MyTravel

Travel Booking Platform Documentation

Full-Stack Application Documentation

Generated: November 29, 2025

Table of Contents

MyTravel - Overview

Setup Guide

Features

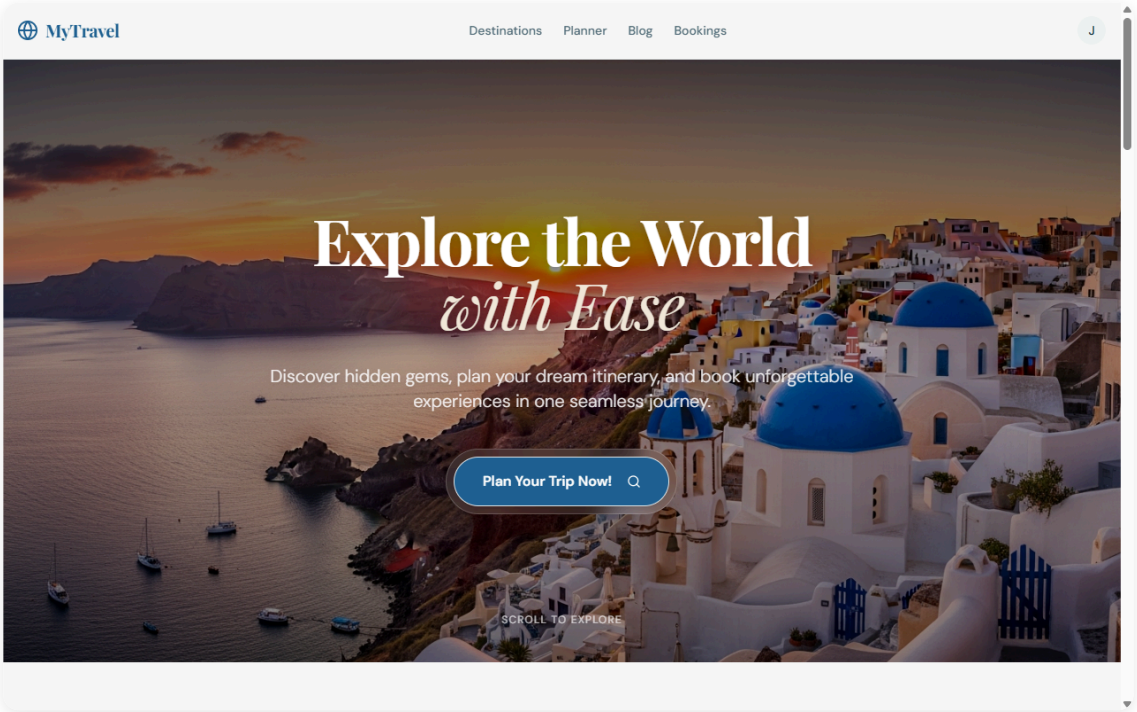
Architecture

Technologies

Error Handling

Appendix A: API Reference

MyTravel - Travel Booking Platform



Screenshot: MyTravel homepage with hero section and destination search

Project Description

MyTravel is a full-stack travel booking platform designed to provide users with a comprehensive travel planning experience. The application serves as a one-stop solution for travelers seeking to discover new destinations, book travel services, and organize their trips efficiently.

The platform enables users to browse an extensive catalog of travel destinations from around the world. Each destination includes detailed information, high-quality imagery, and transparent pricing to help travelers make informed decisions. Users can book flights, hotels, and guided tours directly through the platform, with all selections managed through an integrated shopping cart and checkout system.

A standout feature of MyTravel is the interactive trip planner, which allows users to create custom itineraries using drag-and-drop functionality. This tool empowers travelers to organize their activities day by day, visualize their journey, and adjust plans as needed before finalizing their bookings.

The platform also includes a travel blog section where users can read destination guides, travel tips, and stories from other travelers. Authenticated users have the ability to create and publish their own blog posts, contributing to the community knowledge base.

For returning users, MyTravel provides account management features including booking history, profile customization, and saved preferences. This personalization ensures a streamlined experience for frequent travelers.

The administrative side of the platform offers a comprehensive dashboard for managing the entire system. Administrators can oversee user accounts, process and track bookings, moderate blog content, and monitor platform statistics. This dual-interface approach ensures smooth operation for both end users and platform managers.

Documentation

Document	Description
Setup Guide	Instructions on how to install and run the project
Features	Complete list of pages and features with screenshots
Architecture	Overview of the code structure and organization
Technologies	Libraries and frameworks used in the project
API Reference	Full documentation of all API endpoints
Error Handling	Explanation of error handling strategies

Setup Guide

This guide provides instructions for setting up and running the MyTravel application on your local development environment.

Prerequisites

The following software must be installed on your machine before proceeding with the setup:

- **.NET 10 SDK** - The backend is built on ASP.NET Core 10. Download from dotnet.microsoft.com
- **Node.js (v18 or higher)** - Required for the frontend build tools. Download from nodejs.org
- **npm** - Comes bundled with Node.js and is used for managing frontend dependencies

To verify your installations, run the following commands in your terminal:

```
dotnet --version    # Should output 10.x.x
node --version      # Should output v18.x.x or higher
npm --version       # Should output 9.x.x or higher
```

Installation

1. Install Frontend Dependencies

Navigate to the client project and install the required npm packages:

```
cd mytravel.client
npm install
cd ..
```

2. Configure the Backend (Optional)

The application comes with default configuration settings that work out of the box for local development. The configuration files are located in the `MyTravel.Server` directory:

- `appsettings.json` - Base configuration

- `appsettings.Development.json` - Development-specific settings

The default admin credentials are configured in `appsettings.json`:

```
{
  "AdminCredentials": {
    "Email": "admin@mytravel.com",
    "Password": "Admin@123!"
  }
}
```

For production deployments, these credentials should be changed and stored securely using environment variables or a secrets manager.

Running the Application

The MyTravel solution uses the ASP.NET Core SPA Proxy feature, which means the frontend development server starts automatically when you run the backend. This provides a seamless development experience where both the API and client are available from a single command.

Using Visual Studio

1. Open `MyTravel.slnx` in Visual Studio
2. Set `MyTravel.Server` as the startup project
3. Press `F5` to run with debugging, or `Ctrl+F5` to run without debugging

Visual Studio will automatically start both the backend server and the frontend development server.

Using the Command Line

Navigate to the server project directory and run:

```
cd MyTravel.Server
dotnet run
```

For a more interactive development experience, use `dotnet watch` which provides hot reload capabilities. This automatically rebuilds and restarts the application when code changes are detected:

```
cd MyTravel.Server
dotnet watch
```

The `dotnet watch` command is recommended during active development as it significantly speeds up the feedback loop when making changes to the backend code.

Accessing the Application

Once the application is running, you can access it at the following URLs:

Service	URL	Description
Frontend	http://localhost:49764	Main application interface
Backend API	http://localhost:5083	API endpoints
Scalar API Docs	http://localhost:5083/scalar/v1	Interactive API documentation (development only)

The frontend development server proxies API requests to the backend, so you can interact with the application entirely through the frontend URL during development.

Database

MyTravel uses SQLite as its database, which requires no additional setup. The database file is created automatically when the application first runs and is stored in the `MyTravel.Server` directory as `mytravel.db`.

The database is seeded with sample data on first run, including:

- Sample destinations and activities
- Demo blog posts
- The admin user account

To reset the database, simply delete the `mytravel.db` file and restart the application.

Troubleshooting

Port Already in Use

If you encounter an error indicating that a port is already in use, another application may be using ports 5083 or 49764. You can either stop the conflicting application or modify the ports in the following files:

- `MyTravel.Server/Properties/launchSettings.json` - Backend port configuration
- `mytravel.client/vite.config.ts` - Frontend port configuration

Frontend Not Starting Automatically

If the frontend does not start automatically with the backend, you can run it manually in a separate terminal:

```
cd mytravel.client
npm run dev
```

Certificate Errors

The development server uses HTTPS with a self-signed certificate. If your browser shows a certificate warning, you can proceed by accepting the risk (for development purposes only) or trust the .NET development certificate:

```
dotnet dev-certs https --trust
```

Features

This document provides a comprehensive overview of all pages and features available in the MyTravel application. The platform is divided into two main sections: the public-facing website for travelers and the administrative dashboard for platform management.

Public Pages

Homepage

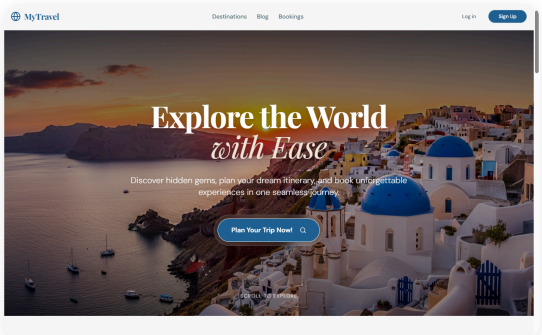
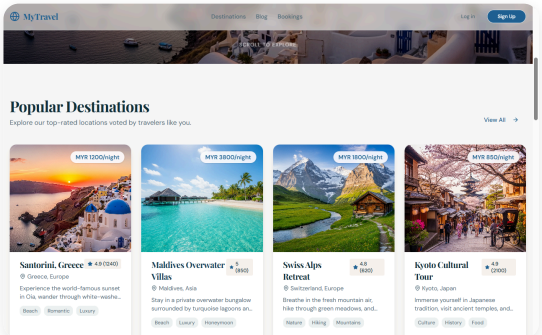
Route: `/`

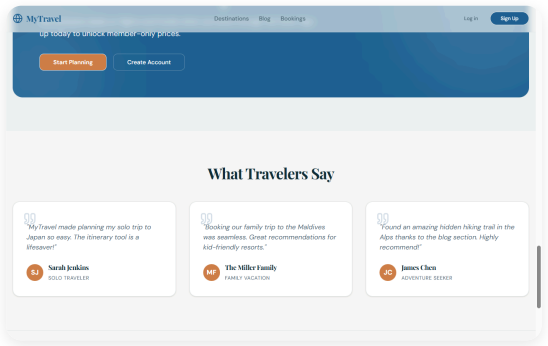
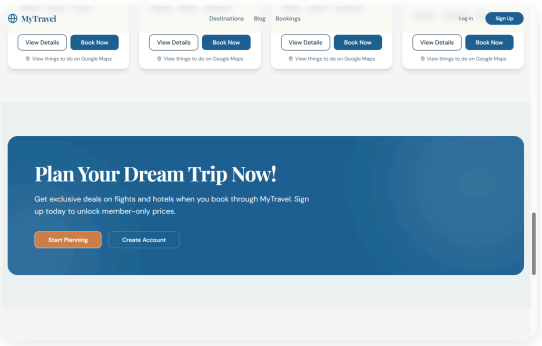
"Explore the World with Ease"

The homepage serves as the main entry point for visitors and establishes the MyTravel brand identity. A prominent hero section displays the website's name alongside a captivating tagline that invites users to begin their travel journey. The hero includes a search interface that allows users to quickly find destinations by location or keyword.

Below the hero, the page highlights top travel destinations with visually appealing cards showcasing popular locations. A testimonials section features reviews from satisfied travelers, building trust with new visitors. Promotional offers and seasonal deals are prominently displayed to encourage immediate engagement. A clear call-to-action button labeled "Plan Your Trip Now!" guides users toward the booking flow.

Screenshots

Hero Section	Featured Destinations
	
Hero with tagline and search	*Top destination cards*

Testimonials	Call to Action Card
	
User reviews and ratings	*Call to Action Card*

User Registration and Login

Routes: `/login` , `/register`

"Create an account to access personalized travel deals."

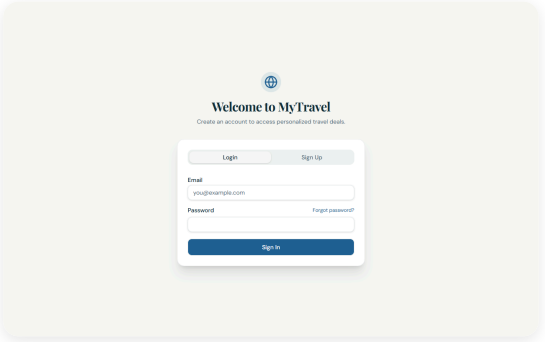
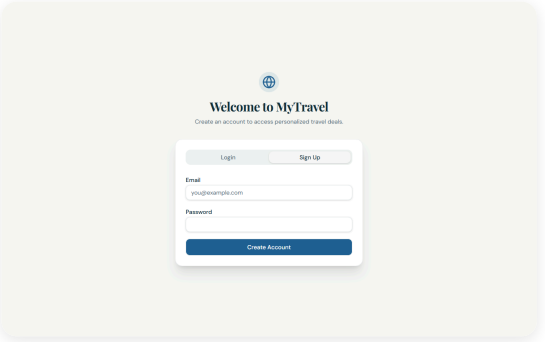
The authentication system provides secure access to personalized features and booking capabilities. The registration form collects essential information including email address, first name, last name, and password with confirmation. Form validation is handled client-side using react-hook-form integrated with Zod schemas, providing immediate feedback on input errors.

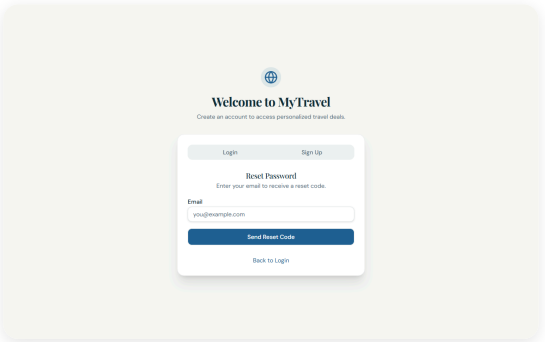
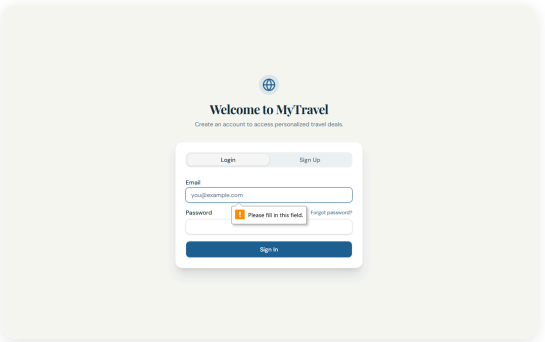
The login form accepts email and password credentials with options for password visibility toggle. Account verification is supported through email confirmation to ensure valid user accounts. A forgot password feature allows users to reset their credentials by receiving a password reset link via email. User sessions are managed through ASP.NET Core Identity with secure cookie-based authentication.

Features:

- Email-based registration with validation
- Account verification via email confirmation
- Forgot password functionality with email reset
- Secure session management
- Protected route redirection after login

Screenshots

Login Form	Registration Form
	
Email and password login	*New account creation*

Password Recovery	Form Validation
	
Password reset request	*Real-time validation feedback*

Destination Browsing Page

Route: `/destinations`

"Discover hidden gems across the world."

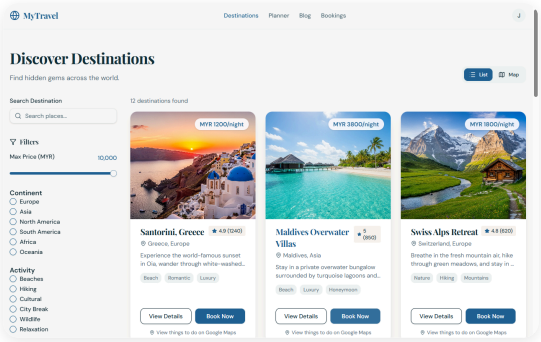
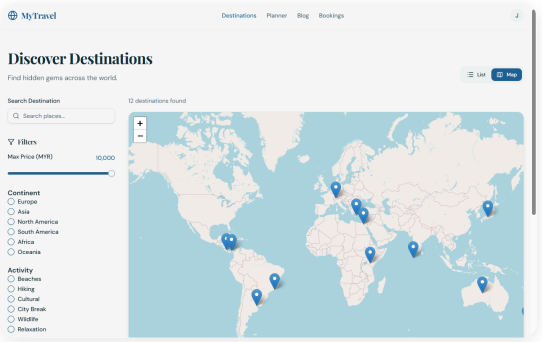
The destinations page presents a comprehensive catalog of all available travel locations with rich visual content. Each destination is displayed as a card featuring high-quality images, descriptive text highlighting key attractions, and aggregated user reviews. Pricing information shows the starting cost to help users quickly assess options within their budget.

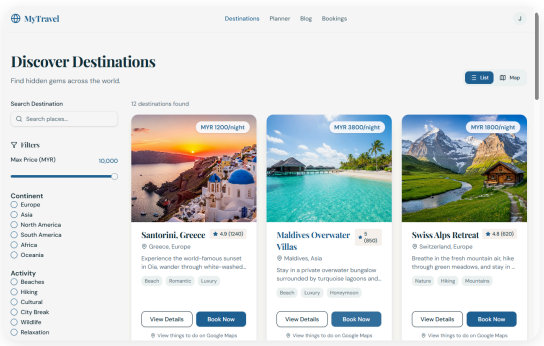
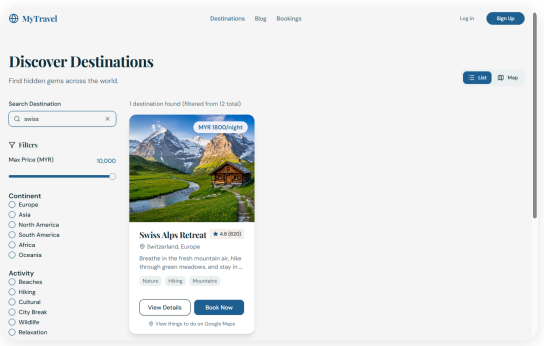
The page offers robust filtering capabilities to help users find their ideal destination. Filters are available for continent selection, activity type (beaches, hiking, cultural experiences, adventure sports), and budget range. Users can toggle between a traditional grid view for browsing and an interactive map view powered by Leaflet for geographical exploration. The search functionality allows users to find specific destinations by name or keyword.

Filter Options:

- Continent/Region selection
- Activity type (beaches, hiking, cultural, adventure)
- Budget range (economy, mid-range, luxury)
- Rating and review scores
- Seasonal availability

Screenshots

Grid View	Map View
	
Card-based destination browsing	*Interactive Leaflet map view*

Filter Panel	Search Results
	
Continent, activity, and budget filters	*Filtered destination results*

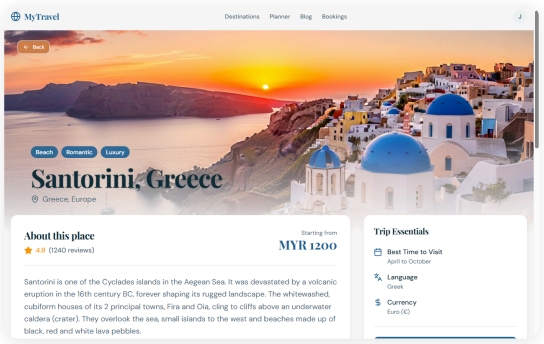
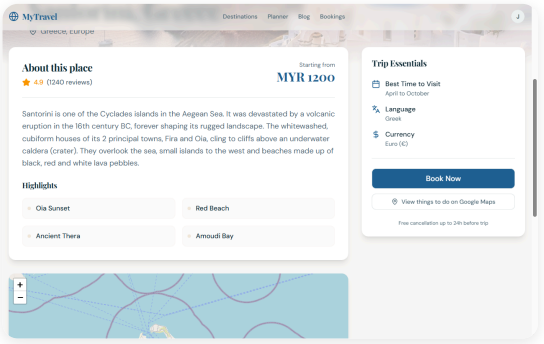
Destination Details

Route: `/destinations/:id`

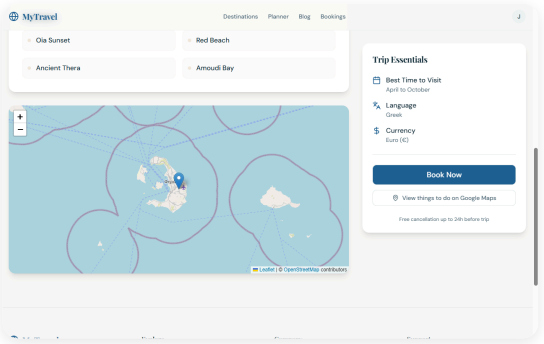
Individual destination pages provide in-depth information about a specific location. The page opens with a gallery of high-quality images showcasing the destination's highlights. Detailed descriptions cover the location's history, culture, climate, and best times to visit.

