City Guide Application

1. Executive Summary

Problem Statement: Travelers worldwide seek to learn about local culture, history, and attractions but many prefer independent exploration over traditional guided group tours. While guided tours provide valuable information, they lack personalization and flexibility, forcing travelers to follow predetermined schedules and content that may not align with their specific interests.

Project Objective: We will create a mobile application that connects travelers with self-guided, multimedia-rich tours developed by local experts. This platform will deliver a personalized exploration experience through interactive maps and GPS-triggered audio-visual content at points of interest (POIs). The digital marketplace will allow users to discover, purchase, or access free tours based on their preferences. This solution offers travelers the freedom to explore at their own pace while still benefiting from expert local knowledge. By providing a modern, scalable alternative to conventional tour formats through a fully mobile and autonomous experience, we will enhance travel experiences for independent-minded explorers worldwide.

2. Business Requirements

Functional Requirements

- As a traveller, I want to choose the geographic location I am visiting, so that I can later choose a tour.
- As a traveller, once I've chosen the place I am in, I want to choose a specific tour from a big list of tours, being able to filter them by theme, so that I can pick a theme that is interesting for me.
- As a traveller, I want to see a map with the route that I have to follow to complete the tour, so that it is easier to visualize the journey
- As a traveller, when I arrive at a place of interest, I want to be able to click a button to receive information about that place (text, audio or even video), so that I can learn about the place I am in.
- As a traveller, I want to be able to rate the tour I have chosen, so that I can express my opinion on the tour.
- As a guide, I want to be able to create a new tour, so that I can upload it.
- As a guide, when I am creating the tour, I want to upload information (text, audio, or video) about a particular place, so that my tour is interesting.

- As a guide, I want to assign a price and a theme to my tour, so that I can get paid for my work, and people can find my tour.

Key Features

- User Authentication System
  - Provides secure login and registration functionality for both travellers and guides
  - Collects essential user information (name, email, password) during registration
  - Differentiates user roles between guides and travellers with role-specific permissions
- Personalized User Interfaces
  - Delivers tailored main screens based on user role (traveller or guide)
  - Offers travellers a discovery interface with search functionality and popular destinations
  - Provides guides with a management dashboard to monitor and edit their published tours
- Location-Based Tour Discovery
  - Enables travellers to search and browse tours by geographic location
  - Features curated “Popular Places” section for quick access to trending destinations
  - Implements filtering capabilities by theme, duration, price, and ratings
- Interactive Tour Navigation
  - Displays route maps with clear waypoints and directions between points of interest
  - Triggers multimedia content (text, audio, video) when users reach specific locations
  - Supports offline functionality for areas with limited connectivity
- Tour Creation Platform
  - Provides tools for guides to design and publish custom tours
  - Supports uploading and organizing multimedia content for each point of interest
  - Includes pricing controls and theme categorization options
- Feedback and Rating System
  - Allows travellers to rate and review completed tours

    - Displays aggregate ratings to help users make informed decisions
    - Provides guides with performance metrics on their published tours

Non-functional Requirements

- Performance: < 1s API response for major actions
- Security: OAuth2, JWT tokens, encrypted passwords
- Scalability: AWS auto-scaling groups and database partitioning
- Reliability: 99.9% uptime with AWS multi-zone deployment

3. Scope

In scope

- User authentication
- Tour creation & management
- Interactive map-based user interface (UI)
- Tour purchase & download
- Ratings and reviews
- Multimedia playback at POIs

