DBCO

DBCO FREE BUILD BRIEF · MASS MESSAGES

The Mass Message System

A free, paste-ready build brief from the DBCO Value Library. Drop the companion **.md** file straight into Claude Code and build the system from the video yourself.

BUILD IT YOURSELF | CLAUDE CODE READY | NO SAAS FEES

Drop this whole file into Claude Code and it'll scaffold you a working mockup of the mass-message system I walk through in the video. This is written to you, the builder. Read it top to bottom once, then paste the prompts at the bottom into Claude Code one at a time.

First — what this actually is (and what it is NOT)

If you run a chatting operation across a bunch of creator accounts, you already feel this problem: **mass messages are money, and mass messages are chaos.**

Every managed account should be blasting the fan list on a regular cadence. That's a huge chunk of revenue — it's the cheapest re-engagement you have. But the second you're running more than two or three creators with a rotating team of chatters across three shifts a day, it falls apart:

- Nobody knows *what* to send. Every chatter freestyles a different message, half of them are lazy one-liners, and quality is all over the place.
- Nobody knows *when it was last sent*. Some accounts get blasted three times a day, others go dark for a week and the fans forget the creator exists.
- The good scripts live in one person's head (or a messy Google Doc), and they never make it to the person actually on shift at 3am.
- When someone skips their mass messages, you find out a week later when revenue dips — not the same day.

So here's the important part, and this is the thing most people get wrong when they try to build this: **the system I'm showing you is NOT an auto-sender.** It does not touch the OnlyFans API. It does not blast anything automatically. **Never build a tool that hits OF's API directly — you'll get the account logged out or flagged.** Everything sends by a human, manually, like a human.

What we actually built is a **centralized scripting + scheduling + distribution system**. Think of it as a teleprompter for your chatters:

1. One person — your chatting manager — **pre-writes the mass-message scripts** and schedules them: this script, for this creator, for this shift slot, on this day.
2. The system figures out **who is on shift right now, and which creator they're covering**, and **surfaces exactly the right script to that chatter** in their panel.
3. The chatter reads it, hits copy, pastes it into OF, and sends. Manually.
4. Meanwhile the system **tracks coverage** — how many days ahead each creator has scripts scheduled — and **fires a Telegram alert** before any creator runs out.

That's it. Pre-write → schedule → surface to the right person → track coverage → alert before runout. No API, no automation, no risk. Just organization at scale.

We cloud-coded the whole thing ourselves — it's a Node/Express backend, a couple of JSON files for storage, and two HTML panels (one for the manager, one for the chatters). You can build the same scaffold yourself with Claude Code in an afternoon. That's what this file is for.

> **One thing up front:** what you'll build from this brief is a **working mockup / scaffold**, not a turnkey production system. It'll have the real data model, the real admin composer, the real chatter panel, the real coverage-alert logic. The one thing you supply is the actual message *content* — the words you send are yours to write, in your own voice and for your own creators. The system treats every message as a plain plug-in string, so you drop your own messages (or your own way of writing them) straight in. More on where that goes below.

The mental model: three surfaces, one source of truth

Before any code, get the shape in your head. There are **three people** who touch this system, and they each see a different surface:

Who	Surface	What they do
Chatting manager (admin)	Composer panel	Writes/schedules the scripts. Bulk-imports a whole week at once. Sees coverage.
Chatter (on shift)	"Mass" tab in their panel	Sees ONLY the scripts for their current shift + creator. Copies, sends, checks off.
The system	Backend + storage	Resolves "who's on shift, covering whom, what should they send" and watches coverage.

And the thing that ties it all together — the **single source of truth** — is your **weekly schedule**. The schedule says "chatter X covers creator Y on the night

slot Tuesday." The mass-message system *reads that schedule* to decide which scripts to show chatter X when they clock in. **Do not build the mass-message targeting as a separate assignment system.** Bind it to the schedule you already have. That's the key architectural insight and it's what makes the whole thing feel automatic.

The pieces you're going to build

Here's the component list. Build them in this order.

