SmartTrip Project Files Guidebook

This guide gives you a **high-level, beginner-friendly tour of the main files and folders** in the SmartTrip project, with a focus on what you actually need to understand to work on the app end-to-end.

It does **not** explain the recommendation algorithm, events, Flask basics, or TypeScript basics in depth – those are already covered in the other Week 1–4 guides. Instead, this document is your **map of the project**: what lives where, why each file exists, and when you will (or won't) touch it.

---

1. Top-Level (Root) Files & Folders

These are the first things you see at the project root.

- **`backend/`**
  Python backend: Flask API, database models, scripts, recommendation engine, event tracking, background jobs.
- **`frontend/`**
  TypeScript/React frontend (Next.js App Router): pages, components, hooks, and client-side tracking.
- **`tests/`**
  All automated tests: backend, integration, and end-to-end UI tests (Playwright). Already covered in the testing guide, but important to remember the location.
- **`frontend/public/`**
  Static assets for the frontend: images, SVG icons, logos, etc. Anything here is served directly by Next.js.
- **`docs/`**
  All learning documentation and generated PDFs (the guides you've been reading), plus the PDF generator script.
- **`frontend/package.json`** / **`frontend/package-lock.json`**
  JavaScript/TypeScript dependencies and scripts for the frontend and tooling (like `md-to-pdf`).
  - You edit `package.json` when adding frontend libraries.
  - `package-lock.json` is auto-managed by npm.
- **`frontend/tsconfig.json`**
  TypeScript configuration for the frontend. Controls strictness, path aliases, and compiler options. You rarely touch this as a beginner, except to add path aliases if needed.
- **`frontend/tailwind.config.ts`**
  Tailwind CSS configuration: which files are scanned for class names, theme extensions, custom colors, etc.
- **`frontend/eslint.config.mjs`**
  ESLint configuration for linting frontend code. Helps keep code style consistent.
- **`frontend/next.config.js`**
  Next.js configuration: rewrites, image domains, experimental flags. Important if you need custom routing or backend proxying.
- **`frontend/postcss.config.mjs`**
  PostCSS configuration used by Tailwind; you almost never touch this directly.

- `pytest.ini`
  Pytest configuration file for backend tests: markers, test path settings, etc.
- `render.yaml`
  Render deployment configuration for the backend. Describes how the Python service is built and run in production.
- `run_tests.py`
  Convenience script to run the project's Python test suite or specific subsets. Useful when you want a one-command test run.
- `README.md`
  Top-level project README: high-level overview, setup instructions, and links to docs. Good starting point when you need a reminder of the big picture.
- `smarttrip_backup.dump`
  Database backup file (PostgreSQL dump). Not part of the application logic; used to restore data into a database instance.

---

2. Frontend Structure (`frontend/src/`)

The frontend is a Next.js App Router app written in TypeScript/React.

2.1 `frontend/src/app/` – Pages & Layout

These are the **actual pages** of the app. Next.js uses the folder structure under `app/` to define routes.

- `frontend/src/app/layout.tsx`
  Root layout for the whole app. Defines global HTML structure, `<head>` tags (metadata), and wraps all pages (e.g., global `<Header />`, `<Footer />`).
  - You'd touch this if you add global providers (context, theming) or layout changes that affect every page.
- `frontend/src/app/page.tsx`
  The homepage (`/`). Entry point for users: high-level marketing or featured trips, and links to search.
- `frontend/src/app/error.tsx`
  Global error boundary for top-level routes. Next.js uses this when an uncaught error happens.
- `frontend/src/app/globals.css`
  Global CSS imported into the app. Tailwind base styles, global resets, any non-Tailwind custom styles.
- `frontend/src/app/search/page.tsx`
  Search form page (`/search`).
  - Uses form state (`useState`) and controlled inputs to capture user preferences.
  - On submit, navigates to the results page with query parameters.
- `frontend/src/app/search/error.tsx`
  Error UI for the search route if something fails there specifically.
- `frontend/src/app/search/results/page.tsx`
  Search results page (`/search/results`).

    - Reads URL search params, calls the backend recommendation API, and displays the list of trips with scores and statuses.
    - Uses the recommendation algorithm output heavily (match_score, is_relaxed, etc.).
- **`frontend/src/app/search/results/loading.tsx`**
  Loading UI for results route. Next.js shows this while data is being fetched.
- **`frontend/src/app/trip/[id]/page.tsx`**
  Dynamic route for a single trip detail page (`/trip/123`).
    - Fetches trip data from backend by ID.
    - Uses tracking hooks (e.g., dwell time, trip view events).
- **`frontend/src/app/auth/page.tsx`**
  Authentication page for user login (Supabase OAuth).
- **`frontend/src/app/auth/callback/page.tsx`**
  OAuth callback handler that processes authentication redirects and links users to the tracking system.