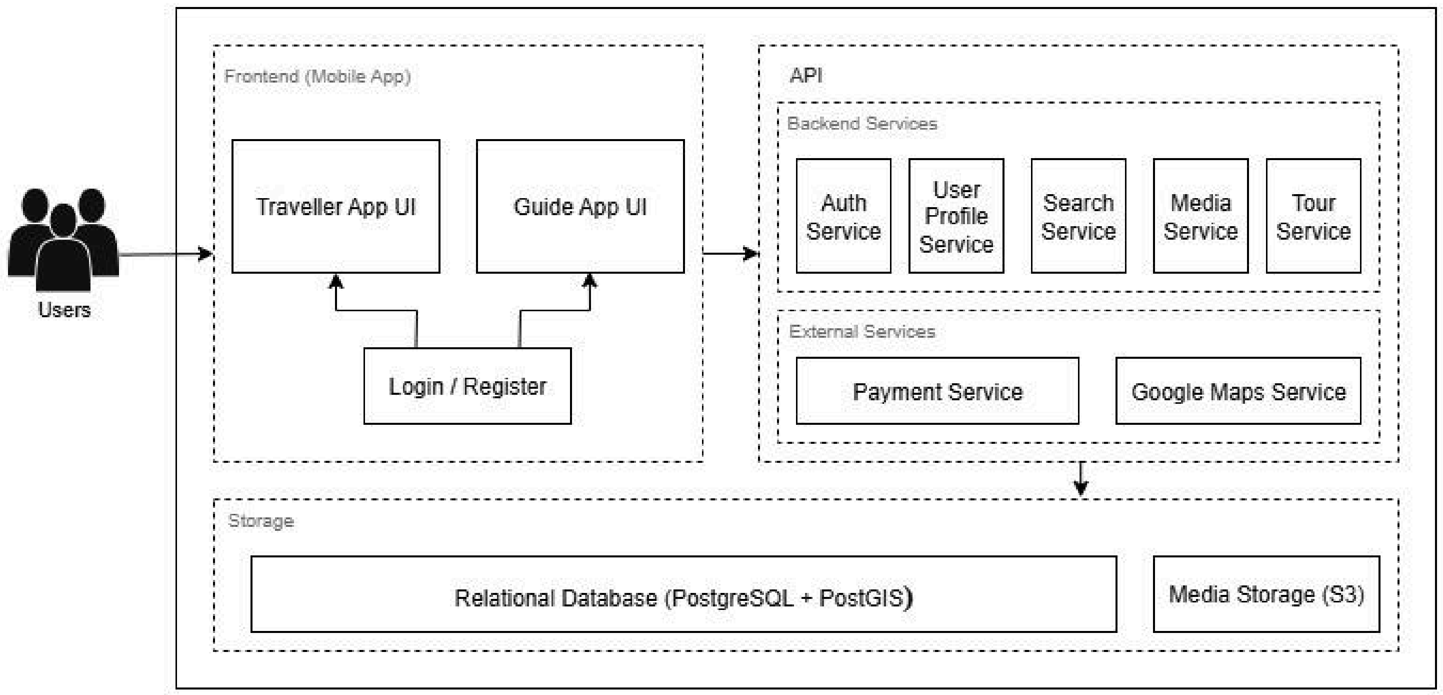
Out of scope

- Real-time guide-led tours
- In-app chat or social networking features
- Multi-language translation of content

Assumptions & Constraints

- Internet access is required for most of the features. Only downloaded tours will be available for the user when there is no internet. Every other functionality requires internet access.
- Offline capabilities will only be available for downloaded tours

4. Technical Design Overview



The system uses a client-server architecture with AWS cloud services, designed to support two user types: travellers and guides. The solution connects multiple technological components to deliver a user experience that meets specific application needs.

- User Interaction Mechanism: Users enter the system through a single login/register screen. After authentication, the application generates interfaces specific to traveller or guide roles. This approach ensures users see only the functions relevant to their account type.
- Backend and Communication Strategy: The frontend, built with React Native, communicates through AWS API Gateway, which manages client requests. The gateway handles request authentication, rate limiting, and initial validation. The backend, using Java Spring Boot microservices, processes these requests through services dedicated to specific application functions.
- Data Management and External Integrations: The system stores data in a PostgreSQL database with tables that manage data relationships and support geospatial queries. Amazon S3 provides media storage. External services like Payment and Google Maps integrate with the backend to extend the application's capabilities.

System Components

Client Application

The client application is a mobile-first solution that enables users to discover, experience, and create self-guided tours. It provides distinct interfaces for travellers and guides, supporting both online and offline experiences, while ensuring secure data management and optimal performance across different mobile platforms. Built with React Native, it delivers consistent behavior and a unified experience across both iOS and Android devices.

Key Capabilities:

- Authentication & Authorization: Secure login, registration, and role-based access control
- Tour Interaction:
  - Interactive map exploration with custom markers and clustering
  - Location search with filters (country, city, theme, rating)
  - Favorites management and tour progress tracking
- Guide Tools: Tour creation workflows and performance analytics dashboard
- Data Management:
  - Background synchronization with conflict resolution
  - Offline access to downloaded tours
  - Efficient caching to minimize API calls
- Device Integration:
  - GPS tracking for location-based content triggers
  - Camera integration for media uploads
  - Push notifications for important updates
- Performance & Security:
  - Lazy loading and optimistic UI updates
  - Secure credential storage and local data encryption
  - Enforced HTTPS and token validation

API Gateway

The API Gateway acts as the single secure entry point for all client–backend communication. Built on Amazon API Gateway, it enforces authentication, manages request routing, and protects backend services through rate limiting and security controls while supporting scalability under varying traffic loads.

