `getBatch()` (see **Using Lookups in Channels**). Omitting TTL, or passing `0`, always returns the latest available value regardless of age.

Cache Status Tab

Open a group and click its **Cache Status** tab.

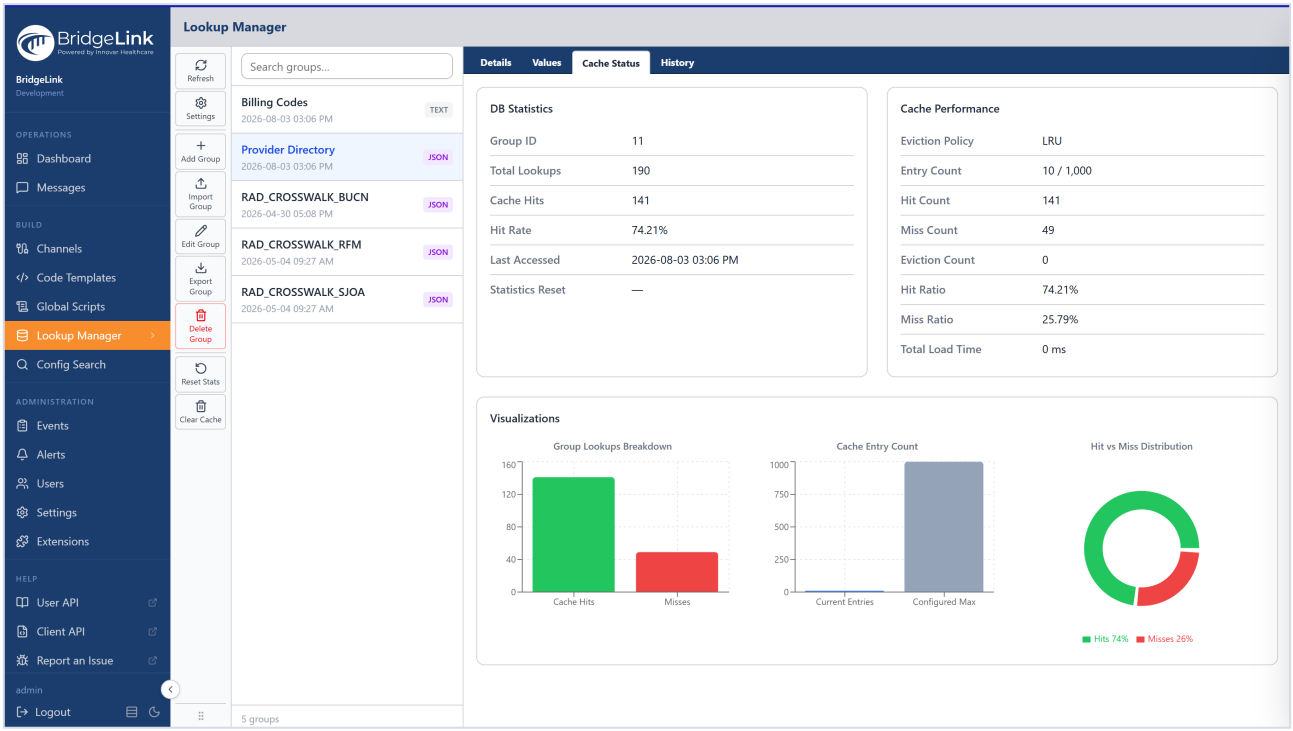

Figure 9: The Cache Status tab, showing the DB Statistics and Cache Performance cards and the lookup and cache charts.

The **DB Statistics** card shows **Total Lookups**, **Cache Hits**, **Hit Rate**, **Last Accessed**, and **Statistics Reset**, the last time these counters were reset.

The **Cache Performance** card shows **Eviction Policy** and **Entry Count** for every group. If the group uses the **LRU** policy, it also shows **Hit Count**, **Miss Count**, **Eviction Count**, **Hit Ratio**, **Miss Ratio**, and **Total Load Time**. A group using the **FIFO** policy does not track these additional counters; the card notes that cache statistics are not supported for that policy.

Below the cards, charts show the group's lookup breakdown (hits versus misses) and, for LRU-cached groups, current entries against the configured maximum, plus a hit-versus-miss distribution.