1. **A storage file for messages** — one JSON file, an array of message objects. (Swap for a real DB later; JSON is perfect for a mockup.)
2. **A data model for a message** — with slot/creator *assignments* and a schedule date. This is the heart of it. Get this right and everything else is easy.
3. **Admin API** — CRUD for messages (create, read, edit, delete) with an idempotency guard.
4. **Chatter API** — the important one: "given who I am and my current shift, what should I send right now?"
5. **Admin composer UI** — a form to write one message, plus a CSV bulk-import pipeline for loading a whole week at once.
6. **Chatter "Mass" tab UI** — collapsible cards, per-line copy buttons, a shift-context header, a multi-creator filter, and a client-side "mark as sent" checklist.
7. **Coverage tracking + Telegram alerts** — the runway algorithm and the low-coverage notification.
8. **Shift-checklist + penalties tie-in** — "Send 3 mass messages" as a checklist item and a penalty type.

Let me walk each one.

---

1 & 2. Storage + the data model (spend your time here)

Store messages in a single JSON file — call it `mass-messages.json` — as a flat array. In the real system this file holds well over a thousand messages spanning a couple of months of daily coverage, and flat JSON handles it fine for a mockup. Have a `loadMM()` / `saveMM()` pair that reads/writes the whole file.

Each message object looks like this:

```
{
  "id": "a uuid",
  "assignments": [
    { "creator_id": "creator-uuid-here", "slots": ["night"] }
  ],
  "lines": [
    { "text": "the full script goes here", "marker": null }
  ],
  "scheduled_date": "2026-07-05",
  "created_at": "iso timestamp",
  "updated_at": "iso timestamp",
  "created_by": "Marko"
}
```

The three fields that matter:

`assignments[]` **— who and which shift this script is for.** Each assignment is `{ creator_id, slots }`. Two magic values make this flexible:

- `creator_id` can be a specific creator's UUID **or the literal string** `"all"` — meaning "every creator."
- `slots` is an array of slot names (`"morning"` , `"evening"` , `"night"`) **or** `["all"]` — meaning "any shift."

So `{ creator_id: "all", slots: ["all"] }` is a script that shows for every creator on every shift. `{ creator_id: "jenny-uuid", slots: ["night"] }` shows only for Jenny on the night shift. This little `creator × slot` matrix, with

`"all"` wildcards on both axes, is the entire targeting model. It's simple and it covers every case.

`lines[]` **— the message content.** An array of `{ text, marker }`. For the mockup, keep it dead simple: **the whole script goes into one line's** `text` **field**, and `marker` stays `null`. (The `marker` field is a vestige of an older per-line design — you can include it in the schema for forward-compat, but you don't need to populate it. Just store the script as one text blob.)

> **The message content is yours to write.** What goes *inside* that `text` — the actual words, how they're structured, what makes them convert — is specific to your creators and your voice. In your scaffold, treat `text` as a plain string: **write your messages by hand, use your own AI prompt, whatever fits your brand**, and drop them in. The system doesn't care what's in the string — it just schedules it and surfaces it to the right chatter at the right time. You build the machine around the message; the message itself you write for your own pages.

`scheduled_date` **— when it shows, with one special case.**

- A date string `"YYYY-MM-DD"` = this script shows for that specific day only.
- `null` **= evergreen** = this script shows *every day, forever*, until you delete it. Great for a creator's always-on default blast. Evergreen creators, by the way, never trigger low-coverage alerts (they're covered by definition) — remember that when you build the alert logic.

Two more things to bake into the model from day one:

Idempotency key. When you bulk-import (see the CSV pipeline below) you POST messages row by row, and you don't want re-running an import to stack duplicates. Compute a dup key per message = `scheduled_date` + a stringified `assignments` + an md5 hash of the `lines`. On create, if a message with that key already exists, **return the existing one instead of inserting a duplicate.** This one guard makes imports safe to re-run.

A normalizer for legacy shapes. If you ever change the assignment format, write a small `normalizeAssignments()` that upgrades old records on read. In the real system this converts a legacy `{ creators[], slots[] }` shape into the modern `assignments[]`. Cheap insurance.

The one genuinely fiddly bit: slots and timezone

Your shifts are wall-clock times in **one fixed timezone** — pick one and commit (we use Europe/Berlin / CET). Define your slots as start/end hours:

- `morning` = 10:00–18:00
- `evening` = 18:00–02:00 (crosses midnight)
- `night` = 02:00–10:00 (crosses midnight)

The gotcha: the **night slot (02:00–10:00)** belongs to the *previous* calendar day's schedule, because someone working 02:00–10:00 Tuesday morning is really on "Monday night's" shift. So for any slot whose start hour is before 10:00, **store its schedule row under `display_date - 1 day`.** The admin UI, the chatter query, and the CSV importer all special-case "start hour < 10 → shift the storage date by a day."