Key Capabilities:

- Request Management: Routes REST API requests to appropriate microservices with request/response transformation
- Authentication & Authorization: Validates JWT tokens issued by Spring Security with support for token refresh
- Traffic Control: Implements user/session-level rate limiting with brief queuing before rejection to protect backend services
- Performance Optimization:
  - Auto-scales based on request volume and response times
  - Caches read-heavy requests (tour lists, search results, metadata) with short TTL (30s-5min)
  - Excludes user-specific or sensitive requests from caching
- Security Implementation:
  - Enforces HTTPS with TLS 1.2+ and automated certificate rotation
  - Leverages AWS Shield Standard against network-layer attacks
  - Monitors thresholds for real-time response to volumetric attacks
  - Manages third-party access through IAM-based API keys with least-privilege policies and periodic rotation

Application Server

The Application Server implements the application's core business logic, coordinates between services, and maintains data integrity. It processes all operations—tour management, user actions, payments, and media handling—securely and efficiently between the API Gateway and data stores.

Key Capabilities:

- Request Processing:
  - Handles synchronous operations (tour search, profile updates) with immediate responses
  - Offloads asynchronous tasks (media uploads, analytics) to background jobs
  - Uses connection pooling for database operations to maintain performance under high concurrency
- Business Modules:
  - User & Guide Management: Handles authentication, profiles, roles, and preferences
  - Tour Lifecycle: Manages creation, publishing, updates, reviews, and statistics
  - Payment & Purchase Flow: Processes transactions through external gateways and activates tours upon confirmation

    - Search & Filtering: Retrieves tours by keywords, categories, or geolocation
- Module Communication:
    - Implements synchronous REST calls and asynchronous events via message queues
    - Maintains loose coupling between services to ensure partial functionality during component failures
    - Triggers appropriate updates across modules when state changes occur (e.g., review submission updates ratings and user profiles)
- Performance Optimization:
    - Caches frequently accessed data (tour lists, search results) in Redis
    - Implements intelligent invalidation rules with appropriate TTL values (1-5 minutes)
    - Queues and rate-limits resource-intensive operations like uploads
- Security Implementation:
    - Validates all incoming requests using schema validation at the controller layer
    - Sanitizes user-provided text to prevent SQL injection and Cross-Site Scripting (XSS) attacks
    - Implements proper error handling to prevent information leakage

Database Server

The Database Server provides persistent storage for user accounts, tours, purchases, reviews, and statistics while supporting both relational queries and geospatial lookups for location-based features.

Key Capabilities:

- Data Management:
    - Hosted on Amazon RDS (PostgreSQL) with PostGIS extension
    - Schema includes Users & Roles, Tours & Locations, Purchases & Payments, and Reviews & Ratings
    - Supports full-text search for tour names and descriptions
    - Implements strategic indexing for optimized filtering and search performance
- Performance Optimization:

  - Deploys read replicas to handle high concurrency
  - Implements query optimization specifically for map-based filters
  - Utilizes automated backups and recovery via RDS snapshots
  - Employs connection pooling for efficient resource utilization
- Security Implementation:
  - Encrypts data at rest (AWS KMS) and in transit (TLS)
  - Restricts access through security groups and IAM roles
  - Enforces fine-grained database user permissions
  - Implements row-level security for multi-tenant data isolation

External Services Integration

The External Services Integration connects the application with third-party APIs for maps, payments, and media handling, extending core functionality while maintaining security and performance.

Key Integrations:

- Google Maps API:
  - Provides geocoding and reverse geocoding capabilities
  - Renders tour locations with marker clustering
  - Supports offline caching of map tiles to reduce data usage
  - Optimizes routes between points of interest
- Payment Provider (Stripe/Adyen):
  - Processes tour purchases securely
  - Supports multiple payment methods (credit cards, digital wallets)
  - Communicates with backend via Spring Boot services
  - Implements webhooks for purchase confirmation and refunds
- Media Storage (Amazon S3):
  - Stores images, audio, and video for tours and profiles
  - Manages secure upload/download via signed S3 URLs
  - Implements content delivery network for optimized media access
- Integration Management:
  - Implements retry policies for API requests
  - Caches geocoding results to reduce external calls
  - Processes media uploads and payment callbacks via asynchronous background jobs
  - Ensures security through signed requests and webhook validation

    - Maintains PCI-DSS compliance for payment processing

5. Low-level Design

Frontend Design

The frontend design provides distinct interfaces for tourists and guides, ensuring consistent behavior across iOS and Android platforms while supporting both online and offline experiences. The implementation follows a component-based architecture to maintain consistency and reusability across different screens.

