# n8n Automation Workflows

Documentation for all n8n automation workflows running at Puffing Billy Railway, including integration details, data flows, credentials, and maintenance notes.

# Overview and Workflow Index

## Overview

This book documents all n8n automation workflows currently deployed at Puffing Billy Railway (PBR). n8n is PBR's self-hosted workflow automation platform, running at https://n8n.pbr.org.au. It integrates internal systems, third-party APIs, and AI services to automate routine IT and operational processes.

| Field | Value |
|---|---|
| Platform | n8n (self-hosted, Docker) |
| URL | https://n8n.pbr.org.au |
| Owner | IT Team — Mitch (IT Manager) |
| Status | Active |

---

## Workflow Index

| Workflow | Status | Trigger | Purpose |
|---|---|---|---|
| BetterImpact > Swift User Ingest | Active | Scheduled — every 24 hours | Syncs people records from BetterImpact into Swift Digital contact groups by engagement type |
| Jitbit External Tool — search_bookstack | Active | Webhook (Jitbit AI External Tool) | Searches BookStack documentation on behalf of the Jitbit AI assistant |
| Jitbit Auto-Triage — Type Field + BookStack + Tech Note | Active | Webhook (Jitbit automation rule) | Classifies new helpdesk tickets by ITIL type, sets a custom field, and posts a private tech note with BookStack references |

| Workflow | Status | Trigger | Purpose |
|---|---|---|---|
| Delete All Swift Users From Group | Inactive — manual run only | Manual trigger | Utility workflow to bulk-remove all contacts from a specified Swift Digital group |

---

# Credentials Overview

The following credential types are used across these workflows. All credentials are stored in n8n and referenced by name — no secrets are stored in workflow parameters.

| Credential Name | Type | Used By |
|---|---|---|
| Anthropic API Key | HTTP Header Auth (`x-api-key`) | Jitbit Auto-Triage |
| Jitbit API Token | HTTP Bearer Auth | Jitbit Auto-Triage, search_bookstack |
| BookStack Token | HTTP Header Auth (`Authorization: Token id:secret`) | Jitbit Auto-Triage, search_bookstack |
| BetterImpact API | HTTP Basic Auth | BetterImpact > Swift User Ingest |
| Swift Digital OAuth2 | OAuth2 | BetterImpact > Swift User Ingest, Delete All Swift Users |

---

# Maintenance Notes

- When adding new workflows, add a row to the Workflow Index table above and create a dedicated page in this book.
- Credential tokens should be rotated in n8n under **Settings > Credentials** when staff change or tokens are revoked.
- The Anthropic API key used for Jitbit Auto-Triage currently uses the `claude-haiku-4-5-20251001` model. Check Anthropic model deprecation notices periodically and update the workflow if the model string changes.
- n8n is running in Docker — refer to the n8n server documentation for upgrade and backup procedures.

# BetterImpact > Swift User Ingest

## Overview

| Field | Value |
|---|---|
| Workflow ID | DmoxddEkpx854rYj |
| n8n URL | https://n8n.pbr.org.au/workflow/DmoxddEkpx854rYj |
| Status | Active |
| Trigger | Schedule — every 24 hours |
| Systems | BetterImpact (source), Swift Digital (destination), n8n Data Table (audit log) |
| Predecessor | BetterImpact>Swift User Ingest_old (inactive, superseded) |

---

## Purpose

This workflow keeps Swift Digital contact groups in sync with BetterImpact, PBR's people management system. It runs once every 24 hours and processes all BetterImpact users whose records have been updated in the past 48 hours.

BetterImpact is treated as the source of truth. For each recently changed person, the workflow determines their engagement type (Volunteer, Staff, Board/Sub-Committee, or Guild Member) and ensures their Swift Digital contact record and group memberships reflect that classification. If a person no longer qualifies for a group they are in, they are removed. If they are not yet in a group they should be in, they are added. If they do not exist in Swift Digital at all, a new contact record is created.

---

## Data Sources

# BetterImpact API

Authenticates via HTTP Basic Auth. Two separate API calls are made at the start of each run:

- **GetAllBIUsers Updated Last 48Hrs** — fetches all users from the *PBR - Whole Team* organisation updated in the last 48 hours, paginated at 100 records per page.
- **GetAllGuildBIUsers Updated last 48Hrs** — fetches all users from the *PBR - GUILD* organisation updated in the last 72 hours (slightly wider window), paginated at 100 records per page.

Both result sets are merged and aggregated before processing. The wider 72-hour window for Guild users is intentional, accounting for the fact that Guild membership changes may be recorded slightly later in BetterImpact.

# Fields Extracted from BetterImpact

For each user, the following fields are extracted and normalised:

| Field | Source in BetterImpact | Notes |
|---|---|---|
| `BI_user_id` | `user_id` | BetterImpact's unique user identifier |
| `first_name`, `last_name`, `title` | Top-level fields | |
| `email` | `email_address` | Parsed with `extractEmail()` to strip display-name formatting |
| `mobile` | `cell_phone` | Spaces removed |
| `volunteer_status` | `memberships[0].volunteer_status` | Used to determine active vs archived status. Value must be `Accepted` to be considered active. |
| `engagement_type` | Custom field category *PBR VOLUNTEER TYPE* | Determines which Swift group the person belongs to |
| `org_name` | `memberships[0].organization_name` | Used to distinguish Guild users in the GUILD org from those appearing in Whole Team |
| `branch`, `sub_branch` | Custom fields *Branch* and *Sub-Branch* | Ampersands (`&`) are replaced with *and* before sending to Swift Digital, which does not accept the `&` character in these fields |
| `country` | Looked up via n8n Data Table mapping from `country_name` | Converted to ISO country code(s) for Swift Digital compatibility |

Users without an email address are filtered out early and never sent to Swift Digital.