Keep this rule in *one* helper and reuse it everywhere. The nice trick we use: the person writing the script always types the **display date** (the human date they think of it as), and the system silently applies the offset. Don't make your author do timezone math.

---

3. Admin API (CRUD + idempotency)

Node/Express. Gate these behind an admin auth check. Routes:

- `GET /api/admin/mass-messages` → return all messages, **enriched** with human-readable `creator_names` and `slot_labels` (resolve the UUIDs/slot keys to names before sending to the UI — the composer needs to show "Jenny · Night", not a raw UUID).

- `POST /api/admin/mass-messages` → validate that `lines` and `assignments` are present and well-formed, run the **idempotency check**, then insert. Return the created (or existing) message.
- `PUT /api/admin/mass-messages/:id` → edit.
- `DELETE /api/admin/mass-messages/:id` → delete.

Nothing exotic. The only non-obvious part is the idempotency guard on POST — build it once, thank yourself later.

---

4. Chatter API — "what should I send right now?" (the important one)

This is the brain. One endpoint:

`GET /api/chatters/mass-messages` — gated behind the chatter's own token. It returns only the scripts relevant to *that chatter's current (or next upcoming) shift*. Here's the resolution logic, step by step — build it exactly in this order:

1. **Figure out the chatter's effective shift.** Prefer an `active` **shift row** if one exists (i.e. they've clocked in — that's the ground truth of what they're working right now). If not clocked in, fall back to today's scheduled slot for them. If nothing today, fall back to their **next upcoming** scheduled entry (so a chatter about to start can prep).

2. **Ask the schedule who they cover.** This is where the weekly schedule is the source of truth. Filter the schedule by `chatter_id + slot_id + stored_date` (remember the night-slot date offset) to get the creator(s) this chatter covers this slot. Add a defensive fallback: if they clocked in for a creator who isn't on the printed schedule, include that creator too.

3. **Filter all messages** down to the ones where:
   - `(creator ∈ my covered creators OR assignment is "all")` **AND**
   - `(slot matches my slot OR assignment slot is "all")` **AND**

    - `(scheduled_date == my effective date OR scheduled_date is null/evergreen)`

4. **Tag each returned message** with which of my creators it matched (`matched_creators`), so the UI can group/label them.
5. **Return a payload** shaped like:

```
{
  "messages": [ ... ],
  "current_slot": "night",
  "shift_context": {
    "type": "current",           // or "upcoming"
    "creator_name": "Jenny",
    "date": "2026-07-05",
    "slot_label": "Night",
    "slot_start": "02:00",
    "slot_end": "10:00"
  }
}
```

That `shift_context` is what lets the chatter panel say "here's your Jenny night shift, 02:00–10:00" at the top. Sweat this endpoint — it's the whole magic trick.

---

5. Admin composer UI (two ways to load messages)

Build this as its own admin page, gated behind a `mass_messages` permission (you don't want every admin in here). Two modes:

Manual mode — for one-off scripts. A form with:

- A **creator × slot matrix** (checkboxes) to build the `assignments[]`.

- A **date picker** for `scheduled_date` (with an "evergreen / no date" option that stores `null`).
- One **big textarea** where you paste the full script.
- Submit → POST one message with the whole textarea as a single `text` line.

CSV bulk-import mode — *this is the real daily workflow.* You're not hand-entering a hundred scripts a week through a form. You prepare a CSV and load the whole thing at once. Build:

1. A drop/paste zone for CSV text.
2. A **preview step** that parses the CSV (use a proper RFC-4180 parser — your message content will have commas, quotes, line breaks, and emojis, so a naive `split(',')` will destroy it), validates each row (date format valid? creator name exists? slot label recognized?), and **cross-references the live schedule** to show *which chatter is currently assigned* to that creator+slot+date. That preview column is gold — the author instantly sees "this script is going to land on David's Tuesday night shift."
3. A preview table: `# / Date / Creator / Slot / Chatter / Message / Status (valid ✓ or error)`.
4. An **Apply** button that POSTs each valid row to the admin create endpoint. The idempotency guard means re-applying is safe.

CSV columns are just: `date, creator, slot, message`.