Available activities are listed with descriptions and pricing, allowing users to plan their experiences. Accommodation options range from budget-friendly to luxury establishments. Users can add flights, hotels, or guided tours to their cart directly from this page, streamlining the booking process.

Screenshots

Image Gallery	Description
	
High-quality destination images	*Details*

Available Activities and Map



Tours and experiences

Itinerary Planner

Route: `/planner`

"Plan your dream trip effortlessly."

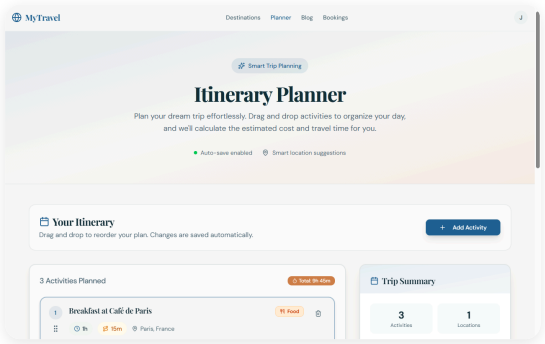
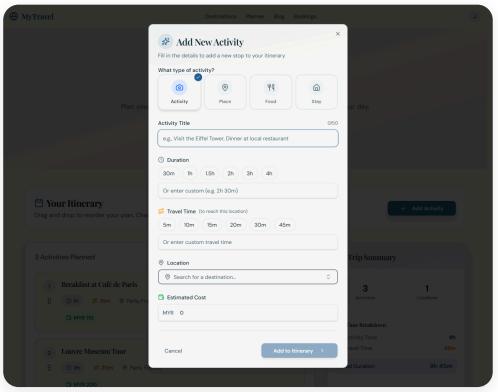
The itinerary planner is an interactive tool that empowers authenticated (registered) users to create custom travel itineraries tailored to their preferences. The interface enables users to select places, accommodations, and activities, then organize them into a coherent travel plan.

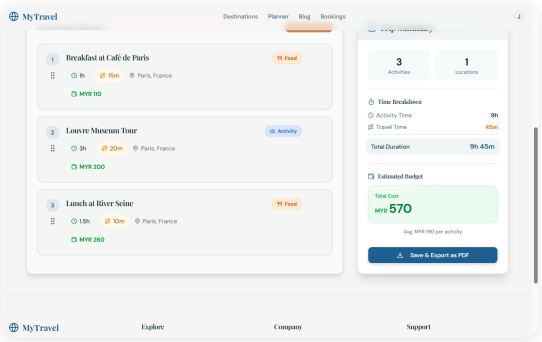
The drag-and-drop functionality, powered by the dnd-kit library, allows users to easily arrange activities across different days of their trip. Users can reorder items, move activities between days, and visualize their complete journey timeline. The planner calculates estimated travel times between locations and provides running cost totals as items are added or removed.

Features:

- Drag-and-drop itinerary creation
- Day-by-day activity organization
- Estimated travel time calculations
- Running cost totals and budget tracking
- Export PDF

Screenshots

Planner Page	Adding Activities
	
Planner Header	*Selecting activities to add*

Itinerary

Activities

Booking System

Routes: `/booking/:id` , `/checkout`

"Book everything you need for your trip in one place."

The booking system provides a unified interface for reserving all travel services. The booking page presents detailed forms for configuring reservations, including travel dates, number of travelers, and specific options such as room types, flight classes, or tour packages. Real-time availability checking ensures users only see bookable options.

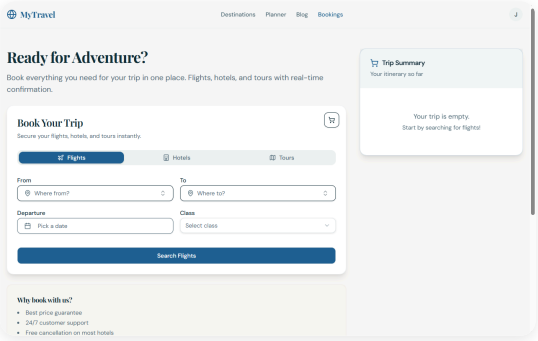
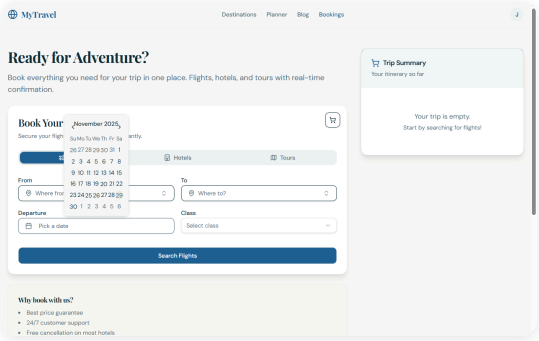
The checkout page consolidates all items from the user's cart into a comprehensive order summary. An itemized breakdown displays each service with its details and pricing. The system calculates total costs including any applicable taxes and fees. Customer information collection includes contact details and any special requirements.

Payment processing is handled securely with real-time confirmation. Upon successful payment, users receive immediate booking confirmation displayed on screen and sent via email. The confirmation includes booking reference numbers, detailed itinerary information, and relevant contact details for each service provider.

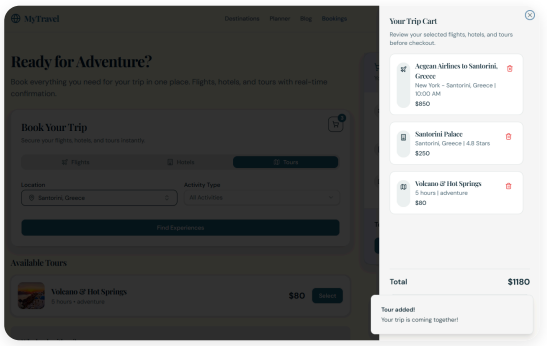
Features:

- Integrated booking for flights, hotels, and tours
- Real-time availability and pricing
- Secure payment processing
- Instant booking confirmation
- Email confirmation with booking details
- Booking reference numbers for each service

Screenshots

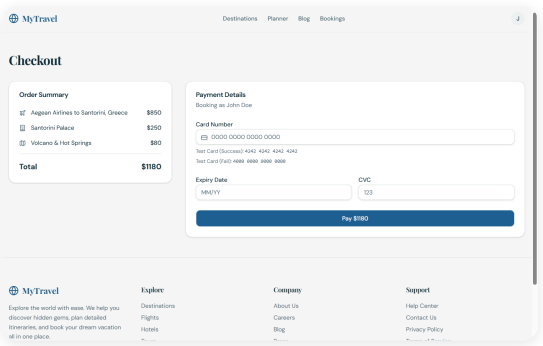
Booking Form	Date Selection
	
Service configuration options	*Travel date selection*

Cart Summary



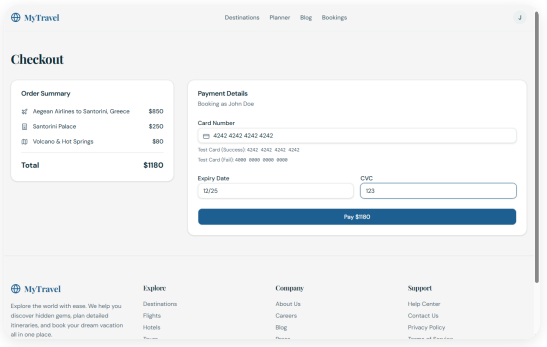
Itemized booking summary

Checkout Page



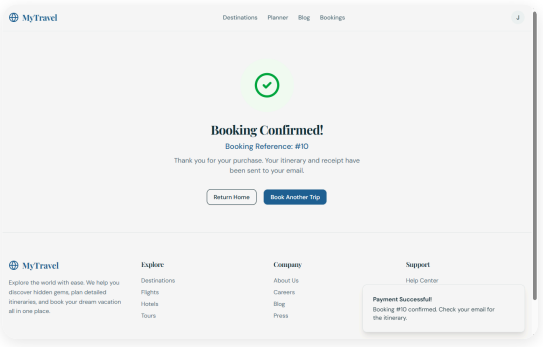
Customer information entry

Payment Processing



Secure payment form

Confirmation



Booking success with reference

Interactive Map Integration

The application integrates interactive mapping capabilities using the Leaflet library with React-Leaflet bindings. Maps are featured prominently on the destinations browsing page, allowing users to explore travel locations geographically.

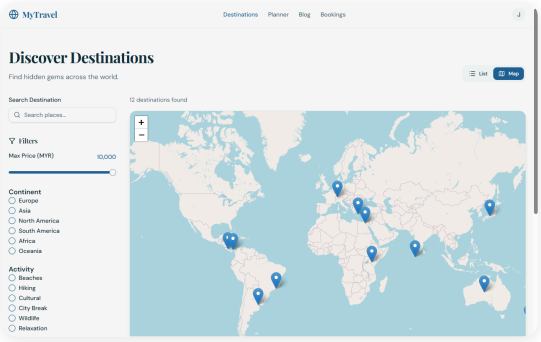
"Explore destinations and travel routes on an interactive map."

Users can pan and zoom across the world map, with destination markers indicating available locations. Clicking a marker reveals a popup with destination summary information and a link to the full details page. The map view provides an intuitive way to discover destinations by region and understand geographical relationships between locations.

On individual destination pages, maps display the specific location along with nearby points of interest, helping users understand the area and plan local exploration.

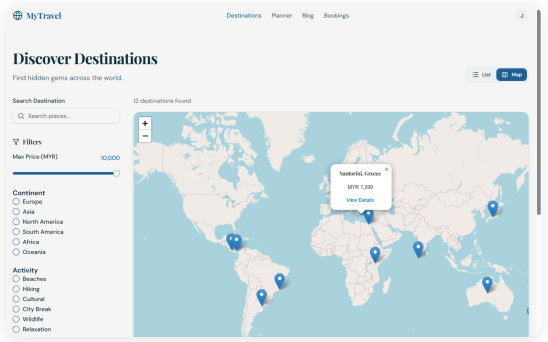
Screenshots

World Map View



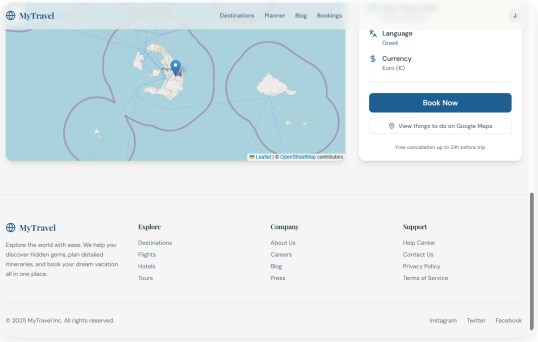
Global destination overview

Marker Popup



Quick destination info

Destination Map



Detailed location view

Travel Blog

Routes: `/blog` , `/blog/:slug` , `/blog/new` , `/blog/edit/:slug`

"Get inspired by amazing travel experiences."

The blog section serves as a community-driven content hub where users can read and contribute travel stories, tips, and destination guides. The main blog page displays posts in a paginated list with preview cards showing titles, excerpts, featured images, and publication dates.

Content is organized into categories to help readers find relevant articles. Categories include solo travel adventures, family trip planning, luxury travel experiences, budget backpacking, cultural immersion, and adventure activities. Users can browse all posts or filter by their preferred category.

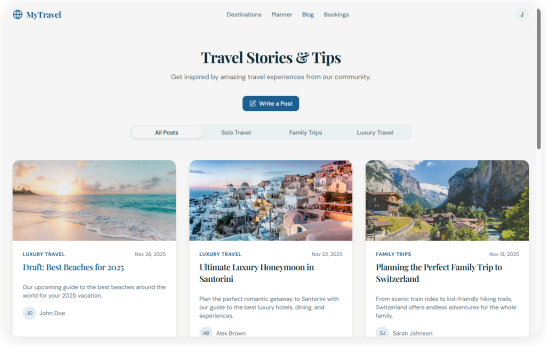
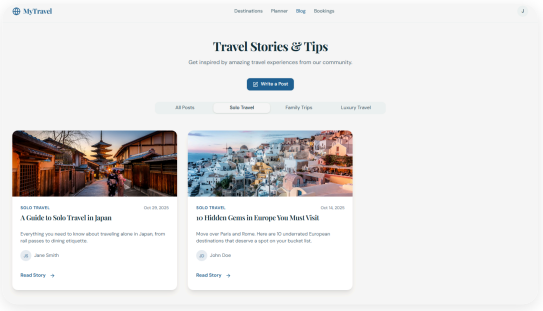
Individual blog posts are rendered with full rich text content using the Lexical editor's read mode. Articles support formatted text, headings, embedded images, and other media elements. Related posts are suggested at the bottom of each article to encourage continued exploration.

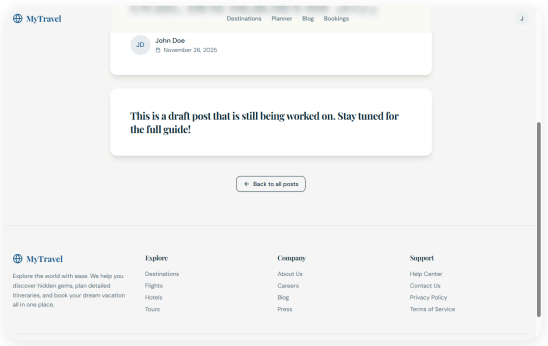
Authenticated users can contribute their own travel stories through the blog editor. The editor provides a rich text interface supporting formatted content, image uploads, and category selection. Authors can save drafts, preview their work, and publish when ready to share with the community.

Content Categories:

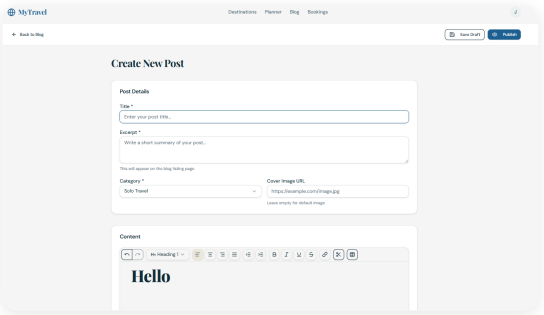
- Solo travel
- Family trips
- Luxury travel
- Budget backpacking
- Cultural experiences
- Adventure activities

Screenshots

Blog Listing	Category Filter
	
Paginated post cards	*Filter by travel category*

Blog Post	Rich Content
	
Full article view	*Formatted text and images*

Blog Editor



Lexical rich text editor

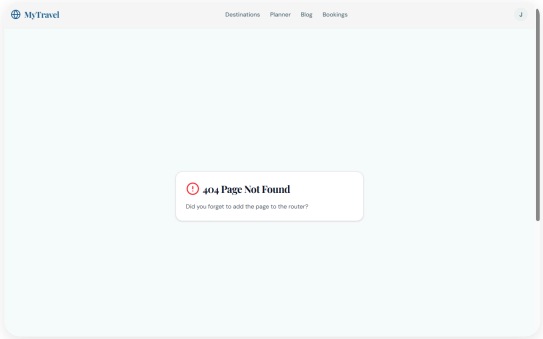
Not Found

Route: `*` (catch-all)

A custom 404 page is displayed when users navigate to a route that does not exist. The page provides a friendly message and links back to the homepage and other main sections of the site, helping users recover from navigation errors gracefully.

Screenshots

404 Page



Page not found with navigation options

Administrative Pages

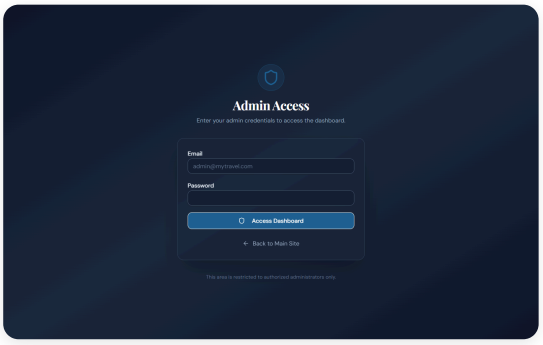
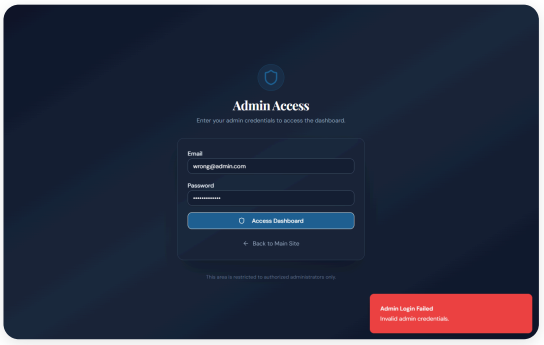
The admin section is protected and requires administrator credentials to access. It provides comprehensive tools for managing all aspects of the platform.

Admin Login

Route: `/admin/login`

The administrative login page is separate from the public user login to maintain security isolation. It accepts the admin email and password configured in the application settings. Successful authentication creates an admin session cookie that grants access to all administrative features. Failed login attempts are logged for security monitoring.

Screenshots

Admin Login	Login Error
	
Administrator authentication	*Invalid credentials handling*

Admin Dashboard

Route: `/admin`

The main dashboard provides a centralized overview of platform statistics, analytics, and recent activity. Key performance metrics are displayed prominently, including total registered users, active bookings, published blog posts, and revenue summaries.

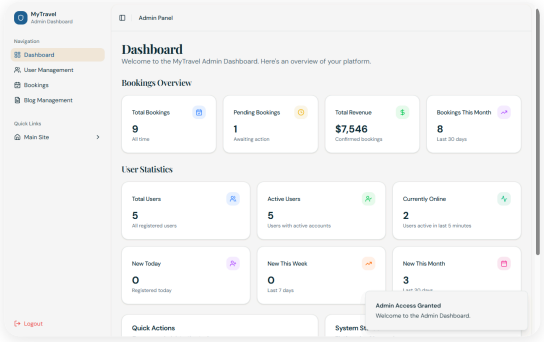
Analytics and Metrics:

- Visitor traffic statistics
- User registration trends
- Booking volume and revenue charts
- Blog engagement metrics
- Active user sessions

Charts built with Recharts visualize trends over time, helping administrators identify patterns and make data-driven decisions. Quick action buttons provide shortcuts to common administrative tasks such as approving pending bookings or reviewing new user registrations. Recent activity feeds show the latest user actions and system events.

Screenshots

Dashboard Overview



Main admin interface

User Management

Route: `/admin/users`

The user management page displays a searchable, paginated table of all registered users. The search functionality allows administrators to find users by name, email, or other attributes. Pagination controls handle large user bases efficiently.