2.2 `frontend/src/api/` – Modular API Client

The frontend API has been restructured into a modular architecture:

- **`frontend/src/api/client.ts`**
  Core API utilities: `apiFetch()` wrapper function that handles authentication, retry logic, error handling, and timeout management.
    - All API modules use this wrapper for consistent behavior.
    - `getAuthHeaders()` function adds JWT tokens from Supabase to requests.
- **`frontend/src/api/v2.ts`**
  V2 API endpoints: `getRecommendations()` , `getTemplates()` , `getOccurrences()` , `getTripById()` .
    - Handles recommendation requests and trip data fetching.
- **`frontend/src/api/events.ts`**
  Event tracking endpoints: `trackEventsBatch()` , `startSession()` .
    - Used by the tracking service to send events to the backend.
- **`frontend/src/api/resources.ts`**
  Resource endpoints: `getLocations()` , `getTripTypes()` , `getTags()` , `getGuides()` .
    - Fetches reference data (countries, trip types, themes, guides).
- **`frontend/src/api/analytics.ts`**
  Analytics endpoints: `getMetrics()` , `getDailyMetrics()` , `getTopSearches()` .
    - Retrieves recommendation metrics and analytics data.
- **`frontend/src/api/system.ts`**
  System endpoints: `healthCheck()` .
    - Health check and system status endpoints.
- **`frontend/src/api/types.ts`**
  TypeScript type definitions shared across API modules.

- `frontend/src/api/index.ts`
  Centralized exports for convenient importing of all API functions.

2.3 `frontend/src/lib/` – Frontend Utilities

- `frontend/src/lib/dataStore.tsx`
  Frontend state management helper (React Context API) for shared reference data (countries, trip types, theme tags).
  - Provides `DataStoreProvider` and hooks: `useCountries()`, `useTripTypes()`, `useThemeTags()`.
  - Caches data fetched from backend API, available to all components.
- `frontend/src/lib/supabaseClient.ts`
  Supabase client initialization and authentication helpers.
  - `getAccessToken()` – Gets JWT token from Supabase session.
  - `getCurrentUser()` – Gets current authenticated user.
- `frontend/src/lib/utils.ts`
  General utility functions used across the frontend.

2.4 `frontend/src/services/` – Business Logic Services

- `frontend/src/services/tracking.service.ts`
  Core frontend tracking module (Phase 1).
  - Manages anonymous ID and session ID via `localStorage`.
  - Queues tracking events, batches them (10 events or 5 seconds), and sends to `/api/events/batch`.
  - Provides functions like `trackPageView`, `trackSearchSubmit`, `trackTripClick`, `trackSaveTrip`, etc.
  - Handles page unload events via `sendBeacon()` for reliable delivery.

2.5 `frontend/src/hooks/` – Custom React Hooks

- `frontend/src/hooks/useTracking.ts`
  React hooks wrapper around tracking service:
  - `usePageView()` – Tracks page views automatically.
  - `useResultsTracking()` – Tracks search result impressions.
  - `useImpressionTracking()` – Tracks when trip cards enter viewport.
  - Exports tracking functions: `trackTripClick`, `trackSaveTrip`, `trackBookingStart`, etc.
  - When you want to track something from a component, you import from here.
- `frontend/src/hooks/useSearch.ts`
  Custom hook for managing search form state and preferences.
- `frontend/src/hooks/useSyncSearchQuery.ts`
  Hook for synchronizing search state with URL query parameters.
- `frontend/src/hooks/useUser.ts`
  Hook for managing authenticated user state.

2.6 `frontend/src/components/` – React Components

- **`frontend/src/components/features/`**
  Feature-specific components:
  - `search/` – Search page components (filters, actions, headers, loading states).
  - `TripResultCard.tsx` – Component for displaying trip results.
  - `LogoutConfirmModal.tsx`, `RegistrationModal.tsx` – User interaction modals.
- **`frontend/src/components/ui/`**
  Reusable UI components:
  - `ClearFiltersButton.tsx` – Button to reset filters.
  - `DualRangeSlider.tsx` – Range slider for budget/duration filters.
  - `SelectionBadge.tsx` – Badge component for selected items.
  - `TagCircle.tsx` – Circular tag/theme selector component.

2.7 `frontend/src/contexts/` – React Contexts

- **`frontend/src/contexts/SearchContext.tsx`**
  Context provider for managing search state across components.