Use **Reset Stats** to clear the **DB Statistics** counters, and **Clear Cache** to evict every entry currently held in memory. **Clear Cache** is disabled when caching is off for the group. Both ask for confirmation first.

Viewing History

Every change to a group's values, meaning a create, update, delete, delete-all, import, or clear-before-import, is recorded. Open a group and click its **History** tab.

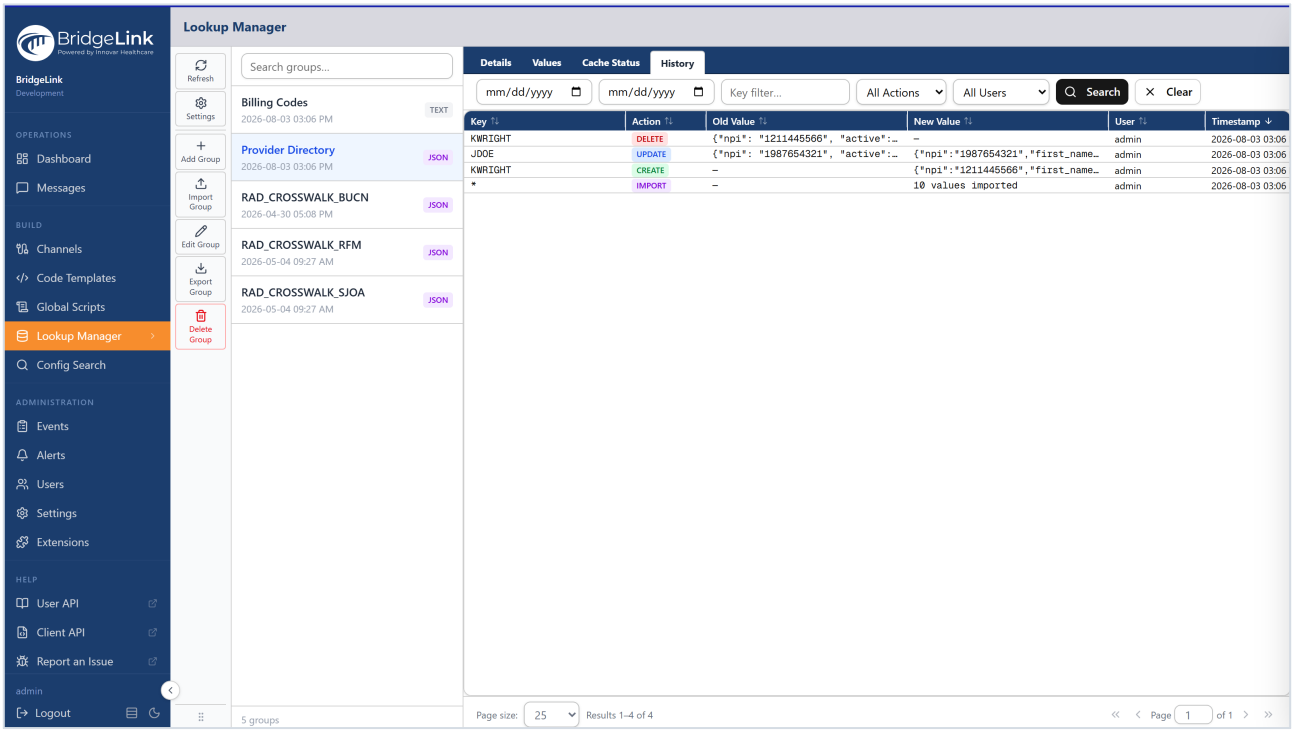

Figure 10: The History tab, showing the filter bar and the audit entry table.

Filter by **Start** and **End** date, a key filter, an **Action** (**CREATE**, **UPDATE**, **DELETE**, **DELETE_ALL**, **IMPORT**, or **CLEAR_ALL**), or a **User**, including **System** for changes made by a script or a scheduled task rather than a signed-in user. Click **Search** to apply the filters, or **Clear** to reset them.

The table shows **Key**, **Action**, **Old Value**, **New Value**, **User**, and **Timestamp** for each entry, newest first by default.