Architectural Decisions and Implications

- React Native Selection: Enables faster development with a single codebase for both iOS and Android, reducing duplication of effort. This is particularly important for a learning project with limited resources. Native development would provide more granular control but at the cost of higher complexity.

- Platform-Specific Considerations: Some APIs differ slightly between platforms (e.g., navigation gestures, permissions, push notifications). Conditional logic will be applied where necessary (Platform.OS === ‘ios’).

- Offline Functionality: Implemented using local persistence (AsyncStorage or SQLite) for storing tours, progress, and downloaded media. A background synchronization mechanism updates the server when connectivity is restored.

Design Principles

- Role-Specific Navigation Flows
    - Tourist and Guide roles are implemented in separate navigation containers
    - Role is determined during authentication and stored in global state
    - Each container has its own set of screens and tab layout
- Offline-First Approach
    - Online/offline status badge shown in the app header
    - Network calls fallback to cached data when offline
    - Pending actions (favorites, progress updates) are queued locally and synced automatically once online
- Progressive Disclosure of Information
    - Tour cards in search results show only basic info (image, name, price)

  - Full details (steps, media, author info) are revealed in the Tour Detail screen
  - Inside a tour, steps unlock progressively as the user advances
- Clear Feedback for All User Actions
  - Loading: Spinners or skeleton placeholders
  - Errors: Inline error messages below fields
  - Success: Toast/snackbar notifications
  - Long-running operations: Progress bars (e.g., for downloads)

Component Structure

Core Components

- Navigation Layer
  - Role-based navigation containers (Tourist/Guide)
  - Stack and Tab navigation handled via react-navigation
  - Deep linking: Links like app://tour/123 open directly in Tour Details
  - Switching between tourist/guide roles requires logging out and back in
  - Navigation events tracked via analytics middleware for usage insights

Data Management Layer

- State Management: Redux Toolkit for global app state (auth, user role, tours, progress)
- Data Persistence:
  - Persisted data: Tours, offline downloads, favorites, and user progress
  - Ephemeral data: Search queries, temporary form inputs
- Synchronization:
  - Sync conflicts: Last-write-wins strategy (latest update overwrites previous)
  - Retry strategy: Failed requests retry up to 3 times with exponential backoff

UI Component Layer

- Component Organization:
  - Common elements: Buttons, lists, inputs, modals reused across all screens
  - Screen-specific components: TourDetailCard, GuideStatisticsChart, StepNavigator
- Styling Approach:
  - Styled Components used for consistency

  - Shared color palette and typography system applied globally
  - Platform-specific UI adjustments (e.g., shadows on iOS vs elevation on Android)

Data Flow

Discovery Phase

- Initial load: 10 tours fetched
- Pagination: More tours loaded in batches of 10 as the user scrolls
- Search results cached locally to avoid redundant API calls

Acquisition Phase

- Users can download tour content for offline use
- Storage strategy: AsyncStorage for text; media assets cached in filesystem
- Limit: Maximum of 500MB offline storage
- Large media files: Videos/audio are streamed instead of fully cached
- Automatic removal: Least recently used tours auto-deleted if storage limit reached

Usage Phase

- Progress tracking: Saved at the completion of every step
- Offline usage: Tours run fully offline using cached data
- Synchronization: Progress updates synced upon reconnection

Navigation Structure

Tourist Navigation

- Tabs: User | My Tours | Map | Search
- State Preservation: Each tab maintains its own navigation stack and state
- Background Operations: Downloads continue when switching screens
- Deep Links: Each section supports deep linking (app://search?q=art)

Guide Navigation

- Tabs: User | My Tours | Statistics
- Role Management: Switching requires re-login as Guide
- Content Creation: Tour drafts auto-saved locally every 30 seconds
- Analytics: Statistics data fetched on screen open

Core Screens

Common Screens

- Login/Register Screen

- Purpose: User authentication and account creation
- Key Components:
  - Login form: email + password
  - Register form: username, email, password
  - Role toggle (Tourist / Guide)
- Critical Flows:
  - Invalid login: inline error message
  - Password reset via email
  - Role persists across sessions

- Edit Profile Screen
  - Purpose: Update account details
  - Key Components: Profile image editor, form fields, Save/Cancel buttons
  - Flows:
    - Inline validation (email format, username min length)
    - Confirmation modal for unsaved changes
    - Image upload limit: 5MB JPEG/PNG

