SmartTrip (Trip Recommendations) — Student Walkthrough

Audience: 3rd-year Industrial & Management student building a first full-stack app
Goal: Understand how this project is structured, how the parts communicate, and why these technology choices make sense.

---

1) What you built (big picture)

This repository is a **full-stack web application** in a single workspace (a "monorepo"):

- **Frontend (UI):** Next.js (React) in `src/` — shows search form, results, and trip details.
- **Backend (API):** Flask (Python) in `backend/` — exposes REST endpoints and runs the recommendation algorithm.
- **Database:** SQL database accessed by SQLAlchemy.
  - In production, designed for **PostgreSQL**.
  - In local development, it can fall back to **SQLite** if `DATABASE_URL` is not set.
- **Data seeding & tooling:** scripts and CSVs to populate trips/countries/tags.

The app works like this:

1. User fills out search preferences in the frontend.
2. Frontend sends those preferences to the backend (`POST /api/recommendations`).
3. Backend validates input, queries the database, scores trips, and returns results.
4. Frontend renders results and explains the match score.

---

2) Repository structure (what each folder/file is for)

At the project root you have:

- **`src/`** : The Next.js app (frontend).
  - **`src/app/`** : Pages/routes (Next.js App Router).
    - `src/app/search/page.tsx` : Search form UI (collects preferences).
    - `src/app/search/results/page.tsx` : Results UI (calls backend and renders recommendations).
    - `src/app/trip/[id]/page.tsx` : Trip detail page.
  - **`src/lib/api.ts`** : A typed API client wrapper around `fetch()` .
- **`public/`** : Static assets served by Next.js (icons, images, logos).
- **`backend/`** : Flask API + SQLAlchemy models + seed scripts.
  - `backend/app.py` : Flask app, endpoints, and recommendation algorithm.
  - `backend/models.py` : SQLAlchemy models (tables + relationships).
  - `backend/database.py` : DB engine + connection pool + session management.
  - `backend/seed.py` / `seed_from_csv.py` : create sample data.
- **`data/` + `*.csv`** (root): CSV data sources used to seed and validate content.
- **`package.json`** : Frontend dependencies/scripts (Next.js/React/Tailwind).
- **`backend/requirements.txt`** : Backend Python dependencies.
- **Deployment configs:** `render.yaml` , `Procfile` , etc.

Why this structure is common

- You keep **UI concerns** (layout, forms, components) separate from **business logic** (recommendation algorithm, database).
- You can deploy frontend and backend independently (typical production setup).
- It matches how teams work: frontend and backend can evolve in parallel with a stable API contract.

---

3) What is the frontend and why these choices?

3.1 Next.js (what it is)

Next.js is a framework built on top of React. It gives you:

- **Routing by folder structure** (your pages live in `src/app/...`).
- **Production build system** (bundling, optimization, minification).
- **Server rendering options** (SSR/SSG) if you need them later.
- A mature ecosystem and strong defaults.

Your project uses **Next.js 14** (see `package.json`).

3.2 React (why React?)

React is a UI library that makes it easier to build interfaces where:

- The UI changes based on state (filters, selections, loading states).
- You reuse components (badges, cards, sliders).
- You handle user interaction (forms, dropdowns, buttons).

This project's search experience is very stateful:

- Selected locations (countries/continents)
- Selected trip type (single)
- Selected themes (up to 3)
- Budget slider and duration range
- Loading states when fetching countries/tags/recommendations

React is well-suited for this because state updates naturally re-render the UI.

3.3 Tailwind CSS (the "wind" you asked about)

The "wind" is **Tailwind CSS**.

Tailwind is a "utility-first" CSS framework. Instead of writing large CSS files, you compose styles using class names:

- `bg-white` (background)
- `text-xl` (font size)
- `rounded-xl` (border radius)
- `shadow-lg` (shadow)
- `flex items-center justify-between` (layout)

Why use Tailwind?

- **Speed:** you design quickly without switching to CSS files.
- **Consistency:** design tokens (spacing, colors) are standardized.
- **Maintainability:** fewer one-off CSS rules.

In this repo you also have:

- **PostCSS** (`postcss`) and **Autoprefixer** (`autoprefixer`) to automatically handle browser compatibility.