Managing Audit History Retention

By default, **History** entries accumulate indefinitely. To automatically prune old entries:

1. Click **Settings** in the toolbar. The **Lookup Settings** dialog opens.

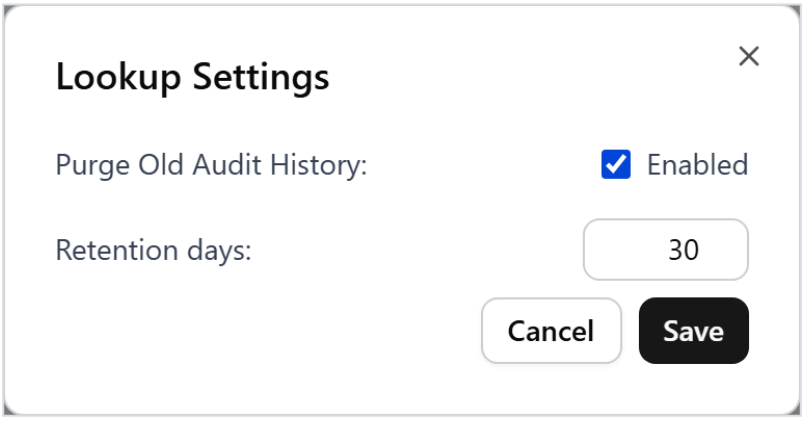

Figure 11: The Lookup Settings dialog, showing the Purge Old Audit History and Retention days controls.

2. Check **Enabled**, next to **Purge Old Audit History**.
3. Set **Retention days**, the number of days of history to keep. Default: `30`.
4. Click **Save**.

BridgeLink checks this setting about once an hour and deletes **History** entries older than the retention period. The setting applies globally, across every lookup group, not per group.

Storage and External Database

By default, Lookup Manager stores every group's values in BridgeLink's own database (Derby, unless your BridgeLink server is configured for another engine), using the same connection BridgeLink itself uses.

To use a separate, external database instead, required for the JSON value type since the default Derby database does not support it, edit the properties file on your BridgeLink server:

`{blDirectory}/conf/dynamic-lookup.properties`

Property	What it does	Default
`useExternalDb`	Set to `true` to use the external database configured below, or `false` to use BridgeLink's own internal database connection.	`false`
`database`	The database engine: `derby`, `mysql`, `postgres`, `oracle`, or `sqlserver`. `derby` does not support the JSON value type.	`postgres`
`database.url`	The JDBC connection URL, for example `jdbc:postgresql://localhost:5432/dynamiclookup`.	`jdbc:postgresql://localhost:5432/<internal-database-name>`
`database.driver`	A JDBC driver class, to override the default for `database` (for example, Microsoft's own SQL Server driver instead of the bundled jTDS driver). Leave blank to use the default.	`(blank)`
`database.username` / `database.password`	Credentials for the external database connection.	`(blank)`
`database.max-connections`	Maximum simultaneous connections in the pool.	`20`
`database.connection.maxretry`	Maximum retries when BridgeLink tries to establish the connection at startup.	`2`

Restart BridgeLink after changing this file for the new settings to take effect.

`database.username` and `database.password` are stored in this file as plain text. Restrict file system access to `dynamic-lookup.properties` the same way you would any other file holding database credentials.

JSON support and minimum database versions:

Database	JSON value type supported
PostgreSQL	Yes, version 9.4 or later
MySQL, including MariaDB	Yes, version 8.0 or later
Microsoft SQL Server	Yes
Oracle	Yes, version 12.2 or later
Derby (the default database)	No

If a group is created with **Value Type JSON** against the default Derby database, the import succeeds, but JSON-specific features, meaning FIELD indexing and `searchValuesByJsonFields()`, are unavailable. Use **TEXT** in that case, or switch to a supported external database first.

There is no screen in WebAdmin for this configuration. It is a server-side file, set up once when you configure your BridgeLink server.

Access Control

In the base product, every signed-in WebAdmin user can use Lookup Manager; there is no built-in permission gating.

On servers using the Advanced Access Control plugin, access to the **Lookup Manager** page and its underlying REST API is controlled by a single permission, assigned to a role the same way as any other section of WebAdmin. A role either has access to Lookup Manager or it does not: there is no separate read-only permission, and no per-group permission. A user with access can view, edit, import, export, and delete every group.