---

# Engagement Type Classification

Each person is routed to one of four Swift Digital groups based on their `engagement_type` value from BetterImpact:

| Engagement Type | Swift Digital Group | Match Logic |
|---|---|---|
| Contains *Volunteer* | PBR_Vols | String contains check |
| Contains *Staff* | PBR_Staff | String contains check |
| Equals *Board Member* or *Sub-Committee Member* | PBR_BoardSubcommittees | Exact match (OR) |
| Equals *PBR Guild Member* AND org is *PBR - GUILD* | PBR_Guild (AllGuild) | Both conditions must match |

Each person is evaluated against all four type checks independently in parallel. A person could theoretically match more than one (e.g. if their engagement type changes mid-run), though in practice a person should match only one.

---

# Processing Logic — Per User

Once a user's engagement type is determined, the workflow follows this decision tree for each type. The logic is identical across all four types — described here using Volunteer as the example:

## Active User Path (volunteer_status = Accepted)

1. **Does the user exist in Swift Digital?** (checked via `contact_ids` — a sentinel value of `zzzz69363c11d9fa7821` indicates no match was found from the email search)
   - **Yes — user exists:** Update their contact details in Swift Digital (name, email, mobile, country, engagement type, BetterImpact user ID). Then check whether they are already a member of the correct group.
     - If **not in the group**: add them to it.
     - If **already in the group**: no group action needed.
   - **No — user does not exist:** Create a new Swift Digital contact record with all their details, and assign them to the correct group in the same API call.

# Inactive/Archived User Path (volunteer_status != Accepted)

1. **Does the user exist in Swift Digital?**
   - **No — user does not exist:** Nothing to do, exit.
   - **Yes — user exists:** Check whether they are a member of the group they should no longer be in.
     - **Not in the group:** Nothing to do, exit.
     - **In the group — and it is their only group:** Delete the contact from Swift Digital entirely.
     - **In the group — and they are in other groups too:** Remove them from this specific group only (do not delete the contact).

The "last group" check (delete vs group-remove) is done by checking whether `group_ids.length <= 1`. If a contact is only in one group and should be removed from it, deleting the contact entirely is the correct action.

---

# Guild User Edge Case

Guild members appear in two BetterImpact API results: the *PBR - Whole Team* response and the *PBR - GUILD* response. To prevent a Guild member from being incorrectly processed as a Whole Team volunteer, the workflow uses a deduplication merge that prefers the *PBR - Whole Team* record when a user appears in both. The `Is Guild?` check then additionally requires that `org_name = PBR - GUILD`, ensuring only genuine Guild organisation records are routed to the Guild group.

There is also a specific filter called *Catch Guild User Archived in BI Whole Team*. This handles a known data pattern in BetterImpact where a Guild member who has been archived in the GUILD organisation still appears in the Whole Team result. This filter passes the user through to the Guild group removal logic, preventing them from being incorrectly left in the Swift Digital Guild group after archival.

---

# Duplicate Contact Detection

After main processing, the workflow runs a secondary check for duplicate Swift Digital contacts. For each processed user, it searches Swift Digital by BetterImpact User ID (`Internal_BIUserId`) to see if more than one contact record is associated with that ID. If duplicates are found, the workflow:

1. Retrieves full details for all duplicate contacts.
2. Identifies the earliest-created contact by `create_stamp`.
3. Deletes the earliest contact, keeping the most recently created one.

The premise is that BetterImpact is the source of truth — if a single BI user ID maps to multiple Swift contacts, the oldest is assumed to be stale and the newest is retained.

---

# n8n Data Table (Audit Log)

After processing, each user record is upserted into an n8n internal Data Table named **BI_Users** (project ID: `JMezsOufXlA5w9MB`, table ID: `ps5tTQfbPJ8IMtfQ`). This provides an internal record of the last-known state of each processed user including their Swift contact ID, group IDs, engagement type, and email. The upsert matches on `BI_user_id`.

This table is used as a debugging and audit reference — it is not consumed by any downstream automated workflow.

---

# Custom Fields Written to Swift Digital

When creating or updating a contact in Swift Digital, the following custom internal fields are populated:

| Swift Field | Value |
|---|---|
| `Internal_EngagementType` | The person's engagement type from BetterImpact |
| `Internal_BIUserid` | The person's numeric BetterImpact user ID |
| `Internal_Branch` | Branch (omitted if null) |
| `Internal_SubBranch` | Sub-branch (omitted if null) |

Note: Branch and Sub-Branch fields were present in an earlier version of the workflow but are not written in the current active version's update/create payloads. They remain in the BetterImpact data extraction step.

---

# API Rate Limiting

Swift Digital API calls are batched at 100 requests per batch with a 7,500ms interval between batches for search and group-read operations. Contact creation for Volunteers uses a 5,000ms interval. This prevents hitting Swift Digital API rate limits during large sync runs.

