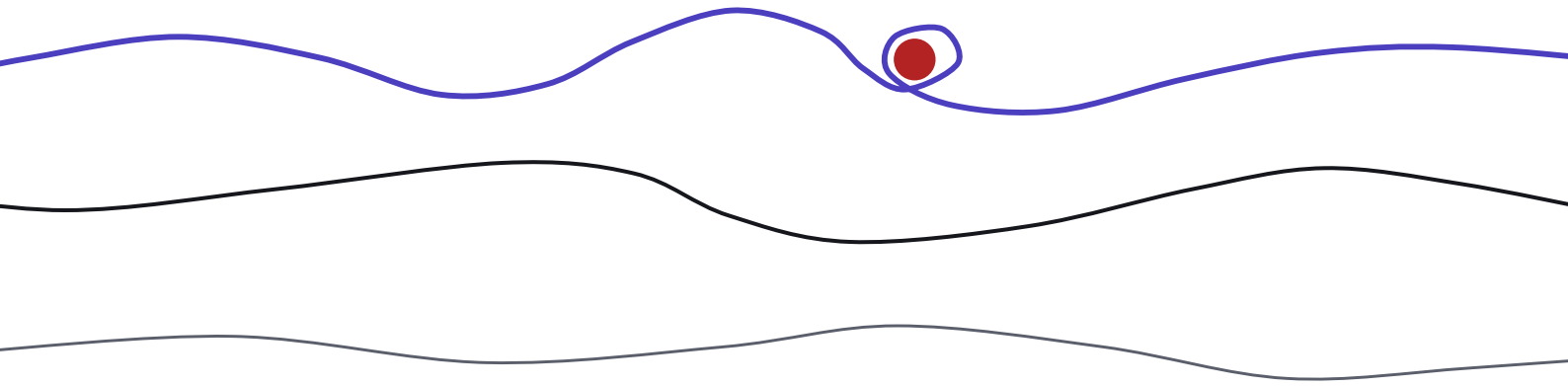
jupiter.au

hello-fresh API

Menu planning service for weekly menus, recipes, ingredients and reviews, running as a Cloudflare Worker with a D1 database.



VERSION

v2.0.0

DATE

25 November 2026

AUTHOR

belikh

SOURCE

github.com/belikh/hello-fresh-API

Contents

00.1 What it does 2
00.2 Quick start 2

Start

01 Install and run 4
01.1 Install dependencies 4
01.2 Configure secrets 4
01.3 Create the database 4
01.4 Start the worker 5
01.5 Log in 5

Guides

02 Deploy to Cloudflare 6
02.1 Create the database 6
02.2 Set the signing secret 6
02.3 Migrate and deploy 6
02.4 Verify 7

Reference

03 API reference 8
03.1 Authentication 8
03.2 Users and customers 8
03.3 Recipes 9
03.4 Ingredients 9
03.5 Classifications 10
03.6 Reviews 11
03.7 Weekly menus 11
03.8 Health 12
03.9 Pagination and errors 12
04 Configuration 12
04.1 wrangler.jsonc 12
04.2 Environment values 13
04.3 npm scripts 13
04.4 seed.sql 13

Concepts

05 **Data model** 15

05.1 Tables and relationships 15

05.2 Deletes 16

05.3 Derived values 16

05.4 Nutrition 16

06 **Authentication** 16

06.1 Registration 17

06.2 Login 17

06.3 The requireUser middleware 17

06.4 Choosing the secret 17

06.5 Why PBKDF2 17

07 **From Flask to Workers** 18

07.1 Stack 18

07.2 Kept from the 2021 API 18

07.3 Changed on purpose 18

07.4 What stayed behind 19

Develop

08 **Test** 20

08.1 Run the suite 20

08.2 How it works 20

08.3 Adding a test 21

hello-fresh API

hello-fresh API manages weekly menus, recipes, ingredients, customers and their reviews. It runs as a Cloudflare Worker and stores data in *D1*. A rewrite of a 2021 Flask service, it keeps the original API contract. See From Flask to Workers for what changed.

00.1

What it does

- Stores recipes with ingredients, nutrition, directions and a *Classification*.
- Puts recipes into the *Weekly menu* for each week and scores the week from customer reviews.
- Secures every endpoint except registration and login with a *JWT* in the `x-access-tokens` header.
- Validates every request body with Zod and returns JSON errors.