2.8 `frontend/src/schemas/` – Zod Validation Schemas

- **`frontend/src/schemas/`**
  Zod schemas for runtime type validation of API responses:
  - `base.ts` – Base schemas and utilities.
  - `resources.ts` – Resource data schemas.
  - `trip.ts` – Trip data schemas.
  - `events.ts` – Event tracking schemas.
  - `analytics.ts` – Analytics data schemas.
  - `index.ts` – Centralized exports.

---

3. Backend Structure (`backend/`)

The backend is a Flask app with SQLAlchemy models and a modular architecture.

3.1 Core Application (`backend/app/`)

- **`backend/app/main.py`**
  Main Flask application entry point.
  - Initializes Flask app, configures CORS, registers blueprints.
  - Registers API blueprints: `api_v2_bp`, `events_bp`, `analytics_bp`, `resources_bp`, `system_bp`.
  - Handles database lifecycle (before_request, teardown_appcontext).
  - Root endpoint and error handlers.
- **`backend/app/core/database.py`**
  Database setup and session management.
  - Provides `engine`, `SessionLocal`, and scoped session `db_session`.
  - `init_db()` function initializes database connection.
  - You rarely change this once it's correct, but you might read it to understand how sessions are created.

- **`backend/app/core/auth.py`**
  JWT authentication and authorization.
  - `get_current_user()` – Verifies JWT token from Supabase, extracts user info.
  - `require_auth` decorator – Requires authentication for protected endpoints.
  - `optional_auth` decorator – Optional authentication (supports guest users).
- **`backend/app/core/config.py`**
  Application configuration and environment variable management.

3.2 Models (`backend/app/models/`)

- **`backend/app/models/trip.py`**
  SQLAlchemy ORM models for trip-related tables:
  - `Company` , `TripTemplate` , `TripOccurrence` (V2 schema).
  - `Country` , `Guide` , `TripType` , `Tag` (reference data).
  - Relationship definitions and `to_dict()` methods for serialization.
- **`backend/app/models/events.py`**
  SQLAlchemy models for event-related tables:
  - `User` , `Session` , `Event` , `EventType` , `TripInteraction` .
  - Understanding these helps you analyze analytics data in the DB.

3.3 API Routes (`backend/app/api/`)

The API is organized into feature-based blueprints:

- **`backend/app/api/v2/routes.py`**
  V2 API blueprint (`api_v2_bp`).
  - `POST /api/v2/recommendations` – Get personalized trip recommendations.
  - `GET /api/v2/trips/<id>` – Get individual trip details.
  - `GET /api/v2/templates/<id>` – Get trip template details.
  - `GET /api/v2/occurrences/<id>` – Get trip occurrence details.
  - Uses V2 schema (templates, occurrences, companies).
- **`backend/app/api/events/routes.py`**
  Events API blueprint (`events_bp`).
  - `POST /api/events/batch` – Batch upload user events.
  - `POST /api/events` – Single event tracking (alternative).
  - `POST /api/session/start` – Initialize user session.
  - `POST /api/user/identify` – Link anonymous user to authenticated user.
- **`backend/app/api/resources/routes.py`**
  Resources API blueprint (`resources_bp`).
  - `GET /api/locations` – Get all countries with continent info.
  - `GET /api/countries` – Get all countries (alternative endpoint).
  - `GET /api/trip-types` – Get all trip type categories.
  - `GET /api/tags` – Get all theme tags.
  - `GET /api/guides` – Get all tour guides.

- **`backend/app/api/analytics/routes.py`**
  Analytics API blueprint (`analytics_bp`).
  - `GET /api/metrics` – Get current recommendation metrics.
  - `GET /api/metrics/daily` – Get daily metrics breakdown.
  - `GET /api/metrics/top-searches` – Get most common search criteria.
  - `POST /api/evaluation/run` – Run algorithm evaluation scenarios.
- **`backend/app/api/system/routes.py`**
  System API blueprint (`system_bp`).
  - `GET /api/health` – Health check endpoint with database status.

3.4 Services (`backend/app/services/`)

- **`backend/app/services/events.py`**
  Event tracking service (Phase 1).
  - `EventService.process_event()` – Validates and stores events.
  - User resolution (anonymous vs authenticated).
  - Session management, trip interaction updates.
  - Defines `VALID_EVENT_TYPES` , `VALID_SOURCES` , and event processing logic.
- **`backend/app/services/recommendation/`**
  Recommendation algorithm package (modular structure):
  - `engine.py` – Main orchestration (`get_recommendations()` function).
  - `scoring.py` – Scoring algorithm implementation.
  - `filters.py` – Query building and filtering logic.
  - `relaxed_search.py` – Relaxed search expansion logic.
  - `constants.py` – Configuration and thresholds (`SCORING_WEIGHTS` , `SCORE_THRESHOLDS`).
  - `context.py` – Preference parsing and normalization.
  - `__init__.py` – Package exports (`get_recommendations`).