> **Robustness tip worth stealing:** when you *generate* the CSV (whatever your message method is), don't hand-assemble the CSV string — mass-message content is full of line breaks, quotes and emojis that break naive CSV building. Generate it programmatically with a real CSV writer (e.g. Python's `csv.writer` with proper quoting) so multi-line, quote-heavy, emoji-heavy content survives the round-trip intact. This is a plumbing tip, not the secret — the *content* of those rows is still yours to bring.

---

6. Chatter "Mass" tab UI

This is what the person on shift actually stares at. Build a **"Mass" tab** in the chatter panel that calls `GET /api/chatters/mass-messages` on load. Then:

- **Header subtitle** with the shift context: *"Messages prepared by your team for your current shift — Jenny · 10:00–18:00."* Convert the slot times from your fixed system timezone into **the chatter's local timezone** so a chatter in a different country sees their own clock.
- **Each message = a collapsible card**, numbered globally with the creator name as a suffix: *"Mass Message 1 — Jenny."* If a card matches multiple creators, show them joined: *"Jenny & Jess."*
- If the chatter covers **multiple creators**, show a **filter bar** at the top: *"Show: All / Jenny / Jess."*
- **Per-line copy button.** Each line of the script renders as a bubble with its own copy-to-clipboard button. This is the single most-used control in the whole system — the chatter's loop is literally *tap copy → paste into OF → send → next.* Make it fast and obvious.
- **"Mark as sent" checkbox** per card. Keep this **purely client-side** for the mockup — an in-memory set, not persisted to the server. It's a visual checklist so the chatter doesn't lose their place, nothing more. (You *can* persist it later, but you don't need to, and not persisting keeps it dead simple.)
- (Optional, forward-compat) If you keep the `marker` field, you can show a little tooltip on lines that carry one — a hint about whether to wait for a reply before sending the next line, or to fire two in a row. In the mockup this is rare-to-never used since you're storing scripts as one blob, so don't over-build it.

---

7. Coverage tracking + Telegram alerts (the safety net)

This is what turns "a scripting tool" into "a system you can trust." You want to know, per creator, **how many days ahead they have scripts scheduled**, and you want a ping *before* anyone runs dry.

The runway algorithm — and here's the subtle part most people get wrong: don't just take the max scheduled date. **Count *consecutive* covered days starting from today.** If a creator has scripts today, tomorrow, and the day after, then a gap, then more next week — their runway is **3 days**, not "covered through next week." Gaps matter. So:

- Start at today. Walk forward day by day as long as each day has at least one scheduled (or evergreen) message for that creator. Stop at the first gap. The number of consecutive covered days = the **runway**.
- Also compute `nextAfterGap` — when coverage resumes after the gap — so the alert can say "covered today but there's a hole Thursday, resumes Saturday."
- **Evergreen creators** (any null-dated message targeting them) are covered forever — never alert on them.

The Telegram alert — run a sweep on a timer (every ~5 minutes, plus a catch-up run shortly after boot). For each active creator, compute the runway. **Fire when** `runway ⩽ threshold`**.** Make the threshold a per-user preference (default 2 days, clamp it to something sane like 0–30). Suppress alerts overnight (e.g. before 08:00 local) so you're not pinging people at 4am. Dedup so each user gets at most one alert per creator per day (a dedup key like `userId|creatorId|today`).

Message format, roughly:

```
📨 Low Mass Message Coverage
👤 Jenny (Agency Name)
📅 Covered until: 2026-07-05  (⚠ Gap after Thu, resumes Sat)
⏳ Runway: 2 days
```

With edge copy for the nasty cases: "⚠ Today uncovered — next scheduled: …" and "No future mass messages scheduled."

Route it to the right people. Send low-coverage alerts to your admins / chatting managers — the people who can actually fix it by writing more scripts — not to the chatters. If you've split your notifications into topic-based bots (I have a separate bot per notification type), this goes to the "content" bot. In a mockup, one bot and a subscriber list is plenty. Let each subscriber opt out and set their own threshold.

Also mirror it in-app. Push the same signal into an in-app "Alerts" panel as a `{ type: 'mm', severity: 'critical', detail: 'No mass messages scheduled' }` item, so the coverage gap is visible in the dashboard even if someone muted Telegram.

Optional — a weekly digest. Roll mass-message coverage together with your other content types (stories, feed posts, whatever you track) into one weekly "content runway" digest (e.g. Monday 9am), plus an instant ping on anything critical. Mass messages become one of several coverage types in that digest.