00.2

Quick start

RUN IT LOCALLY

```
npm install
cp .dev.vars.example .dev.vars
npm run db:migrate:local
npm run db:seed:local
npm run dev
```

The worker serves `http://localhost:8787`. Log in with the seeded account `user` / `password`:

FIRST REQUEST

```
TOKEN=$(curl -s -X POST http://localhost:8787/api/v1/login -u user:password | jq -r
.token)
curl -s http://localhost:8787/api/v1/recipes -H "x-access-tokens: $TOKEN"
```

Classification
A recipe category, such as Meat & Veggis. A recipe belongs to one classification.

D1
Cloudflare's serverless SQLite database. The binding name in this project is DB.

JWT
A JSON Web Token. The API signs HS256 tokens with SECRET_KEY and accepts them in the x-access-tokens header.

Mapping
A row in weekly_recipe_map that puts a recipe into a weekly menu.

Soft delete
A delete that keeps the row and clears is_publish or is_active, so list endpoints stop returning it.

Weekly menu
A calendar week of recipes, identified by a unique week number.

SEE ALSO

- Install and run

- API reference

SECTION

Start

Install it and run it for the first time.

01

Install and run

Set up the worker locally with a D1 database, seed data and a running dev server.

You need Node.js 20 or newer and npm. The steps below give you a worker on `http://localhost:8787` with a local D1 database and a seeded demo user.

01.1

Install dependencies

INSTALL

```
npm install
```

`npm install` runs `wrangler types`, which generates `worker-configuration.d.ts` from `wrangler.jsonc` and `.dev.vars`.

01.2

Configure secrets

LOCAL SECRETS

```
cp .dev.vars.example .dev.vars
```

`.dev.vars` holds `SECRET_KEY`, the HMAC key for login tokens. wrangler dev loads it automatically and never commits it. Set a different value for anything you keep.

01.3

Create the database

MIGRATE AND SEED

```
npm run db:migrate:local
npm run db:seed:local
```

The first command applies `migrations/*.sql` to the local D1 database. The second loads `seed.sql`: two recipes with ingredients, week 46 with both recipes mapped, one customer review, and the accounts `user` / `password` (API user) and `alice` / `password` (customer).

01.4

Start the worker

DEV SERVER

```
npm run dev
```

wrangler serves the worker and the local database. Check it is up:

HEALTH CHECK

```
curl -s http://localhost:8787/healthz
```

TEXT

```
{"status":"ok"}
```

01.5

Log in

GET A TOKEN

```
TOKEN=$(curl -s -X POST http://localhost:8787/api/v1/login -u user:password | jq -r
.token)
curl -s http://localhost:8787/api/v1/recipes -H "x-access-tokens: $TOKEN"
```

> RESET THE LOCAL DATABASE
>
> Delete `.wrangler/state` to start again. Both commands above recreate everything.

SEE ALSO

- Configuration
- API reference

SECTION

Guides

Steps for one task each.

02

Deploy to Cloudflare

Create the D1 database, set the signing secret and ship the worker.

You need a Cloudflare account and `wrangler` logged in (`npx wrangler login`).

02.1

Create the database

CREATE D1

```
npx wrangler d1 create hello-fresh
```

wrangler prints a `database_id`. Paste it into `d1_databases` in `wrangler.jsonc`, replacing the placeholder `00000000-0000-0000-0000-000000000000`.

02.2

Set the signing secret

PRODUCTION SECRET

```
npx wrangler secret put SECRET_KEY
```

Paste a long random value when prompted. This key signs every *JWT*; changing it invalidates existing tokens.

02.3

Migrate and deploy

SHIP IT

```
npm run db:migrate:remote
npm run deploy
```

`db:migrate:remote` applies `migrations/*.sql` to the production database. `deploy` uploads the worker and prints its URL.

02.4

Verify

CHECK THE DEPLOYMENT

```
curl -s https://hello-fresh-api.<your-subdomain>.workers.dev/healthz
```

TEXT

```
{"status":"ok"}
```

> SEED DATA IS LOCAL ONLY
>
> `seed.sql` is for development. To seed production, review it first and run `npm run db:seed:remote`.

SEE ALSO

- Configuration

SECTION

Reference

Every command, option and file format.

03

API reference

Every endpoint with its method, body fields and status codes.

All paths live under `/api/v1`. Every endpoint except `POST /api/v1/register`, `POST /api/v1/login` and `GET /healthz` needs an `x-access-tokens` header with a *JWT*. Requests and responses are JSON.