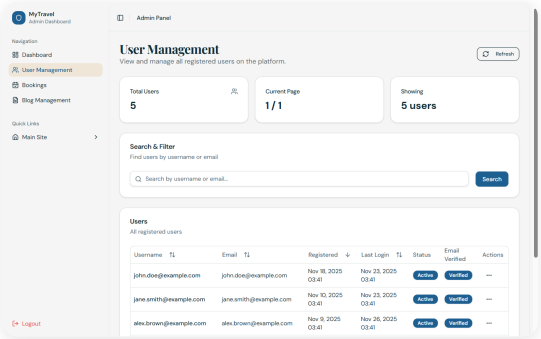
Management Capabilities:

- View detailed user profiles
- Edit user information
- Toggle account active/inactive status
- Delete user accounts
- View registration dates and last login times
- Monitor user activity patterns

Administrators can activate or deactivate accounts as needed for moderation purposes. The interface provides clear visual indicators of account status and important dates.

Screenshots

User List



Paginated user table

Booking Management

Route: `/admin/bookings`

This page provides complete visibility into all bookings made through the platform. The booking list can be filtered by status categories: pending (awaiting confirmation), confirmed (approved and scheduled), cancelled (user or admin cancelled), and completed (past bookings).

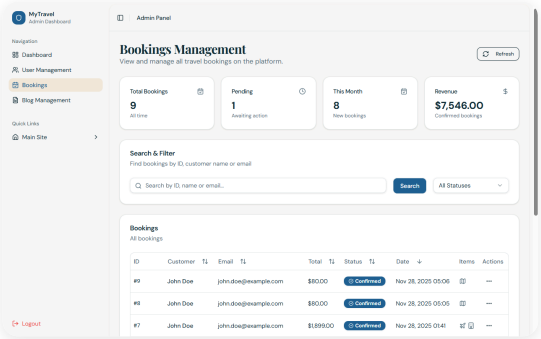
Management Features:

- Search bookings by customer name or email
- Filter by booking status
- View detailed booking information
- Update booking status
- Track revenue by period
- Export booking data

Administrators can view comprehensive details for each booking including customer information, itemized services, payment status, and timeline. Status updates trigger notification emails to customers, keeping them informed of any changes.

Screenshots

Booking List



All bookings overview

Blog Management

Route: `/admin/blog`

The blog management interface provides administrative control over all blog content. The page lists all posts regardless of publication status, giving administrators visibility into both published articles and pending drafts.

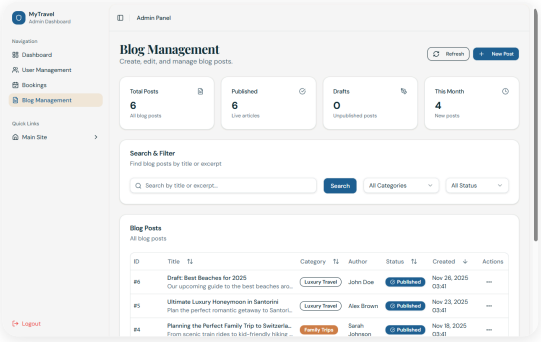
Management Capabilities:

- View all posts (published and drafts)
- Edit existing content
- Toggle publication status
- Delete posts
- Review content before publication
- Monitor blog statistics

Statistics display engagement metrics for blog content, helping administrators understand which topics resonate with readers. Content moderation tools ensure quality and appropriateness of community contributions.

Screenshots

Post List



All posts overview

Search and Filtering

The application provides comprehensive search and filtering capabilities throughout the platform. A global search bar in the navigation allows users to find destinations and blog articles from any page.

Search Features:

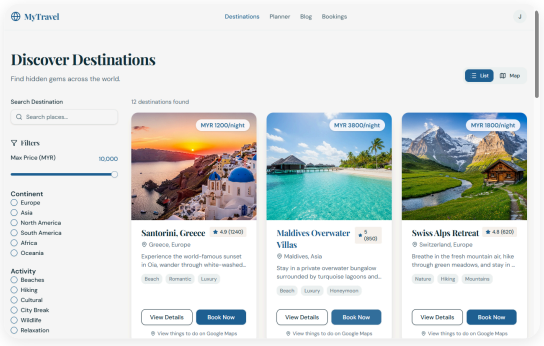
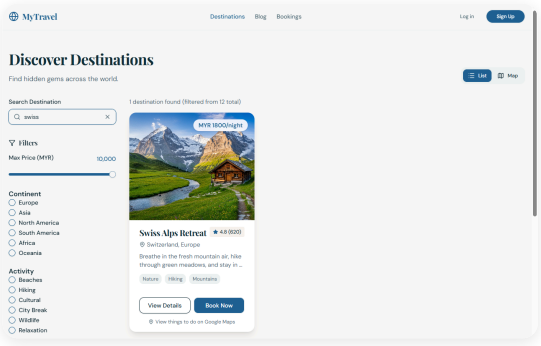
- Destination search by name or keyword
- Blog article search by title or content
- Real-time search suggestions

Advanced Filters:

- Price range sliders
- User rating thresholds
- Seasonal availability (best time to visit)
- Activity type categorization
- Continent and region selection

Filters can be combined to narrow results precisely. Filter selections are preserved during navigation, allowing users to refine their search incrementally.

Screenshots

Filter Panel	Search Results
	
Continent, activity, and budget filters	*Filtered destination results*

Core Features

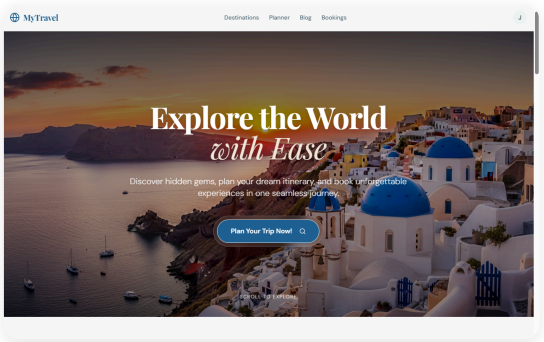
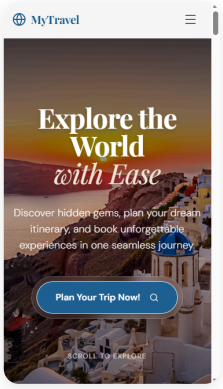
Responsive Design and Styling

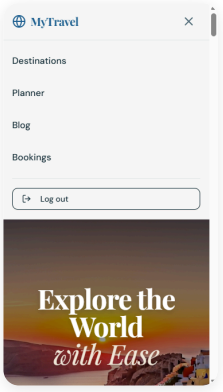
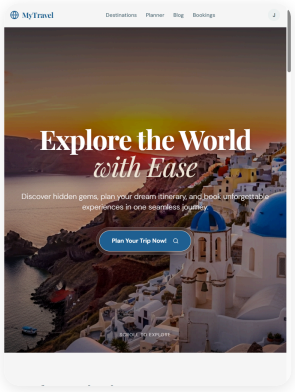
The entire application is built with a visually appealing and user-friendly layout designed to enhance the user experience. High-quality images showcase destinations and create visual appeal throughout the site. Clear typography using modern font stacks ensures readability across all devices.

Responsive design principles implemented with Tailwind CSS ensure the interface adapts seamlessly to different screen sizes. On mobile devices, navigation collapses into an accessible hamburger menu, images resize appropriately, and layouts stack vertically for comfortable scrolling. Tablet and desktop views utilize available screen space with multi-column layouts and expanded navigation.

Intuitive navigation patterns guide users through the site logically. Consistent visual hierarchy helps users understand page structure and find information quickly. Interactive elements provide clear feedback through hover states, focus indicators, and transition animations powered by Framer Motion.

Screenshots

Desktop Layout	Mobile Layout
	
Full desktop experience	*Mobile-optimized view*

Mobile Navigation	Tablet View
	
Hamburger menu expanded	*Tablet layout*

Error Handling

"Ensure smooth user experiences by providing meaningful error messages."

The application implements comprehensive error handling to maintain a smooth user experience even when issues occur. Payment failures display clear messages explaining the issue and suggesting resolution steps. Unavailable services are communicated with helpful alternatives when possible.

Network errors are caught and displayed with retry options. Form validation errors appear inline next to the relevant fields with specific guidance on how to correct the input. Server errors are logged for debugging while users see friendly messages that avoid technical jargon.

Toast notifications provide non-intrusive feedback for both successful actions and error conditions. The notification system uses the Sonner library to display consistent, accessible messages that automatically dismiss after an appropriate duration.

For detailed information about error handling implementation, see Error Handling.

Analytics Integration

The application integrates PostHog analytics to track user behavior and optimize the platform experience. Analytics capture page views, feature usage patterns, and conversion metrics while respecting user privacy.

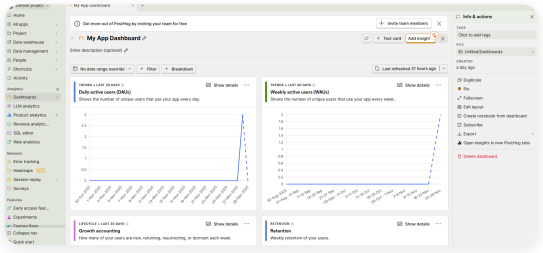
Tracked Metrics:

- Page view counts and user flows
- Feature engagement (planner usage, blog reads)
- Booking funnel conversion rates
- Search and filter usage patterns
- Error occurrence frequency

Analytics data informs decisions about content prioritization, feature development, and user experience improvements. The integration is configured to comply with privacy requirements and can be adjusted based on user consent preferences.

Screenshots

Page Analytics



Page view tracking

Real-time Form Validation

Forms throughout the application provide immediate feedback using react-hook-form integrated with Zod validation schemas. Users see validation errors as they type, helping them correct issues before submission. This client-side validation reduces server load and improves the user experience by providing instant feedback.

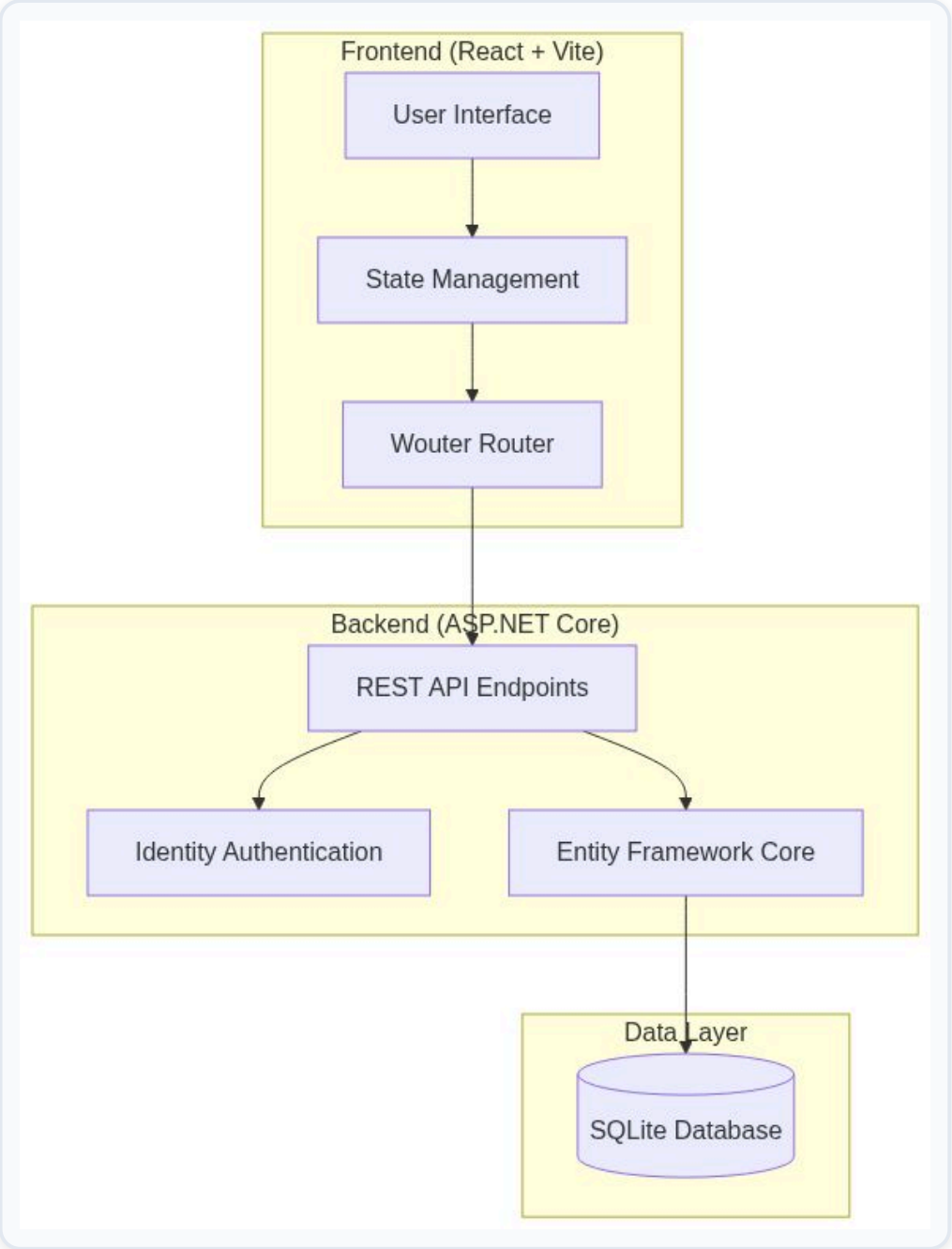
Validation rules enforce data quality requirements including email format verification, password strength requirements, required field completion, and logical constraints such as departure dates preceding return dates.

Architecture

This document provides an overview of the MyTravel application's code structure and organization. The project follows a modern full-stack architecture with a clear separation between the frontend React application and the backend ASP.NET Core API.

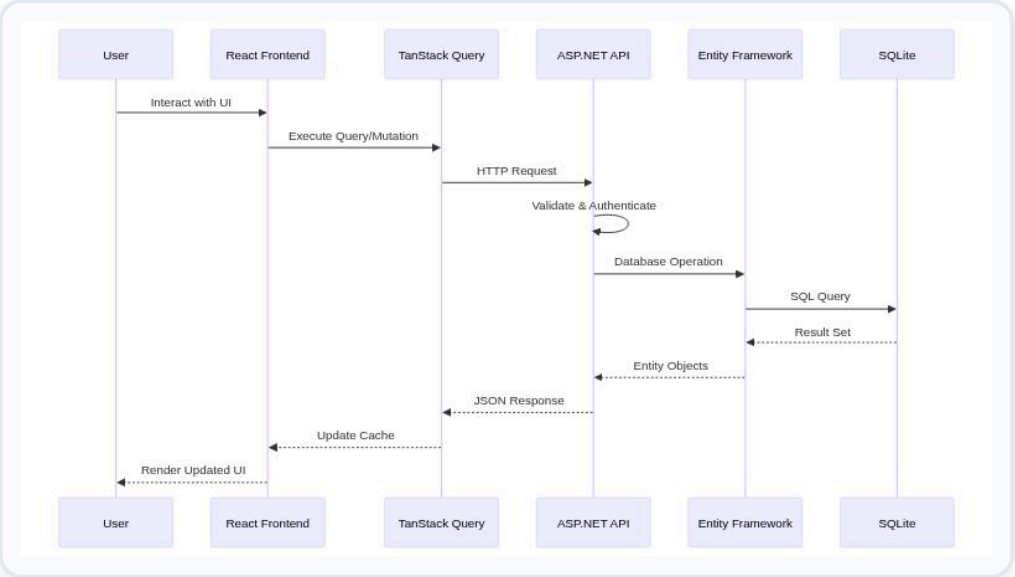
High-Level Architecture

The application consists of two main components that communicate over HTTP. The frontend handles all user interface concerns while the backend manages data persistence, business logic, and authentication.



