Petto

Petto is a Progressive Web Application designed to help reunite lost pets with their owners.

Copilot Instructions

ALWAYS MUST FOLLOW RULES FROM INSTRUCTIONS FILE to provide correct dependencies versions and avoid deprecations and errors.

Features

- **Login, Logout, Register and Password recovery**
- **Create a Collection of Pets**
  - Add, Edit, Read, Remove a Pet
  - Upload up to 5 images per pet (first image is treated as the cover)
  - Reorder images before submission; backend stores additional images in discrete fields (`picture2`..`picture5`) and exposes a unified `pictures[]` array
  - Generate Pet QR Code in square or circle canvas to print and glue in the pet medal
- **Generate Pet Flyers and A4 Posters**
  - The user could create print ready flyers of lost pet. It must include pet picture, description, indications and pet QR code.

Monorepo Structure

Petto is organized as a monorepo using pnpm workspaces:

- `backend/`: FastAPI backend (Python)
- `apps/frontend/`: SvelteKit frontend (TypeScript)
- `packages/`: (optional) Shared libraries or modules (e.g., `packages/shared/`)

Workspace configuration: see `pnpm-workspace.yaml`

Technologies

- **Backend:** Python 3, FastAPI, SQLite (scalable to MySQL/PostgreSQL)
- **Frontend:** SvelteKit 5, Prisma ORM, Tailwind CSS, Paraglide i18n

Backend Setup & Start Guide

Prerequisites

- Python 3.8+
- (Recommended) Virtual environment tool: `venv` or `virtualenv`

Installation

1. Navigate to the backend app directory:

```
cd backend
```

2. Create and activate a virtual environment (optional but recommended):

```
python3 -m venv venv
source venv/bin/activate
```

3. Install dependencies:

```
pip install -r requirements.txt
```

Running the Backend

Start the FastAPI server using Uvicorn:

```
uvicorn main:app --reload
```

- The API will be available at `http://127.0.0.1:8000`
- Health check endpoint: `GET /_health`

Project Structure

- `backend/main.py`: FastAPI application entry point
- `backend/routers/`: API route modules (users, pets, qrcode, banners, pet_location)
- `backend/models.py`: Database models
- `backend/database.py`: Database configuration
- `apps/frontend/`: SvelteKit frontend app

Autenticación y autorización (end-to-end)

Este proyecto implementa autenticación basada en JWT con tokens de acceso (corto plazo) y tokens de refresco (largo plazo), más autorización en rutas del frontend. El flujo cubre inicio de sesión, acceso a páginas protegidas, refresco automático del token y cierre de sesión.

- Backend (FastAPI):
  - `POST /api/login`: emite `access_token` y `refresh_token`.
  - `GET /api/users/me`: devuelve el usuario autenticado (requiere `Authorization: Bearer <access_token>`).
  - `POST /api/token/refresh`: intercambia un `refresh_token` válido por un nuevo `access_token` (y opcionalmente nuevo `refresh_token`).
- Frontend (SvelteKit 5):

- Helper centralizado `src/lib/utils/api.ts`: añade `Authorization` cuando `requireAuth: true`, usa el `fetch` de `load` si se le pasa, y refresca automáticamente al recibir 401.
- Layout `routes/+layout.ts`: durante `load`, consulta `api/users/me` mediante el helper; si no hay sesión/tokens válidos, las páginas protegidas redirigen a `/login` con `returnUrl`.
- Guardia SSR opcional `src/lib/utils/protect-route.ts`: para +page.ts protegidas, valida la sesión del layout y redirige si no existe.
- Guardia de cliente `src/lib/components/ProtectedRoute.svelte`: asegura que el usuario esté autenticado al renderizar contenido protegido en el cliente.
- Almacén de sesión `src/lib/stores/session.ts`: guarda el usuario actual; los tokens se almacenan en `localStorage`.

Secuencia completa (Mermaid)

```
sequenceDiagram
   autonumber
   actor U as Usuario
   participant FE as SvelteKit (Frontend)
   participant AH as API Helper (api.ts)
   participant LS as TokenStore (localStorage)
   participant BE as Backend (FastAPI)

   Note over U,FE: Acceso a página protegida (ej. /pets)
   U->>FE: Navega a ruta
   FE->>AH: +layout load → GET api/users/me { requireAuth: true, fetchFn }
   alt Sin tokens en LS
      AH-->>FE: 401/No auth
      FE-->>U: Redirige a /login?returnUrl=...
   else Con access_token en LS
      AH->>BE: GET /api/users/me (Bearer access_token)
      alt 200 OK
         BE-->>AH: Datos de usuario
         AH-->>FE: user
         FE->>LS: (Opcional) tokens ya presentes
         FE-->>U: Renderiza página
      else 401 (expirado)
         BE-->>AH: 401 Unauthorized
         AH->>BE: POST /api/token/refresh { refresh_token }
         alt 200 OK (refresh válido)
            BE-->>AH: Nuevo access_token (+ opcional refresh_token)
            AH->>LS: Actualiza tokens
            AH->>BE: Reintenta GET /api/users/me
            BE-->>AH: 200 + user
            AH-->>FE: user
            FE-->>U: Renderiza página
         else 401/400 (refresh inválido)
            BE-->>AH: Error de refresh
            AH->>LS: Limpia tokens
            FE-->>U: Redirige a /login?returnUrl=...
         end
      end
```

```
    end

    Note over U,FE: Inicio de sesión
    U->>FE: Envía credenciales
    FE->>BE: POST /api/login { email, password }
    alt Credenciales válidas
       BE-->>FE: access_token + refresh_token
       FE->>LS: Guarda tokens
       FE-->>U: Redirige a returnUrl o home
    else Inválidas
       BE-->>FE: 401 Unauthorized
       FE-->>U: Muestra error
    end

    Note over U,FE: Cierre de sesión
    U->>FE: Click en logout
    FE->>LS: Elimina access_token y refresh_token
    FE-->>U: Redirige a /login
```