3.4 Frontend dependencies (what's installed)

From `package.json` :

- `next` (Next.js framework)
- `react` , `react-dom`
- `tailwindcss` , `postcss` , `autoprefixer`
- `typescript` + React/Node types
- `lucide-react` (icon library)
- `clsx` (helps build conditional className strings)

---

4) What is the backend and why these choices?

4.1 Python (why Python for backend?)

Python is a common backend choice especially when your project includes **algorithms and data logic**, because:

- **Fast iteration:** you can prototype and change logic quickly.
- **Readable code:** easier to reason about, especially for scoring logic.
- **Great ecosystem:** strong libraries for data, testing, and scripting.

Tradeoffs vs other popular backend languages:

- **Node.js (JavaScript/TypeScript):**
  - Pros: one language across frontend/backend; large ecosystem.
  - Cons: complex data logic can become harder to maintain without discipline.
- **Java/Spring or C#/.NET:**
  - Pros: strong structure, enterprise tooling, performance.
  - Cons: heavier setup, slower iteration for small teams/projects.
- **Go:**
  - Pros: performance and concurrency.
  - Cons: slower iteration; less "batteries included" for rapid prototypes.

For *this* project, Python is a strong match because the recommendation engine is central.

4.2 Flask (why Flask?)

Flask is a lightweight Python web framework.

Why it fits here:

- Simple routing (`@app.route('/api/recommendations', methods=['POST'])`).
- Easy JSON APIs (`jsonify(...)`).
- Easy to deploy with Gunicorn.
- Great for a first full-stack project because the mental model is small.

4.3 SQLAlchemy (why an ORM?)

SQLAlchemy lets you:

- Define database tables as Python classes (`Trip` , `Country` , `Tag`).

- Express queries in Python while still generating SQL.
- Keep relationships consistent (e.g., a trip has a country, guide, tags).

ORMs help beginners because you don't have to write every query in raw SQL. But you still get performance tools (indexes, eager loading).

4.4 PostgreSQL (why Postgres?)

PostgreSQL is a popular relational database because:

- Strong reliability and data integrity (transactions, constraints).
- Great performance for relational queries.
- Works well with SQLAlchemy.
- Industry standard for production.

Your backend includes `psycopg2-binary`, which is the Postgres driver used by SQLAlchemy.

4.5 SQLite fallback (why it exists)

In `backend/database.py`:

- If `DATABASE_URL` is not set, it defaults to `sqlite:///./smarttrip.db`.

This is helpful because:

- SQLite requires no server to run locally.
- Quick development and testing.

Production typically uses Postgres (especially on platforms like Render).

---

5) How the frontend talks to the backend (the API contract)

5.1 The base URL

Frontend uses:

- `NEXT_PUBLIC_API_URL` environment variable, or
- fallback `http://localhost:5000`

This appears in:

- `src/lib/api.ts` (`API_BASE_URL`)
- `src/app/search/page.tsx` and `src/app/search/results/page.tsx` (`API_URL`)

Important concept:

- Frontend and backend are separate servers.
- Frontend must know the backend URL.

5.2 What is an API endpoint?

An endpoint is a URL on your backend that accepts a request and returns a response.

Examples in `backend/app.py`:

- `GET /api/health`
- `GET /api/locations`
- `GET /api/trip-types`

- `GET /api/tags`
- `POST /api/recommendations`

5.3 The recommendation call (most important request)

Frontend (results page) builds preferences from query parameters and sends:

- `POST /api/recommendations`
- JSON body with user preferences

Example payload (based on `src/app/search/results/page.tsx`):

```
{
  "selected_countries": [45, 67],
  "selected_continents": ["Asia"],
  "preferred_type_id": 3,
  "preferred_theme_ids": [12, 15],
  "min_duration": 10,
  "max_duration": 14,
  "budget": 12000,
  "difficulty": 2,
  "year": "2026",
  "month": "3"
}
```

Backend responds with:

- `data`: list of trips
- `match_score`: integer 0–100
- `match_details`: human-readable reasons
- metadata: `primary_count`, `relaxed_count`, `score_thresholds`, etc.

5.4 Why CORS exists

Because the frontend runs on one origin (e.g., `http://localhost:3000`) and the backend on another (`http://localhost:5000`), browsers enforce security rules.

`flask-cors` is used so the backend explicitly allows the frontend to call it.