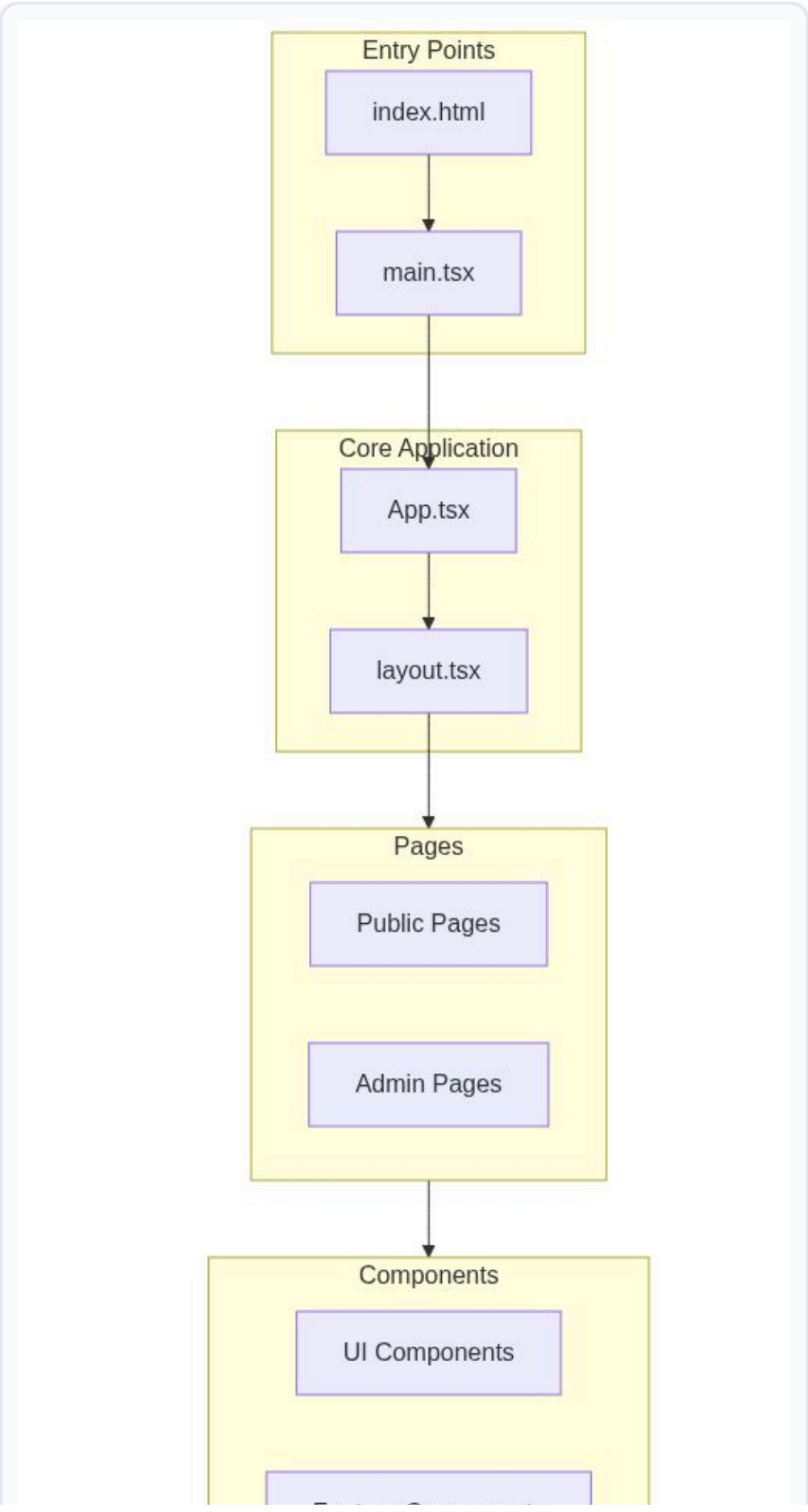
Request Flow

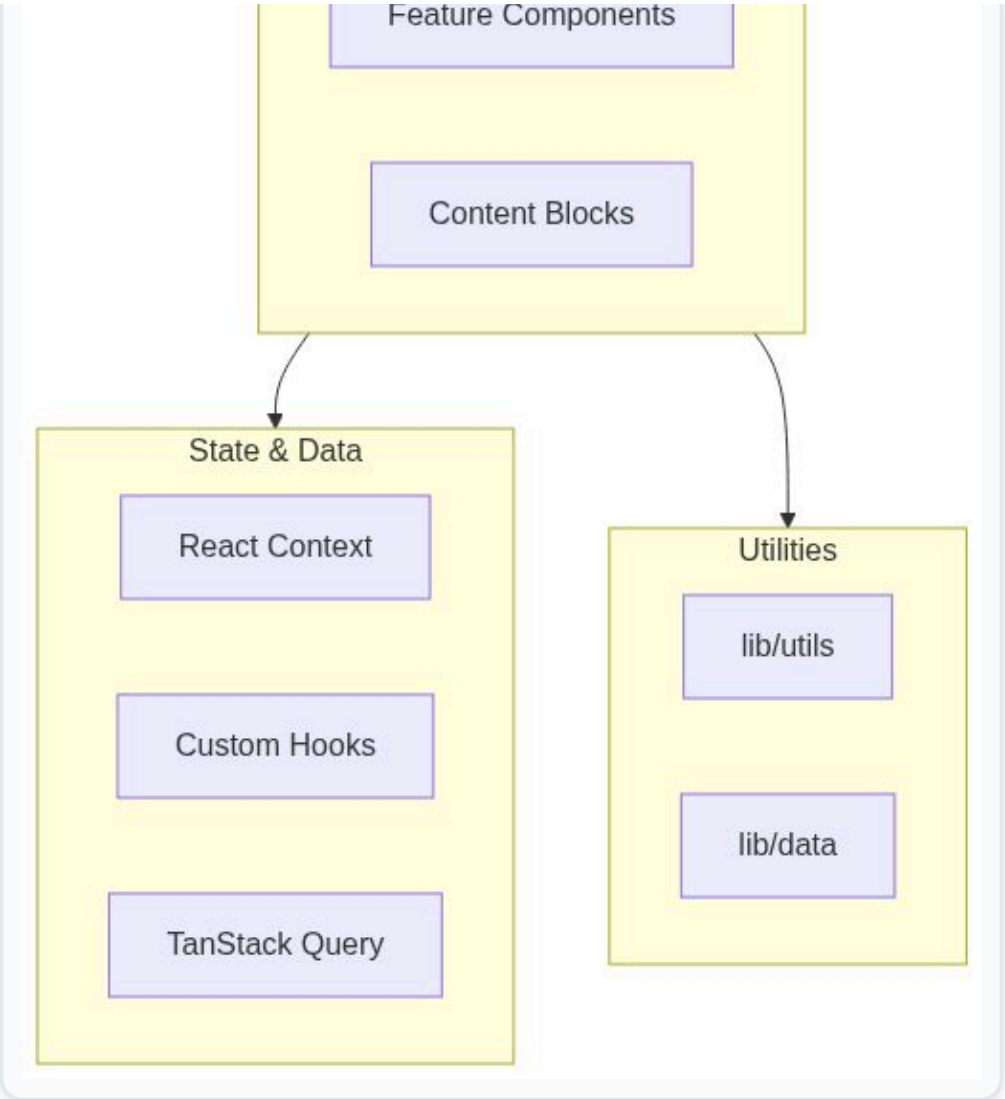
When a user interacts with the application, requests flow through several layers before reaching the database. This diagram illustrates the typical request lifecycle.



Frontend Architecture

The frontend is built with React and TypeScript, using Vite as the build tool. The codebase follows a feature-based organization pattern with clear separation of concerns.


Entry Points
index.html
main.tsx
Core Application
App.tsx
layout.tsx
Pages
Public Pages
Admin Pages
Components
UI Components



Frontend Directory Structure

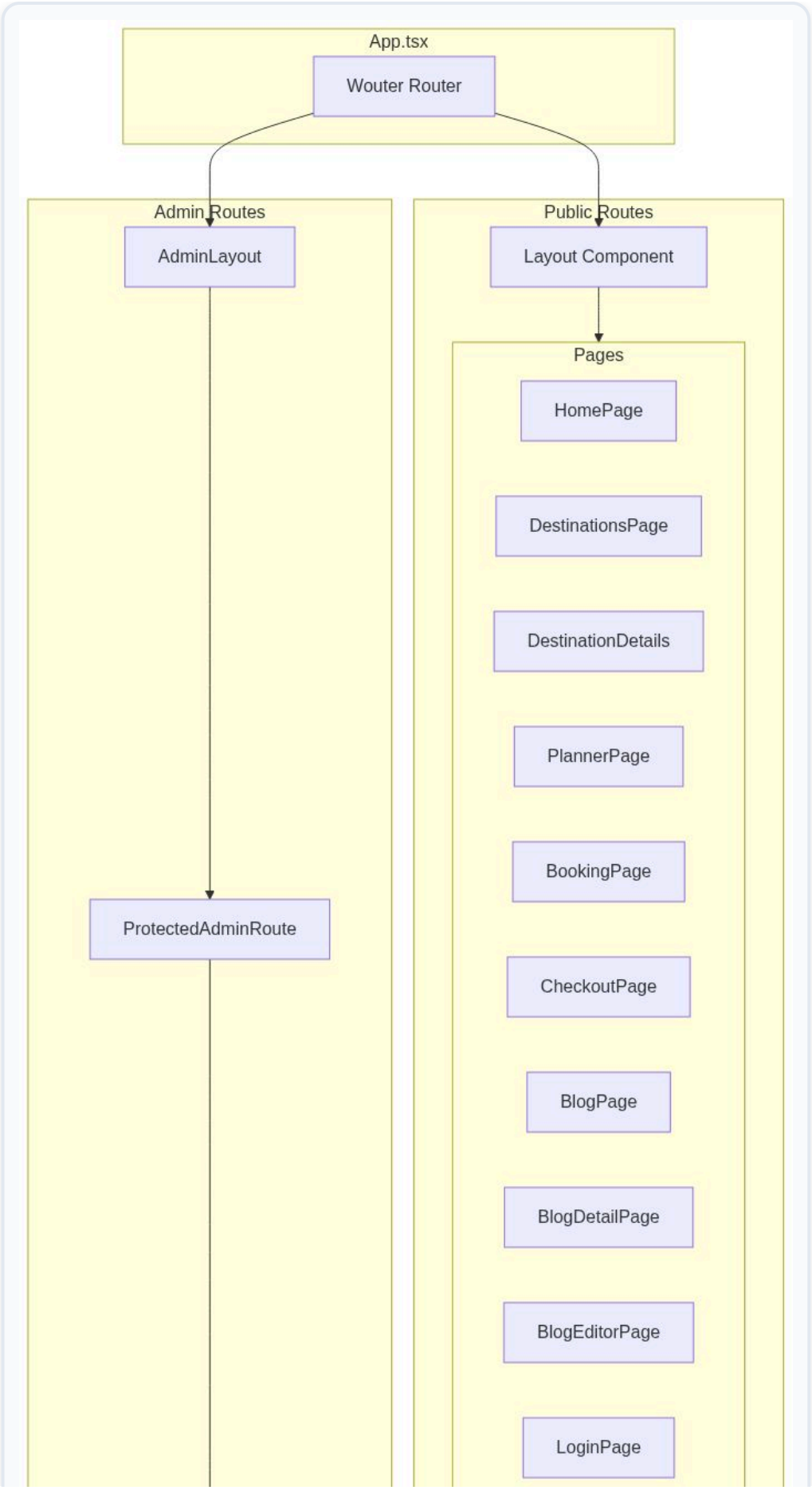
```
mytravel.client/
├── client/
│   ├── index.html                # HTML entry point
│   ├── public/                   # Static assets (images, fonts)
│   └── src/
│       ├── App.tsx               # Root component with routing
│       ├── main.tsx              # React entry point
│       ├── index.css             # Global styles (Tailwind)
│       │
│       ├── components/           # Reusable components
│       │   ├── admin/            # Admin-specific components
│       │   │   ├── admin-layout.tsx
│       │   │   ├── protected-admin-route.tsx
│       │   │   └── ...
│       │   ├── blocks/           # Content block components
│       │   ├── editor/           # Lexical editor components
│       │   ├── ui/               # Base UI components (55+)
│       │   │   ├── button.tsx
```

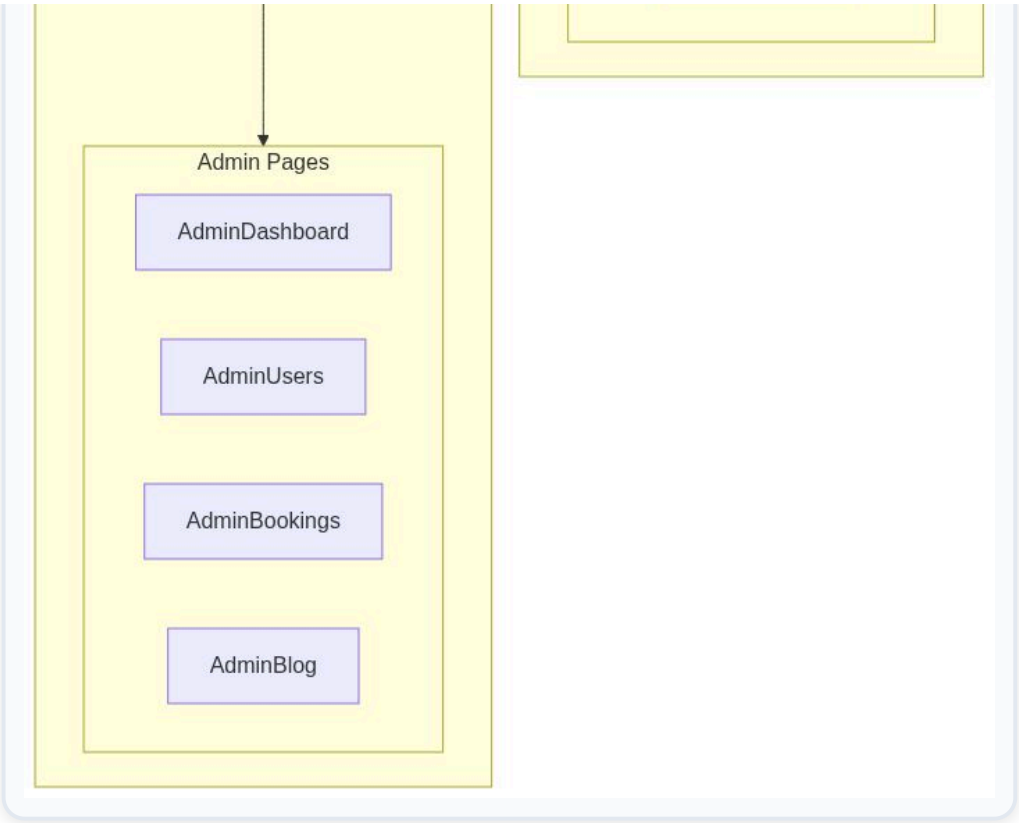
```
│   │   │   ├── card.tsx
│   │   │   ├── dialog.tsx
│   │   │   └── ...
│   │   ├── booking-form.tsx
│   │   ├── cart-sheet.tsx
│   │   ├── destination-card.tsx
│   │   ├── hero.tsx
│   │   ├── itinerary-planner.tsx
│   │   ├── layout.tsx
│   │   └── trip-summary.tsx
│   │
│   ├── context/              # React context providers
│   │   ├── blog-context.tsx  # Blog state management
│   │   └── cart-context.tsx  # Shopping cart state
│   │
│   ├── hooks/                # Custom React hooks
│   │   ├── use-admin-auth.tsx # Admin authentication
│   │   ├── use-auth.tsx      # User authentication
│   │   ├── use-mobile.tsx    # Responsive detection
│   │   ├── use-ping.tsx      # Activity tracking
│   │   └── use-toast.ts      # Toast notifications
│   │
│   ├── lib/                  # Utilities and data
│   │   ├── data.ts           # API client functions
│   │   ├── destinations.ts   # Destination data
│   │   ├── mock-data.ts      # Development mock data
│   │   ├── queryClient.ts    # TanStack Query config
│   │   └── utils.ts          # Helper functions
│   │
│   └── pages/                # Page components
│       ├── admin/            # Admin pages
│       │   ├── admin-blog.tsx
│       │   ├── admin-bookings.tsx
│       │   ├── admin-dashboard.tsx
│       │   ├── admin-login.tsx
│       │   └── admin-users.tsx
│       ├── blog.tsx
│       ├── blog-detail.tsx
│       ├── blog-editor.tsx
│       ├── booking.tsx
│       ├── checkout.tsx
│       ├── destination-details.tsx
│       ├── destinations.tsx
│       ├── home.tsx
```

```
│           ├── login.tsx
│           ├── not-found.tsx
│           └── planner.tsx
│
├── attached_assets/              # Generated images
├── components.json               # shadcn/ui configuration
├── eslint.config.js              # ESLint configuration
├── package.json                  # Dependencies and scripts
├── postcss.config.js             # PostCSS configuration
├── tailwind.config.js            # Tailwind CSS configuration
├── tsconfig.json                 # TypeScript configuration
└── vite.config.ts                # Vite build configuration
```

Component Hierarchy

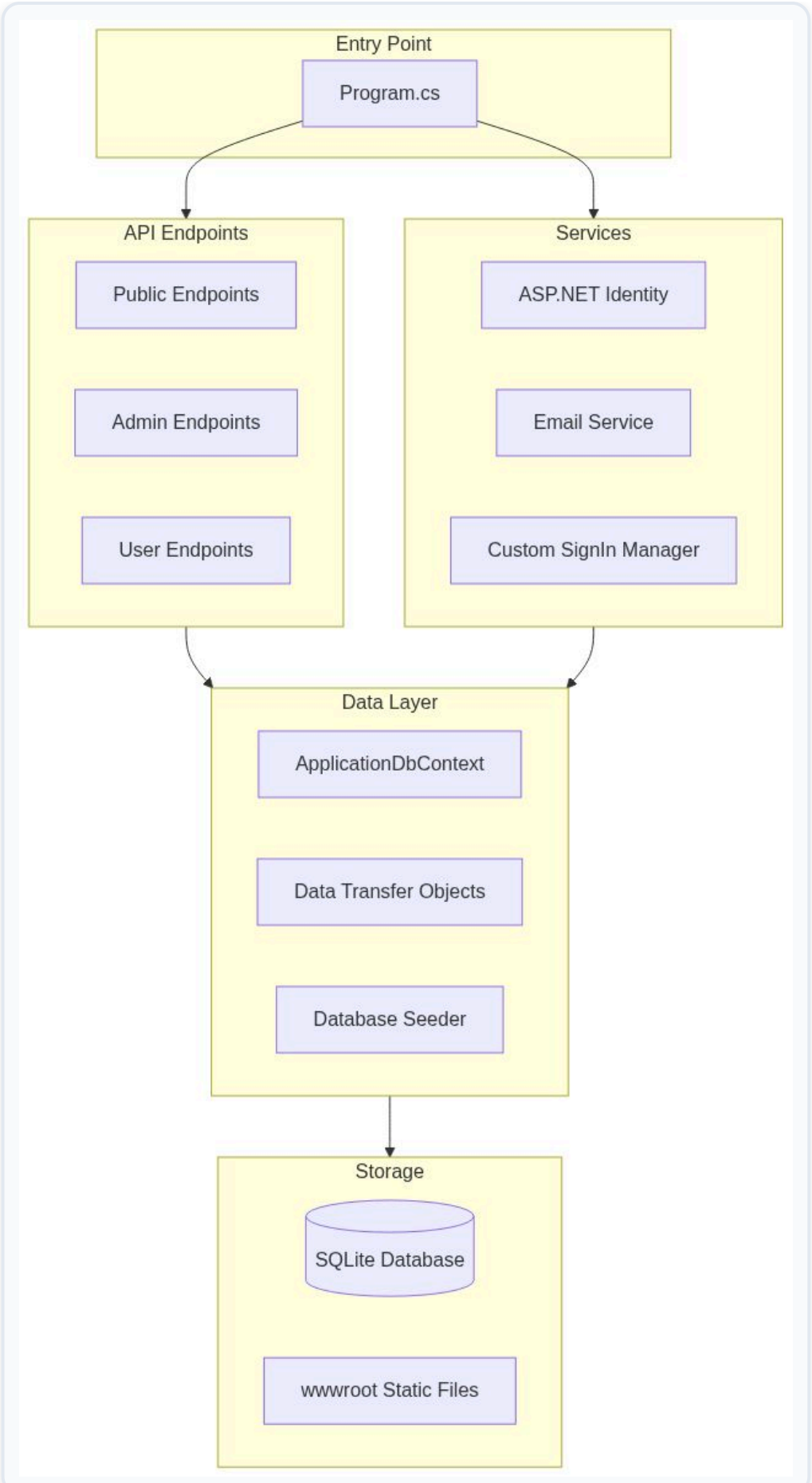
This diagram shows the relationship between major frontend components and how they compose to form the user interface.


App.tsx
Wouter Router
Admin Routes
AdminLayout
ProtectedAdminRoute
Public Routes
Layout Component
Pages
HomePage
DestinationsPage
DestinationDetails
PlannerPage
BookingPage
CheckoutPage
BlogPage
BlogDetailPage
BlogEditorPage
LoginPage



Backend Architecture

The backend is built with ASP.NET Core using a minimal API approach. It follows a layered architecture with clear separation between endpoints, data access, and business logic.