Channel scripts reach the same data through the `LookupHelper` namespace directly on the server, independent of any WebAdmin permission. Anyone who can create, edit, or deploy a channel that calls `LookupHelper` can read and write every lookup group from that channel's script; BridgeLink does not enforce a separate per-group permission for script access. If a lookup group holds sensitive reference data, restrict who can edit and deploy channels accordingly.

Code Template Reference

Every `LookupHelper` method is available as a ready-made snippet in the Code Templates editor's reference panel, under **Lookup Table Functions**. Drag a snippet into your script to insert a working example that already uses the correct namespace, argument order, and error handling.

Snippet name	Calls
Lookup Value by Key	`LookupHelper.get(group, key)`
Lookup Value by Key with TTL	`LookupHelper.get(group, key, ttlHours, ttlMinutes)`
Lookup Value with Default Fallback	`LookupHelper.get(group, key, defaultValue)`
Lookup Value with TTL and Default Fallback	`LookupHelper.get(group, key, ttlHours, ttlMinutes, defaultValue)`
Lookup Values Matching Pattern	`LookupHelper.getMatching(group, keyPattern)`
Lookup Values Matching Pattern (Custom Limit)	`LookupHelper.getMatching(group, keyPattern, limit)`
Lookup Values Count Matching Pattern	`LookupHelper.getMatchingCount(group, keyPattern)`
Batch Lookup by Keys	`LookupHelper.getBatch(group, keys)`
Batch Lookup by Keys with TTL	`LookupHelper.getBatch(group, keys, ttlHours, ttlMinutes)`
Lookup Key Existence in Group	`LookupHelper.exists(group, key)`
Get Lookup Cache Statistics	`LookupHelper.getCacheStats(group)`
Set Lookup Value by Key	`LookupHelper.set(group, key, value)`
Delete Lookup Value by Key	`LookupHelper.deleteValue(group, key)`
Delete All Lookup Values in Group	`LookupHelper.deleteAllValues(group)`
Import Lookup Values	`LookupHelper.importValues(group, payload, clearExisting)`
Put Lookup Value If Absent	`LookupHelper.putIfAbsent(group, key, value)`
Compare and Swap Lookup Value	`LookupHelper.compareAndSwap(group, key, expectedValue, newValue)`
Update Lookup Value by Delta	`LookupHelper.updateValueByDelta(group, key, delta)`
Search lookup values by JSON filter (Advanced)	`LookupHelper.searchValuesByJsonFields(groupName, keyPattern, filterJson)`
Creates a lookup group	`LookupHelper.createGroup(payload)`

Snippet name	Calls
Deletes a lookup group	`LookupHelper.deleteGroup(group)`

See **The LookupHelper Namespace**, above, for the complete set of method signatures, including the hours-only TTL forms that do not have their own snippet.

REST API Reference

Every endpoint lives under `/v1/lookups`. Try any endpoint directly from your BridgeLink server's interactive API documentation, at `https://<your-server>:8443/api`, under the **Lookup Table** section. Each entry has a **Try It Out** button that prefills the request body, runs the call, and shows the resulting `curl` command, request URL, response body, and HTTP status code.

Group Operations

Method and Path	Description
`GET /v1/lookups/groups`	Returns every lookup group.
`POST /v1/lookups/groups`	Creates a new lookup group.
`GET /v1/lookups/groups/{groupId}`	Returns a group by ID.
`PUT /v1/lookups/groups/{groupId}`	Updates a group.
`DELETE /v1/lookups/groups/{groupId}`	Deletes a group.
`GET /v1/lookups/groups/name/{name}`	Returns a group by name.
`GET /v1/lookups/groups/{groupId}/export`	Exports a group's configuration and values.
`GET /v1/lookups/groups/{groupId}/exportPaged`	Exports a group's values in pages, using `offset` and `limit` query parameters, for large groups.
`POST /v1/lookups/groups/import`	Imports a group's configuration and values. If a group with the same name already exists and `updateIfExists=true` is passed, its existing values are deleted and replaced.
`GET /v1/lookups/groups/{groupId}/statistics`	Returns the group's usage and cache statistics.
`POST /v1/lookups/groups/{groupId}/statistics/reset`	Resets the group's usage statistics.
`GET /v1/lookups/groups/{groupId}/audit`	Returns the group's audit entries, paginated.
`POST /v1/lookups/groups/{groupId}/audit/search`	Searches the group's audit entries with filters. The plain `GET /audit` endpoint above ignores filters; use this one to filter by key, action, user, or date range.