03.1

Authentication

METHOD	PATH	BODY	RESPONSES
POST	`/api/v1/register`	`name`, `password`	200 `{message}`; 409 name taken; 422 missing field
POST	`/api/v1/login`	HTTP Basic (`Authorization: Basic ...`)	200 `{token}`; 401 bad credentials

A failed login sends `WWW-Authenticate: Basic realm="login required"`. The token expires after 30 minutes. Send it as `x-access-tokens: <token>`; a missing or invalid token gets 403 `{message}`.

03.2

Users and customers

METHOD	PATH	BODY	RESPONSES
GET	`/api/v1/user`		200 `{users, meta}`
GET	`/api/v1/customer`		200 `{users, meta}`

METHOD	PATH	BODY	RESPONSES
GET	`/api/v1/customer/id/:id`		200 `{users, meta}`; 404
POST	`/api/v1/customer`	`name`, `password`	200 `{message}`; 409 name taken; 422 missing field
PUT	`/api/v1/customer/id/:id`	`name`, `admin`, `is_active`, `password` (any subset)	200 `{customer, meta}`; 400; 404
DELETE	`/api/v1/customer/id/:id`		204 [[term:Soft delete

`GET /api/v1/user` returns `{public_id, name, admin}` per user. Customer entries add `id` and `is_active`. Password hashes never appear in a response.

03.3

Recipes

METHOD	PATH	BODY	RESPONSES
GET	`/api/v1/recipes`		200 `{recipes, meta}`; 404 when no published recipe exists
GET	`/api/v1/recipes/:page`		200; 404 when the page is empty
GET	`/api/v1/recipes/id/:id`		200 `{recipes, meta}`; 404
POST	`/api/v1/recipes`	see below	201 `{Recipe, meta}`; 400 unknown classification; 409 name taken; 422 invalid
PUT	`/api/v1/recipes/id/:id`	any subset of the create fields	200 `{Recipe, meta}`; 400; 404
DELETE	`/api/v1/recipes/id/:id`		204 soft delete; 400 mapped to a menu; 404

Create fields: `recipe_name` (required), `classification` (required, a classification id), `description`, `num_of_servings` (default 2), `cook_time` (default 30), `directions`, `is_publish` (default false), `nutrition` (object of strings, default `{}`).

A recipe response carries `classification` (its id), `nutrition` (object), `has_links` (true when a *Mapping* references it) and `ingredients` (see below).

03.4

Ingredients

METHOD	PATH	BODY	RESPONSES
GET	`/api/v1/ingredient`		200 `{Ingredient, meta}`; 404 when empty
GET	`/api/v1/ingredient/:page`		200; 404 when the page is empty
GET	`/api/v1/ingredient/id/:id`		200; 404
POST	`/api/v1/ingredient`	`ingredient_name`, `recipe` (recipe id), `unit`, `count`	201; 409 duplicate name for the recipe or unknown recipe
PUT	`/api/v1/ingredient/id/:id`	any subset of the create fields	200; 400; 404
DELETE	`/api/v1/ingredient/id/:id`		204 hard delete; 404

`ingredient_name` plus `recipe` must be unique together.

03.5

Classifications

METHOD	PATH	BODY	RESPONSES
GET	`/api/v1/recipes-classification`		200 `{recipes-classification, meta}`; 404 when empty
GET	`/api/v1/recipes-classification/:page`		200; 404 when the page is empty
GET	`/api/v1/recipes-classification/id/:id`		200; 404
POST	`/api/v1/recipes-classification`	`name`, `is_publish`	201; 409 name taken; 422 invalid
PUT	`/api/v1/recipes-classification/id/:id`	`name`, `is_publish`	200; 400; 404
DELETE	`/api/v1/recipes-classification/id/:id`		204 soft delete; 400 recipes reference it; 404

`name` holds at most 35 characters.

03.6

Reviews

Recipe reviews and menu reviews share one shape. The foreign key field is `recipe` (a recipe id) or `menu` (the id of a *Weekly menu*); `customer` is a customer id.

METHOD	PATH	BODY	RESPONSES
GET	`/api/v1/recipe-review`		200 `{recipe-review, meta}`; 404 when empty
GET	`/api/v1/recipe-review/:page`		200; 404
GET	`/api/v1/recipe-review/id/:id`		200; 404
POST	`/api/v1/recipe-review`	`review`, `summary`, `recipe`, `customer`, `score` (1 to 5, default 5), `is_publish`	201; 409 one review per customer and recipe, or unknown ids; 422 invalid
PUT	`/api/v1/recipe-review/id/:id`	`review`, `summary`, `score`, `is_publish`	200; 400; 404
DELETE	`/api/v1/recipe-review/id/:id`		204 soft delete; 404

Replace `recipe-review` with `weekly-menu-review` and `recipe` with `menu` for menu reviews. A response entry names its recipe or menu and its customer, and carries `score`, `summary`, `pub_date` and `is_publish`.