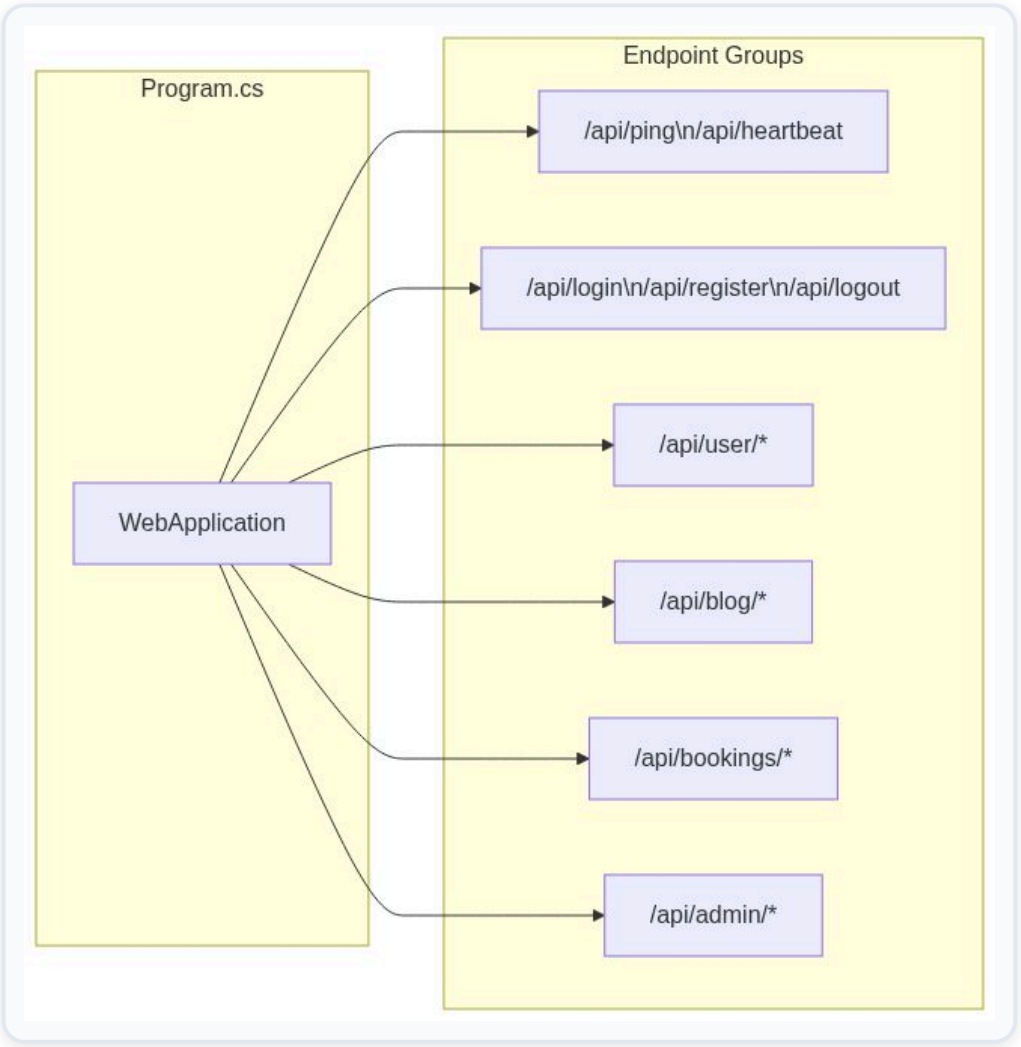

Entry Point
Program.cs
API Endpoints
Public Endpoints
Admin Endpoints
User Endpoints
Services
ASP.NET Identity
Email Service
Custom SignIn Manager
Data Layer
ApplicationDbContext
Data Transfer Objects
Database Seeder
Storage
SQLite Database
wwwroot Static Files

Backend Directory Structure

```
MyTravel.Server/
├── Program.cs                    # Application entry point
│                                 # - Service configuration
│                                 # - Middleware pipeline
│                                 # - Endpoint registration
│
├── Data/                         # Data access layer
│   ├── ApplicationDbContext.cs   # EF Core context
│   │                             # - DbSet definitions
│   │                             # - Entity configurations
│   │                             # - Model definitions
│   └── DbSeeder.cs               # Database seeding
│                                 # - Sample destinations
│                                 # - Demo blog posts
│                                 # - Admin user creation
│
├── DTOs/                         # Data Transfer Objects
│   ├── BlogDtos.cs               # Blog request/response models
│   ├── BookingDtos.cs            # Booking request/response models
│   ├── UserDtos.cs               # User request/response models
│   └── WeatherForecast.cs        # Sample DTO
│
├── Endpoints/                    # API endpoint definitions
│   ├── ActivityEndpoints.cs      # Ping and heartbeat tracking
│   ├── AdminBlogEndpoints.cs     # Admin blog CRUD operations
│   ├── AdminBookingEndpoints.cs  # Admin booking management
│   ├── AdminEndpoints.cs         # Admin auth and user management
│   ├── BlogEndpoints.cs          # Public blog endpoints
│   ├── BookingEndpoints.cs       # Public booking endpoints
│   ├── UserEndpoints.cs          # User profile endpoints
│   └── WeatherEndpoints.cs       # Sample weather endpoint
│
├── Services/                     # Business logic services
│   ├── CustomSignInManager.cs    # Extended sign-in logic
│   └── EmailSender.cs            # Email sending service
│
├── Properties/
│   └── launchSettings.json       # Development launch profiles
│
├── Views/                        # Server-side views (if any)
│   └── PageBlockTypes/
│
├── wwwroot/                      # Static file serving
│
├── appsettings.json              # Base configuration
├── appsettings.Development.json  # Development configuration
├── MyTravel.Server.csproj        # Project file
└── MyTravel.Server.http          # HTTP request testing
```

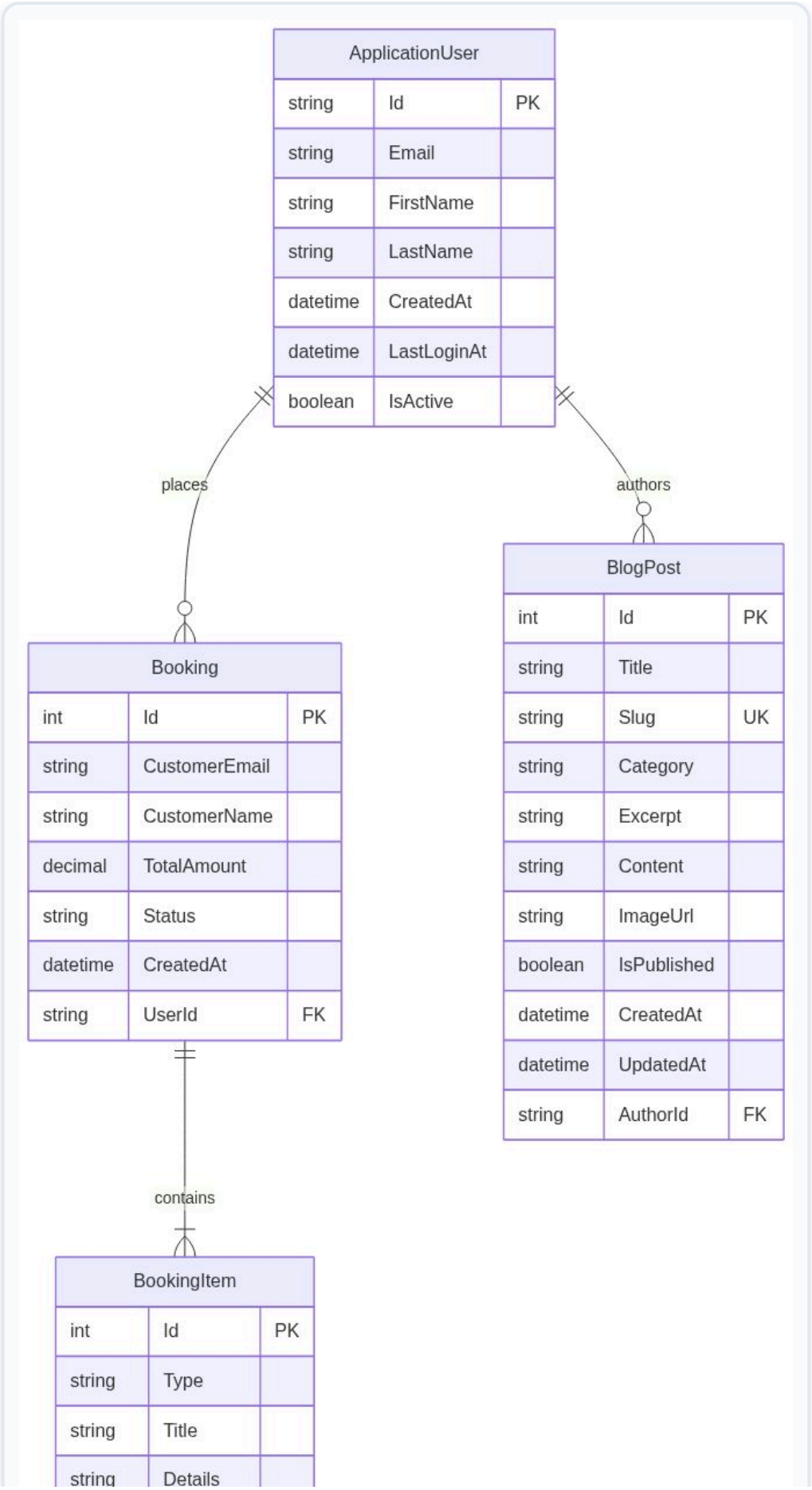
API Endpoint Organization

The API endpoints are organized into logical groups using ASP.NET Core's minimal API pattern with extension methods.



Data Model

The application uses Entity Framework Core with SQLite for data persistence. The data model centers around users, bookings, and blog content.

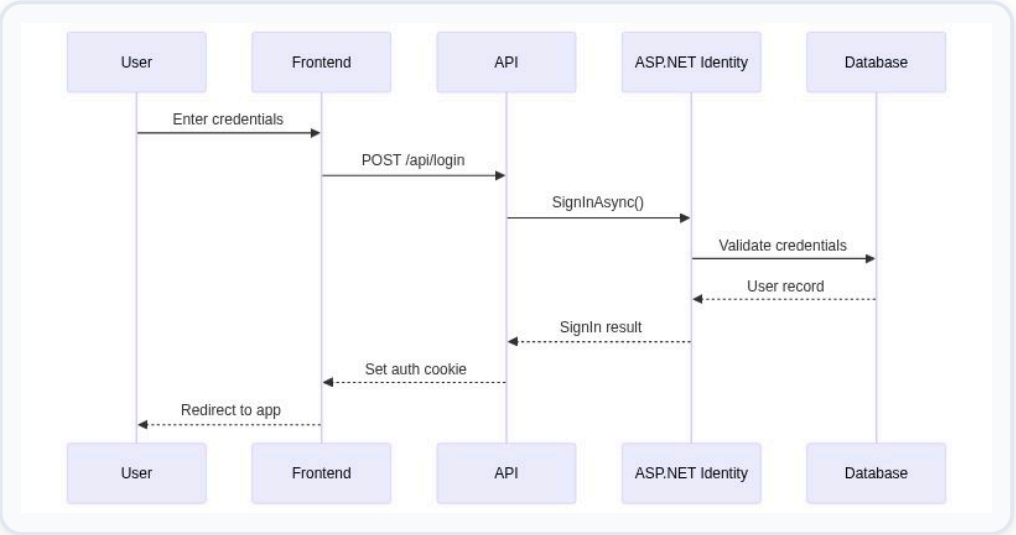

ApplicationUser
string Id PK
string Email
string FirstName
string LastName
datetime CreatedAt
datetime LastLoginAt
boolean IsActive
places
authors
Booking
int Id PK
string CustomerEmail
string CustomerName
decimal TotalAmount
string Status
datetime CreatedAt
string UserId FK
BlogPost
int Id PK
string Title
string Slug UK
string Category
string Excerpt
string Content
string ImageUrl
boolean IsPublished
datetime CreatedAt
datetime UpdatedAt
string AuthorId FK
contains
BookingItem
int Id PK
string Type
string Title
string Details

decimal	Price	
int	BookingId	FK

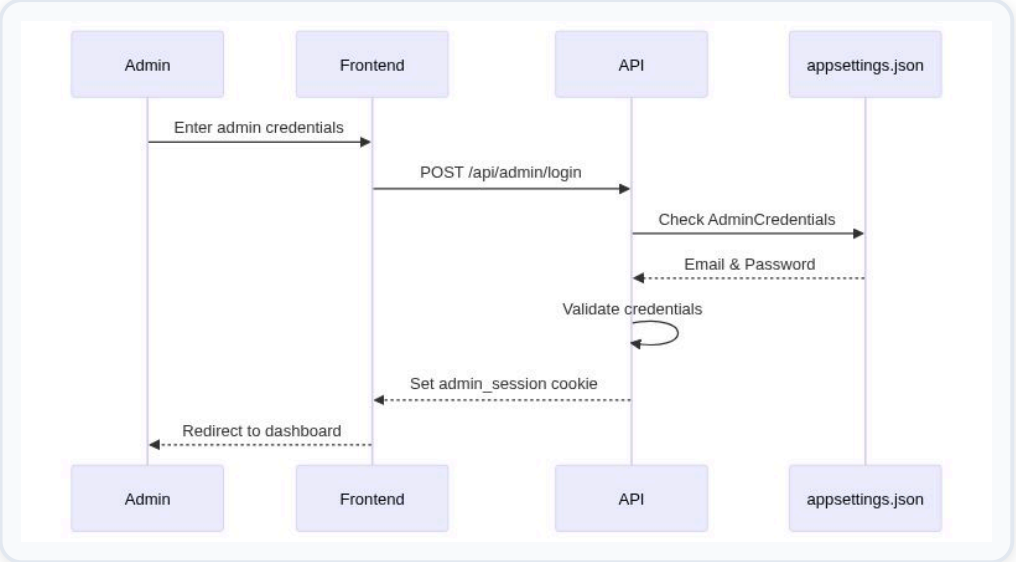
Authentication Flow

The application implements two separate authentication flows: one for regular users and one for administrators.

User Authentication

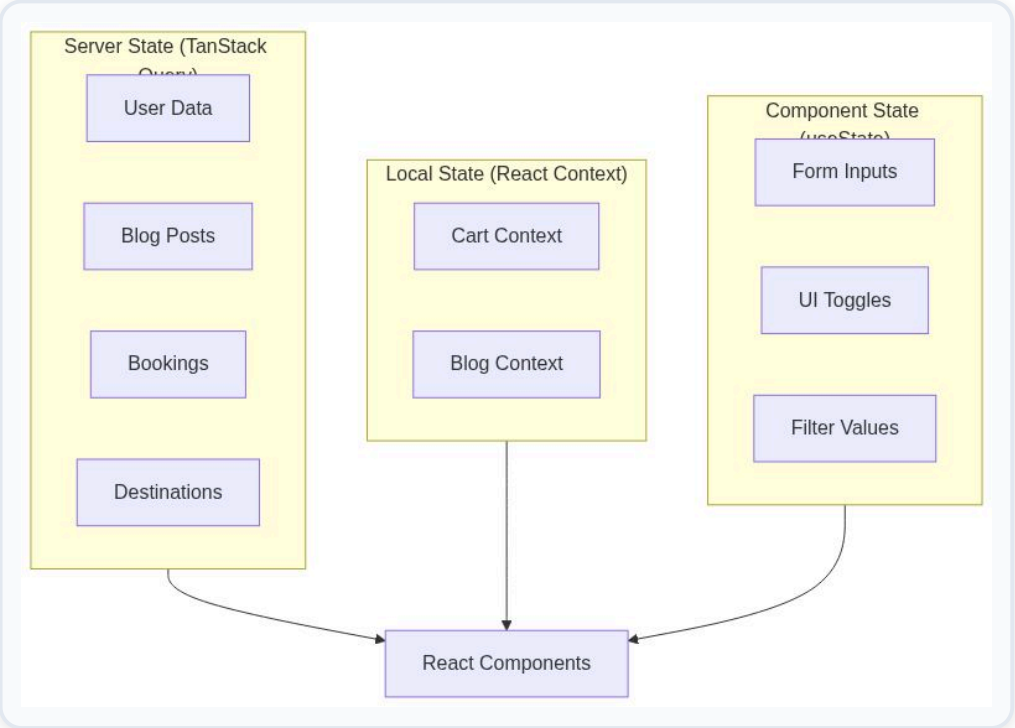


Admin Authentication



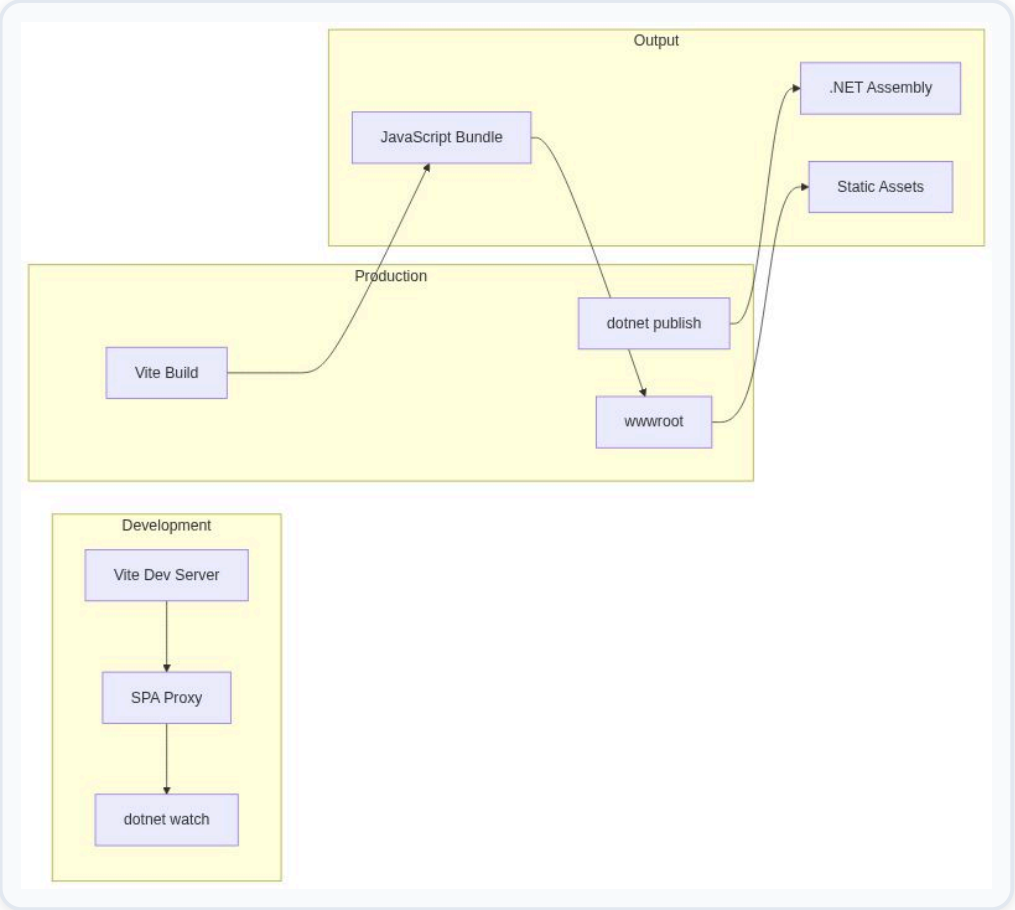
State Management

The frontend uses a combination of React Context and TanStack Query for state management. Local UI state is managed with React hooks, while server state is cached and synchronized using TanStack Query.



Build and Deployment

The application uses different build configurations for development and production environments.



Development Mode

In development, the Vite development server runs on port 49764 and proxies API requests to the ASP.NET Core server on port 5083. The SPA Proxy middleware automatically starts the Vite server when the .NET application launches.

