Architecture Overview

Complete technical architecture documentation for the SmartTrip project, including technology choices, design patterns, API endpoints, and key files organized by data pipeline.

Table of Contents

Programming Languages

TypeScript (Frontend)

Why TypeScript was chosen: TypeScript provides static type checking for the frontend React application, catching errors at compile-time rather than runtime. This is critical for a production application where type safety prevents bugs and improves developer experience.

Pros vs Alternatives:

- **vs JavaScript**: Compile-time error detection, better IDE support, refactoring safety, self-documenting code through types
- **vs Flow**: Better ecosystem support, more mature tooling, larger community, Microsoft backing
- **vs Dart**: TypeScript integrates seamlessly with existing JavaScript/React ecosystem, no need to learn new language

Cons:

- Additional compilation step (minimal with Next.js)
- Learning curve for developers unfamiliar with types
- Can be verbose for simple cases

Interview Answer: "We chose TypeScript for the frontend because it provides static type safety that catches errors before they reach production. In a complex application with multiple API calls, state management, and user interactions, TypeScript helps prevent common bugs like passing wrong data types to functions or accessing undefined properties. The type system also serves as documentation, making the codebase more maintainable. While it adds a compilation step, Next.js handles this seamlessly, and the benefits far outweigh the minimal overhead. We use TypeScript 5 with strict mode enabled to maximize type safety."

Python (Backend)

Why Python was chosen: Python was selected for the backend due to its excellent ecosystem for data processing, ease of development, and strong libraries for database interaction (SQLAlchemy). The recommendation algorithm involves complex scoring logic that benefits from Python's readability and expressiveness.

Pros vs Alternatives:

- **vs Node.js**: Better for data processing and analytics, more mature ORM (SQLAlchemy), easier to write complex algorithms
- **vs Java**: Faster development, less boilerplate, better for rapid prototyping, simpler deployment
- **vs Go**: More libraries available, easier to learn, better for data manipulation, larger developer pool
- **vs Ruby**: Better performance, more widely used in enterprise, stronger typing support

Cons:

- Slower than compiled languages (not an issue for this use case)
- Global Interpreter Lock (GIL) limits true parallelism (not relevant for web requests)
- Can be less performant than Go/Rust for high-concurrency scenarios

Interview Answer: "Python was chosen for the backend because it excels at data processing and complex business logic. Our recommendation algorithm involves multi-factor scoring with weights, filtering, and ranking - Python's expressiveness makes this code readable and maintainable. SQLAlchemy provides excellent ORM capabilities for our PostgreSQL database, and Flask offers a lightweight framework perfect for our REST API needs. Python's ecosystem also includes excellent libraries for analytics (which we use for metrics aggregation) and testing. While Python isn't the fastest language, for our use case of handling web requests and database queries, performance is more than adequate, and developer productivity is more important."

Frontend Technologies

Next.js 14 (App Router)

Why Next.js was chosen: Next.js provides server-side rendering, automatic code splitting, and file-based routing, which improves performance and SEO. The App Router (introduced in Next.js 13) offers better data fetching patterns and layout support.

Pros vs Alternatives:

- **vs Create React App**: Built-in SSR/SSG, better performance, automatic optimization, file-based routing
- **vs Remix**: More mature ecosystem, larger community, better documentation, Vercel integration
- **vs Gatsby**: Better for dynamic content, simpler routing, more flexible data fetching
- **vs Vite + React**: Built-in SSR, better production optimizations, integrated routing

Cons:

- Learning curve for App Router (newer paradigm)
- More opinionated than plain React
- Can be overkill for simple static sites

Interview Answer: "We chose Next.js 14 with the App Router because it provides server-side rendering out of the box, which improves initial page load times and SEO. The App Router's file-based routing system makes it intuitive to organize pages, and features like loading states, error boundaries, and parallel routes are built-in. Next.js also handles code splitting automatically, ensuring users only download the JavaScript they need. For a production application that needs to be fast and SEO-friendly, Next.js provides these optimizations without requiring custom webpack configuration. The App Router, while newer, offers better data fetching patterns with async components and server components, reducing client-side JavaScript."

React 18

Why React was chosen: React is the industry standard for building user interfaces, with a massive ecosystem, excellent developer tools, and a component-based architecture that promotes reusability.

Pros vs Alternatives:

- **vs Vue**: Larger ecosystem, more job market demand, better TypeScript support, more third-party libraries
- **vs Angular**: Lighter weight, more flexible, easier to learn, better for smaller teams
- **vs Svelte**: More mature, larger ecosystem, better tooling, more developers familiar with it

Cons:

- Requires additional libraries for routing, state management (though Next.js handles routing)
- Can have performance issues with large lists (solved with virtualization if needed)

Interview Answer: "React 18 was chosen because it's the most widely adopted UI library with excellent TypeScript support and a massive ecosystem. React's component-based architecture allows us to build reusable UI components (like trip cards, filters, search forms) that can be composed together. React 18's concurrent features like automatic batching improve performance, and features like Suspense help with loading states. The large ecosystem means we can find solutions for common problems (like form handling, date pickers) without building from scratch. React's popularity also means it's easier to hire developers familiar with it, and there's extensive documentation and community support."