Detalles y mejores prácticas

- En `api.ts`, usa siempre `fetchFn` del `load` de SvelteKit cuando esté disponible para evitar warnings y mantener SSR consistente.
- Todas las llamadas autenticadas deberían pasar por el helper (`requireAuth: true`) para beneficiarse del refresco automático.
- Las páginas protegidas pueden combinar: comprobación en `+layout.ts` (hidrata la sesión), `protect-route.ts` para SSR y `ProtectedRoute.svelte` en cliente.
- En fallo de refresh, se limpian tokens y se redirige a `/login?returnUrl=...`.
- Los mensajes de UI e i18n se gestionan con Paraglide; el almacén de sesión mantiene el usuario actual.

Pet API Data Models

The Pet endpoints now use **explicit Pydantic schemas** instead of auto-generated ones. This gives us clearer versioning, validation control, and security guarantees.

Schema overview:

Schema	Purpose	Fields
`PetBase`	Shared core fields	`name`, `pet_type`, `picture`, `notes`, `status` (default `at_home`)
`PetCreate`	Creation payload (client -> server)	Inherits `PetBase` + `owner_id` (currently accepted but ignored in favor of authenticated user)
`PetUpdate`	Update payload (partial)	All fields optional (`name`, `pet_type`, `picture`, `notes`, `status`, `owner_id`)
`PetOut`	Response model	`id`, `owner_id`, plus all `PetBase` fields + `pictures[]` (ordered list; element 0 is cover)

Important security rule: the backend **always overrides** `owner_id` with the authenticated user on create/update. A future cleanup will remove `owner_id` from the create payload entirely once the frontend is updated.

Multi-image Model Mapping

Internally the database model maintains legacy `picture` plus optional `picture2` .. `picture5` columns for a maximum of 5 images. The API serializer normalizes these into:

```
PetOut.pictures = [picture, picture2, picture3, picture4,
picture5].filter(Boolean)
```

When creating or updating a pet the client can send both:

```
{
   "picture": "cover.jpg",            // must match pictures[0]
   "pictures": ["cover.jpg", "side.jpg", ...]
}
```

The backend maps array indices 1..4 into `picture2`..`picture5`. Extra elements beyond 5 are rejected at validation time. Missing or empty arrays fallback to a placeholder image on the frontend (`DEFAULT_PET_IMAGE`).

Frontend components (`PetCard`, `PetDetail`) now use a shared helper `getPetCover(pet)` from `src/lib/utils/pet.ts` to consistently select and normalize the cover image URL.

Enum values:

```
PetType   = Cat | Dog | Lizard | Hamster | Bird | Other
PetStatus = at_home | lost | found
```

Migration Note

Previously the backend used `pydantic_model_creator(Pet, ...)` to auto-generate `PetIn`/`PetOut`. These were replaced by explicit classes (`PetCreate`, `PetUpdate`, `PetOut`) in order to:

- Enforce ownership rules explicitly.
- Support partial updates cleanly.
- Avoid accidental field exposure if the ORM model gains internal attributes later.
- Provide stable documentation for external consumers.

If you still have clients posting `owner_id`, they will continue to work for now, but should be updated to omit it—treat it as deprecated.

Planned Follow-ups

- Remove `owner_id` from create form submissions (derive entirely server-side).
- Introduce a `PATCH /pets/{id}` endpoint using `PetUpdate` for semantic partial updates.
- Add a serializer helper to reduce repetition when constructing `PetOut` (partially complete with current normalize logic).
- Add tests covering ownership override, status transitions, and multi-image mapping (length >5 rejection, ordering persistence).

Frontend Setup & Start Guide

Technologies

- **Framework:** SvelteKit 5
- **Styling:** Tailwind CSS
- **ORM:** Prisma (for data modeling)
- **i18n:** Paraglide

Installation

1. Navigate to the frontend app directory:

```
cd apps/frontend
```

2. Install dependencies:

```
pnpm install
# or
npm install
# or
yarn install
```

Running the Frontend

Start the SvelteKit dev server:

```
pnpm run dev
# or
npm run dev
# or
yarn dev
```

- The app will be available at `http://localhost:5173` (default)

Building & Preview

To build for production:

```
pnpm run build
```

To preview the production build:

```
pnpm run preview
```

Project Structure

- `apps/frontend/src/routes/`: Main SvelteKit routes
- `apps/frontend/src/lib/paraglide/messages/`: i18n message files (auto-generated JS exports)
- `apps/frontend/app.css`: Tailwind CSS entry
- `apps/frontend/package.json`: Project dependencies and scripts

Developer Workflows

- Unit tests: `pnpm run test:unit` (Vitest)
- E2E tests: `pnpm run test:e2e` (Playwright)
- Lint/format: `pnpm run lint`, `pnpm run format`

i18n (Paraglide)

- Messages are imported as named exports and called as functions (e.g., `m.button_cancel()`).
- Locale switching: `setLocale('en')`, etc. Messages update reactively if used via Svelte stores.

Additional Notes

- The backend uses SQLite by default. For production, consider switching to MySQL or PostgreSQL.
- API documentation is available at `/docs` when the server is running.

Monorepo Commands

- Install all dependencies for all apps/packages:

  ```
  pnpm install
  ```

- Run backend:

  ```
  cd backend
  uvicorn main:app --reload
  ```

- Run frontend:

```
cd apps/frontend
pnpm run dev
```

Contributing

Pull requests and suggestions are welcome!

License

Specify your license here.