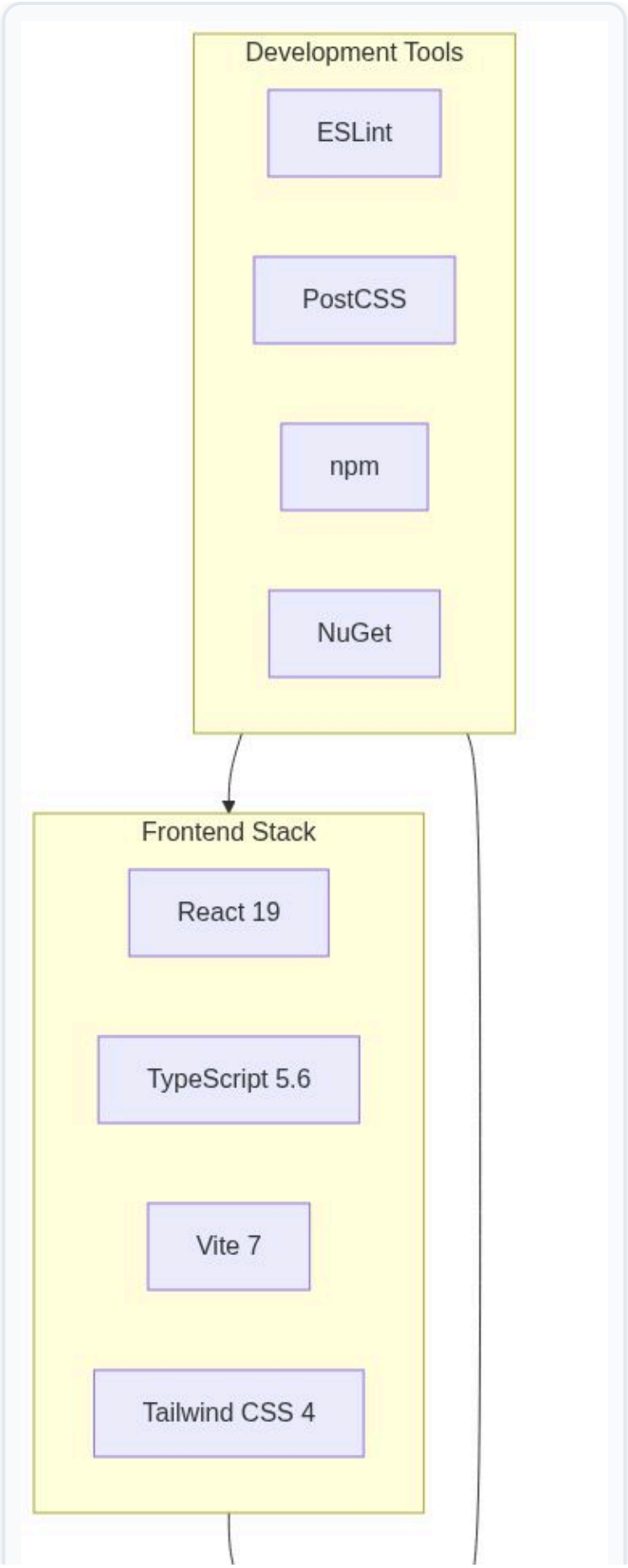
Production Mode

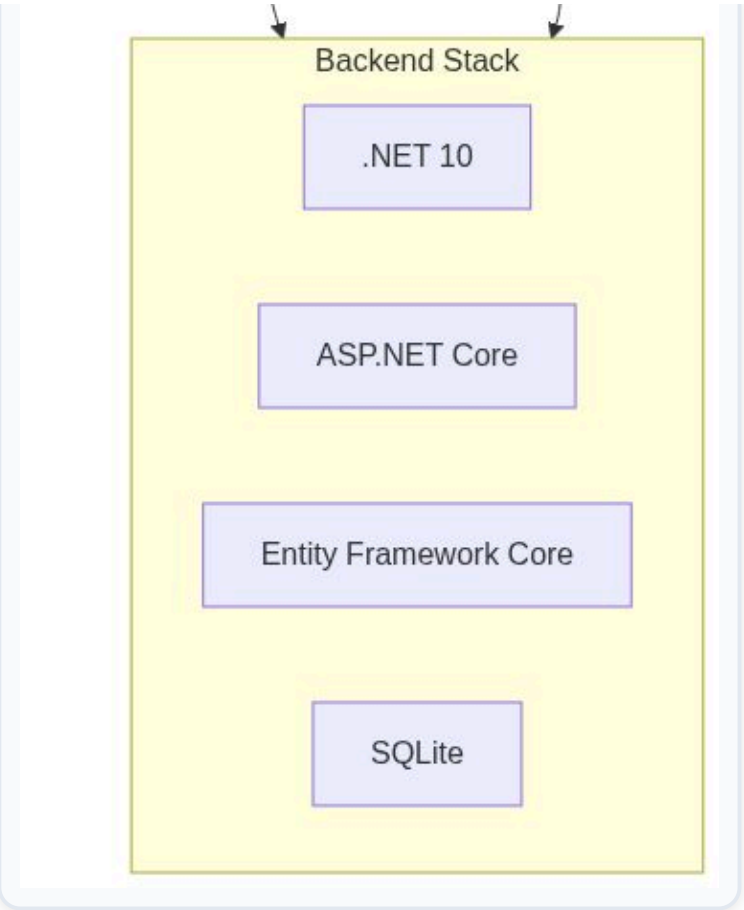
For production, Vite builds optimized static assets that are served from the ASP.NET Core wwwroot directory. The .NET application serves both the API and the static frontend files.

Technologies

This document provides a comprehensive overview of the libraries, frameworks, and tools used in the MyTravel application. The project leverages modern technologies for both frontend and backend development to deliver a robust and maintainable codebase.

Technology Stack Overview


Development Tools
ESLint
PostCSS
npm
NuGet
Frontend Stack
React 19
TypeScript 5.6
Vite 7
Tailwind CSS 4



Frontend Technologies

The frontend is built with React and TypeScript, utilizing a modern toolchain for development and production builds.

Core Framework

Library	Version	Purpose
React	^19.2.0	The core UI library for building component-based user interfaces. React 19 introduces improved performance and new features like automatic batching and concurrent rendering.
React DOM	^19.2.0	Provides DOM-specific methods for rendering React components to the browser.
TypeScript	5.6.3	Adds static typing to JavaScript, enabling better tooling, code navigation, and compile-time error detection.

Build Tools

Library	Version	Purpose
Vite	^7.1.9	A modern build tool that provides extremely fast development server startup and hot module replacement (HMR). Vite uses native ES modules during development for instant updates.
@vitejs/plugin-react	^4.5.2	Official Vite plugin for React that enables Fast Refresh and JSX transformation.
ESLint	^9.28.0	A pluggable linting utility for JavaScript and TypeScript. Identifies and reports on patterns found in code to maintain code quality.
PostCSS	(bundled)	A tool for transforming CSS with JavaScript plugins. Used by Tailwind CSS for processing utility classes.

Styling

Library	Version	Purpose
Tailwind CSS	^4.1.14	A utility-first CSS framework that provides low-level utility classes for building custom designs. Enables rapid UI development without leaving HTML.
tailwindcss-animate	^1.0.7	Adds animation utilities to Tailwind CSS for creating smooth transitions and keyframe animations.
class-variance-authority	^0.7.1	A utility for creating variant-based component styles. Simplifies conditional class application.
clsx	^2.1.1	A utility for constructing className strings conditionally. Works well with Tailwind's utility classes.
tailwind-merge	^3.3.1	Intelligently merges Tailwind CSS classes without style conflicts. Prevents duplicate or conflicting utilities.

Routing

Library	Version	Purpose
wouter	^3.3.5	A minimalist routing library for React. Provides hooks-based routing with a small footprint (~1.5KB). Chosen for its simplicity and performance over React Router.

State Management and Data Fetching

Library	Version	Purpose
@tanstack/react-query	^5.60.5	Powerful data synchronization library for fetching, caching, and updating server state. Handles loading states, error handling, and cache invalidation automatically.
React Context	(built-in)	Used for local application state like shopping cart and blog editor state. Provides component-level state sharing without external dependencies.

UI Components

The application uses Radix UI primitives as the foundation for accessible, unstyled UI components. These are styled with Tailwind CSS to match the application design.

Library	Version	Purpose
@radix-ui/react-accordion	^1.2.11	Collapsible content sections with accessibility support.
@radix-ui/react-alert-dialog	^1.1.14	Modal dialogs for important confirmations and alerts.
@radix-ui/react-aspect-ratio	^1.1.7	Maintains consistent aspect ratios for images and media.
@radix-ui/react-avatar	^1.1.10	User avatar display with fallback support.
@radix-ui/react-checkbox	^1.3.2	Accessible checkbox inputs with custom styling.
@radix-ui/react-collapsible	^1.1.11	Expandable/collapsible content areas.
@radix-ui/react-dialog	^1.1.14	Modal dialog windows for forms and content.
@radix-ui/react-dropdown-menu	^2.1.15	Dropdown menus with keyboard navigation.
@radix-ui/react-hover-card	^1.1.14	Content cards that appear on hover.
@radix-ui/react-label	^2.1.7	Accessible form labels.
@radix-ui/react-menubar	^1.1.14	Application menu bars with submenus.
@radix-ui/react-navigation-menu	^1.2.13	Site navigation with dropdowns.
@radix-ui/react-popover	^1.1.14	Floating content panels anchored to triggers.
@radix-ui/react-progress	^1.1.7	Progress indicators and loading bars.

Library	Version	Purpose
@radix-ui/react-radio-group	^1.3.7	Radio button groups with accessibility.
@radix-ui/react-scroll-area	^1.2.9	Custom scrollable areas with styled scrollbars.
@radix-ui/react-select	^2.2.5	Custom select dropdowns with search support.
@radix-ui/react-separator	^1.1.7	Visual dividers between content sections.
@radix-ui/react-slider	^1.3.5	Range sliders for numeric input.
@radix-ui/react-slot	^1.2.3	Utility for component composition.
@radix-ui/react-switch	^1.2.5	Toggle switches for boolean settings.
@radix-ui/react-tabs	^1.1.12	Tabbed interfaces for content organization.
@radix-ui/react-toast	^1.2.14	Toast notification system.
@radix-ui/react-toggle	^1.1.9	Toggle buttons for binary states.
@radix-ui/react-toggle-group	^1.1.10	Groups of mutually exclusive toggles.
@radix-ui/react-tooltip	^1.2.7	Informational tooltips on hover.

Forms and Validation

Library	Version	Purpose
react-hook-form	^7.66.0	Performant form library with minimal re-renders. Provides form state management, validation, and submission handling.
@hookform/resolvers	^5.0.1	Validation resolvers for react-hook-form. Connects external validation libraries like Zod.
zod	^3.25.76	TypeScript-first schema validation library. Defines validation rules with full type inference.

Rich Text Editor

Library	Version	Purpose
lexical	^0.38.2	An extensible text editor framework by Meta. Provides the foundation for the blog post editor with support for rich formatting.
@lexical/react	^0.38.2	React bindings for Lexical editor.
@lexical/rich-text	^0.38.2	Rich text editing capabilities including bold, italic, headings.
@lexical/list	^0.38.2	Ordered and unordered list support.
@lexical/link	^0.38.2	Hyperlink editing and display.
@lexical/code	^0.38.2	Code block formatting.
@lexical/table	^0.38.2	Table editing support.
@lexical/selection	^0.38.2	Text selection utilities.
@lexical/utils	^0.38.2	Common utility functions.

Maps and Geolocation

Library	Version	Purpose
leaflet	^1.9.4	Open-source JavaScript library for interactive maps. Provides map rendering, markers, and user interaction.
react-leaflet	^5.0.0	React components for Leaflet maps. Enables declarative map configuration in JSX.
@types/leaflet	^1.9.18	TypeScript type definitions for Leaflet.

Drag and Drop

Library	Version	Purpose
@dnd-kit/core	^6.3.1	Core drag and drop functionality. Provides the foundation for the itinerary planner's drag-and-drop interface.
@dnd-kit/sortable	^10.0.0	Sortable list utilities built on dnd-kit. Enables reordering of itinerary items.
@dnd-kit/utilities	^3.2.2	Helper utilities for dnd-kit implementations.

Animations

Library	Version	Purpose
framer-motion	^12.23.24	Production-ready animation library for React. Powers page transitions, micro-interactions, and complex animations throughout the application.

Charts and Data Visualization

Library	Version	Purpose
recharts	^2.15.4	Composable charting library built on React and D3. Used in the admin dashboard for displaying statistics and trends.

Date and Time

Library	Version	Purpose
date-fns	^3.6.0	Modern JavaScript date utility library. Provides functions for parsing, formatting, and manipulating dates.
react-day-picker	^8.10.1	Flexible date picker component. Used for selecting travel dates in booking forms.

Icons

Library	Version	Purpose
lucide-react	^0.545.0	Beautiful, consistent icon set with React components. Provides 1000+ icons used throughout the interface.

Notifications

Library	Version	Purpose
sonner	^2.0.7	Toast notification library with a clean API. Displays success, error, and informational messages to users.

Analytics

Library	Version	Purpose
posthog-js	^1.298.1	Product analytics platform integration. Tracks user behavior, feature usage, and conversion metrics.

Utilities

Library	Version	Purpose
cmdk	^1.1.1	Command menu component for keyboard-driven interfaces.
embla-carousel-react	^8.6.0	Carousel/slider component with touch support.
input-otp	^1.4.2	One-time password input component.
next-themes	^0.4.6	Theme switching utility (light/dark mode).
react-resizable-panels	^3.0.2	Resizable panel layouts for complex interfaces.
vaul	^1.1.2	Drawer component for mobile-friendly slide-out panels.

Backend Technologies

The backend is built on ASP.NET Core with Entity Framework Core for data access.

Core Framework

Package	Version	Purpose
.NET	10.0	The runtime and SDK for building the server application. .NET 10 provides the latest performance improvements and language features.
ASP.NET Core	10.0	Web framework for building APIs and web applications. Provides routing, middleware, dependency injection, and HTTP handling.

Authentication and Authorization

Package	Version	Purpose
Microsoft.AspNetCore.Identity.EntityFrameworkCore	10.0.0	Identity system for user authentication. Provides user management, password hashing, role-based authorization, and session management.

Data Access

Package	Version	Purpose
Microsoft.EntityFrameworkCore.Sqlite	10.0.0	SQLite database provider for Entity Framework Core. Enables file-based database storage without requiring a separate database server.
Microsoft.EntityFrameworkCore.Tools	10.0.0	Command-line tools for EF Core migrations and database management.

API Documentation

Package	Version	Purpose
Scalar.AspNetCore	2.11.0	Modern, beautiful API documentation UI for OpenAPI specifications. Provides an interactive interface for exploring and testing API endpoints.
Microsoft.AspNetCore.OpenApi	10.0.0	OpenAPI/Swagger support for API documentation. Automatically generates API documentation from endpoint definitions. Works with Scalar for interactive documentation.

Development Tools

Package	Version	Purpose
Microsoft.AspNetCore.SpaProxy	10.*	Development proxy for Single Page Applications. Automatically starts the Vite development server when running the .NET application.

Development Environment

Recommended Tools

Tool	Purpose
Visual Studio 2022	Full-featured IDE for .NET development with integrated debugging, testing, and deployment tools.
Visual Studio Code	Lightweight editor with excellent TypeScript and React support through extensions.
Node.js 18+	JavaScript runtime required for frontend build tools and development server.
.NET 10 SDK	Required for building and running the backend application.

Browser Support

The application targets modern browsers with ES2020+ support:

- Chrome 90+
- Firefox 90+
- Safari 14+
- Edge 90+

Development Dependencies

Library	Version	Purpose
@types/react	^19.1.8	TypeScript definitions for React.
@types/react-dom	^19.1.6	TypeScript definitions for React DOM.
@eslint/js	^9.28.0	ESLint core JavaScript rules.
eslint-plugin-react-hooks	^5.2.0	ESLint rules for React Hooks best practices.
eslint-plugin-react-refresh	^0.4.20	ESLint plugin for React Fast Refresh compatibility.
globals	^16.2.0	Global variable definitions for ESLint.
typescript-eslint	^8.33.1	TypeScript support for ESLint.

Error Handling

This document explains the error handling strategies implemented throughout the MyTravel application. Comprehensive error handling ensures users receive meaningful feedback when issues occur, maintaining a smooth experience even during failures.

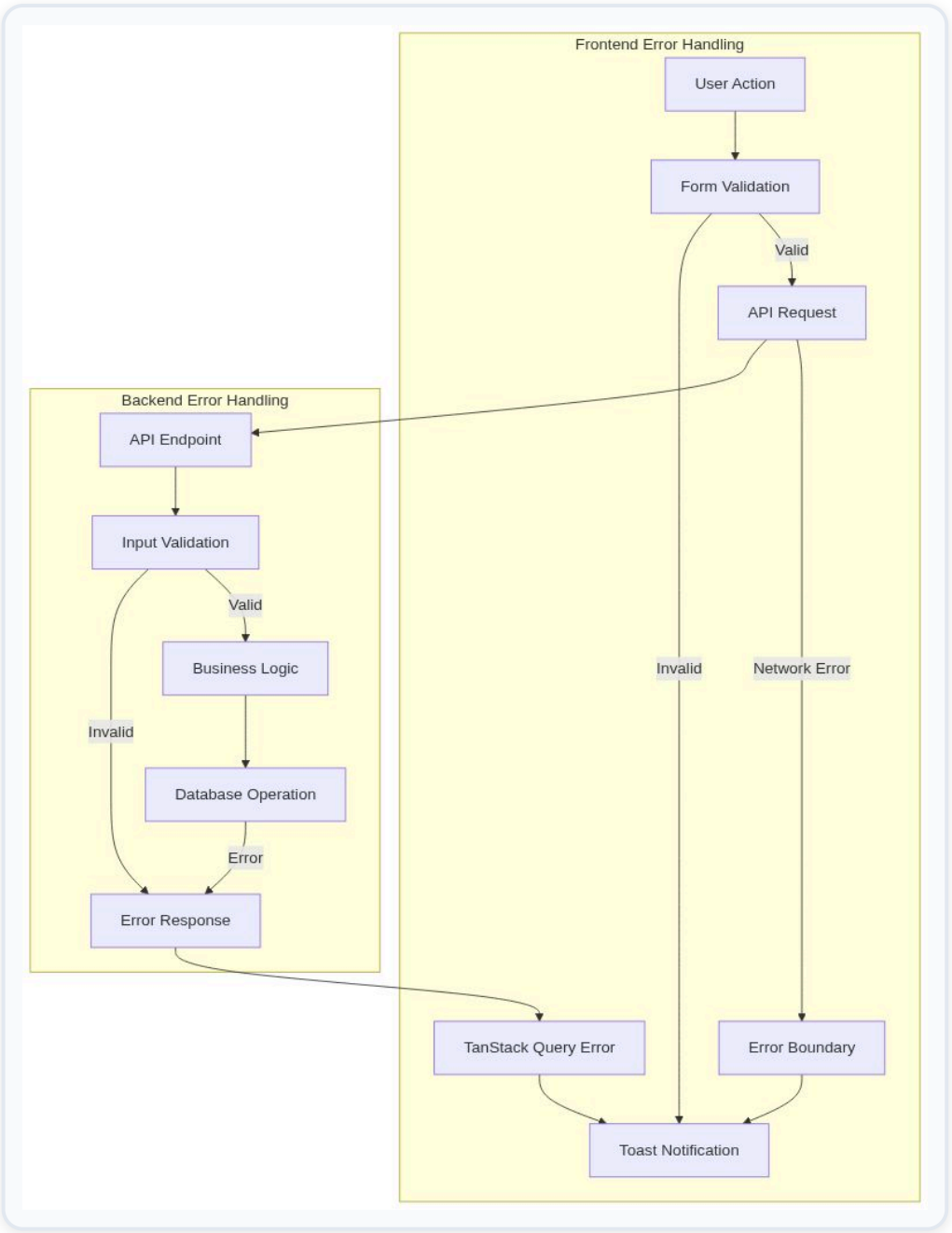
Error Handling Philosophy

"Ensure smooth user experiences by providing meaningful error messages."