03.7

Weekly menus

METHOD	PATH	BODY	RESPONSES
GET	`/api/v1/weekly-menu`		200 `{weekly-menu, meta}`; 404 when empty
GET	`/api/v1/weekly-menu/:page`		200; 404 when the page is empty
GET	`/api/v1/weekly-menu/id/:id`		200; 404
POST	`/api/v1/weekly-menu`	`week_number`, `is_publish` (default true)	201; 409 week number taken; 422 invalid
PUT	`/api/v1/weekly-menu/id/:id`	`week_number`, `is_publish`	200; 400; 404
DELETE	`/api/v1/weekly-menu/id/:id`		204 soft delete; 404

A menu response carries `recipes` (its mappings) and `score` (the mean review score, 0 with no reviews).

METHOD	PATH	RESPONSES
POST	`/api/v1/weekly-menu/id/:id/recipe/id/:recipeId`	201 maps the recipe; 409 already mapped; 404 unknown id
DELETE	`/api/v1/weekly-menu/id/:id/recipe/id/:recipeId`	202 unmaps; 404 not mapped or unknown id

03.8

Health

METHOD	PATH	RESPONSES
GET	`/healthz`	200 `{status: "ok"}`, no auth

03.9

Pagination and errors

List endpoints return 10 rows per page. `meta` carries `page`, `per_page` and `page_url`. An empty list answers 404 with `{status, url, error}`, matching the 2021 API. Validation failures answer 400 or 422 with `{error}` and a `details` object naming each bad field. Unhandled failures answer 500 with `{status, url, error}`.

SEE ALSO

- Data model
- Configuration

04

Configuration

Every file, script and environment value the project reads.

04.1

wrangler.jsonc

KEY	VALUE	MEANING
`name`	`hello-fresh-api`	The deployed worker name.
`main`	`src/index.ts`	The entry module.
`compatibility_date`	`2026-09-01`	The runtime behaviour set.
`d1_databases[].binding`	`DB`	The binding name the code reads as `env.DB`.

KEY	VALUE	MEANING
`d1_databases[].database_name`	`hello-fresh`	The D1 database name.
`d1_databases[].database_id`	placeholder	Replace with the id from `wrangler d1 create` before deploying.
`d1_databases[].migrations_dir`	`migrations`	Where `d1 migrations apply` finds SQL files.
`observability.enabled`	`true`	Sends worker logs to the Cloudflare dashboard.

04.2 Environment values

NAME	WHERE	MEANING
SECRET_KEY	`.dev.vars` locally, `wrangler secret` in production	The HMAC-SHA256 key that signs login tokens. Required.

`.dev.vars` is gitignored. Copy `.dev.vars.example` to create it.

04.3 npm scripts

SCRIPT	WHAT IT DOES
`prepare`	Runs `wrangler types` to generate `worker-configuration.d.ts`. Runs on `npm install`.
`dev`	Serves the worker and a local D1 database on port 8787.
`deploy`	Uploads the worker to Cloudflare.
`typecheck`	Runs `tsc --noEmit`.
`test`	Runs the Vitest suite inside workerd.
`db:migrate:local`	Applies `migrations/*.sql` to the local database.
`db:migrate:remote`	Applies `migrations/*.sql` to the production database.
`db:seed:local`	Loads `seed.sql` into the local database.
`db:seed:remote`	Loads `seed.sql` into the production database.

04.4 seed.sql

The seed creates these accounts and rows. It is for development only.

WHAT	VALUE
API user	`user` / `password`
Customer	`alice` / `password`
Classifications	Meat & Veggis, Veggie (published), Family (unpublished)
Recipes	Duck Breast with Cheesy Chive Mash, Southeast Asian Chicken Coconut Soup, each with two ingredients
Weekly menu	Week 46 with both recipes mapped and one 4-star review
Week 47	Empty menu

Password hashes use PBKDF2-SHA256 with a fixed demo salt, so the file reproduces the same rows every run. Never load it into a database the internet can reach.

SEE ALSO

- Install and run
- Test

SECTION

Concepts

How it works and why it is built this way.

