Nudge Backend (MVP) — API + Schema + Worker

This repo is the MVP backend for **capturing items (URLs or pasted text)**, extracting readable text asynchronously for URL items via a **background worker**, and exposing a stable HTTP contract for the frontend.

Local dev default: Docker Postgres + FastAPI on http://localhost:8000.

High-level architecture

Components

- **FastAPI API service** (synchronous request handling; no scraping inside API handlers)
- **Postgres database** (canonical state)
- **Background worker** (polls DB for queued URL items, fetches pages, extracts readable text, updates DB state)

Design rule of thumb

- **API creates and reads items.**
- **Worker performs extraction and moves items through statuses.**
- **items.status is the single source of truth** for UI state.

Item lifecycle (state machine)

items.status values:

- queued — created from a URL; waiting for worker
- processing — claimed by a worker (row-locked) and being processed
- succeeded — canonical text is available (item_content.canonical_text)
- needs_user_text — worker could not extract; UI should request paste fallback
- failed — rare, terminal internal error

Typical transitions:

- URL flow: queued → processing → succeeded OR queued → processing → needs_user_text
- Pasted-text flow: created as succeeded immediately (no worker)

items.status_detail is a **human-readable** message intended to help the UI explain non-happy paths (retrying, paste needed, etc.).

Database schema (4 tables)

users

- Minimal user identity (UUID primary key).
- Users are created lazily when first seen.

items

Represents one "saved thing" from the user.

Key fields:

- id (UUID)
- user_id (FK → users)
- status (enum)
- source_type (url or pasted_text)
- requested_url (nullable; set for URL items)
- final_text_source (nullable enum; set when succeeded)
- title (nullable; optional future enrichment)
- created_at, updated_at

item_content (1:1 with items)

Stores text blobs:

- user_pasted_text (optional; user-provided)
- extracted_text (optional; raw worker output)
- canonical_text (optional; what frontend should summarize)

For succeeded items:

- canonical_text is the field to summarize
- final_text_source tells where canonical text came from

extraction_attempts

Append-only history of worker attempts:

- (item_id, attempt_no) unique
- Captures error codes, HTTP status, final URL after redirects, etc.

This table makes the system debuggable without digging through logs.

HTTP API contract

Base URL (local): http://localhost:8000

All endpoints are scoped by a user ID derived from the X-User-Id header (details below).

POST /items

Create an item from a URL and/or pasted text.

Request body:

```
{
  "url": "https://example.com/article",
  "pasted_text": "optional text",
  "prefer_pasted_text": false
}
```

Rules:

- Must include at least one of url or pasted_text.
- If pasted_text is provided and (prefer_pasted_text is true OR url is missing):
    - the item is created as succeeded immediately
    - canonical_text = pasted_text
- Otherwise (URL path):
    - item is created as queued

    - worker will extract later

Response:

```
{
  "id": "uuid",
  "status": "queued | succeeded"
}
```

GET /items

List items for the current user (metadata only).

Query params:

- limit (default 20, max 100)
- cursor (optional keyset cursor)

Response:

```
{
  "items": [
    {
      "id": "uuid",
      "status": "queued|processing|needs_user_text|succeeded|failed",
      "status_detail": "optional human readable",
      "source_type": "url|pasted_text",
      "requested_url": "optional",
      "final_text_source": "optional",
      "title": "optional",
      "created_at": "timestamp",
      "updated_at": "timestamp"
    }
  ],
```

"next_cursor": "optional"

}

GET /items/{id}?include_content=true

Fetch one item. If include_content=true, includes the content object.

Response includes the same metadata fields as list, plus:

"content": {

"user_pasted_text": "optional",

"extracted_text": "optional",

"canonical_text": "optional",

"updated_at": "timestamp"

}

PATCH /items/{id}/text

Fallback: user pastes text for an item that is in needs_user_text.

Request body:

{ "pasted_text": "..." }

Rules:

- If item is **not** needs_user_text, backend returns **409 Conflict**.
- On success:
    - item becomes succeeded
    - canonical_text = pasted_text
    - final_text_source = user_pasted_text
    - returns the updated item (with content)

Background worker behavior

The worker runs in a loop:

1. **Requeue stale processing items**

- Any processing item stuck longer than WORKER_STALE_PROCESSING_MINUTES (default 15) is set back to queued.