Tailwind CSS 3.4

Why Tailwind CSS was chosen: Tailwind provides utility-first CSS that enables rapid UI development without writing custom CSS. It's particularly useful for building consistent, responsive designs quickly.

Pros vs Alternatives:

- **vs CSS Modules**: Faster development, consistent spacing/colors, built-in responsive design, smaller bundle size (with purging)
- **vs Styled Components**: No runtime overhead, better performance, easier to maintain, no JavaScript in CSS
- **vs Bootstrap**: More flexible, better customization, utility-first approach, smaller bundle
- **vs Material-UI**: More flexible design system, not tied to Material Design, better performance

Cons:

- HTML can become verbose with many utility classes
- Learning curve for utility-first approach
- Can be harder to read for developers unfamiliar with it

Interview Answer: "Tailwind CSS was chosen for its utility-first approach that enables rapid UI development. Instead of writing custom CSS for every component, we use utility classes that provide consistent spacing, colors, and responsive breakpoints. This means we can build complex layouts quickly without context-switching between HTML and CSS files. Tailwind's purging removes unused CSS in production, resulting in smaller bundle sizes. The utility classes are also self-documenting - seeing `flex items-center justify-between` immediately tells you what the layout does. While the HTML can become verbose, the trade-off is faster development and easier maintenance. We configured Tailwind with custom colors and spacing to match our design system."

Lucide React

Why Lucide React was chosen: Lucide provides a comprehensive set of icons with React components, offering better tree-shaking and TypeScript support compared to icon fonts.

Pros vs Alternatives:

- **vs Font Awesome**: Better tree-shaking, smaller bundle, TypeScript support, SVG-based (scalable)
- **vs Material Icons**: More icon variety, better React integration, consistent design
- **vs Heroicons**: More icons available, actively maintained, better documentation

Cons:

- Smaller ecosystem than Font Awesome
- Less brand recognition

Interview Answer: "Lucide React was chosen for its comprehensive icon set with excellent React and TypeScript support. Unlike icon fonts, Lucide icons are SVG components that can be tree-shaken, meaning only the icons we actually use are included in the bundle. This results in smaller bundle sizes. The icons are also fully customizable through props (size, color, stroke width) and work seamlessly with Tailwind CSS. TypeScript support means we get autocomplete and type checking for icon names, preventing typos. While Font Awesome has more icons, Lucide's selection covers all our needs, and the better React integration and smaller bundle size make it the better choice for our project."

Backend Technologies

Flask 3.0

Why Flask was chosen: Flask is a lightweight, flexible Python web framework that provides just enough structure without being overly opinionated. It's perfect for building REST APIs with fine-grained control.

Pros vs Alternatives:

- **vs Django**: Lighter weight, more flexible, less boilerplate, better for APIs, easier to customize
- **vs FastAPI**: More mature, larger ecosystem, simpler for basic REST APIs, better documentation
- **vs Express.js**: Better Python ecosystem integration, SQLAlchemy support, better for data processing
- **vs Pyramid**: Simpler, more widely used, easier to learn, better community support

Cons:

- Less built-in features than Django (requires more manual setup)
- No built-in admin panel (not needed for API)
- Smaller ecosystem than Django

Interview Answer: "Flask was chosen because it's a lightweight, flexible framework perfect for building REST APIs. Unlike Django, which includes many features we don't need (admin panel, template engine, etc.), Flask gives us just the essentials and allows us to add only what we need. This results in a simpler, more maintainable codebase. Flask's blueprint system allows us to organize routes by feature (v2 API, events, analytics, resources, system), making the codebase modular. Flask also integrates seamlessly with SQLAlchemy and provides fine-grained control over request/response handling. For a REST API that doesn't need Django's full-stack features, Flask is the perfect balance of simplicity and power. We use Flask 3.0 which includes modern Python features and improved type hints."

SQLAlchemy 2.0

Why SQLAlchemy was chosen: SQLAlchemy is the most mature and feature-rich Python ORM, providing excellent database abstraction, relationship handling, and query building capabilities.

Pros vs Alternatives:

- **vs Django ORM**: More flexible, better for complex queries, works with any Python framework, more powerful
- **vs Peewee**: More features, better documentation, larger community, more mature

- **vs SQLObject**: More actively maintained, better performance, more features
- **vs Raw SQL**: Type safety, relationship handling, migration support, less error-prone

Cons:

- Steeper learning curve than simpler ORMs
- Can be verbose for simple queries
- More complex than needed for very simple applications

Interview Answer: "SQLAlchemy 2.0 was chosen because it's the most powerful and flexible Python ORM available. It provides excellent abstraction over PostgreSQL while still allowing us to write raw SQL when needed for complex queries. SQLAlchemy's relationship system handles our complex data model (TripTemplate → TripOccurrence → Guide, many-to-many relationships with tags and countries) elegantly. The ORM's query building capabilities are essential for our recommendation algorithm, which needs to build dynamic queries based on user preferences. SQLAlchemy 2.0's new API is more Pythonic and provides better type hints, improving developer experience. The session management with scoped sessions ensures thread-safe database access in our Flask application. While it has a learning curve, the power and flexibility it provides are essential for our use case."