05

Data model

The nine DI tables, their relationships and how deletes work.

The schema in `migrations/0001_init.sql` has nine tables. Every id is an `INTEGER PRIMARY KEY AUTOINCREMENT`.

05.1

Tables and relationships

TABLE	PURPOSE	REFERENCES
`users`	API accounts that log in and receive tokens	
`customers`	People who write reviews; separate table from `users`	
`recipe_classifications`	Categories such as Meat & Veggis	
`recipes`	Recipes with nutrition (a JSON text column)	`recipe_classifications`
`ingredients`	Ingredient lines with unit and count	`recipes`
`weekly_menus`	One row per `week_number`	
`weekly_recipe_map`	*Mapping* of a recipe into a week	`weekly_menus`, `recipes`
`recipe_reviews`	A customer scores a recipe	`recipes`, `customers`
`weekly_menu_reviews`	A customer scores a week	`weekly_menus`, `customers`

Uniqueness rules beyond the primary keys:

TABLE	UNIQUE ON
`users`, `customers`	`public_id`, `name`
`recipes`	`recipe_name`
`recipe_classifications`	`name`
`ingredients`	`ingredient_name` + `recipe_id`
`weekly_menus`	`week_number`
`recipe_reviews`	`recipe_id` + `customer_id`
`weekly_menu_reviews`	`menu_id` + `customer_id`

Review scores carry a `CHECK (score BETWEEN 1 AND 5)` constraint, so an out-of-range score fails the insert even if a route missed it.

05.2

Deletes

Two kinds of delete exist:

- ***Soft delete*** for recipes, classifications, reviews and menus: the row stays and `is_publish` becomes 0. List endpoints filter on `is_publish = 1`; reads by id still find the row.
- **Hard delete** for ingredients: the row goes.

Deleting is refused with 400 when other rows depend on the resource: a recipe that appears in a *Mapping*, or a classification that recipes reference. Customers soft delete through `is_active` instead of `is_publish`.

05.3

Derived values

A recipe response carries `has_links` (true when a mapping references it) and its full ingredient list. A weekly menu response carries `recipes` (its mappings) and `score`, the mean of its menu reviews, or 0 with no reviews. Both are computed per request, not stored.

05.4

Nutrition

The 2021 Postgres schema stored nutrition as an hstore. D1 has no hstore, so `nutrition` is a JSON text column. Routes parse it on read and stringify it on write, so the API keeps accepting and returning a JSON object.

SEE ALSO

- API reference
- Authentication

06

Authentication

How registration, login and token checks work, and what they are built on.

The API authenticates with a *JWT* signed HS256. `src/auth.ts` implements signing, verification and password hashing on the Web Crypto API, with no crypto dependency.

06.1

Registration

`POST /api/v1/register` takes `name` and `password`. The route hashes the password with PBKDF2-SHA256, 100 000 iterations, a 16-byte random salt, and stores it as `pbkdf2$<iterations>$<salt>$<hash>`. It also generates a `public_id` with `crypto.randomUUID()`. A duplicate name answers 409.

06.2

Login

`POST /api/v1/login` reads HTTP Basic credentials. The route loads the user, re-derives the hash and compares it in constant time. On success it signs a token with `SECRET_KEY`:

JSON

```
{
  "public_id": "1b671a64-40d5-491e-99b0-da01ff1f3341",
  "iat": 1764000000,
  "exp": 1764001800
}
```

The token lives 30 minutes. Verification recomputes the HMAC over the signed parts, compares it in constant time, then checks `exp`.

06.3

The requireUser middleware

`src/index.ts` applies `requireUser` to every route except `register`, `login` and `healthz`. The middleware:

1. Reads `x-access-tokens` from the request. Missing: 403 `{message: "a valid token is missing"}`.
2. Verifies the signature and expiry. Invalid: 403 `{message: "token is invalid"}`.
3. Loads the user by `public_id`. Deleted: 403.
4. Puts the user on the Hono context for the route handler.

06.4

Choosing the secret

`SECRET_KEY` comes from the environment: `.dev.vars` under wrangler dev, `wrangler secret` in production. Anyone holding the key can mint tokens for any user, so treat it like a password and rotate it by deploying a new value, which invalidates every existing token.

06.5

Why PBKDF2

The 2021 service used werkzeug hashes (scrypt on modern versions). Workers expose no scrypt or bcrypt, so the rewrite uses PBKDF2, the hash Web Crypto provides. The stored format records its iteration count, so raising `PBKDF2_ITERATIONS` later lets `verifyPassword` still check old hashes.