---

# Credentials

| Credential | Type | Used For |
|---|---|---|
| BetterImpact API | HTTP Basic Auth | All BetterImpact API calls |
| Swift Digital OAuth2 | OAuth2 | All Swift Digital API calls |

---

# Maintenance Notes

- **Schedule:** Runs every 24 hours. The 48-hour lookback window means a missed run will still catch changes from the previous cycle on the next run.
- **BetterImpact engagement type values:** If PBR adds new engagement types in BetterImpact, the *Is Vol?*, *Is Staff?*, *Is Board Or Sub-Committee?*, and *Is Guild?* nodes must be updated to include the new values, and a corresponding Swift group and routing branch must be added.
- **Swift Digital group IDs** are stored in the n8n Global Constants node (not hardcoded in individual nodes). If group IDs change in Swift Digital, update the Global Constants node only.
- **Country code mapping** is maintained in the n8n Data Table referenced by ID `wRiJ3eMdq66p8Efh`. If new countries appear in BetterImpact, add a mapping row to that table.
- **The sentinel value** `zzzz69363c11d9fa7821` is used as a placeholder to indicate no Swift contact was found for a given email. This is an internal n8n pattern — do not remove it from the contact_ids field checks.
- **The _old workflow** (`0FJoBOuSCoGadbOY`) is inactive and retained for reference only. Do not activate it.

# Jitbit External Tool — search_bookstack

## Overview

| Field | Value |
|---|---|
| Workflow ID | qLw7S1Rr0eznKDhi |
| n8n URL | https://n8n.pbr.org.au/workflow/qLw7S1Rr0eznKDhi |
| Status | Active |
| Trigger | Webhook — called by Jitbit AI as a registered External Tool |
| Webhook URL | `https://n8n.pbr.org.au/webhook/jitbit-search-bookstack` |
| Systems | Jitbit Helpdesk (caller), BookStack (search target) |

---

## Purpose

This workflow acts as a bridge between the Jitbit AI assistant and PBR's internal BookStack documentation wiki. It is registered in Jitbit as an **External Tool** named `search_bookstack`. When the Jitbit AI is answering a helpdesk ticket and determines that relevant internal documentation may exist, it automatically calls this tool, passing search keywords extracted from the ticket. The workflow queries BookStack and returns a formatted list of matching pages — titles, types, URLs, and content previews — which the AI incorporates into its response to the technician.

---

## How It Works

1. **Jitbit AI calls the webhook** via HTTP POST with a JSON body containing a `query` parameter — search keywords extracted by the AI from the ticket subject and body.
2. **Search BookStack** — the workflow calls `GET /api/search` on BookStack with the query string, requesting up to 8 results. Authentication uses the BookStack API token.

3. **Build response** — a Set node formats the results into a plain-text string. Each result includes: name, content type (page, chapter, or book), full URL, and a content preview snippet (up to 200 characters, HTML tags stripped).
4. **Respond to Jitbit** — the workflow returns a JSON response to the Jitbit AI containing the formatted `result` string and a `total_found` count.

---

# Jitbit External Tool Configuration

This workflow is registered in Jitbit under **Admin > AI Features > External Tools** with the following settings:

| Field | Value |
|---|---|
| Name | `search_bookstack` |
| URL | `https://n8n.pbr.org.au/webhook/jitbit-search-bookstack` |
| Description | Search PBR's internal IT documentation wiki (BookStack) for relevant articles, configuration guides, troubleshooting procedures, and technical documentation. Use this when a ticket involves a known system, technology, or procedure that may be documented internally. |

# Parameters

| Name | Description | Required |
|---|---|---|
| `query` | Search terms extracted from the ticket — keywords describing the system or issue (e.g. "Proxmox iSCSI timeout" or "PA-440 IPsec VPN") | Yes |

---

# Response Format

The workflow returns a JSON object to the Jitbit AI:

```
{
  "result": "Page Title (page): https://bookstack.pbr.org.au/books/book-slug/page/page-slug
  Preview: content snippet...
```

```
Another Page (book): https://...",
  "total_found": 29
}
```

The `result` field is a formatted plain-text string the AI can read directly. `total_found` is the total number of matching results in BookStack (not just the 8 returned).

---

# Nodes

| Node | Type | Purpose |
|---|---|---|
| Receive from Jitbit AI | Webhook (POST, responseMode: responseNode) | Entry point — receives the search query from Jitbit AI |
| Search BookStack | HTTP Request (GET) | Calls `https://bookstack.pbr.org.au/api/search` with the query and count=8 |
| Build Jitbit Response | Set | Formats the BookStack results array into a plain-text string; strips HTML tags from preview_html.content; falls back to "No preview available" if content is empty |
| Respond to Jitbit | Respond to Webhook | Returns the JSON result to the Jitbit AI caller |

---

# Credentials

| Credential | Type | Used For |
|---|---|---|
| BookStack Token | HTTP Header Auth (`Authorization: Token id:secret`) | BookStack search API |

---

# Maintenance Notes

- The BookStack API token is stored as an HTTP Header Auth credential in n8n. Regenerate it at `https://bookstack.pbr.org.au` under the Claude_AI user profile if access is revoked.
- The result count is capped at 8 to keep AI responses focused. This can be adjusted in the Search BookStack node query parameter `count` if broader results are needed.

- BookStack search uses keyword matching. If searches return poor results, the issue is likely in the quality of keywords the Jitbit AI is passing — this is controlled by Jitbit's AI system prompt, not this workflow.
- Books and chapters without page-level content return "No preview available" in the result — this is expected behaviour, as BookStack only generates preview snippets for pages.