2. **Claim a batch of queued URL items**
    - Selects items status=queued and source_type=url
    - Uses SELECT … FOR UPDATE SKIP LOCKED so multiple workers can run safely
    - Sets status to processing and commits quickly
3. **Process each claimed item**
    - Fetch HTML with httpx (redirects on)
    - Reject non-HTML content types
    - Enforce WORKER_MAX_BYTES cap
    - Extract readable text via trafilatura with a BeautifulSoup fallback
    - Require minimum length (default ~600 chars)
4. **Write results**
    - Creates an extraction_attempts row for each attempt
    - On success: writes item_content.extracted_text + canonical_text, sets item succeeded
    - On failure: classifies retryable vs non-retryable
        - Retryable HTTP statuses: 429, 5xx (plus timeouts)
        - Non-retryable: most other 4xx, non-HTML, too big, etc.
    - Retries are capped: **max_attempts = 2**
    - If out of retries or non-retryable: sets needs_user_text

Batch size note

If fewer than WORKER_BATCH_SIZE items exist, the worker will process what it claimed and then continue polling normally. If it claims **0** items, it sleeps for WORKER_POLL_SECONDS and tries again.

Local development setup (Docker Postgres + API)

1) Start Postgres + API

From repo root:

docker compose up --build

This starts:

- Postgres on localhost:5432
- API on localhost:8000

2) Run database migrations

The compose file includes an optional “tools” profile service called migrate.

Run:

docker compose --profile tools run --rm migrate

3) Run the worker

You have two clean options:

Option A (recommended for dev): run worker inside the API container

Because the API container already has deps installed and the code mounted:

docker compose exec api python -m app.worker

To run one batch and exit (useful for debugging):

docker compose exec api python -m app.worker --once

Option B: run worker in a separate local terminal (non-container)

If you run it outside Docker, your DATABASE_URL must point to localhost:

Example (PowerShell / bash)

export DATABASE_URL="postgresql+psycopg://postgres:postgres@localhost:5432/nudge?sslmode=disable"

python -m app.worker

Environment variables

Required:

- DATABASE_URL — SQLAlchemy URL for Postgres

Optional:

- DEV_USER_ID — UUID used when X-User-Id header is missing (default is 00000000-0000-0000-0000-000000000001)

Worker tuning (all optional):

- WORKER_POLL_SECONDS (default 3)
- WORKER_BATCH_SIZE (default 5)
- WORKER_HTTP_CONNECT_TIMEOUT (default 5)
- WORKER_HTTP_READ_TIMEOUT (default 20)
- WORKER_MAX_BYTES (default 2,000,000)
- WORKER_USER_AGENT (default NudgeBot/0.1)
- WORKER_LOG_LEVEL (default INFO)

Frontend integration notes

Authentication / user scoping: X-User-Id

This MVP uses “fake auth” for developer ergonomics:

- If the request includes header X-User-Id: <uuid>, that UUID is the user scope.
- If the header is missing, the backend uses DEV_USER_ID.

What frontend should do:

- In development, **always send X-User-Id** (fixed UUID constant per developer or per test user).
- This lets you simulate multiple users and ensures you don’t accidentally mix item lists between devs.

Example header:

X-User-Id: 11111111-1111-1111-1111-111111111111

Polling strategy

For a digest-first UX:

- After POST /items, poll GET /items (or GET /items/{id}) until status is terminal:
    - terminal: succeeded, needs_user_text, failed
- If needs_user_text, show the paste UI and call PATCH /items/{id}/text.

Quick curl examples (copy/paste)

Create URL item:

```
curl -s -X POST http://localhost:8000/items \
  -H "Content-Type: application/json" \
  -H "X-User-Id: 11111111-1111-1111-1111-111111111111" \
  -d '{"url":"https://en.wikipedia.org/wiki/FastAPI"}'
```

List items:

```
curl -s http://localhost:8000/items \
  -H "X-User-Id: 11111111-1111-1111-1111-111111111111"
```

Get item + content:

```
curl -s "http://localhost:8000/items/<ITEM_ID>?include_content=true" \
  -H "X-User-Id: 11111111-1111-1111-1111-111111111111"
```

Paste fallback (only valid if status is needs_user_text):

```
curl -i -X PATCH "http://localhost:8000/items/<ITEM_ID>/text" \
  -H "Content-Type: application/json" \
  -H "X-User-Id: 11111111-1111-1111-1111-111111111111" \
  -d '{"pasted_text":"Full article text here..."}'
```

Troubleshooting

“Missing required environment variable DATABASE_URL”

The API reads DATABASE_URL at import time via app/settings.py. Ensure your environment (or compose) sets it.

Worker never processes items

Common causes:

- Worker not running
- Items were created with prefer_pasted_text=true (they skip worker)
- Items are stuck in processing (worker crashed mid-run); the stale requeue will eventually move them back to queued (default 15 minutes)

PATCH /items/{id}/text returns 409 Conflict

This is expected unless the item is currently needs_user_text. The UI should only enable paste flow in that status.