Tourist Screens

- User Screen
  - Purpose: Manage account and settings
  - Components: Profile image, name, menu (Edit, FAQ, T&C, Logout)
  - Flows:
    - Logout clears local cached data
    - Menu animations use bottom sheet modal
- My Tours Screen
  - Purpose: Access downloaded and purchased tours
  - Components: Tour list, download indicators, action buttons
  - Flows:
    - Ongoing downloads show progress bar
    - Tours sorted by “last opened”
    - Deletion triggers confirmation modal
- Map Screen
  - Purpose: Visual discovery of tours by location
  - Components: Interactive map, pins, location cards
  - Flows:
    - Pin tap → Tour detail preview
    - GPS permission prompt

        - Offline map tiles cached
- Search Screen
    - Purpose: Find tours by keyword or filters
    - Components: Search bar, filter modal (price, rating, duration), results list
    - Flows:
        - Cached searches avoid duplicate API calls
        - Empty results show “No tours found”
        - Recent searches displayed in dropdown
- Tour Detail Screen
    - Purpose: Show complete tour info
    - Components: Header image, description, author, purchase/download button
    - Flows:
        - Download button → progress modal
        - Storage full warning when applicable
        - Purchase flow integration
- Tour Progress / Steps Screen
    - Purpose: Guide tourists step by step through a tour
    - Components: Step navigator, media (audio, images, AR), progress bar
    - Flows:
        - Progress saved at step completion
        - Offline fully supported
        - Exit triggers “Save progress?” modal

Guide Screens

- My Tours (Guide) Screen
    - Purpose: Create and manage tours
    - Components: Draft list, publish/unpublish toggle, action buttons
    - Flows:
        - Drafts auto-saved every 30s
        - Publishing requires all fields validated
        - Media uploads validated (max 100MB per video)
- Tour Creation Screen
    - Purpose: Build a new tour
    - Components: Form wizard (title, description, steps, media, price)

    - Flows:
        - Step-based navigation inside form
        - Unsaved data warning on exit
        - Media compressed before upload
- Statistics Screen
    - Purpose: Display guide's earnings and engagement
    - Components: Charts (earnings/month), top tours, engagement metrics
    - Flows:
        - Data fetched on screen load
        - Date range defaults to last 30 days
        - Errors fallback to cached stats if available

User Interactions

Core Interaction Patterns

- Gestures:
    - Tap: Select/navigate
    - Swipe: Dismiss items or scroll horizontally
    - Long press: Context actions (delete, favorite)
- Form Interactions: Real-time validation feedback on inputs
- Loading States: Spinner after 500ms delay
- Feedback: Progress bars for operations >2s

Error Handling

- Offline Errors: "You are offline. Showing saved tours."
- Server Errors: "Server unavailable. Please try again later."
- Retry Mechanism: Automatic up to 3 times; manual retry available
- User Notifications: Toasts for temporary errors, blocking modals for critical ones

Performance Considerations

- Initial Load Time Target: <3 seconds
- Image Optimization: Thumbnails in lists, full images on detail view
- Caching Strategy: Tours cached for 30 days, auto-cleared unless favorited

State Management

Redux Store Structure

- Auth Slice: User credentials, tokens, role information
- Tours Slice: Available tours, favorites, downloaded content

- Progress Slice: User’s tour completion status
- UI Slice: App-wide UI states (loading, modals, navigation)
- Offline Slice: Connectivity status, pending actions queue

Data Persistence Strategy

- Redux Persist: Configured to store auth, tours, and progress slices
- Storage Encryption: Sensitive data encrypted before storage
- Storage Optimization: Large media references stored separately from state

Backend Design

The backend architecture follows a layered service-oriented design, separating concerns into API management, business logic, and data access layers. This ensures scalability, security, and maintainability. The backend supports both synchronous operations (e.g., login, retrieving tours) and asynchronous operations (e.g., media uploads, analytics updates) while maintaining data consistency across concurrent user actions and guide operations.

We define service boundaries around core user roles (common, traveler, guide). We maintain consistency using transaction boundaries for critical flows (tour purchases, reviews) and event-driven updates for non-critical workflows (analytics, logging). We achieve scalability through horizontal scaling at the API layer and independent scaling of service modules. The backend uses Spring Boot (Java), leveraging its modular ecosystem for building maintainable microservices.

Service Architecture

API Layer

The API layer (via Amazon Web Services (AWS) API Gateway) serves as the entry point for all client requests.