# Jitbit Auto-Triage — Type Field + BookStack + Tech Note

## Overview

| Field | Value |
|---|---|
| Workflow ID | llP1pezJvYAGKjYA |
| n8n URL | https://n8n.pbr.org.au/workflow/llP1pezJvYAGKjYA |
| Status | Active |
| Trigger | Webhook — called by a Jitbit automation rule on every new ticket |
| Webhook URL | `https://n8n.pbr.org.au/webhook/jitbit-ticket-triage` |
| Systems | Jitbit Helpdesk (caller), Anthropic Claude API (classifier), BookStack (documentation search), Jitbit API (write-back) |
| AI Model | claude-haiku-4-5-20251001 |

---

## Purpose

This workflow automatically triages every new Jitbit helpdesk ticket. When a ticket is created, Jitbit fires an automation rule that POSTs the ticket details to this n8n webhook. The workflow then:

1. Uses Claude Haiku (Anthropic API) to classify the ticket as an ITIL type and extract a triage summary and BookStack search keywords.
2. Sets the ticket's **Type** custom field in Jitbit to the classified value.
3. Searches BookStack for relevant internal documentation using the AI-extracted keywords.
4. Posts a private tech-only comment to the ticket containing the ITIL type, a triage summary, and links to relevant BookStack pages.

This gives attending technicians an immediate structured summary and relevant documentation links before they even open the ticket.

---

# Jitbit Automation Rule

The workflow is triggered by a Jitbit automation rule (not a Jitbit AI External Tool — the trigger is a direct HTTP call, not routed through the Jitbit AI assistant). The rule is configured as:

| Setting | Value |
|---|---|
| Trigger | Ticket is created |
| Action | Send HTTP request |
| Method | POST |
| URL | `https://n8n.pbr.org.au/webhook/jitbit-ticket-triage` |
| Post Data | `ticket_id=#ticketId#&subject=#subject#&body=#body#&category=#category#&tags=#tags#` |

Jitbit substitutes the `#ticketId#`, `#subject#`, `#body#`, `#category#`, and `#tags#` tokens with the actual ticket values before sending the POST request.

---

# How It Works

## Step 1 — Receive webhook from Jitbit

The webhook node receives the POST body from Jitbit containing `ticket_id`, `subject`, `body`, `category`, and `tags`.

## Step 2 — Classify with Claude

The workflow POSTs to the Anthropic Messages API (`https://api.anthropic.com/v1/messages`) using the `claude-haiku-4-5-20251001` model. The system prompt instructs Claude to act as an ITIL-aligned IT helpdesk triage assistant for PBR and respond with raw JSON only (no markdown). The user message contains the ticket subject, category, tags, and body.

Claude returns a JSON object with three fields:

| Field | Description |
|---|---|
| `type` | ITIL classification — one of: Incident, Problem, Service Request, Change Request, Event |
| `bookstack_query` | 3-6 keyword search terms for finding relevant internal documentation |
| `triage_summary` | 2-3 sentence plain-English summary of the issue for the attending technician |

The Parse Classification node strips any markdown code fences from the response (Claude Haiku occasionally wraps JSON in backticks despite instructions) before parsing the JSON.

# Step 3 — Parallel branches

After parsing, two branches run in parallel:

- **Branch A — Set Type Custom Field:** POSTs to `https://helpdesk.pbr.org.au/api/SetCustomField` with `ticketId`, `fieldId=1` (the Type field), and the option ID corresponding to the classified type.
- **Branch B — Search BookStack:** Calls `https://bookstack.pbr.org.au/api/search` with the AI-extracted keywords, returning up to 5 results.

Both branches feed into a Merge node that waits for both to complete before continuing.

# Step 4 — Build and post tech note

The Build Tech Note node constructs a private comment body combining the ITIL type, triage summary, and formatted BookStack links with content previews. The `ticket_id` and triage fields are read directly from the Parse Classification node output (not from the merge) to ensure they are never overwritten by the empty response body from the SetCustomField API call.

The comment is posted to Jitbit via `POST /api/comment` with `forTechsOnly=true`, making it visible only to technicians.

# Step 5 — Respond

The workflow returns a JSON confirmation to Jitbit: `{ "result": "Triage complete. Type set to: X. Tech note posted to ticket." }`.

---

# ITIL Type to Custom Field Option ID Mapping

The Jitbit Type custom field (Field ID: 1) uses dropdown option IDs. The mapping is hardcoded in the Parse Classification node:

| ITIL Type | Option ID | Definition |
|---|---|---|
| Incident | 1 | Unplanned interruption to service (e.g. outage, crash, failure, offline, broken) |
| Problem | 28 | Underlying root cause investigation of one or more recurring incidents |
| Service Request | 3 | Routine pre-approved request (e.g. password reset, new laptop, software install) |
| Change Request | 4 | Planned alteration, addition, or removal of IT systems |
| Event | 29 | Automated monitoring alert (e.g. server monitoring trigger) |

If Claude returns an unrecognised type value, the option ID defaults to 1 (Incident).

---

# Example Tech Note Output

```
□□AI Triage

Type: Incident

Summary: The network printer at Belgrave station has been offline since 8am, preventing
staff from printing boarding passes. Immediate investigation of printer connectivity
and network status is required.

□□Relevant Documentation:
• Fault Finding - Belgrave Ticket Printers (page)
  https://bookstack.pbr.org.au/books/printers/page/fault-finding-belgrave-ticket-printers
  Issues with any of the Zebra ZD421 Belgrave Ticket printers where it is not possible to
```

```
ge...

• Printer Setup Guide (page)
  https://bookstack.pbr.org.au/books/endpoint-devices/page/printer-setup
  Steps to configure network printers at PBR sites...
```