The application follows these core principles for error handling:

1. **User-Friendly Messages** - Technical errors are translated into understandable language
2. **Graceful Degradation** - The application remains functional even when individual features fail
3. **Appropriate Feedback** - Users receive timely notifications about the success or failure of their actions
4. **Logging for Debugging** - Errors are captured for developer troubleshooting without exposing sensitive details to users

Error Flow Architecture



Frontend Error Handling

API Request Helper

The application uses a centralized API request function that standardizes error handling for all HTTP requests. This is implemented in `lib/queryClient.ts`.

```
async function throwIfResNotOk(res: Response) {
  if (!res.ok) {
    const text = (await res.text()) || res.statusText;
    throw new Error(`${res.status}: ${text}`);
  }
}

export async function apiRequest(
  method: string,
  url: string,
  data?: unknown | undefined,
): Promise<Response> {
  const res = await fetch(url, {
    method,
    headers: data ? { "Content-Type": "application/json" } : {},
    body: data ? JSON.stringify(data) : undefined,
    credentials: "include",
  });

  await throwIfResNotOk(res);
  return res;
}
```

This helper function:

- Checks if the response status indicates an error (`!res.ok`)
- Extracts the error message from the response body or uses the status text
- Throws an `Error` with both the status code and message for handling upstream

TanStack Query Error Handling

TanStack Query is configured with custom error handling behavior in `queryClient.ts`:

```
type UnauthorizedBehavior = "returnNull" | "throw";

export const getQueryFn: <T>(options: {
  on401: UnauthorizedBehavior;
}) => QueryFunction<T> =
  ({ on401: unauthorizedBehavior }) =>
  async ({ queryKey }) => {
    const res = await fetch(queryKey.join("/") as string, {
      credentials: "include",
    });

    if (unauthorizedBehavior === "returnNull" && res.status === 401) {
      return null;
    }

    await throwIfResNotOk(res);
    return await res.json();
  };

export const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      queryFn: getQueryFn({ on401: "throw" }),
      refetchInterval: false,
      refetchOnWindowFocus: false,
      staleTime: Infinity,
      retry: false,
    },
    mutations: {
      retry: false,
    },
  },
});
```

Key configuration choices:

- **401 Handling** - Configurable behavior for unauthorized responses (return null or throw)
- **No Automatic Retry** - Failed requests are not automatically retried to prevent redundant server load
- **No Refetch on Focus** - Prevents unexpected data refreshes when switching browser tabs

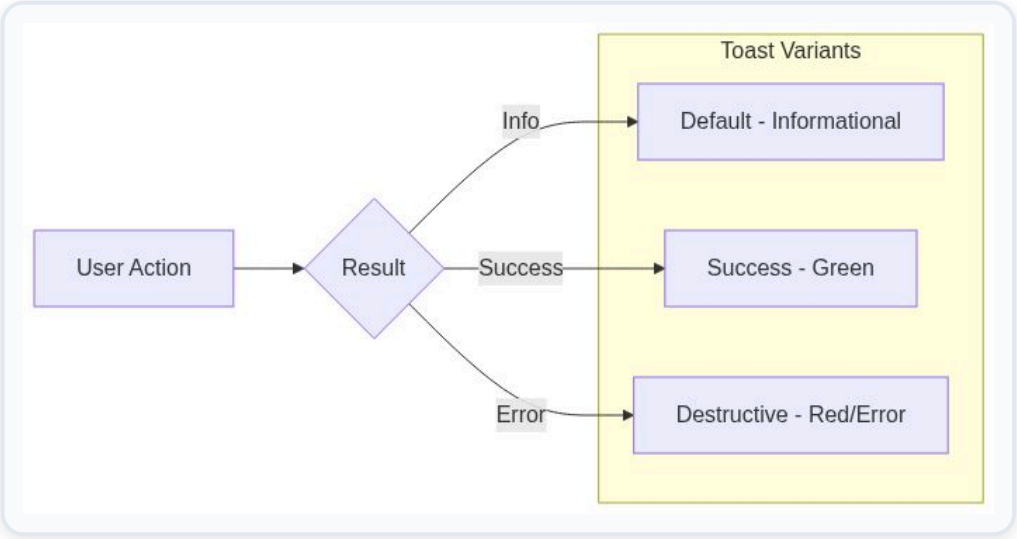
Mutation Error Handling Pattern

When performing mutations (create, update, delete operations), errors are handled in the `onError` callback:

```
const mutation = useMutation({
  mutationFn: async (data: FormData) => {
    const response = await apiRequest("POST", "/api/bookings", data);
    return response.json();
  },
  onSuccess: (data) => {
    toast({
      title: "Success",
      description: "Your booking has been confirmed!",
    });
    // Navigate or update UI
  },
  onError: (error: Error) => {
    toast({
      title: "Booking Failed",
      description: error.message || "Unable to complete your booking. Please try again.
      variant: "destructive",
    });
  },
});
```

Toast Notification System

The application uses a toast notification system (Sonner/Radix Toast) for user feedback. Toasts support multiple variants for different message types.



Toast Usage Examples:

```
// Success notification
toast({
  title: "Profile Updated",
  description: "Your changes have been saved successfully.",
});

// Error notification
toast({
  title: "Error",
  description: "Failed to save changes. Please try again.",
  variant: "destructive",
});

// Informational notification
toast({
  title: "Session Expiring",
  description: "Your session will expire in 5 minutes.",
});
```

Form Validation with Zod

Forms use react-hook-form with Zod schemas for client-side validation. This catches errors before they reach the server.

```
import { useForm } from "react-hook-form";
import { zodResolver } from "@hookform/resolvers/zod";
import { z } from "zod";

const loginSchema = z.object({
  email: z.string().email("Please enter a valid email address"),
  password: z.string().min(6, "Password must be at least 6 characters"),
});

type LoginForm = z.infer<typeof loginSchema>;

function LoginPage() {
  const form = useForm<LoginForm>({
    resolver: zodResolver(loginSchema),
    defaultValues: {
      email: "",
      password: "",
    },
  });

  const onSubmit = async (data: LoginForm) => {
    try {
      await apiRequest("POST", "/api/login", data);
      // Handle success
    } catch (error) {
      form.setError("root", {
        message: "Invalid email or password",
      });
    }
  };

  return (
    <Form {...form}>
      <FormField
        control={form.control}
        name="email"
        render={({ field }) => (
          <FormItem>
            <FormLabel>Email</FormLabel>
            <FormControl>
              <Input {...field} />
            </FormControl>
            <FormMessage /> {/* Displays validation errors */}
          </FormItem>
        )}
      />
      {/* ... */}
    </Form>
  );
}
```

Validation Error Display:

State	User Experience
Empty required field	“This field is required” appears below input
Invalid email format	“Please enter a valid email address”
Password too short	“Password must be at least 6 characters”
Server validation error	Error message displayed at form level

404 Not Found Page

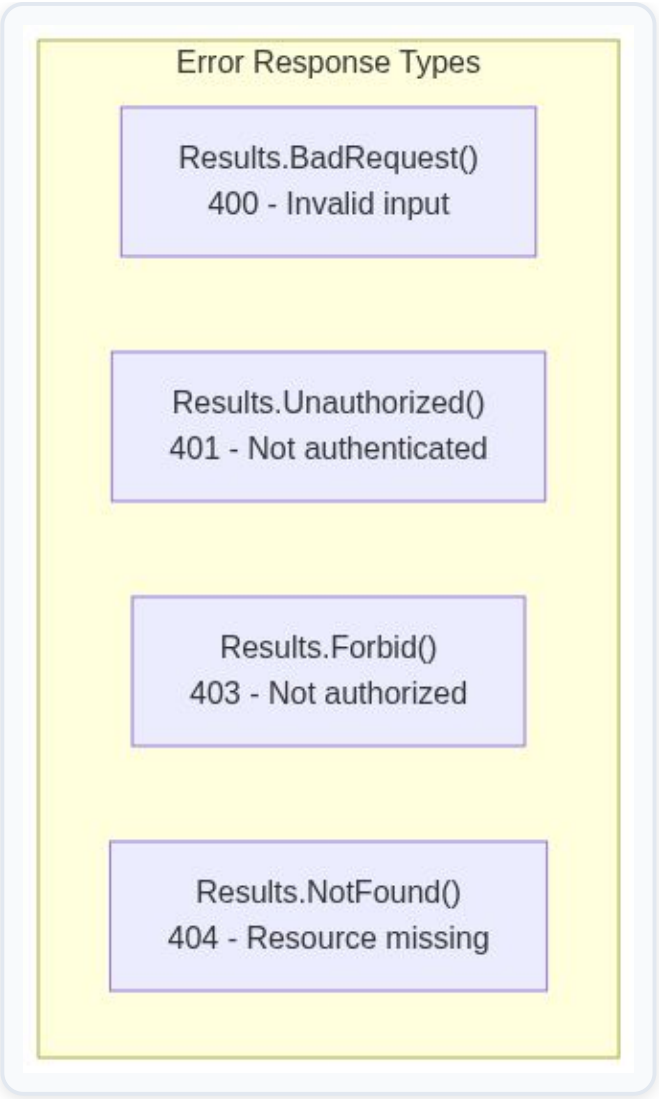
The application includes a custom 404 page for handling invalid routes:

```
// pages/not-found.tsx
export default function NotFound() {
  return (
    <div className="flex flex-col items-center justify-center min-h-screen">
      <h1 className="text-6xl font-bold text-gray-900">404</h1>
      <p className="text-xl text-gray-600 mt-4">Page not found</p>
      <p className="text-gray-500 mt-2">
        The page you're looking for doesn't exist or has been moved.
      </p>
      <Link href="/" className="mt-6">
        <Button>Return to Homepage</Button>
      </Link>
    </div>
  );
}
```

Backend Error Handling

Standardized Error Responses

The backend uses ASP.NET Core’s `Results` class to return consistent error responses:



Error Response Examples:

```
// 400 Bad Request - Validation errors
if (string.IsNullOrWhiteSpace(request.Title))
{
    return Results.BadRequest(new { message = "Title is required" });
}

if (request.Items == null || request.Items.Count == 0)
{
    return Results.BadRequest(new { message = "At least one booking item is required" }
}

// 401 Unauthorized - Authentication required
var adminCookie = httpContext.Request.Cookies["admin_session"];
if (adminCookie != "authenticated")
{
    return Results.Unauthorized();
}

// 403 Forbidden - Access denied
if (booking.UserId != null && booking.UserId != userId)
{
    return Results.Forbid();
}

// 404 Not Found - Resource doesn't exist
var post = await db.BlogPosts.FindAsync(id);
if (post == null)
{
    return Results.NotFound(new { message = "Blog post not found" });
}
```

Input Validation

All API endpoints validate input before processing:

```
// Booking endpoint validation
app.MapPost("/api/bookings", async (CreateBookingRequest request, ...) =>
{
    // Validate required items
    if (request.Items == null || request.Items.Count == 0)
    {
        return Results.BadRequest(new { message = "At least one booking item is required
    }

    // Validate guest checkout requirements
    if (user.Identity?.IsAuthenticated != true)
    {
        if (string.IsNullOrWhiteSpace(request.CustomerEmail) ||
            string.IsNullOrWhiteSpace(request.CustomerName))
        {
            return Results.BadRequest(new {
                message = "Customer email and name are required for guest checkout"
            });
        }
    }

    // Validate enum values
    if (!Enum.TryParse<BookingStatus>(request.Status, true, out var statusEnum))
    {
        return Results.BadRequest(new { message = "Invalid status value" });
    }

    // ... proceed with valid request
});
```

Non-Critical Operation Error Handling

For operations that shouldn't block the main request (like sending emails), errors are caught and logged but don't fail the overall operation:

```
// Create booking
db.Bookings.Add(booking);
await db.SaveChangesAsync();

// Try to send confirmation email (non-critical)
try
{
    await emailSender.SendConfirmationLinkAsync(
        new ApplicationUser { Email = booking.CustomerEmail },
        booking.CustomerEmail,
        emailBody);
}
catch
{
    // Email sending failed, but booking was created successfully
    // Log the error for investigation but don't fail the request
}

// Return success - booking was created
return Results.Created($"/api/bookings/{booking.Id}", new
{
    id = booking.Id,
    message = "Booking created successfully"
});
```

Authentication Error Handling

The admin authentication system validates credentials against configuration:

```
app.MapPost("/api/admin/login", async (
    AdminLoginRequest request,
    IConfiguration config,
    HttpContext httpContext) =>
{
    var adminEmail = config["AdminCredentials:Email"];
    var adminPassword = config["AdminCredentials:Password"];

    if (request.Email == adminEmail && request.Password == adminPassword)
    {
        // Set secure cookie and return success
        httpContext.Response.Cookies.Append("admin_session", "authenticated",
            new CookieOptions
            {
                HttpOnly = true,
                Secure = true,
                SameSite = SameSiteMode.Strict,
                Expires = DateTimeOffset.UtcNow.AddHours(8)
            });

        return Results.Ok(new { message = "Admin login successful", isAdmin = true });
    }

    // Invalid credentials - return 401 without revealing which field was wrong
    return Results.Unauthorized();
});
```

Database Error Handling

Entity Framework operations are wrapped in appropriate error handling:

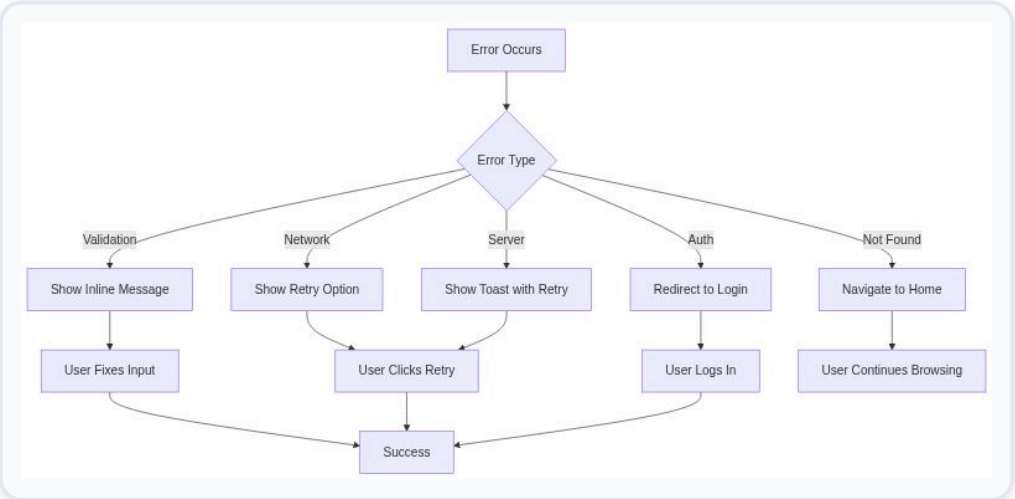
```
// Delete user with Identity Manager (handles cascading constraints)
var result = await userManager.DeleteAsync(user);
if (!result.Succeeded)
{
    return Results.BadRequest(new {
        message = "Failed to delete user",
        errors = result.Errors
    });
}

return Results.Ok(new { message = "User deleted successfully" });
```

Error Scenarios and User Experience

Common Error Scenarios

Scenario	Frontend Handling	Backend Response	User Message
Invalid form input	Inline validation message	N/A (client-side)	"Please enter a valid email address"
Wrong password	Toast notification	401 Unauthorized	"Invalid email or password"
Session expired	Redirect to login	401 Unauthorized	"Your session has expired. Please log in again."
Resource not found	404 page or toast	404 Not Found	"The requested item could not be found"
Network error	Toast notification	N/A	"Unable to connect. Please check your internet connection."
Server error	Toast notification	500 Internal Error	"Something went wrong. Please try again later."
Payment failure	Toast + form state	400 Bad Request	"Payment could not be processed. Please verify your payment details."
Booking unavailable	Toast notification	400 Bad Request	"This booking is no longer available"

Error Recovery Patterns



Best Practices Summary

Frontend

1. **Always handle errors in mutations** - Use `onError` callbacks to display user feedback
2. **Validate before submitting** - Use Zod schemas with react-hook-form
3. **Provide specific messages** - Tell users what went wrong and how to fix it
4. **Use appropriate toast variants** - `destructive` for errors, default for success
5. **Handle loading states** - Show spinners/skeletons during async operations

Backend

1. **Return appropriate status codes** - 400 for validation, 401 for auth, 404 for missing resources
2. **Include helpful messages** - Explain what went wrong in the response body
3. **Don't expose sensitive information** - Avoid revealing system internals in error messages
4. **Handle non-critical failures gracefully** - Don't fail the whole request if email sending fails
5. **Validate all input** - Never trust client-side validation alone

Appendix A: API Reference

This appendix provides comprehensive documentation for all API endpoints available in the MyTravel application. The API follows RESTful conventions and returns JSON responses.

Base URL

```
Development: http://localhost:5083/api
Production: https://your-domain.com/api
```

API Documentation UI

The API includes interactive documentation using **Scalar**, a modern OpenAPI documentation interface. In development mode, you can access:

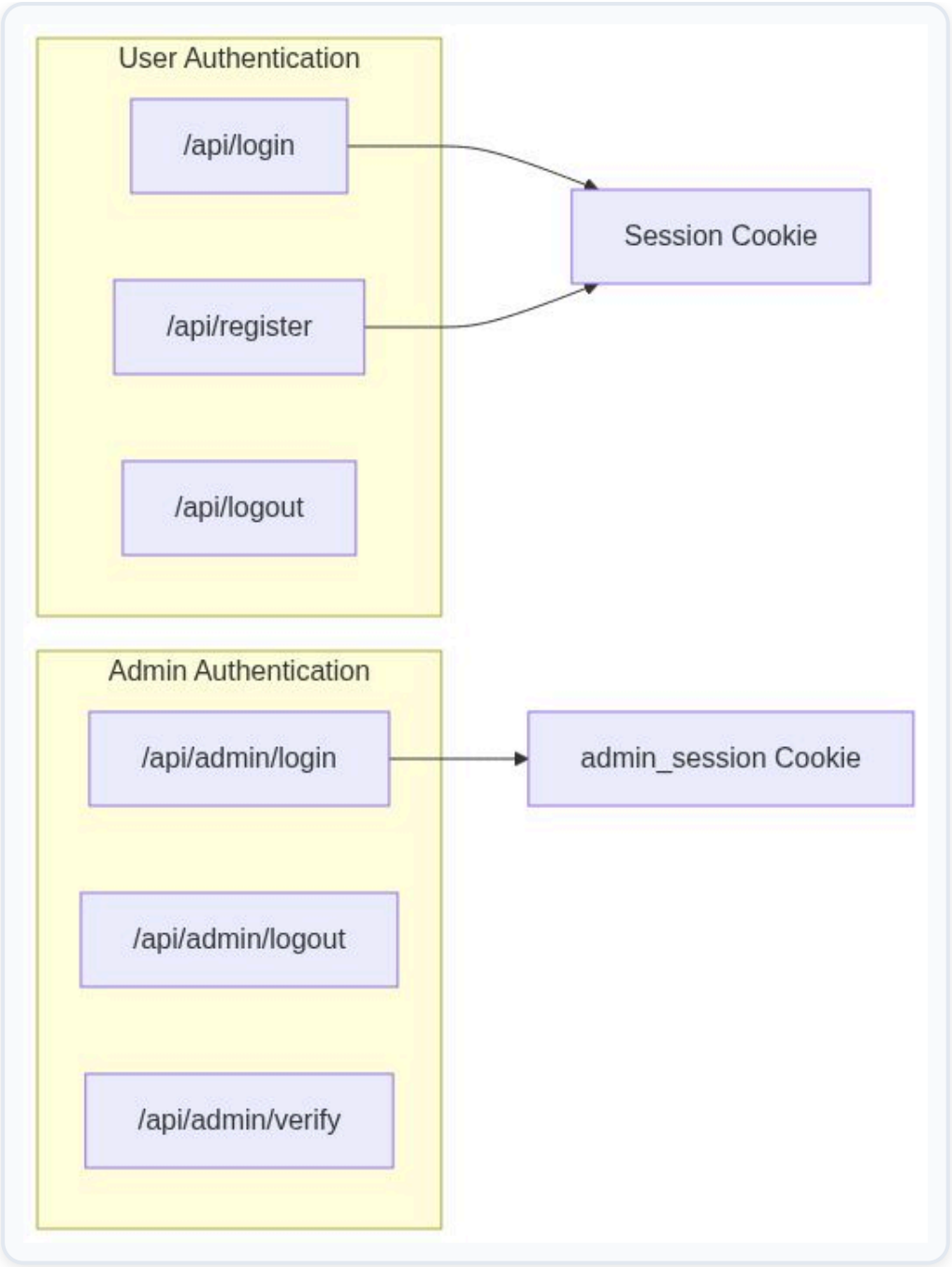
URL	Description
`/scalar/v1`	Interactive Scalar API documentation UI
`/openapi/v1.json`	Raw OpenAPI/Swagger specification

Scalar provides a beautiful, modern interface for exploring and testing API endpoints directly in your browser.