Gunicorn

Why Gunicorn was chosen: Gunicorn is a production-ready WSGI HTTP server for Python applications. It's the standard choice for deploying Flask applications in production.

Pros vs Alternatives:

- **vs uWSGI**: Simpler configuration, easier to set up, more widely used, better documentation
- **vs Waitress**: Better performance, more features, production-proven, better for high traffic
- **vs Flask development server**: Production-ready, handles multiple workers, better performance, proper process management

Cons:

- Requires reverse proxy (nginx) for static files and SSL (handled by Render)
- More complex than development server (expected for production)

Interview Answer: "Gunicorn was chosen as the production WSGI server because it's the industry standard for deploying Python web applications. Unlike Flask's development server, Gunicorn can handle multiple worker processes, making it suitable for production traffic. It's battle-tested, well-documented, and integrates seamlessly with deployment platforms like Render. Gunicorn handles process management, worker restarts, and graceful shutdowns, which are essential for production reliability. We configure it to bind to the port provided by Render's environment and use multiple workers to handle concurrent requests. While we could use uWSGI, Gunicorn's simplicity and widespread adoption make it the better choice for our deployment needs."

Database & ORM

PostgreSQL 12+ (via Supabase)

Why PostgreSQL was chosen: PostgreSQL is a powerful, open-source relational database with excellent support for complex queries, JSON data, and ACID transactions. Supabase provides managed PostgreSQL with additional features.

Pros vs Alternatives:

- **vs MySQL**: Better JSON support, more advanced features, better for complex queries, ACID compliance
- **vs SQLite**: Better for production, supports concurrent writes, better performance, more features

- **vs MongoDB**: ACID transactions, relational data integrity, SQL queries, better for structured data
- **vs Supabase (Firebase alternative)**: PostgreSQL is more powerful than Firestore, SQL queries, better for analytics

Cons:

- More complex than SQLite (not needed for production)
- Requires more setup than NoSQL (worth it for data integrity)

Interview Answer: "PostgreSQL was chosen because it's the most advanced open-source relational database, providing excellent support for our complex data model with relationships, constraints, and transactions. We use Supabase, which provides managed PostgreSQL with additional features like real-time subscriptions, authentication, and a dashboard. PostgreSQL's JSONB support allows us to store flexible metadata (like trip properties) without schema changes, while maintaining the benefits of a relational database for our core data. The database handles our complex queries for the recommendation algorithm efficiently, and ACID transactions ensure data integrity. Supabase's connection pooling helps manage database connections efficiently, and the managed service reduces operational overhead. PostgreSQL's maturity and feature set make it the ideal choice for a production application with complex data relationships."

Supabase (Database Provider)

Why Supabase was chosen: Supabase provides managed PostgreSQL with built-in authentication, connection pooling, and a dashboard, reducing operational complexity.

Pros vs Alternatives:

- **vs AWS RDS**: Simpler setup, built-in auth, better developer experience, more affordable for small projects
- **vs Heroku Postgres**: More features, better pricing, modern platform, better documentation
- **vs DigitalOcean Managed Databases**: Built-in authentication, better developer tools, Supabase ecosystem
- **vs Self-hosted PostgreSQL**: No server management, automatic backups, connection pooling, built-in features

Cons:

- Less control than self-hosted
- Vendor lock-in (mitigated by using standard PostgreSQL)
- Can be more expensive at scale than self-hosted

Interview Answer: "Supabase was chosen because it provides managed PostgreSQL with additional features that simplify development and deployment. The built-in authentication system integrates seamlessly with our frontend, eliminating the need to build our own auth system. Supabase's connection pooling (Session pooler) is essential for handling concurrent requests efficiently, and the dashboard provides an easy way to inspect data and run queries. Supabase uses standard PostgreSQL, so we're not locked into proprietary features - we can migrate to another PostgreSQL provider if needed. The managed service handles backups, updates, and scaling, allowing us to focus on building features rather than managing infrastructure. For a startup or small team, Supabase provides the right balance of features, simplicity, and cost."

Authentication & Security

Supabase Authentication (OAuth + JWT)

Why Supabase Auth was chosen: Supabase provides a complete authentication solution with OAuth providers (Google, etc.), email/password, and JWT token management, eliminating the need to build custom auth.

Pros vs Alternatives:

- **vs Auth0**: More affordable, simpler setup, integrated with database, better for small projects
- **vs Firebase Auth**: Works with PostgreSQL (not Firestore), more flexible, better for relational data
- **vs Custom JWT implementation**: Pre-built, secure, OAuth support, session management, less code to maintain
- **vs Passport.js**: Integrated with database, simpler setup, built-in user management

Cons:

- Less customizable than custom implementation
- Vendor dependency (mitigated by JWT standard)
- Can be overkill if only need basic auth

Interview Answer: "Supabase Authentication was chosen because it provides a complete, secure authentication solution without requiring us to build and maintain our own auth system. It supports multiple OAuth providers (Google, GitHub, etc.) and email/password authentication out of the box. The JWT tokens are standard and can be verified on our backend using the JWT secret, ensuring security. Supabase handles user management, password reset, email verification, and session management, significantly reducing development time and security risks. The authentication integrates seamlessly with our PostgreSQL database, automatically creating user records. While we could build custom auth, using Supabase allows us to focus on our core business logic (recommendations) rather than authentication infrastructure. The JWT tokens are verified on every API request that requires authentication, and guest users can still use the application without logging in."