# Nodes

| Node | Type | Purpose |
|---|---|---|
| Receive from Jitbit AI | Webhook (POST, responseMode: responseNode) | Receives ticket data from Jitbit automation rule |
| Classify with Claude | HTTP Request (POST) | Calls Anthropic API with ticket content; returns ITIL type, triage summary, and search query |
| Parse Classification | Set | Strips markdown fences; parses JSON; maps type string to option ID; extracts ticket_id and subject from webhook input |
| Set Type Custom Field | HTTP Request (POST) | Calls Jitbit `/api/SetCustomField` to set Field ID 1 to the classified type option ID |
| Search BookStack | HTTP Request (GET) | Searches BookStack with AI-extracted keywords; returns up to 5 results |
| Combine Triage + Docs | Merge (combineByPosition) | Waits for both parallel branches to complete |
| Build Tech Note | Set | Assembles the private comment body; reads triage fields from Parse Classification node directly to avoid merge overwrite |
| Post Tech Note to Jitbit | HTTP Request (POST) | Posts private tech-only comment to the ticket via Jitbit `/api/comment` |
| Respond to Jitbit | Respond to Webhook | Returns confirmation JSON to Jitbit |

# Credentials

| Credential | Type | Used For |
|---|---|---|
| Anthropic API Key | HTTP Header Auth (`x-api-key`) | Claude Haiku API calls |

| Credential | Type | Used For |
|---|---|---|
| Jitbit API Token | HTTP Bearer Auth | SetCustomField and comment POST calls to Jitbit |
| BookStack Token | HTTP Header Auth (`Authorization: Token id:secret`) | BookStack search API |

---

# Known Quirks

- **Claude Haiku and markdown fences:** Despite the system prompt instructing raw JSON output, Claude Haiku 4.5 occasionally wraps its response in markdown code fences (). The Parse Classification node strips these defensively before parsing.
- **Merge node field overwrite:** The SetCustomField API returns an empty response body ( `{}`), which causes a `combineByPosition` merge to overwrite the triage fields from the other branch. This is worked around by having the Build Tech Note node reference the Parse Classification node directly via `.item.json.*` rather than relying on  from the merge output.
- **UpdateTicket does not support custom fields:** The Jitbit `/api/UpdateTicket` endpoint does not accept custom field parameters. The dedicated `/api/SetCustomField` endpoint must be used instead.

---

# Maintenance Notes

- **AI model:** The workflow uses `claude-haiku-4-5-20251001`. Check Anthropic's model deprecation schedule periodically. To update the model, edit the `jsonBody` parameter in the Classify with Claude node.
- **Type option IDs:** If the Jitbit Type custom field options are changed (added, renamed, or reordered), update the option ID mapping object in the Parse Classification node. Current IDs can be verified via `GET https://helpdesk.pbr.org.au/api/customfields`.
- **Jitbit API token:** The token is stored as an HTTP Bearer Auth credential. Regenerate at `https://helpdesk.pbr.org.au/User/Token` if the token expires or is revoked.
- **Anthropic API key:** Stored as HTTP Header Auth with name `x-api-key`. Update in n8n credentials if the key is rotated.
- **BookStack credential:** Uses a `Token id:secret` format. Regenerate under the Claude_AI user in BookStack admin if access is revoked.

# Delete All Swift Users From Group

## Overview

| Field | Value |
|---|---|
| Workflow ID | l0KQdZd8IGiJNuLa |
| n8n URL | https://n8n.pbr.org.au/workflow/l0KQdZd8IGiJNuLa |
| Status | Inactive — manual execution only |
| Trigger | Manual (Execute Workflow button in n8n) |
| Systems | Swift Digital |

---

## Purpose

This is a utility workflow used to bulk-delete all contacts from a specified Swift Digital contact group. It is not scheduled and must be manually executed. Before running, the target group ID must be set in the n8n Global Constants node.

**Warning:** This workflow calls the Swift Digital contact DELETE API, which permanently removes the contacts from Swift Digital entirely — it does not merely remove them from the group. Use with care.

---

## How It Works

1. **Manual trigger** — workflow is started manually from within n8n.
2. **Global Constants** — reads the target group ID(s) from the n8n Global Constants node. The `constants` field is expected to contain one or more group IDs.
3. **Split Out group IDs** — if multiple group IDs are present in constants, they are split into individual items for iteration.

4. **Get Users in Group** — calls `GET https://v3.api.swiftdigital.com.au/request/mailhouse/mailgroup/readmembers` with the group ID to retrieve all contact IDs in the group.
5. **Split Out contact IDs** — splits the returned array of contact IDs into individual items.
6. **Remove Users from Swift** — calls `DELETE https://v3.api.swiftdigital.com.au/request/mailhouse/contact/delete` for each contact ID, permanently deleting them from Swift Digital.

---

# Usage Instructions

1. Open the workflow in n8n: https://n8n.pbr.org.au/workflow/l0KQdZd8IGiJNuLa
2. Open the **Global Constants** node and set the `constants` value to the Swift Digital group ID(s) you want to clear.
3. Save the workflow.
4. Click **Execute Workflow**.
5. Monitor the execution to confirm all contacts were removed.

Do not activate (publish) this workflow. It should always remain inactive and only be run on demand.