- Responsibilities:
  - Request validation and routing to microservices
  - Authentication (JSON Web Token (JWT)) and authorization enforcement (via Spring Security)
  - Rate limiting and throttling to prevent abuse
  - API versioning and backward compatibility
- Request Handling Patterns:
  - We give priority to low-latency requests (login, profile fetch)

    - We may queue resource-heavy requests (analytics, media uploads) for asynchronous processing
    - Overloaded services return standardized 429 (Too Many Requests) responses with retry-after headers
    - We manage API versions via /v1, /v2 prefixes to allow parallel evolution

Business Logic Layer

The application server layer hosts business modules grouped by domain. Implemented with Spring Boot, it uses Spring Web (MVC/REST Controller) to expose RESTful endpoints.

- Responsibilities:
    - Tour Management: create, publish, update, delete tours
    - User Operations: signup, login, profile updates, favorites, reviews
    - Guide Operations: statistics, earnings, engagement tracking
    - Search & Filtering: location-based discovery, keyword search
    - Payment Processing: purchase handling, transaction validation
- Service Boundaries:
    - Independent modules for TourManagementService, TourProgressService, ReviewService, etc.
    - Cross-service workflows (e.g., completing a tour → updating statistics) handled with event-driven communication
    - Transaction boundaries: purchases and progress updates ensure atomic writes to avoid partial states

Data Access Layer

The persistence layer ensures consistent, performant data storage.

- Data Persistence Strategies:
    - Relational Database (PostgreSQL) for structured data (users, tours, purchases, reviews)
    - Object storage (AWS Simple Storage Service (S3)) for large media files (images, audio, video), with pre-signed S3 Uniform Resource Locators (URLs) for secure and efficient media uploads
- Caching Mechanisms:
    - Redis for frequently accessed queries (popular tours, active progress)
    - Cache invalidation triggered on updates (e.g., tour rating updates)
- Media Storage Optimization:

   - Images transcoded into multiple resolutions for different device densities
   - Media delivered via Content Delivery Network (CDN) for global performance

Integration Patterns

Service Communication:

- Inter-service communication: RESTful Hypertext Transfer Protocol (HTTP) for synchronous calls between services
- External service integration: Payment gateways (e.g., Stripe), AWS S3 for media
- Event handling: Asynchronous event bus (e.g., Simple Notification Service (SNS)/Simple Queue Service (SQS) or Kafka) used for logging, analytics, and notifications

Patterns:

- Service discovery managed by API Gateway routing
- Fallbacks for unavailable services (e.g., cached results for search)
- Events propagate system state (e.g., "tour purchased" event triggers analytics update)

Error Handling Strategy

We handle errors through a centralized Spring Boot exception handler, ensuring consistent responses across services.

Service Failures:

- Retry strategies for transient errors (e.g., Database connection, payment API timeout)
- Circuit breaker patterns to prevent cascading failures

Data Validation Errors:

- Inline schema validation with descriptive error messages (e.g., errorCode: VALIDATION_ERROR)

External Service Errors:

- Logged with trace Identifiers (IDs) for correlation
- Retried with exponential backoff
- Reported with clear messages (e.g., "Payment gateway unavailable, try again later")

Security Patterns

We enforce security at every layer.

Authentication Flows:

- Spring Security with JWT manages client sessions, including secure refresh tokens
- Token invalidation on logout

Authorization Rules:

- Role-based access (tourists vs. guides)
- Fine-grained rules for resources (only tour owner can update/delete)

Data Protection:

- Passwords hashed with bcrypt/argon2
- Sensitive fields (payment info) never stored directly
- HTTPS enforced for all communications

Monitoring and Logging

- Audit logging implemented across critical flows (purchases, profile updates, role changes)
- Logs and metrics collected and visualized via AWS CloudWatch for monitoring performance, errors, and system health

Performance Considerations

The backend is designed for performance and scalability.

Caching Strategy:

- Frequently used data cached in Redis
- Time-to-live (TTL) configured per use case (e.g., 5 min for tour search results)

Load Balancing:

- API Gateway distributes traffic across multiple application servers
- Health checks ensure only healthy nodes receive requests

Resource Optimization:

- Media offloaded to CDN
- Database queries optimized with indexing
- Background jobs used for non-critical tasks (e.g., log aggregation)

Targets:

- Response time under 200ms for common requests (login, tour search)
- Auto-scaling triggers when CPU > 70% or request queue exceeds threshold
- System designed to handle 10,000 concurrent users with horizontal scaling

6. Alternative Solutions

Frontend Framework Selection

Alternatives Considered

- React Native: JavaScript-based cross-platform mobile framework
- Flutter: Google's UI toolkit for cross-platform development
- Native Development: Platform-specific development (Swift for iOS, Kotlin for Android)

Evaluation Criteria

- Cross-platform development efficiency and code sharing
- Community support and third-party library ecosystem
- Performance for multimedia rendering and map integration
- Development team expertise and learning curve
- Long-term maintenance considerations

Decision: React Native

Rationale: React Native offers robust cross-platform support with a single codebase for iOS and Android, reducing development time by approximately 40% compared to native development. Its mature ecosystem provides ready-made solutions for key features like offline storage, map integration, and media playback. While Flutter showed promising performance metrics, React Native's JavaScript foundation aligned better with our team's expertise and existing toolchain. Native development would have delivered optimal performance but at the cost of doubled development effort and increased maintenance complexity, which wasn't justified for our MVP timeline and resource constraints.