JWT (JSON Web Tokens)

Why JWT was chosen: JWT provides stateless authentication, allowing the backend to verify user identity without maintaining server-side sessions. This is essential for a REST API.

Pros vs Alternatives:

- **vs Session-based auth**: Stateless (scales better), works across domains, no server-side storage needed
- **vs OAuth 2.0 tokens**: JWT is the token format used by OAuth 2.0, standard and widely supported
- **vs API keys**: More secure, user-specific, can include user data, standard format

Cons:

- Tokens can't be revoked easily (mitigated by short expiration)
- Larger than session IDs (minimal impact)
- Must be stored securely on client (handled by Supabase)

Interview Answer: "JWT tokens are used for authentication because they provide stateless authentication that's perfect for REST APIs. When a user logs in through Supabase, they receive a JWT token that contains their user ID and email. This token is sent with every API request in the Authorization header. Our Flask backend verifies the token using the JWT secret from Supabase, extracting the user information without needing to query the database or maintain server-side sessions. This stateless approach scales better and simplifies our architecture. The tokens are signed and include expiration times, ensuring security. While JWTs can't be easily revoked (they're valid until expiration), we use relatively short expiration times, and Supabase handles token refresh. For our use case, the benefits of stateless authentication outweigh the limitations."

Production Servers & Deployment

Vercel (Frontend)

Why Vercel was chosen: Vercel is the platform created by the Next.js team, providing seamless deployment, automatic SSL, CDN, and excellent Next.js optimization.

Pros vs Alternatives:

- **vs Netlify**: Better Next.js integration, faster builds, better performance, created by Next.js team
- **vs AWS Amplify**: Simpler setup, better Next.js support, easier configuration, better developer experience
- **vs Self-hosted**: No server management, automatic deployments, CDN, SSL, zero configuration
- **vs Heroku**: Better for static/SSR sites, faster, more modern, better Next.js support

Cons:

- Less control than self-hosted
- Can be expensive at very high traffic (not an issue for most projects)
- Vendor lock-in (mitigated by Next.js being portable)

Interview Answer: "Vercel was chosen for frontend deployment because it's the platform created by the Next.js team, providing the best possible integration and optimization for Next.js applications. Vercel automatically detects Next.js projects and optimizes builds, handles server-side rendering, and provides edge functions for global performance. The platform offers automatic SSL certificates, CDN distribution, and zero-configuration deployments from Git. Every push to the main branch triggers a new deployment, and preview deployments are created for pull requests. Vercel's edge network ensures fast page loads worldwide. While we could self-host or use other platforms, Vercel's seamless Next.js integration and excellent developer experience make it the obvious choice. The free tier is generous for development, and the platform scales automatically with traffic."

Render (Backend)

Why Render was chosen: Render provides simple, reliable deployment for Python applications with automatic SSL, health checks, and easy environment variable management.

Pros vs Alternatives:

- **vs Heroku**: More affordable, better free tier, simpler pricing, modern platform
- **vs AWS EC2**: Simpler setup, no server management, automatic deployments, better for small teams
- **vs DigitalOcean App Platform**: Better Python support, simpler configuration, better documentation
- **vs Self-hosted**: No server management, automatic SSL, health checks, easy scaling

Cons:

- Less control than self-hosted
- Can have cold starts on free tier (mitigated by paid plans)
- Vendor lock-in (mitigated by using standard technologies)

Interview Answer: "Render was chosen for backend deployment because it provides a simple, reliable platform for deploying Python Flask applications. Render automatically builds and deploys from our Git repository, handles SSL certificates, and provides health check endpoints. The platform supports environment variables, database connections, and automatic scaling. Render's free tier is generous for development and small projects, and the paid tiers provide better performance and no cold starts. The platform integrates seamlessly with our PostgreSQL database on Supabase through connection strings. While we could use AWS or self-host, Render's simplicity and developer-friendly approach allow us to focus on building features rather than managing infrastructure. The platform handles process management, logging, and monitoring, reducing operational overhead."

Design Patterns

1. Service Layer Pattern

Location: `backend/app/services/recommendation/` (package), `backend/app/services/events.py`

Why this pattern was chosen: Separates business logic from API routes, making code testable and reusable. Business logic can be tested without HTTP mocks, and services can be reused by different interfaces (REST API, background jobs, CLI scripts).

Pros vs Alternatives:

- **vs Fat Controllers**: Business logic is testable independently, reusable across interfaces, clearer separation of concerns
- **vs Active Record**: Better for complex operations across multiple entities, more flexible
- **vs Domain-Driven Design**: Simpler, sufficient for our domain complexity, less over-engineering

Interview Answer: "The Service Layer pattern separates business logic from HTTP concerns. Our API routes (`app/api/v2/routes.py`) handle HTTP requests and responses, while services (`app/services/recommendation/` , `app/services/events.py`) contain the actual business logic. The recommendation service has been restructured into a modular package with separate modules for engine orchestration, scoring, filtering, relaxed search, constants, and context parsing. This separation allows us to test business logic without mocking HTTP requests, and services can be reused by different interfaces (REST API, background scripts, future GraphQL API). For example, the recommendation algorithm in the `recommendation` package can be called from the API endpoint, a CLI script, or a background job. This pattern makes the codebase more maintainable and testable."