---

# Nodes

| Node | Type | Purpose |
|---|---|---|
| When clicking Execute workflow | Manual Trigger | Entry point — started manually only |
| Global Constants | Global Constants | Provides the target group ID(s) |
| Split Out1 | Split Out (`constants`) | Iterates over group IDs if multiple are configured |
| Get Users in Group | HTTP Request (GET) | Retrieves all contact IDs in the target group from Swift Digital |
| Split Out | Split Out (`contact_ids`) | Splits the contact ID array into individual items |
| Remove Users from Swift | HTTP Request (DELETE) | Permanently deletes each contact from Swift Digital |

---

# Credentials

| Credential | Type | Used For |
| --- | --- | --- |
| Swift Digital OAuth2 | OAuth2 | All Swift Digital API calls |

---

# Maintenance Notes

- This workflow must remain **inactive** at all times. Do not publish it.
- The target group ID must be manually set in Global Constants before each run — it is not persisted between executions.
- This workflow deletes contacts from Swift Digital entirely, not just from a group. Ensure this is the intended behaviour before running. If you only need to remove contacts from a group without deleting them, the Swift Digital API endpoint `/request/mailhouse/mailgroup/removemembers` should be used instead — this would require a workflow modification.
- The BetterImpact > Swift User Ingest workflow will re-create contacts on its next run if the deleted users still exist as active members in BetterImpact. This workflow is therefore typically used to reset a group before a clean resync.

# CustomLinc TOL → Queue-it Availability JSON

## Overview

| | |
|---|---|
| **Workflow ID** | T76MXW3QRGcoIBbo |
| **Webhook URL** | https://n8n.pbr.org.au/webhook/availability/tol |
| **n8n editor** | https://n8n.pbr.org.au/workflow/T76MXW3QRGcoIBbo |
| **Trigger** | HTTP GET/POST webhook |
| **Season** | 2026-06-26 → 2026-07-19 (24 dates) |
| **Product** | TOL (18:00 Lakeside Twilight Train) |
| **Base capacity** | 256 pax / date |
| **WC capacity** | 3 slots / date |
| **Status** | Active (published) |
| **Active version** | 391004ab-dcc3-48e9-92e7-62cc2a70b478 |
| **Last updated** | 2026-05-27 |
| **Author** | Mitch Fraser / Claude AI |

---

## Purpose

This n8n workflow is the backend for PBR's Queue-it integration for the 2026 TOL season. Queue-it queries this endpoint to determine which dates are sold, limited, or available, and uses the data to gate the onsale queue appropriately.

The endpoint combines two independent data sources:

- **CustomLinc availability API** — real-time checkout availability (`apiPax`): what the next customer can actually book right now
- **CustomLinc SQL (Txn/Booking tables)** — confirmed booking truth: what has actually been paid and is genuinely sold

This dual-source approach is required because the CustomLinc availability API is influenced by in-cart (pending) bookings, operator sales caps, and Manifest counter drift — none of which should drive the public SOLD/LIMITED classification.

---

# Architecture: Node Flow

```
Availability Webhook
  → CustomLinc Login  (parallel)
  → Query Bookings    (SQL via Microsoft SQL node)
  → Ensure One Item   (passthrough - ensures Classify fires even if SQL returns 0 rows)
  → Build Date Chunks (splits season into 10-day API request chunks)
  → Loop Chunks
      → Fetch CustomLinc Availability  (POST per chunk)
      → [loop back]
  → Classify          (Code node - cross-joins SQL + API, outputs payload)
  → Respond JSON      (HTTP 200 with JSON body)
```

The CustomLinc Login and Query Bookings nodes run in parallel at workflow start. Both results are available by the time Classify executes after the loop completes.

---

# SQL Query: Query Bookings

The SQL query runs against the CustomLinc MS SQL database (subscriber 22). It joins Manifest (per-date capacity) against confirmed (statusType=0) and pending (statusType=6) Txn/Booking records.

## Key design decisions

- Manifest is the **primary table** (LEFT JOIN). Dates with no bookings still appear (pax values default to 0 via COALESCE).
- `statusType IN (0, 6)` — includes both Confirmed (0) and Pending Web Payment (6) bookings as separate columns.
- WC (Wheelchair Add-on fare code) is counted separately from standard pax (ADULT, CHILD, TODDLER).
- Booking counts are tracked per status for ops monitoring.
- Hardcoded season window: 2026-06-26 to 2026-07-20 (exclusive upper bound).

# Output columns

| Column | Description |
|---|---|
| travel_date | Date of departure |
| pax_max | Manifest.max — base capacity (typically 256) |
| manifest_totalPax | Manifest counter (cumulative, never decremented on cancel — known drift issue) |
| manifest_totalAvailable | Manifest available counter |
| pax_confirmed | Paid pax from statusType=0 bookings (ADULT+CHILD+TODDLER) |
| wc_confirmed | Paid WC slots from statusType=0 bookings |
| pax_pending | In-cart pax from statusType=6 bookings (Pending Web Payment) |
| wc_pending | In-cart WC slots from statusType=6 bookings |
| booking_count_confirmed | Number of confirmed booking records |
| booking_count_pending | Number of pending booking records |

# Manifest counter drift

The Manifest.totalPax counter increments on booking but does **not** decrement on cancellation. This is a confirmed CustomLinc bug. The endpoint does not rely on Manifest counters for classification — it uses direct Txn aggregation instead. Manifest mismatches are flagged in `monitoring.manifestMismatches[]` for ops awareness only.

---

# Classification Model: Pending-Blind

Sold/limited classification is driven exclusively by **confirmed seats** (statusType=0). Pending bookings (statusType=6, i.e. in-cart / Pending Web Payment) are tracked and reported in monitoring but do **not** affect the public classification or the dynamic message.