Method and Path	Description
`POST /v1/lookups/groups/{groupId}/cache/clear`	Clears the group's cache.
`POST /v1/lookups/cache/clear`	Clears every group's cache.
`GET /v1/lookups/databaseInfo`	Returns the connected database's type and version.

Value Operations

Method and Path	Description
`GET /v1/lookups/groups/{groupId}/values`	Returns a group's values, with optional pagination (`offset`, `limit`) and key filtering (`pattern`).
`POST /v1/lookups/groups/{groupId}/values`	Imports key-value pairs into an existing group. Pass `clearExist=true` to clear the group first.
`GET /v1/lookups/groups/{groupId}/values/{key}`	Returns a single value by key.
`PUT /v1/lookups/groups/{groupId}/values/{key}`	Sets or updates a single value. Request body: `{ "value": "..." }`.
`DELETE /v1/lookups/groups/{groupId}/values/{key}`	Deletes a single value by key.
`POST /v1/lookups/groups/{groupId}/values/batch`	Retrieves multiple values for an array of keys in one request.

To merge values into an existing group without losing what is already there, use `POST /v1/lookups/groups/{groupId}/values` rather than `POST /v1/lookups/groups/import`, which is destructive when a group of the same name already exists.

JSON groups and `POST /v1/lookups/groups/import`: this endpoint accepts an inline `values` map alongside the group configuration, but for a group with **Value Type JSON**, those inline values are not reliably imported. Import the group with an empty `values` map first, then load its values separately with `POST /v1/lookups/groups/{groupId}/values`. This is exactly what WebAdmin's own **Import Group** dialog does.

Import File Format Reference

Lookup Manager accepts two file formats: a CSV file for one group's values, and a JSON file for a complete group. Both are specified here.

CSV Values File

Used by **Import Values** on a group's **Values** tab and by `POST /v1/lookups/groups/{groupId}/values/import`.

- Two columns: `key`, `value`, in that order.
- The first row is treated as a header and skipped, regardless of its text.
- Quote any field containing a comma or a quote character; double any quote characters inside a quoted field (standard CSV quoting).
- One value per row; keys must be unique within the group.

```
key,value
99213,"Office Visit, Level 3"
99214,"Office Visit, Level 4"
```

Export Values produces this same shape, so an exported file can be re-imported without changes.

JSON Group File

The same JSON shape is used by WebAdmin's **Import Group** dialog, **File** source, and by `POST /v1/lookups/groups/import`. An exported group, from **Export Group** or `GET /v1/lookups/groups/{groupId}/export`, can be re-imported without changes.

Top-Level Envelope

```
{
  "group": { },
  "values": { }
}
```

`group` is the group's configuration. `values` is a map of keys to value strings.

The group Object

Field	Type	Required	Notes
`name`	string	Yes	The group's unique name.
`description`	string	No	Free-text description.
`version`	string	Yes	A free-text version label, for example `"1.0.0"`.

Field	Type	Required	Notes
`cacheSize`	integer	Yes	Maximum cached entries. `0` disables caching.
`cachePolicy`	string	Yes	`LRU` or `FIFO`.
`valueType`	string	No	`TEXT` (default) or `JSON`.
`statisticsEnabled`	boolean	No	Defaults to `true`.
`extra`	object	Conditional	Required when `valueType` is `JSON`. See below.

The extra Object (JSON groups only)

Field	Type	Required	Notes
`groupId`	integer	Yes	`Set to 0 on a new import; BridgeLink assigns the real ID.`
`jsonIndexMode`	string	Yes	`NONE or FIELD.`
`indexedJsonFields`	string	Conditional	`Required when jsonIndexMode is FIELD.`

`indexedJsonFields` is a string containing a JSON-encoded array, not a native JSON array. To index the `code` and `category` fields, the literal characters in the file must be:

```
"indexedJsonFields": "[\"code\",\"category\"]"
```

Sending a native array, without the surrounding string quotes, fails with a deserialization error.

The values Map

`values` is always a map of string to string, even for a JSON group. Each value is a JSON-encoded string, with every inner quote backslash-escaped:

```
"values": {
  "JDOE": "{\"npi\":\"1987654321\",\"first_name\":\"John\",\"last_name\":\"Doe\"}"
}
```

Most JSON libraries produce this automatically when you serialize a string that happens to contain JSON. Hand-editing this shape is error-prone.