---

2. Repository Pattern (Implicit via SQLAlchemy)

Location: SQLAlchemy models in `backend/app/models/trip.py` , `backend/app/models/events.py`

Why this pattern was chosen: SQLAlchemy ORM models already provide repository-like functionality (query methods, relationships). Adding explicit repository classes would create unnecessary abstraction.

Pros vs Alternatives:

- **vs Explicit Repository Classes**: Leverages SQLAlchemy's built-in capabilities, reduces code duplication, simpler
- **vs Data Access Objects**: SQLAlchemy models already serve this purpose, no need for additional layer
- **vs Active Record**: More flexible, better for complex transactions, Unit of Work pattern

Interview Answer: "We use an implicit Repository pattern through SQLAlchemy ORM models. Our models (`TripTemplate` , `TripOccurrence` , `User` , etc.) provide data access methods through SQLAlchemy's query API. For example, `db_session.query(TripTemplate).filter(...).all()` acts as a repository method. We chose not to create explicit repository classes because SQLAlchemy already provides this functionality, and adding another layer would create unnecessary abstraction and code duplication. The models serve dual purposes: domain objects representing business entities and data access through SQLAlchemy's query interface. This approach is simpler while still providing the benefits of the Repository pattern."

---

3. Blueprint Pattern (Flask)

Location: `backend/app/api/v2/routes.py` , `backend/app/api/events/routes.py` , `backend/app/api/analytics/routes.py` , `backend/app/api/resources/routes.py` , `backend/app/api/system/routes.py`

Why this pattern was chosen: Flask Blueprints allow organizing routes by feature, making the codebase modular and maintainable. Each feature (v2 API, events, analytics, resources, system) has its own blueprint in a dedicated subfolder.

API Organization:

- **System API** (`app/api/system/`) - Health checks and system operations
- **Resources API** (`app/api/resources/`) - Read-only reference data (countries, guides, trip types, tags)
- **V2 API** (`app/api/v2/`) - Trip recommendations and data using V2 schema
- **Events API** (`app/api/events/`) - User event tracking and session management
- **Analytics API** (`app/api/analytics/`) - Metrics and evaluation endpoints

Pros vs Alternatives:

- **vs Single routes file**: Better organization, easier to maintain, modular, can be developed by different team members
- **vs Separate Flask apps**: Simpler, shared configuration, easier deployment
- **vs Django apps**: More flexible, lighter weight, better for APIs

Interview Answer: "Flask Blueprints allow us to organize routes by feature. We have separate blueprints for the V2 API (`api_v2_bp`), events tracking (`events_bp`), analytics (`analytics_bp`), resources (`resources_bp`), and system endpoints (`system_bp`). Each blueprint is in its own subfolder under `app/api/` and is registered in `main.py` . This modular approach makes the codebase easier to navigate and maintain. For example, all recommendation-related endpoints are in the V2 blueprint, while all analytics endpoints are in the analytics blueprint. This organization scales well as the application grows, and different team members can work on different blueprints without conflicts."

---

4. Context API Pattern (React)

Location: `frontend/src/lib/dataStore.tsx`

Why this pattern was chosen: React Context API provides global state management for reference data (countries, trip types, tags) that's needed across multiple components. Simpler than Redux for this use case.

Pros vs Alternatives:

- **vs Redux**: Simpler, less boilerplate, sufficient for our needs, built into React
- **vs Zustand**: Built into React, no additional dependency, simpler API
- **vs Prop drilling**: Avoids passing props through many components, cleaner code
- **vs Local state**: Shared data loaded once, available everywhere, better performance

Interview Answer: "We use React Context API for global state management of reference data (countries, trip types, theme tags). This data is fetched once when the app loads and needs to be available across multiple components (search form, filters, trip cards). Context API is simpler than Redux for this use case - we don't need complex state management, just shared read-only data. The `DataStoreProvider` wraps the app in `layout.tsx` , and components can access the data using hooks like `useCountries()` , `useTripTypes()` , and `useThemeTags()` . This avoids prop drilling and ensures data is loaded once and cached. For more complex state (like user preferences or search results), we use local component state or URL parameters."

---

5. Custom Hooks Pattern (React)

Location: `frontend/src/hooks/useTracking.ts`

Why this pattern was chosen: Custom hooks encapsulate tracking logic, making it reusable across components and keeping components clean. Hooks like `usePageView()` and `useResultsTracking()` abstract away tracking implementation.

Pros vs Alternatives:

- **vs Inline tracking code**: Reusable, testable, keeps components clean, consistent tracking
- **vs Higher-Order Components**: Simpler, more flexible, better TypeScript support
- **vs Render props**: Cleaner syntax, easier to use, more modern

Interview Answer: "Custom hooks encapsulate tracking logic, making it reusable and keeping components clean. For example, `usePageView()` automatically tracks page views when a component mounts, and `useResultsTracking()` handles impression tracking for search results. Components simply call these hooks without needing to know the implementation details of the tracking service. This pattern promotes code reuse - any component that needs to track page views can use `usePageView()`. The hooks also make testing easier - we can test tracking logic independently of components. This pattern is a React best practice for sharing stateful logic between components."