3.5 Schemas (`backend/app/schemas/`)

- **`backend/app/schemas/`**
  Pydantic schemas for request/response validation:
  - `base.py` – Base schemas and utilities.
  - `trip.py` – Trip data schemas (`TripOccurrenceSchema` , etc.).
  - `resources.py` – Resource data schemas.
  - `utils.py` – Schema utilities and helpers.

3.6 Recommender Module (`backend/recommender/`)

- **`backend/recommender/logging.py`**
  Logging helpers for recommendation requests (Phase 0).
  - `RecommendationLogger.log_request()` – Generates `request_id` , logs inputs and outputs for analysis.
  - Stores logs in `recommendation_requests` table for analytics.
- **`backend/recommender/metrics.py`**
  Aggregates performance metrics (e.g., response times, score distributions) from recommendation runs.

    - `MetricsAggregator` class computes metrics from logged requests.
    - `get_current_metrics()`, `aggregate_daily_metrics()`, `get_top_searches()` methods.
- **`backend/recommender/evaluation.py`**
  Scenario evaluator: runs predefined test scenarios to evaluate the quality of recommendations against expectations.
    - Connects to test cases like `tests/backend/test_05_recommender.py` and `tests/integration/test_recommendations.py`.
- **`backend/recommender/README.md`**
  Additional documentation focused on the recommender architecture, scenarios, and evaluation approach.

These files are essential if you want to **improve or debug the recommendation engine quality** beyond basic scoring changes.

3.7 Scheduler & Background Jobs

- **`backend/scripts/`**
  One-off and recurring maintenance scripts organized by category:
    - **`backend/scripts/analytics/`** – Analytics and metrics scripts:
        - `aggregate_daily_metrics.py` – Aggregates daily metrics for dashboards.
        - `aggregate_trip_interactions.py` – Recalculates `TripInteraction` counters.
    - **`backend/scripts/data_gen/`** – Data generation scripts:
        - `seed.py`, `seed_from_csv.py` – Seed the database with initial or CSV-based data.
    - **`backend/scripts/db/`** – Database management scripts:
        - `check_schema.py`, `verify_schema.py`, `verify_seed.py` – Sanity checks for DB schema and seed data.
        - `cleanup_sessions.py` – Clean up old or invalid sessions.
        - `export_data.py`, `import_data.py` – Data import/export operations.
    - **`backend/scripts/_archive/`** – Archived/legacy scripts (e.g., old scoring analysis).

  You typically **run** these scripts via `python -m backend.scripts.<category>.<name>` or via a scheduled job in production. They are not imported by the app at runtime (except where explicitly wired by a scheduler).

3.8 Migrations (`backend/migrations/`)

- **`backend/migrations/`**
  Database migration scripts for schema changes:
    - `_001_add_recommendation_logging.py` – Adds recommendation logging tables.
    - `_002_add_user_tracking.py` – Adds user and session tracking tables.
    - `_003_add_companies.py` – Adds company support.
    - `_004_refactor_trips_to_templates.py` – Migrates to V2 schema (templates/occurrences).
    - `_005_normalize_events_and_load_scenarios.py` – Normalizes events schema and loads scenarios.
    - `_006_add_properties_jsonb.py` – Adds JSONB properties column.
    - `run_schema_v2_migration.py` – Main migration runner script.

3.9 Scenarios & Personas (`backend/scenarios/`)

- **`backend/scenarios/README.md`**
  Explains predefined user personas and test scenarios used to evaluate recommendations.
- **`backend/scenarios/generated_personas.json`**
  Machine-generated persona data used in evaluation and analytics scripts. Helps simulate diverse user profiles.

---

4. Tests (`tests/`)

While there is a full Testing & Deployment guide, here is how the test tree relates to the rest of the project.