---

6) How the recommendation algorithm is implemented (conceptual)

6.1 Hard filters vs scoring

In `backend/app.py`, the algorithm uses:

- **Hard filters:** eliminate trips that don't match minimum requirements.
- **Scoring:** rank the remaining trips by how well they match.

Why this is a good approach:

- Filters ensure results are not "nonsense".
- Scoring creates a ranking without requiring perfect matches.

6.2 Two-tier results: Primary + Relaxed

If strict filtering yields too few results, the system runs a relaxed search.

- **Primary tier:** strict filters, best results.
- **Relaxed tier:** wider filters, but with penalties (so they rank lower).

This prevents the “no results” frustration.

6.3 Eager loading (performance concept)

The backend uses `joinedload` / `selectinload` to avoid the N+1 query problem.

Why you should care:

- Without eager loading, one page load can create hundreds of database queries.
- With eager loading, it stays around a few queries.

---

7) How to run the project locally (mental model)

You run **two servers**:

1. Backend (Flask) — typically on port 5000
2. Frontend (Next.js) — typically on port 3000

Backend

- Install Python deps from `backend/requirements.txt`
- Set `DATABASE_URL` (optional; if not, SQLite is used)
- Run `python backend/app.py` (or `python app.py` if you run from `backend/`)

Frontend

- Install Node deps: `npm install`
- Set `NEXT_PUBLIC_API_URL=http://localhost:5000`
- Run `npm run dev`

---

8) Why this is a good “first full-stack” design

Clear separation of responsibilities

- **Frontend:** Collects inputs, displays outputs.
- **Backend:** Implements business rules and data access.
- **Database:** Stores the truth.

This separation makes debugging and learning much easier.

Strong real-world relevance

This architecture is similar to many real products:

- React/Next.js frontend
- API backend
- Postgres database
- ORM
- Deployed on managed hosting

---

9) “Why” answers (quick reference)

- **Why Python?** Fast iteration, readable, great for algorithms, huge ecosystem.
- **Why Flask?** Simple mental model; ideal for building a JSON API quickly.
- **Why Postgres?** Reliable, production-grade relational database.
- **Why SQLAlchemy?** Defines tables and relationships in Python; safer and easier than raw SQL for many tasks.
- **Why Next.js?** Routing + build pipeline + production best practices.
- **Why React?** Excellent for interactive UI with many states (filters, loading).
- **Why Tailwind?** Fast UI development and consistent styling.

---

10) Suggested learning path (use this project as your syllabus)

Step A — Understand the UI → API call

- Start at `src/app/search/page.tsx` (where the user chooses preferences)
- Follow `router.push()` to `src/app/search/results/page.tsx`
- Find the `fetch(` ${API_URL}/api/recommendations `)` call

Questions to ask:

- What preferences do we send?
- How are they encoded into URL params?
- How does the backend return the result structure?

Step B — Understand the backend endpoint

- Open `backend/app.py`
- Find `@app.route('/api/recommendations', methods=['POST'])`

Questions to ask:

- What does it validate?
- What does it filter?
- How does it score?
- How does it sort?

Step C — Understand the database model

- Open `backend/models.py`

Questions to ask:

- What tables exist?
- Why do we use a join table `TripTag` ?
- Which columns are indexed and why?

Step D — Understand the data flow end-to-end

Try one “happy path”:

1. Choose a continent + trip type + 2 themes.
2. Run recommendations.
3. Inspect the JSON response in browser network tab.
4. Match fields from response to UI rendering.

---

11) Report export to Word/PDF

Use the conversion guide:

- `REPORT_CONVERSION_GUIDE.md`

If you want a single “best command” for Word export:

```
pandoc PROJECT_WALKTHROUGH_STUDENT_GUIDE.md -o SmartTrip_Project_Walkthrough.docx --toc --
toc-depth=3 -s
```

---

12) Glossary (simple)

- **Frontend**: the UI users see.
- **Backend**: server logic; handles requests and data.
- **API**: contract between frontend and backend.
- **REST**: common API style: endpoints + JSON.
- **ORM**: maps database tables to code objects.
- **SQL**: language used to query relational databases.
- **CORS**: browser security policy; backend must allow frontend origin.
- **Environment variable**: configuration value set outside code (URLs, secrets).

Next step: If you want, I can generate a “chaptered” version of this guide (like a textbook), and add small exercises after each chapter (with expected answers) so you can learn by doing.