---

6. Singleton Pattern

Location: `backend/app/services/events.py` (implicit through module-level service instance)

Why this pattern was chosen: Event tracking service needs consistent state (session management, user resolution) across all API endpoints. A single instance ensures consistency.

Pros vs Alternatives:

- **vs Multiple instances**: Ensures consistent state, prevents duplicate sessions, simpler access
- **vs Dependency Injection**: Simpler for our use case, no DI framework needed
- **vs Module-level instance**: Lazy initialization allows graceful startup, better testability

Interview Answer: "We use a Singleton pattern (implicitly) for the event tracking service. The service manages user sessions and event processing, and having a single instance ensures consistent state across all API endpoints. For example, if a user makes multiple requests, the same service instance resolves the user and manages their session consistently. While we could use dependency injection, the Singleton pattern is simpler for our use case and doesn't require a DI framework. The service is instantiated at the module level and reused across all requests, ensuring efficient resource usage and consistent behavior."

File Structure

```
trip-recommendations/
├── frontend/                          # Next.js frontend application
│   ├── src/
│   │   ├── app/                       # Next.js App Router (file-based routing)
│   │   │   ├── layout.tsx             # Root layout with providers
│   │   │   ├── page.tsx               # Home page
│   │   │   ├── auth/                  # Authentication pages
│   │   │   ├── search/                # Search functionality
│   │   │   └── trip/[id]/             # Dynamic trip detail pages
│   │   ├── components/                # Reusable React components
│   │   ├── hooks/                     # Custom React hooks
│   │   ├── lib/                       # Utilities and state management
│   │   ├── api/                       # Modular API client (v2, events, resources, analytics)
```

```
│   │   │   ├── client.ts            # Core API utilities (apiFetch, auth, retry logic)
│   │   │   ├── v2.ts                # V2 API endpoints (recommendations, templates)
│   │   │   ├── events.ts            # Event tracking endpoints
│   │   │   ├── resources.ts         # Resource data endpoints (countries, guides, types,
tags)
│   │   │   ├── analytics.ts          # Analytics endpoints
│   │   │   ├── system.ts            # System endpoints (health check)
│   │   │   ├── types.ts             # TypeScript type definitions
│   │   │   └── index.ts             # Centralized exports
│   │   └── services/                 # Tracking service (tracking.service.ts)
│   ├── public/                       # Static assets
│   └── package.json                  # Dependencies and scripts
│
├── backend/                          # Flask backend application
│   ├── app/
│   │   ├── main.py                   # Flask app entry point
│   │   ├── core/                     # Core functionality (database, auth, config)
│   │   ├── models/                   # SQLAlchemy database models
│   │   ├── services/                 # Business logic services
│   │   │   ├── recommendation/      # Recommendation algorithm package
│   │   │   │   ├── engine.py        # Main orchestration (get_recommendations)
│   │   │   │   ├── scoring.py       # Scoring algorithm
│   │   │   │   ├── filters.py       # Query building and filtering
│   │   │   │   ├── relaxed_search.py # Relaxed search expansion
│   │   │   │   ├── constants.py     # Configuration and thresholds
│   │   │   │   ├── context.py       # Preference parsing and normalization
│   │   │   │   └── __init__.py      # Package exports
│   │   │   └── events.py            # Event processing service
│   │   └── api/                      # API route handlers (Flask Blueprints)
│   │       ├── v2/                   # V2 API endpoints
│   │       ├── events/               # Event tracking endpoints
│   │       ├── analytics/            # Analytics endpoints
│   │       ├── resources/            # Resource data endpoints
│   │       └── system/               # System endpoints (health check)
│   ├── recommender/                  # Recommendation engine module (logging, metrics,
evaluation)
│   ├── migrations/                   # Database migration scripts
│   ├── scripts/                      # Utility scripts
│   └── requirements.txt              # Python dependencies
│
└── docs/                             # Project documentation
```

API Endpoints by Data Pipeline

Pipeline 1: Recommendation Pipeline

Flow: User Search → Frontend Form → API Call → Recommendation Algorithm → Results Display

Endpoints:

- `POST /api/v2/recommendations` - Get personalized trip recommendations
  - **Request**: User preferences (countries, trip types, themes, budget, duration, difficulty)

- **Response**: Ranked trip recommendations with match scores (0-100)
- **Key File**: `backend/app/api/v2/routes.py` (lines 512-632)

Supporting Endpoints:

- `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

Pipeline 2: Event Tracking Pipeline

Flow: User Interaction → Frontend Tracking → Batch Events → Backend Processing → Database Storage

Endpoints:

- `POST /api/events/batch` - Batch upload user events
  - **Request**: Array of events (page views, clicks, impressions, searches)
  - **Response**: Success confirmation with processed event count
  - **Key File**: `backend/app/api/events/routes.py` (lines 234-318)
- `POST /api/session/start` - Initialize user session
  - **Request**: Device info, user agent, screen size
  - **Response**: Session ID and user ID (anonymous or authenticated)
  - **Key File**: `backend/app/api/events/routes.py` (lines 43-138)
- `POST /api/user/identify` - Link anonymous user to authenticated user
  - **Request**: Supabase user ID
  - **Response**: Success confirmation
  - **Key File**: `backend/app/api/events/routes.py` (lines 319-372)
- `POST /api/events` - Single event (alternative to batch)
  - **Request**: Single event object
  - **Response**: Success confirmation
  - **Key File**: `backend/app/api/events/routes.py` (lines 139-233)

Pipeline 3: Resource Data Pipeline

Flow: Component Mount → API Call → Reference Data → Caching → UI Display

Endpoints:

- `GET /api/locations` - Get all countries with continent info
  - **Response**: List of countries with names (English/Hebrew) and continents
  - **Key File**: `backend/app/api/resources/routes.py` (lines 15-109)
- `GET /api/countries` - Get all countries (alternative endpoint)
  - **Response**: List of countries
  - **Key File**: `backend/app/api/resources/routes.py` (lines 110-138)
- `GET /api/trip-types` - Get all trip type categories

    - **Response**: List of trip types (Adventure, Cultural, etc.)
    - **Key File**: `backend/app/api/resources/routes.py` (lines 245-283)
- `GET /api/tags` - Get all theme tags
    - **Response**: List of theme tags (Wildlife, Photography, etc.)
    - **Key File**: `backend/app/api/resources/routes.py` (lines 284-325)
- `GET /api/guides` - Get all tour guides
    - **Response**: List of guides with names and descriptions
    - **Key File**: `backend/app/api/resources/routes.py` (lines 176-244)

Pipeline 4: Analytics Pipeline

Flow: Raw Data Logging → On-Demand Aggregation → Metrics Calculation → API Response

Endpoints:

- `GET /api/metrics` - Get current recommendation metrics
    - **Response**: Total requests, average response time, result counts, score distribution
    - **Key File**: `backend/app/api/analytics/routes.py` (lines 28-75)
- `GET /api/metrics/daily` - Get daily metrics breakdown
    - **Response**: Daily statistics for each day (request counts, response times, etc.)
    - **Key File**: `backend/app/api/analytics/routes.py` (lines 76-144)
- `GET /api/metrics/top-searches` - Get most common search criteria
    - **Response**: Top countries, trip types, themes searched
    - **Key File**: `backend/app/api/analytics/routes.py` (lines 145-198)
- `POST /api/evaluation/run` - Run algorithm evaluation scenarios
    - **Request**: Scenario IDs or "all"
    - **Response**: Evaluation results with accuracy metrics
    - **Key File**: `backend/app/api/analytics/routes.py` (lines 199-261)

Pipeline 5: Authentication Pipeline

Flow: User Login → OAuth Callback → JWT Token → API Requests with Auth Header

Endpoints:

- Frontend handles authentication through Supabase client
- Backend verifies JWT tokens on protected endpoints
- No dedicated auth endpoints (handled by Supabase + JWT verification)

Key Files:

- `frontend/src/lib/supabaseClient.ts` - Supabase client and auth helpers
- `backend/app/core/auth.py` - JWT verification middleware

Pipeline 6: System Health Pipeline

Flow: Health Check Request → Database Query → Status Response

Endpoints:

- `GET /api/health` - Health check endpoint
  - **Response**: Service status, database connection status, table counts
  - **Key File**: `backend/app/api/system/routes.py` (lines 15-57)

Key Files by Data Pipeline

Pipeline 1: Recommendation Pipeline

Frontend:

- `frontend/src/app/search/page.tsx` - Search form component
  - Collects user preferences, tracks search submission, navigates to results
- `frontend/src/app/search/results/page.tsx` - Results display page
  - Fetches recommendations, displays trip cards, tracks impressions
- `frontend/src/api/` - Modular API client structure
  - `client.ts` - Core API utilities (`apiFetch()` wrapper, authentication, error handling, retry logic)
  - `v2.ts` - V2 API endpoints (`getRecommendations()` , `getTemplates()` , `getOccurrences()`)
  - `index.ts` - Centralized exports for convenient importing
  - All API modules use `apiFetch()` for consistent authentication, retry logic, and error handling

Backend:

- `backend/app/api/v2/routes.py` - V2 API routes
  - `get_recommendations_v2()` endpoint (lines 423-632)
  - Formats request, calls service, logs request, returns results
- `backend/app/services/recommendation/` - Recommendation algorithm package
  - `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`)
- `backend/recommender/logging.py` - Request logging
  - `RecommendationLogger.log_request()` - Logs raw request/response data
  - Stores in `recommendation_requests` table for analytics

Pipeline 2: Event Tracking Pipeline

Frontend:

- `frontend/src/services/tracking.service.ts` - Event tracking service
  - Manages anonymous/session IDs, device detection
  - Event queuing, batching (10 events or 5 seconds)
  - Sends events to backend via `POST /api/events/batch`
- `frontend/src/hooks/useTracking.ts` - Tracking hooks

    - `usePageView()` - Tracks page views
    - `useResultsTracking()` - Tracks search result impressions
    - `useFilterTracking()` - Tracks filter changes
    - `trackSearchSubmit()` - Tracks search submissions
    - `flushPendingEvents()` - Forces immediate event send