---

8. Shift-checklist + penalties tie-in

The last 10% that makes people actually *do it*. Wire the mass-message system into your shift accountability:

- Add **"Send 3 Mass Messages"** as item #1 on the per-shift checklist every chatter completes. (Three per shift is our standard — pick your own number.)
- Add **"Skipped mass messages"** as a formal **penalty type** in your penalties system.

Now the loop is closed: the script is surfaced to the chatter → the checklist expects them to send it → skipping it is a named, trackable penalty → the coverage sweep catches the upstream gap before it even gets to the chatter. Three layers, one behavior. That's why it holds up at scale.

Copy-paste build prompts for Claude Code

Paste these into Claude Code **one at a time, in order.** Let each one finish and eyeball the result before moving on. Adjust names to your stack.

Prompt 1 — data model + storage

```
Build a Node/Express backend with a JSON-file store for a "mass messages"
Create mass-messages.json (an array) with load/save helpers.
Define the message model: { id (uuid), assignments:[{creator_id, slots:[]
Rules:
- creator_id may be a UUID or the literal "all"; slots may be slot names
- scheduled_date is "YYYY-MM-DD", or null meaning "evergreen / show every
- For the mockup, store the whole message as ONE line: lines = [{ text: <
Add an idempotency key = scheduled_date + JSON(assignments) + md5(lines);
Define three shift slots as wall-clock hours in ONE fixed timezone: morni
Leave a clearly-commented function generateMessageText() that returns a p
```

Prompt 2 — admin CRUD API

```
Add admin-gated Express routes for mass messages:
GET /api/admin/mass-messages (return all, enriched with human creator_nam
POST (validate lines+assignments, run the idempotency check, insert),
PUT /:id (edit), DELETE /:id.
Gate everything behind an admin auth check and a `mass_messages` permissi
```

Prompt 3 — the chatter "what do I send now" endpoint

```
Add GET /api/chatters/mass-messages, gated behind the chatter's token.
Resolve the chatter's effective shift: prefer an "active" clocked-in shif
Use the weekly schedule as the source of truth for which creator(s) that
Return only messages where (creator in my creators OR "all") AND (slot ma
Tag each message with matched_creators. Return { messages, current_slot,
```

Prompt 4 — admin composer UI + CSV import

```
Build an admin composer page for mass messages with two modes.
Manual: a creator×slot checkbox matrix builds assignments[], a date picke
CSV bulk-import: a paste/drop zone; parse with a proper RFC-4180 parser (
```

Prompt 5 — chatter "Mass" tab UI

```
Build a "Mass" tab in the chatter panel that calls GET /api/chatters/mass
Header subtitle shows shift context ("Messages prepared by your team for
Each message is a collapsible card numbered globally with the creator nam
If the chatter covers multiple creators, show a filter bar (All / per-cre
Each line renders as a bubble with its own copy-to-clipboard button.
Add a per-card "Mark as sent" checkbox that is purely client-side (in-mem
```

Prompt 6 — coverage runway + Telegram alerts

```
Add a coverage sweep on a ~5-minute timer (plus a catch-up run ~50s after
For each active creator, compute the runway = number of CONSECUTIVE cover
Fire a low-coverage alert when runway ⩽ a per-user threshold (default 2,
Message: "📨 Low Mass Message Coverage / 👤 <creator> (<agency>) / 📅 Cov
Also push the same signal into an in-app Alerts panel as { type:'mm', sev
```

Prompt 7 — accountability tie-in

```
Add "Send 3 Mass Messages" as the first item on the per-shift chatter che
Add "Skipped mass messages" as a penalty type in the penalties system.
```

What you'll have when you're done

A working mockup where: a manager writes and schedules scripts (one at a time or a whole week via CSV), the schedule decides who covers whom, a chatter clocks in and instantly sees exactly the right scripts for their shift

with one-tap copy buttons, and the system watches every creator's coverage runway and pings the managers before anyone runs dry — all without ever touching the OF API.

The one thing you supply is the message content itself. The `generateMessageText()` function is a placeholder — that's where your own messages go. The *words* of the scripts, how they're structured, what makes a fan actually reply and buy, are specific to your creators and your voice, so you write those yourself and drop them in. Everything around it — the part that makes the system survive contact with a real team across real shifts — is what this brief builds for you.

Build the machine from this file, then write your own messages to run through it.

Built with Claude Code. You can build this yourself with Claude Code too — that's the whole point.