- **`tests/backend/`**
  - `test_01_db_schema.py` – Checks DB schema and constraints.
  - `test_02_api.py` – API endpoints behavior (trips, V2, recommendations).
  - `test_03_analytics.py` – Analytics, events, metrics.
  - `test_04_cron.py` – Scheduler and background jobs.
  - `test_05_recommender.py` – Recommendation algorithm behavior (scoring, filters, relaxed results, logging).
- **`tests/integration/`**
  - `test_algorithm.py` , `test_recommendations.py` – Integration tests that run the full recommendation stack.
  - `test_api_endpoints.py` – Multi-endpoint flows.
  - `test_event_tracking.py` – Frontend→backend→DB event flows.
  - `test_search_scenarios.py` – Realistic search flows combining filters, results, and scoring.
- **`tests/e2e/`**
  - `test_06_ui_desktop.py` , `test_07_ui_mobile.py` – UI layout and interaction checks on different viewports.
  - `test_08_search_flow.py` , `test_09_trip_details.py` , `test_10_visual_feedback.py` – End-to-end user flows (search → results → trip details → feedback), using Playwright.
- **`tests/conftest.py`**
  Global test configuration and fixtures:
  - Creates `client` (Flask test client), `db_session` , sample entities (`sample_trip` , `sample_country` , etc.).
  - Defines `pytest` markers like `@pytest.mark.api` , `@pytest.mark.recommender` , `@pytest.mark.e2e` .
- **`tests/requirements-test.txt`**
  Python dependencies needed to run tests only (pytest, coverage, Playwright, etc.).

---

5. Deployment & Infrastructure Files

5.1 Render (Backend)

- **`backend/requirements.txt`**
  Python dependency list for the backend (Flask, SQLAlchemy, etc.). Render uses this to build the image.

- `backend/Procfile`
  Tells Render (or Heroku-style platforms) how to start the web process, e.g.:
  `web: gunicorn app:app`
  You'd adjust this if the entrypoint changes (e.g., to a different app module).
- `backend/runtime.txt`
  Indicates the Python runtime version (e.g., `python-3.11.x`).
- `render.yaml` **(root)**
  High-level Render configuration: defines services (web, worker), environment variables, build/launch commands, and health check paths.

5.2 Vercel (Frontend)

No explicit Vercel config file is required by default; Vercel auto-detects Next.js. The main things you configure are:

- **Environment variable** `NEXT_PUBLIC_API_URL` – Points to your deployed backend.
- `frontend/next.config.js` – If you need rewrites, custom headers, or experimental flags.

5.3 Tooling & Quality

- `frontend/eslint.config.mjs` – Linting rules for TypeScript/React.
- `frontend/tailwind.config.ts` – Tailwind theme and content paths.
- `pytest.ini` – Pytest config (markers, options).
- `docs/generate_learning_pdfs.py` – Script that converts all learning Markdown guides into PDFs using `md-to-pdf`.

---

6. Which Files to Focus on First

Because this is your **first app**, here's a suggested priority for deeply understanding the codebase:

1. **Frontend Core**
   - `frontend/src/app/layout.tsx` – How the app shell is structured.
   - `frontend/src/app/page.tsx` – Homepage and how it links into search.
   - `frontend/src/app/search/page.tsx` – Search form and URL parameters.
   - `frontend/src/app/search/results/page.tsx` – How results are fetched and rendered.
   - `frontend/src/app/trip/[id]/page.tsx` – Trip detail flow.
2. **Frontend Utilities**
   - `frontend/src/api/` – Modular API client (client.ts, v2.ts, events.ts, resources.ts).
   - `frontend/src/services/tracking.service.ts` – Core tracking service.
   - `frontend/src/hooks/useTracking.ts` – Tracking hooks wrapper.
   - `frontend/src/lib/dataStore.tsx` – Shared state patterns for reference data.
3. **Backend Essentials**
   - `backend/app/main.py` – Main Flask app, blueprint registration, database lifecycle.
   - `backend/app/models/trip.py` – How Trips/Countries/Guides/Tags are represented.
   - `backend/app/models/events.py` – Event tracking models.
   - `backend/app/services/events.py` – Event tracking service.
   - `backend/app/api/events/routes.py` – Event tracking API endpoints.
   - `backend/app/api/v2/routes.py` – V2 API endpoints (recommendations).

- `backend/app/services/recommendation/` – Recommendation algorithm package.

4. **Quality & Testing**
   - `tests/backend/test_02_api.py` – What the API is expected to do.
   - `tests/backend/test_05_recommender.py` – How the algorithm is expected to behave.
   - `tests/integration/test_recommendations.py` – End-to-end recommendation flows.
5. **Deployment & Operations**
   - `render.yaml`, `backend/Procfile`, `backend/runtime.txt`, `backend/requirements.txt` – How the backend is deployed.
   - `frontend/package.json`, `frontend/next.config.js`, `frontend/tailwind.config.ts`, `frontend/tsconfig.json` – How the frontend is built and configured.

You **don't need to fully understand every script in `backend/scripts/` or every test file** on day one. Start with the core flows (search → recommend → view trip) and the files above, then expand to scripts and advanced tooling when you're comfortable.

As you work, you can always come back to this guidebook as a reference when you stumble upon a file you don't recognize.