SEE ALSO

- From Flask to Workers
- API reference

07

From Flask to Workers

What the 2026 rewrite changed and which parts of the 2021 contract it kept.

The 2021 implementation was Python 3, Flask, Peewee and Postgres behind Docker Compose, with pytest and a Postman collection. The 2026 rewrite targets Cloudflare Workers and D1 and drops Docker entirely.

07.1

Stack

LAYER	2021	2026
Runtime	Flask dev server in Docker	Cloudflare Workers
Routing	Flask blueprints-free routes	Hono
Database	Postgres + hstore	D1 (SQLite, nutrition as JSON)
Auth	PyJWT + werkzeug password hashes	Web Crypto: HS256 JWTs, PBKDF2 hashing
Validation	ad-hoc dictionary checks	Zod schemas per route
Tests	pytest against a Docker Compose Postgres	Vitest inside workerd against real D1

07.2

Kept from the 2021 API

- Paths, including the `/recipes/:page` and `/recipes/id/:id` shapes.
- Response keys: `Recipe`, `Ingredient`, `recipes-classification`, `recipe-review`, `weekly-menu-review`, `WeeklyMenu`, `users`, with the same capitalisation.
- Pagination of 10 rows per page with `meta.page`, `meta.per_page` and `meta.page_url`.
- Soft deletes through `is_publish`, customer deactivation through `is_active`, and 400 for resources other rows depend on.
- 404 for an empty list or a missing row, and a 30-minute token.

07.3

Changed on purpose

CHANGE	WHY
Password hashes never appear in a response	The 2021 user and customer listings returned `password` hashes.
PBKDF2 instead of werkzeug hashing	Workers expose no scrypt or bcrypt. See Authentication.
Duplicate registration answers 409, not 403	A conflict is not a forbidden request.
Validation answers 422 or 400 with JSON	The 2021 error paths could return HTML.
`WWW-Authenticate` with the correct header name	The 2021 code sent `WWW.Authentication`, which is not a header.
PUT answers 200 everywhere	The 2021 weekly-menu PUT answered 204 with a body.
`nutrition` is a JSON object	D1 has no hstore.

CHANGE	WHY
Login reads `Authorization: Basic`	Same as 2021 in effect; the old header name check was unreliable.

07.4

What stayed behind

`data/` holds the assignment's sample payloads, and `PostmanCollectionE2ETestDev.json` the 2021 E2E collection. Both still describe this API.

SEE ALSO

- Data model
- Test

SECTION

Develop

Build, test and release the project.

08

Test

Run the Vitest suite, which executes inside workerd against a real D1 database.

`npm test` runs the suite in `test/` with the Cloudflare Workers Vitest integration. Tests execute inside workerd, the same runtime that serves production, against a real local D1 database. No Docker and no mocks for storage.

08.1

Run the suite

TEST AND TYPECHECK

```
npm test
npm run typecheck
```

TEXT

```
Test Files  1 passed (1)
     Tests  7 passed (7)
```

08.2

How it works

Three files set the stage:

FILE	ROLE
`vitest.config.ts`	Loads `@cloudflare/vitest-plugin`, reads `migrations/` and exposes them as the `TEST_MIGRATIONS` binding.
`test/apply-migrations.ts`	Setup file that applies the migrations to D1 before any test runs.

FILE	ROLE
`test/helpers.ts`	`api()`, which calls the Hono app with real bindings, and `registerAndLogin()`, which returns headers with a valid token.

Each test creates its own user with a unique name through `registerAndLogin()`, so tests never depend on each other's data.

08.3

Adding a test

Import the helpers and call `api()` with a path and a fetch init:

TS

```ts
import { describe, expect, it } from 'vitest';
import { api, registerAndLogin, uniqueName } from './helpers';

describe('recipes', () => {
  it('rejects an unknown classification', async () => {
    const headers = await registerAndLogin(uniqueName('user'));
    const res = await api('/api/v1/recipes', {
      method: 'POST',
      headers,
      body: JSON.stringify({ recipe_name: 'Ghost', classification: 9999 }),
    });
    expect(res.status).toBe(400);
  });
});
```

The suite covers the auth guards (403, 401) and the registration paths (200, 409, 422). One end-to-end flow creates a classification, a recipe and an ingredient, maps and unmaps a weekly menu, reviews both, and walks every delete path.

SEE ALSO

- Configuration
- Deploy to Cloudflare