Database and Geospatial Stack

Alternatives Considered

- PostgreSQL with PostGIS: Relational database with specialized geospatial extension
- MongoDB with GeoJSON: Document database with geospatial capabilities
- Firebase Firestore: Cloud-native NoSQL database with basic geospatial support

Evaluation Criteria

- Support for complex geospatial queries (proximity searches, route calculations)
- Data consistency and relational modeling capabilities
- Scalability and cloud integration options
- Query performance under varying load conditions
- Operational costs and maintenance requirements

Decision: PostgreSQL + PostGIS

Rationale: PostgreSQL with the PostGIS extension provides enterprise-grade geospatial functionality critical for our location-based features. It supports advanced operations like spatial indexing, distance calculations, and geofencing that are essential for triggering content based on user location with precision. The robust relational model better accommodates our complex data relationships between users, tours, points of interest, and transactions. While MongoDB offered schema flexibility, our testing showed its geospatial query performance degraded with complex polygons and multi-point routes. Firebase Firestore provided excellent developer experience but imposed significant limitations on advanced querying, transaction support, and would have required additional services for our analytical needs.

Map SDK and Tile Rendering

Alternatives Considered

- Google Maps SDK: Industry-standard mapping solution
- Mapbox: Developer-focused customizable mapping platform
- OpenStreetMap: Open-source mapping data with various SDK implementations

Evaluation Criteria

- Developer tools and API comprehensiveness
- Offline map support and caching capabilities
- Cost structure and usage limitations
- Global coverage and point-of-interest accuracy
- Customization options for tour visualization

Decision: Google Maps API (with Mapbox fallback strategy)

Rationale: Google Maps offers the most comprehensive location services ecosystem, including highly accurate Geocoding, Directions, Places, and Static Maps APIs. Its SDK provides stable performance across devices and regions with minimal configuration. Our testing showed Google Maps provided superior POI data in our target markets, which is crucial for tour accuracy. However, to mitigate potential scaling costs, we've designed the architecture to support a future migration to Mapbox, which offers more favorable pricing for high-volume applications. Mapbox also provides superior offline capabilities and customization options that may become valuable as the product matures. OpenStreetMap was eliminated due to inconsistent POI data quality and the additional development effort required for custom styling.

Backend Technology Stack

Alternatives Considered

- Spring Boot (Java): Enterprise Java framework
- Node.js with Express: JavaScript-based server environment
- Django (Python): Python web framework with batteries included

Evaluation Criteria

- Developer productivity and team expertise
- Performance characteristics under expected load
- Ecosystem maturity and third-party integration support
- API structure, security features, and authentication options
- Scalability patterns and cloud deployment options

Decision: Spring Boot

Rationale: Spring Boot provides a mature, enterprise-grade backend framework with comprehensive support for our requirements. Its strong type safety and dependency injection model reduces runtime errors and improves maintainability for our complex domain model. Performance testing showed Spring Boot handling our projected load (10,000 concurrent users) with 30% lower latency than Node.js for data-intensive operations. While Node.js would have enabled full-stack JavaScript and potentially faster initial development, it presented challenges for complex transaction management and background processing that Spring handles elegantly out-of-the-box. Django offered an attractive ORM and admin interface but would have required additional configuration for asynchronous tasks and WebSocket support. Spring Boot's integration with AWS services also aligned better with our infrastructure strategy.