Backend:

- `backend/app/api/events/routes.py` - Event API endpoints
    - `start_session()` - Initializes user session (lines 43-138)
    - `track_event_batch()` - Processes batch events (lines 234-318)
    - `identify_user()` - Links anonymous to authenticated user (lines 319-372)
- `backend/app/services/events.py` - Event processing service
    - `EventService.process_event()` - Validates and stores events
    - User resolution (anonymous vs authenticated)
    - Session management, trip interaction updates
- `backend/app/models/events.py` - Event data models
    - `User`, `Session`, `Event`, `TripInteraction` models
    - Relationships and computed fields

Pipeline 3: Resource Data Pipeline

Frontend:

- `frontend/src/lib/dataStore.tsx` - Global state management (optional pattern)
    - `DataStoreProvider` - Context provider for reference data
    - `useCountries()`, `useTripTypes()`, `useThemeTags()` - Custom hooks
    - Fetches data once, caches in context, available to all components
- `frontend/src/api/resources.ts` - Resource API client
    - `getLocations()`, `getCountries()`, `getTripTypes()`, `getTags()`, `getGuides()` functions
    - Uses `apiFetch()` from `client.ts` for consistent authentication and error handling
- `frontend/src/app/search/page.tsx` - Uses reference data
    - Fetches locations, trip types, tags on component mount via `resources.ts` API
    - Populates form dropdowns and filters

Backend:

- `backend/app/api/resources/routes.py` - Resource endpoints
    - `get_locations()` - Returns countries with continents (lines 15-109)
    - `get_trip_types()` - Returns trip type categories (lines 245-283)
    - `get_tags()` - Returns theme tags (lines 284-325)
    - `get_guides()` - Returns tour guides (lines 176-244)
- `backend/app/models/trip.py` - Data models
    - `Country`, `TripType`, `Tag`, `Guide` models
    - `to_dict()` methods for serialization

Pipeline 4: Analytics Pipeline

Backend:

- `backend/app/api/analytics/routes.py` - Analytics endpoints
  - `get_current_metrics()` - Current recommendation metrics (lines 28-75)
  - `get_daily_metrics()` - Daily breakdown (lines 76-144)
  - `get_top_searches()` - Most common searches (lines 145-198)
  - `run_evaluation()` - Algorithm evaluation (lines 199-261)
- `backend/recommender/metrics.py` - Metrics aggregation
  - `MetricsAggregator.get_current_metrics()` - Computes current metrics
  - `MetricsAggregator.aggregate_daily_metrics()` - Daily aggregation
  - Queries `recommendation_requests` table on-demand
- `backend/recommender/logging.py` - Request logging
  - Logs every recommendation request to `recommendation_requests` table
- `backend/scripts/analytics/aggregate_daily_metrics.py` - Manual aggregation script
  - Can be run manually or scheduled via cron
  - Populates `recommendation_metrics_daily` table (if it exists)

Pipeline 5: Authentication Pipeline

Frontend:

- `frontend/src/lib/supabaseClient.ts` - Supabase client
  - `getAccessToken()` - Gets JWT token from Supabase session
  - `getCurrentUser()` - Gets current authenticated user
  - Initializes Supabase client with project URL and anon key
- `frontend/src/app/auth/page.tsx` - Login page
  - Supabase OAuth login (Google, etc.)
  - Email/password authentication
- `frontend/src/app/auth/callback/page.tsx` - OAuth callback
  - Handles OAuth redirect, exchanges code for session
  - Links user to tracking system

Backend:

- `backend/app/core/auth.py` - JWT verification
  - `get_current_user()` - Verifies JWT token, extracts user info
  - `require_auth` decorator - Requires authentication
  - `optional_auth` decorator - Optional authentication (guest support)
- `frontend/src/api/client.ts` - Core API client utilities
  - `getAuthHeaders()` - Adds JWT token to request headers (async function)
  - `apiFetch()` - Wrapper function that automatically includes auth token in all API requests
  - All API modules (`v2.ts` , `events.ts` , `resources.ts` , etc.) use `apiFetch()` for consistent authentication

---

Pipeline 6: System Health Pipeline

Backend:

- `backend/app/api/system/routes.py` - System endpoints

    - `health_check()` - Health check endpoint (lines 15-57)
    - Queries database for table counts
    - Returns service status and database health
- `backend/app/core/database.py` - Database connection
    - `is_database_error()` - Detects database connection errors
    - Connection pooling, session management

Summary

This architecture overview documents all technologies, design patterns, API endpoints, and key files in the SmartTrip project, organized by data pipeline. Each technology choice includes justification, pros/cons vs alternatives, and detailed interview answers explaining the reasoning behind the selection.

The project uses a modern, scalable architecture with:

- **Frontend**: Next.js 14 + React 18 + TypeScript for type-safe, performant UI
- **Backend**: Flask + SQLAlchemy + Python for flexible, maintainable API
- **Database**: PostgreSQL (Supabase) for robust data storage
- **Authentication**: Supabase Auth + JWT for secure user management
- **Deployment**: Vercel (frontend) + Render (backend) for simple, reliable hosting

The architecture follows best practices with clear separation of concerns, modular design, and comprehensive error handling.