Complete Example: TEXT Group

```
{
  "group": {
    "name": "Billing Codes",
    "description": "Standard billing codes for claims",
    "version": "1.0",
    "cacheSize": 1000,
    "cachePolicy": "LRU",
```



```
    "valueType": "TEXT",
    "statisticsEnabled": true
  },
  "values": {
    "99213": "Office Visit, Level 3",
    "99214": "Office Visit, Level 4"
  }
}
```

Complete Example: JSON Group with FIELD Indexing

```
{
  "group": {
    "name": "Provider Directory",
    "description": "Internal staff code to provider metadata",
    "version": "1",
    "cacheSize": 1000,
    "cachePolicy": "LRU",
    "valueType": "JSON",
    "statisticsEnabled": true,
    "extra": {
      "groupId": 0,
      "jsonIndexMode": "FIELD",
      "indexedJsonFields": "[\"npi\",\"last_name\"]"
    }
  },
  "values": {
    "JDOE": "{\"npi\":\"1987654321\",\"first_name\":\"John\",\"last_name\":\"Doe\"}",
    "MSMITH": "{\"npi\":\"1122334455\",\"first_name\":\"Mary\",\"last_name\":\"Smith\"}"
  }
}
```


Troubleshooting

Symptom	Likely cause	What to do
No **Lookup Manager** item appears under **Build**.	On servers using the Advanced Access Control plugin, your role does not have the **Lookup Manager** permission; otherwise, the underlying extension is not enabled.	Ask your administrator to grant the permission, or confirm **Lookup Table Management System** shows **Enabled** on the **Settings > Extensions** page.
`ReferenceError: LookupTable is not defined`	An older script used a `LookupTable` namespace, which does not exist.	Use `LookupHelper` instead. Drag a snippet from the Code Templates panel to avoid this entirely.
A JSON value's fields read as `undefined`.	The script read a field directly from the raw value, without calling `JSON.parse()` first.	Always call `JSON.parse()` on the return value of `get()` before reading a field. For `searchValuesByJsonFields()`, also coerce the value with `String(...)` first.
Importing a JSON file reports success, but no values appear.	The import replaces a group's existing values entirely; a stale WebAdmin view, or an empty or misplaced `values` map in the file, are the most common causes.	Reselect the group to refresh the view. Confirm `values` is non-empty and sits at the top level of the file, not nested inside `group`.
`putIfAbsent()` returns `false`, but the key does not seem to exist.	`putIfAbsent()` returns `false` both when the key already exists and when an internal error occurred; it does not distinguish the two.	Check the BridgeLink server log for an underlying error around the time of the call.
Creating a group with **Value Type JSON** fails, or JSON-specific features are unavailable.	BridgeLink is using the default Derby database, which does not support the JSON value type.	Configure an external database that supports it. See **Storage and External Database**.
An **Advanced Search** or channel-script JSON filter using **Contains** or **Not Contains** fails with a validation error.	Neither operator is currently supported, for any value type.	Use **Equal** or **Not Equal** instead.
A placeholder entry keeps matching real searches by mistake.	An indexed field on the placeholder's value, for example `target_code`, was set to the real, unresolved code instead of a sentinel like `"UNK"`.	Always set indexed fields on a placeholder to a sentinel value, never to the unresolved input.

Symptom	Likely cause	What to do
Cache Status shows no **Hit Count**, **Miss Count**, or **Eviction Count** for a group.	The group uses the **FIFO** cache policy, which does not track these counters, or **Cache Size** is `0`.	Switch to **LRU** if you need these counters. **Cache Policy** can be changed at any time; **Value Type** cannot.

Support

Customers with an active Innovar Healthcare support plan can contact support at `support@innovarhealthcare.com` for help with Lookup Manager. Include:

- Your BridgeLink version
- The affected group's name, value type, and, if relevant, JSON index mode
- The exact error message or `errorCode`, if one appeared
- Relevant entries from the group's **History** tab, or from the BridgeLink server log

For help designing a lookup schema for a large integration project, such as a multi-system crosswalk or a bulk migration from an existing reference table, Innovar Healthcare's professional services team is also available.