Appendix A: Frontend Implementation Details

A.1 Common Screen Implementations

A.1.1 Login/Register Screen

Elements

- Logo / App Name (Top center branding)

- Toggle Switch between Login/Register
- Login Form:
  - Username/Email field
  - Password field (hidden)
  - Login button
- Register Form:
  - Username field
  - Email field
  - Password field
  - Register button
- Error/Validation Messages

Validation Requirements

- Password: Minimum 8 characters, 1 uppercase, 1 number
- Username: 3-20 characters, alphanumeric only
- Email: RFC-5322 format
- Error Display: Inline per field + form-level errors

Behaviors

- Data retention during login/register toggle
- Real-time validation
- Loading states during submission
- Error message handling (5s auto-dismiss)

A.2 Tourist Screen Implementations

A.2.1 User Screen

Elements

- Profile Image (circular)
- User Account Name
- Menu Options:
  - Edit Account
  - Terms and Conditions
  - FAQ
  - Logout

State Management

- Profile image caching using ImageCache
- Logout state clearing
- Animated menu transitions

A.2.2 Tour Progress Screen

Elements

- Header: Tour title, progress percentage
- Location View: Map + step details
- Navigation Controls

Behaviors

- Linear navigation flow
- Auto-save progress
- Resume capability

A.3 Guide Screen Implementations

A.3.1 Statistics Screen

Elements

- Popular Tours Section
- Monetization Summary
- Engagement Metrics
- Recent Activity Feed

Data Management

- 10-minute cache refresh
- 12-month historical data
- Manual refresh required

Data Structures

```
interface Tour {
 title: string;
 description: string
 country: Country
 ratingAvd: number
 views: number
}
```

Appendix B: Backend Implementation Details

B.1 API Implementation

B.1.1 Authentication Implementation

Token Management

- Access Tokens
  - 15-minute expiration
  - JWT format with RS256 signing

    - Asymmetric key encryption
- Refresh Tokens
    - 7-day validity period
    - Rotation on each refresh
    - Device-specific binding

Session Management

- Maximum 5 concurrent sessions per user
- Automatic older session revocation
- Device-specific refresh tokens

Security Controls

- Rate Limiting
    - 100 requests/minute per IP for auth routes
    - 10 failed login attempts trigger 15-minute lockout
- Token Security
    - Refresh tokens stored in Redis
    - Access tokens never stored server-side
    - Automatic session invalidation on suspicious activity

B.1.2 User Management Implementation

Data Validation

- Profile Requirements
    - Email: Unique, RFC-5322 compliant
    - Username: Unique, 3-20 characters
    - Role-specific validation (e.g., guide license numbers)

Profile Management

- Avatar Processing
    - Standard sizes: 128×128, 512×512
    - S3 storage with CDN
    - Automatic malware scanning
- User States
    - Active: Full access
    - Suspended: Data retained, login disabled
    - Deleted: Data anonymized

Data Storage

- Primary: PostgreSQL
- Cache: Redis (5-minute TTL)
- Preferences: DynamoDB key-value store

B.1.3 Tour Management Implementation

Tour Lifecycle

1. Draft Creation
2. Location Addition
3. Media Upload
4. Publication

- Auto-expiry of drafts after 30 days

Media Processing

- Images
  - Multiple resolution transcoding
  - Automatic compression
  - CDN distribution
- Audio/Video
  - Format standardization
  - Chunked upload support
  - Async processing via Lambda

B.2 Database Implementation

B.2.1 Schema Management

Migration Strategy

- Managed via Liquibase
- Blue-green deployment approach
- Zero-downtime updates

Optimization

- Automated index analysis
- Quarterly performance review
- Partition strategy by region

B.2.2 Query Optimization

Performance Tuning

- Covering indexes for common queries
- Batch loading implementation
- N+1 query prevention

Monitoring

- Query performance tracking
- Index usage statistics
- Regular execution plan analysis

B.3 Infrastructure Implementation

B.3.1 Scaling Strategy

Auto-scaling Rules

- CPU threshold: 70%
- Latency threshold: 500ms
- Memory per task: 512MB
- vCPU per task: 0.25

Load Distribution

- Application Load Balancer
- Round-robin algorithm
- No session stickiness

B.3.2 Caching Architecture

Cache Layers

- L1: Application memory
- L2: Redis cluster
- CDN: Static assets

Cache Policies

- Tours: 5-minute TTL
- Sessions: 15-minute TTL
- Analytics: 1-hour TTL
- Maximum 256MB per Redis shard

B.4 Monitoring Implementation

B.4.1 Performance Monitoring

Key Metrics

- Response time
- Error rates
- Resource utilization
- Business KPIs

Alert Thresholds

- Error rate > 2%
- Response time > 1s
- CPU usage > 80%

B.4.2 Logging Strategy

Log Categories

- Application logs (JSON format)
- Security events
- Audit trail
- Performance metrics

Retention Policy

- Application logs: 30 days
- Security logs: 1 year
- Audit trails: 1 year
- Metrics: 90 days

B.4.3 Error Handling

Recovery Procedures

- Automatic retry with exponential backoff
- Circuit breaker for external services
- Fallback mechanisms for critical services

Error Documentation

- Standardized error codes
- User-friendly messages
- Detailed internal logging