# Classification inputs per date

| Variable | Source | Description |
|---|---|---|
| `paxConfirmed` | SQL statusType=0 | Paid seats — the revenue truth |

| | | |
|---|---|---|
| `paxPending` | SQL statusType=6 | Seats in active checkout carts |
| `apiPax` | CustomLinc API | Seats available to buy right now (reduced by confirmed + pending + any operator cap) |
| `isAvailable` | API .isAvailable | Whether the operator has stopped sales |

# effectiveCap derivation

If the API reports fewer seats than `max - confirmed - pending`, an operator-set sales cap is inferred:

```
remainingIfNoCap = max(paxMax - paxConfirmed - paxPending, 0)
if isAvail AND apiPax < remainingIfNoCap:
    effectiveCap = apiPax + paxConfirmed + paxPending
else:
    effectiveCap = null
```

effectiveCap is suppressed when `!isAvail` (stop-sold) because in that case apiPax=0 is caused by operator action, not a cap.

# confirmedRemaining (classification source)

```
effectiveCeiling = effectiveCap ?? paxMax
confirmedRemaining = max(effectiveCeiling - paxConfirmed, 0)
```

`confirmedRemaining` is the key classifier input. It represents how many confirmed seats are genuinely still available, ignoring pending carts.

# Classification rules

| Condition | Result |
|---|---|
| `!isAvail` (operator stop-sold) | `sold[]` + `stopSold[]` |
| `confirmedRemaining === 0` | `sold[]` (truly sold out by confirmed bookings) |
| `confirmedRemaining <= 20` | `limited[]` |
| `confirmedRemaining > 20` | plenty (not in any list) |

# apiBlocked flag (monitoring only)

```
isApiBlocked = (apiPax === 0) AND (confirmedRemaining > 0) AND isAvail
```

When pending bookings fill remaining seats, the API returns apiPax=0 but confirmed seats remain. This is an ops signal — pending bookings will expire and release those seats. The date does **not** appear in `sold[]` or `dynamicMessage`. It is reported in `monitoring.apiBlocked[]`.

# boundBy field

| Value | Meaning |
|---|---|
| `"capacity"` | No bookings on this date |
| `"confirmed"` | Confirmed bookings are the limiting factor |
| `"cap"` | Operator-set sales cap is binding (effectiveCap is set) |

# Wheelchair classification (pending-blind)

WC follows the same pending-blind rule. WC cap = 3 seats. WC limited threshold = 1 remaining.

```
wcConfirmedRemaining = max(3 - wcConfirmed, 0)
effectiveWcRemaining = dateSold ? 0 : wcConfirmedRemaining
if effectiveWcRemaining <= 0  → wheelchairSold[]
if effectiveWcRemaining === 1  → wheelchairLimited[]
```

---

# Output Payload

## Public classification arrays

| Field | Type | Description |
|---|---|---|
| `sold` | string[] | Tokens for sold-out dates (e.g. "26/6"). Driven by confirmedRemaining=0 or stop-sold. |
| `limited` | string[] | Tokens for limited dates (confirmedRemaining 1–20). |

| | | |
|---|---|---|
| `stopSold` | string[] | Subset of sold[] where isAvailable=false (operator stop-sold). |
| `apiBlocked` | string[] | Dates where apiPax=0 but confirmedRemaining>0. Ops monitoring only — not in dynamic message. |
| `detail` | object | Token → apiPax. Live checkout figure for Queue-it. Reflects confirmed + pending + cap. |
| `wheelchairSold` | string[] | WC sold-out dates. |
| `wheelchairLimited` | string[] | WC limited dates (1 remaining). |
| `wheelchairDetail` | object | Token → wcConfirmedRemaining (pending-blind). |
| `dynamicMessage` | string | Queue-it dynamic message string. Format: "SOLD: d1, d2 LIMITED: d3 WC: d4". Empty if no sold/limited dates. |
| `updatedAt` | ISO 8601 | Timestamp of this response. |

# dynamicMessage format

```
SOLD: 26/6, 4/7
LIMITED: 11/7
WC: 26/6
```

Lines are only included if the respective array is non-empty. The message is blank if all dates are available. Queue-it reads this field to populate availability messaging on the waiting room page.

# monitoring object

The `monitoring` object is not consumed by Queue-it but is available for ops dashboards and troubleshooting.

## monitoring.season (season-level aggregates)

| Field | Description |
|---|---|
| totalCapacity | Sum of pax_max across all dates |
| totalConfirmedPax | Total paid pax across season |
| totalConfirmedWc | Total paid WC slots |

| | |
|---|---|
| totalConfirmedBookings | Booking record count (confirmed) |
| totalPendingPax | Total in-cart pax across season |
| totalPendingBookings | Booking record count (pending) |
| totalInflightPax | Alias of totalPendingPax |
| percentSold | totalConfirmedPax / totalCapacity × 100 |
| datesTotal | Dates processed by the classifier |
| datesWithBookings | Dates with at least one confirmed booking |
| datesWithPending | Dates with at least one pending booking |
| datesSold / datesLimited | Classification counts |
| datesStopSold | Operator stop-sold count |
| datesApiBlocked | Dates where apiPax=0 but confirmedRemaining>0 |
| datesManifestMismatch | Dates where Txn sum ≠ Manifest counter |
| datesCapped | Dates with an operator-set sales cap detected |

# monitoring.byDate

Per-date object keyed by token (e.g. "26/6") with full classification detail:

```
{
  "26/6": {
    "max": 256,
    "confirmed": 2,          // paid pax
    "wcConfirmed": 0,
    "pending": 4,            // in-cart pax
    "wcPending": 0,
    "bookingsConfirmed": 1,
    "bookingsPending": 1,
    "inflight": 4,           // alias of pending (from SQL statusType=6)
    "confirmedRemaining": 254, // classification source
    "apiPax": 250,           // live checkout figure
    "apiBlocked": false,
    "effectiveCap": null,    // operator cap if detected
    "boundBy": "confirmed",  // "capacity" | "confirmed" | "cap"
    "manifestTotalPax": 237,
    "manifestTotalAvailable": 19,
    "manifestDelta": -231,   // negative = Manifest counter ahead of Txn (cancelled bookings
not decremented)
```

```
    "percentSold": 0.78,
    "isAvailable": true,
    "availabilityString": "plenty"
  }
}
```


# monitoring.topInflight / topConfirmed

Top-5 dates sorted by inflight (pending) pax or confirmed pax respectively. Useful for ops dashboards.

# monitoring.manifestMismatches

Dates where the Txn-derived sum (confirmed + pending) differs from the Manifest counter. Reported for visibility of the CustomLinc counter drift bug.

# monitoring.capped

Dates where an operator-set sales cap was detected (effectiveCap is non-null), sorted by cap size ascending.

# monitoring.apiBlocked

Detailed array of dates where checkout is blocked by pending bookings (not truly sold). Each entry includes: confirmed, pending, confirmedRemaining.

# monitoring.health

| Field | Description |
|---|---|
| sqlRowCount | Rows returned by SQL query |
| sqlExpected | Expected row count (24 for full season) |
| sqlOk | true if sqlRowCount === sqlExpected |
| apiResponseCount | Number of CustomLinc API chunk responses received |
| executionDurationMs | Classify node execution time in ms |
| wcCap | WC cap constant (3) |
| limitedThreshold | Limited threshold constant (20) |
| confirmedFilter | statusType IN (0,6) — SQL filter in use |
| classificationSource | confirmedRemaining |
| detailSource | apiPax |
| inflightSource | statusType=6 |

# Design Decisions & Rationale

## Why pending-blind classification?

Pending bookings (statusType=6) represent seats in active checkout carts. They auto-expire after a timeout if payment is not completed. Classifying a date as SOLD because of pending bookings would:

- Show false SOLD status to Queue-it and customers
- Prevent genuine sales when carts expire
- Cause a poor customer experience during high-demand onsales

The `apiBlocked` flag is provided for ops monitoring so the team is aware when this condition exists, without affecting public availability messaging.

## Why apiPax in detail[] rather than confirmedRemaining?

The `detail[token]` value is the live checkout figure that Queue-it can pass to a checkout integration. It reflects what the next customer can actually book, accounting for confirmed + pending + operator cap. Using `confirmedRemaining` here would overstate available seats (it ignores in-cart bookings).

## Why suppress effectiveCap when stop-sold?

When a date is stop-sold (!isAvail), apiPax is forced to 0 by operator action. The residual cap calculation would produce a false cap equal to confirmed pax. Suppressing effectiveCap in this case avoids confusing monitoring output.

## Manifest counter not used for classification

The Manifest.totalPax counter is unreliable for classification because cancelled bookings do not decrement it. On 2026-06-26 the counter shows 237 (from earlier test bookings) while the actual Txn sum is 2 confirmed + 4 pending. The endpoint uses direct Txn aggregation (statusType 0 and 6) as the authoritative source. Manifest data is retained in monitoring for counter drift visibility.

# SQL Manifest-join pattern

The SQL query uses Manifest as the primary table (not Booking), joined to Txn/Booking via LEFT JOINs. This ensures all 24 season dates appear in the result even if they have zero bookings. An earlier version grouped by b.dateTimeOfTravel which collapsed booking-less Manifest rows into NULLs.

# Operator cap detection

CustomLinc allows operators to set a per-departure sales cap via the UI. The cap column was not found in any accessible schema table (Manifest, ManifestAllocation, Movement, ProductOption all checked). The cap is inferred from the residual: if apiPax < (max - confirmed - pending), the cap must equal (apiPax + confirmed + pending). This is a forward-compatible inference that works regardless of where the cap is stored.

---

# Known Issues

# Silent failure mode

If the CustomLinc availability API returns a response with null/empty `productAvailability`, the workflow still returns HTTP 200 with `sqlOk: true` but no date data in the payload. Queue-it would receive an empty availability structure. This edge case is not yet handled with an explicit error response. **Status: open.**

# Manifest counter drift (CustomLinc bug)

Cancelling a booking does not decrement the Manifest.totalPax counter. This causes growing divergence between the counter and actual bookings on dates with cancellation activity. Does not affect classification (Txn is used directly). Reported to CustomLinc support. **Status: vendor bug, monitoring only.**

# Pending booking expiry timing

Pending bookings (statusType=6) auto-expire on a CustomLinc internal timeout. The expiry duration is not documented. During high-demand onsales, many carts may be open simultaneously, causing `apiBlocked` dates to appear transiently. These resolve automatically when carts expire without payment.

---

# Seasonal Maintenance

At the start of each new TOL season, the following constants must be updated in both the SQL query and the Build Date Chunks node:

- `SEASON_START` — first date of the new season
- `SEASON_END` — last date of the new season
- `SQL_EXPECTED_ROWS` — number of dates in the season (for health check)
- SQL WHERE clause date bounds (hardcoded in the Query Bookings node)

Also verify:

- CustomLinc Auth credentials are still valid (Login node)
- The SQL subscriber ID (22) has not changed
- The product code (TOL) matches the new season's product
- Queue-it dynamic message format matches the new season's campaign configuration