Authentication

The API uses cookie-based authentication. There are two authentication mechanisms:

1. **User Authentication** - Managed by ASP.NET Core Identity with session cookies
2. **Admin Authentication** - Uses a separate `admin_session` cookie



Response Format

All API responses follow a consistent format:

Success Response

```
{
  "data": { ... },
  "message": "Operation successful"
}
```

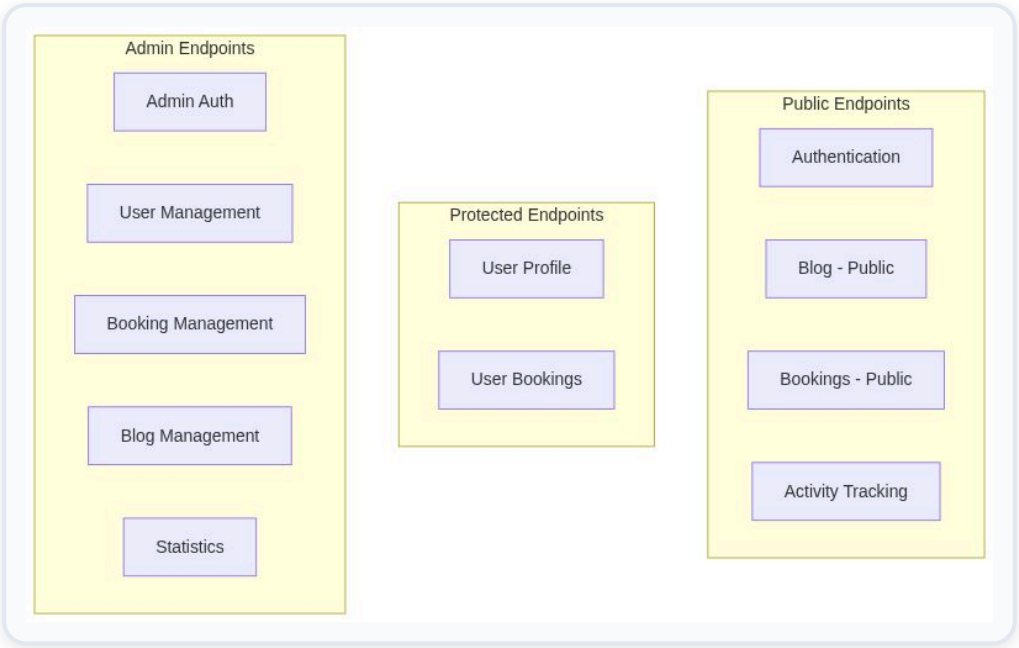
Error Response

```
{
  "message": "Error description"
}
```

Paginated Response

```
{
  "items": [ ... ],
  "totalCount": 100,
  "page": 1,
  "pageSize": 10,
  "totalPages": 10
}
```

API Endpoint Groups



Authentication Endpoints

Register User

Creates a new user account.

```
POST /api/register
```

Request Body:

Field	Type	Required	Description
`email`	string	Yes	User's email address
`password`	string	Yes	Password (min 6 characters, requires uppercase, lowercase, digit, special char)
`firstName`	string	No	User's first name
`lastName`	string	No	User's last name

Example Request:

```
{
  "email": "user@example.com",
  "password": "SecurePass123!",
  "firstName": "John",
  "lastName": "Doe"
}
```

Success Response (200):

```
{
  "message": "Registration successful"
}
```

Error Response (400):

```
{
  "message": "Email already registered"
}
```

Login User

Authenticates a user and creates a session.

```
POST /api/login
```

Request Body:

Field	Type	Required	Description
`email`	string	Yes	User's email address
`password`	string	Yes	User's password

Example Request:

```
{
  "email": "user@example.com",
  "password": "SecurePass123!"
}
```

Success Response (200):

Sets authentication cookie and returns:

```
{
  "message": "Login successful"
}
```

Error Response (401):

```
{
  "message": "Invalid credentials"
}
```

Logout User

Ends the current user session.

```
POST /api/logout
```

Authentication: Required

Success Response (200):

```
{}
```

User Endpoints

Get Current User

Returns basic information about the authenticated user.

```
GET /api/user
```

Authentication: Required

Success Response (200):

```
{
  "name": "user@example.com",
  "isAuthenticated": true
}
```

Error Response (401):

```
{
  "message": "Unauthorized"
}
```

Get User Profile

Returns detailed profile information for the authenticated user.

```
GET /api/user/profile
```

Authentication: Required

Success Response (200):

```
{
  "id": "user-guid",
  "email": "user@example.com",
  "firstName": "John",
  "lastName": "Doe",
  "fullName": "John Doe"
}
```

Update User Profile

Updates the authenticated user's profile information.

```
PUT /api/user/profile
```

Authentication: Required

Request Body:

Field	Type	Required	Description
`firstName`	string	No	Updated first name
`lastName`	string	No	Updated last name

Example Request:

```
{
  "firstName": "Jonathan",
  "lastName": "Smith"
}
```

Success Response (200):

```
{
  "id": "user-guid",
  "email": "user@example.com",
  "firstName": "Jonathan",
  "lastName": "Smith",
  "fullName": "Jonathan Smith"
}
```

Blog Endpoints (Public)

Get All Published Blog Posts

Returns a paginated list of published blog posts.

```
GET /api/blog
```

Query Parameters:

Parameter	Type	Default	Description
`page`	integer	1	Page number
`pageSize`	integer	10	Items per page
`category`	string	null	Filter by category

Example Request:

```
GET /api/blog?page=1&pageSize=10&category=solo-travel
```

Success Response (200):

```
{
  "posts": [
    {
      "id": 1,
      "title": "10 Tips for Solo Travelers",
      "slug": "10-tips-for-solo-travelers",
      "category": "solo-travel",
      "excerpt": "Discover essential tips for your solo adventure...",
      "content": "{\"root\":{...}}",
      "imageUrl": "/images/blog/solo-travel.jpg",
      "published": true,
      "createdAt": "2025-11-01T10:00:00Z",
      "updatedAt": "2025-11-01T10:00:00Z",
      "publishedAt": "2025-11-01T10:00:00Z",
      "author": {
        "id": "author-guid",
        "fullName": "Jane Writer"
      }
    }
  ],
  "totalCount": 25,
  "page": 1,
  "pageSize": 10,
  "totalPages": 3
}
```

Get Blog Post by Slug

Returns a single published blog post by its URL slug.

```
GET /api/blog/{slug}
```

Path Parameters:

Parameter	Type	Description
`slug`	string	The URL-friendly post identifier

Example Request:

```
GET /api/blog/10-tips-for-solo-travelers
```

Success Response (200):

```
{
  "id": 1,
  "title": "10 Tips for Solo Travelers",
  "slug": "10-tips-for-solo-travelers",
  "category": "solo-travel",
  "excerpt": "Discover essential tips for your solo adventure...",
  "content": "{\"root\":{...}}",
  "imageUrl": "/images/blog/solo-travel.jpg",
  "published": true,
  "createdAt": "2025-11-01T10:00:00Z",
  "updatedAt": "2025-11-01T10:00:00Z",
  "publishedAt": "2025-11-01T10:00:00Z",
  "author": {
    "id": "author-guid",
    "fullName": "Jane Writer"
  }
}
```

Error Response (404):

```
{
  "message": "Blog post not found"
}
```

Booking Endpoints

Create Booking

Creates a new booking with one or more items. Supports both authenticated users and guest checkout.

```
POST /api/bookings
```

Authentication: Optional (guest checkout supported)

Request Body:

Field	Type	Required	Description
`customerEmail`	string	Yes*	Customer email (*required for guests)
`customerName`	string	Yes*	Customer name (*required for guests)
`paymentReference`	string	No	External payment reference
`items`	array	Yes	Array of booking items

Booking Item Object:

Field	Type	Required	Description
`type`	string	Yes	Item type: `Flight`, `Hotel`, or `Tour`
`title`	string	Yes	Item title/name
`details`	string	No	Additional details
`price`	decimal	Yes	Item price
`imageUrl`	string	No	Item image URL

Example Request:

```
{
  "customerEmail": "guest@example.com",
  "customerName": "Guest User",
  "paymentReference": "PAY-123456",
  "items": [
    {
      "type": "Flight",
      "title": "New York to Paris",
      "details": "Round trip, Economy class",
      "price": 850.00,
      "imageUrl": "/images/flights/nyc-paris.jpg"
    },
    {
      "type": "Hotel",
      "title": "Hotel Le Marais",
      "details": "3 nights, Double room",
      "price": 450.00,
      "imageUrl": "/images/hotels/le-marais.jpg"
    }
  ]
}
```

Success Response (201):

```
{
  "id": 42,
  "message": "Booking created successfully",
  "totalAmount": 1300.00,
  "status": "Confirmed"
}
```

Error Response (400):

```
{
  "message": "At least one booking item is required"
}
```

Get User's Bookings

Returns all bookings for the authenticated user.

```
GET /api/bookings
```

Authentication: Required

Success Response (200):

```
{
  "bookings": [
    {
      "id": 42,
      "customerEmail": "user@example.com",
      "customerName": "John Doe",
      "totalAmount": 1300.00,
      "status": "Confirmed",
      "createdAt": "2025-11-15T14:30:00Z",
      "confirmedAt": "2025-11-15T14:30:00Z",
      "cancelledAt": null,
      "paymentReference": "PAY-123456",
      "items": [
        {
          "id": 1,
          "type": "Flight",
          "title": "New York to Paris",
          "details": "Round trip, Economy class",
          "price": 850.00,
          "imageUrl": "/images/flights/nyc-paris.jpg"
        }
      ]
    }
  ]
}
```

Get Booking by ID

Returns a specific booking by ID.

```
GET /api/bookings/{id}
```

Authentication: Required (user can only view their own bookings)

Path Parameters:

Parameter	Type	Description
`id`	integer	Booking ID

Success Response (200):

```
{
  "id": 42,
  "customerEmail": "user@example.com",
  "customerName": "John Doe",
  "totalAmount": 1300.00,
  "status": "Confirmed",
  "createdAt": "2025-11-15T14:30:00Z",
  "confirmedAt": "2025-11-15T14:30:00Z",
  "cancelledAt": null,
  "paymentReference": "PAY-123456",
  "items": [ ... ]
}
```

Error Response (404):

```
{
  "message": "Booking not found"
}
```

Error Response (403):

Returns `Forbid` if user tries to access another user's booking.

Activity Tracking Endpoints

Ping (Visitor Tracking)

Tracks anonymous visitor activity. Creates a visitor ID cookie if not present.

```
GET /api/ping
```

Success Response (200):

```
{
  "status": "ok"
}
```

Sets `visitor_id` cookie if not already present.

Heartbeat (User Activity)

Tracks authenticated user activity for real-time online status.

```
POST /api/heartbeat
```

Authentication: Optional (only tracks authenticated users)

Success Response (200):

```
{}
```

Admin Authentication Endpoints

Admin Login

Authenticates an administrator using credentials from configuration.

```
POST /api/admin/login
```

Request Body:

Field	Type	Required	Description
`email`	string	Yes	Admin email
`password`	string	Yes	Admin password

Example Request:

```
{
  "email": "admin@mytravel.com",
  "password": "Admin@123!"
}
```

Success Response (200):

Sets `admin_session` cookie (HttpOnly, Secure, 8-hour expiry) and returns:

```
{
  "message": "Admin login successful",
  "isAdmin": true
}
```

Error Response (401):

Returns `Unauthorized` for invalid credentials.

Admin Logout

Ends the admin session.

```
POST /api/admin/logout
```

Success Response (200):

```
{
  "message": "Admin logged out"
}
```

Verify Admin Session

Checks if the current session has admin privileges.

```
GET /api/admin/verify
```

Success Response (200):

```
{
  "isAdmin": true
}
```

Error Response (401):

Returns `Unauthorized` if no valid admin session.

Admin User Management Endpoints

Get All Users

Returns a paginated, searchable list of all users.

```
GET /api/admin/users
```

Authentication: Admin required

Query Parameters:

Parameter	Type	Default	Description
`page`	integer	1	Page number
`pageSize`	integer	10	Items per page
`search`	string	null	Search by username or email
`sortBy`	string	“createdAt”	Sort field: `username`, `email`, `createdAt`, `lastLoginAt`
`sortOrder`	string	“desc”	Sort direction: `asc` or `desc`

Example Request:

```
GET /api/admin/users?page=1&pageSize=20&search=john&sortBy=createdAt&sortOrder=desc
```

Success Response (200):

```
{
  "users": [
    {
      "id": "user-guid",
      "userName": "john@example.com",
      "email": "john@example.com",
      "createdAt": "2025-10-01T08:00:00Z",
      "lastLoginAt": "2025-11-28T09:30:00Z",
      "isActive": true,
      "emailConfirmed": true
    }
  ],
  "totalCount": 150,
  "page": 1,
  "pageSize": 20,
  "totalPages": 8
}
```

Get User by ID

Returns detailed information for a specific user.

```
GET /api/admin/users/{id}
```

Authentication: Admin required

Path Parameters:

Parameter	Type	Description
`id`	string	User ID (GUID)

Success Response (200):

```
{
  "id": "user-guid",
  "userName": "john@example.com",
  "email": "john@example.com",
  "createdAt": "2025-10-01T08:00:00Z",
  "lastLoginAt": "2025-11-28T09:30:00Z",
  "isActive": true,
  "emailConfirmed": true
}
```

Error Response (404):

```
{
  "message": "User not found"
}
```

Update User

Updates a user's information.

```
PUT /api/admin/users/{id}
```

Authentication: Admin required

Request Body:

Field	Type	Required	Description
`userName`	string	No	New username
`isActive`	boolean	No	Active status
`emailConfirmed`	boolean	No	Email confirmation status

Example Request:

```
{
  "userName": "john.doe@example.com",
  "isActive": true,
  "emailConfirmed": true
}
```

Success Response (200):

```
{
  "message": "User updated successfully",
  "user": {
    "id": "user-guid",
    "userName": "john.doe@example.com",
    "email": "john@example.com",
    "createdAt": "2025-10-01T08:00:00Z",
    "lastLoginAt": "2025-11-28T09:30:00Z",
    "isActive": true,
    "emailConfirmed": true
  }
}
```

Toggle User Status

Toggles a user's active/inactive status.

```
PATCH /api/admin/users/{id}/toggle-status
```

Authentication: Admin required

Success Response (200):

```
{
  "message": "User deactivated successfully",
  "isActive": false
}
```

Delete User

Permanently deletes a user account.

```
DELETE /api/admin/users/{id}
```

Authentication: Admin required

Success Response (200):

```
{
  "message": "User deleted successfully"
}
```

Error Response (400):

```
{
  "message": "Failed to delete user",
  "errors": [ ... ]
}
```

Get Admin Statistics

Returns platform-wide statistics for the admin dashboard.

```
GET /api/admin/stats
```

Authentication: Admin required

Success Response (200):

```
{
  "totalUsers": 1250,
  "activeUsersCount": 1180,
  "currentlyOnline": 45,
  "newUsersToday": 12,
  "newUsersThisWeek": 87,
  "newUsersThisMonth": 320
}
```

Admin Booking Management Endpoints

Get All Bookings

Returns a paginated list of all bookings with filtering and sorting.

```
GET /api/admin/bookings
```

Authentication: Admin required

Query Parameters:

Parameter	Type	Default	Description
`page`	integer	1	Page number
`pageSize`	integer	10	Items per page
`search`	string	null	Search by email, name, or booking ID
`status`	string	null	Filter by status: `Pending`, `Confirmed`, `Cancelled`, `Completed`
`sortBy`	string	"createdAt"	Sort field: `customerName`, `customerEmail`, `totalAmount`, `status`, `createdAt`
`sortOrder`	string	"desc"	Sort direction: `asc` or `desc`

Success Response (200):

```
{
  "bookings": [
    {
      "id": 42,
      "customerEmail": "user@example.com",
      "customerName": "John Doe",
      "totalAmount": 1300.00,
      "status": "Confirmed",
      "createdAt": "2025-11-15T14:30:00Z",
      "confirmedAt": "2025-11-15T14:30:00Z",
      "cancelledAt": null,
      "paymentReference": "PAY-123456",
      "items": [ ... ]
    }
  ],
  "totalCount": 500,
  "page": 1,
  "pageSize": 10,
  "totalPages": 50
}
```

Get Booking by ID (Admin)

Returns detailed booking information.

```
GET /api/admin/bookings/{id}
```

Authentication: Admin required

Success Response (200):

Same structure as user booking response.

Update Booking Status

Updates the status of a booking.

```
PATCH /api/admin/bookings/{id}/status
```

Authentication: Admin required

Request Body:

Field	Type	Required	Description
`status`	string	Yes	New status: `Pending`, `Confirmed`, `Cancelled`, `Completed`

Example Request:

```
{
  "status": "Completed"
}
```

Success Response (200):

```
{
  "message": "Booking status updated successfully",
  "id": 42,
  "status": "Completed"
}
```

Error Response (400):

```
{
  "message": "Invalid status value"
}
```

Get Booking Statistics

Returns booking-related statistics.

```
GET /api/admin/bookings/stats
```

Authentication: Admin required

Success Response (200):

```
{
  "totalBookings": 500,
  "pendingBookings": 25,
  "confirmedBookings": 350,
  "cancelledBookings": 50,
  "completedBookings": 75,
  "totalRevenue": 425000.00,
  "bookingsToday": 8,
  "bookingsThisWeek": 45,
  "bookingsThisMonth": 180
}
```

Admin Blog Management Endpoints

Get All Blog Posts (Admin)

Returns all blog posts including drafts with filtering and sorting.

```
GET /api/admin/blog
```

Authentication: Admin required

Query Parameters:

Parameter	Type	Default	Description
`page`	integer	1	Page number
`pageSize`	integer	10	Items per page
`search`	string	null	Search in title or excerpt
`category`	string	null	Filter by category
`status`	string	null	Filter: `published` or `draft`
`sortBy`	string	"createdAt"	Sort field: `title`, `category`, `published`, `publishedAt`, `createdAt`
`sortOrder`	string	"desc"	Sort direction: `asc` or `desc`

Success Response (200):

```
{
  "posts": [
    {
      "id": 1,
      "title": "Draft Post",
      "slug": "draft-post",
      "category": "adventure",
      "excerpt": "This is a draft...",
      "content": "{...}",
      "imageUrl": null,
      "published": false,
      "createdAt": "2025-11-20T10:00:00Z",
      "updatedAt": "2025-11-20T10:00:00Z",
      "publishedAt": null,
      "author": null
    }
  ],
  "totalCount": 30,
  "page": 1,
  "pageSize": 10,
  "totalPages": 3
}
```

Get Blog Post by ID (Admin)

Returns a specific blog post by ID (including drafts).

```
GET /api/admin/blog/{id}
```

Authentication: Admin required

Success Response (200):

Same structure as public blog post response.

Create Blog Post

Creates a new blog post.

```
POST /api/admin/blog
```

Authentication: Admin required

Request Body:

Field	Type	Required	Description
`title`	string	Yes	Post title
`authorId`	string	No	Author user ID
`category`	string	No	Category (default: "solo-travel")
`excerpt`	string	No	Short description
`content`	string	Yes	Lexical editor JSON content
`imageUrl`	string	No	Featured image URL
`published`	boolean	No	Publish immediately (default: false)

Example Request:

```
{
  "title": "New Adventure Guide",
  "category": "adventure",
  "excerpt": "Discover the best adventure destinations...",
  "content": "{\"root\":{\"children\":[...]}}",
  "imageUrl": "/images/blog/adventure.jpg",
  "published": false
}
```

Success Response (201):

```
{
  "id": 15,
  "slug": "new-adventure-guide",
  "message": "Blog post created successfully"
}
```

Error Response (400):

```
{
  "message": "Title is required"
}
```

Update Blog Post

Updates an existing blog post.

```
PUT /api/admin/blog/{id}
```

Authentication: Admin required

Request Body:

Field	Type	Required	Description
`title`	string	No	Updated title (regenerates slug)
`category`	string	No	Updated category
`excerpt`	string	No	Updated excerpt
`content`	string	No	Updated content
`imageUrl`	string	No	Updated image URL
`published`	boolean	No	Publication status

Success Response (200):

```
{
  "message": "Blog post updated successfully",
  "id": 15,
  "slug": "updated-adventure-guide"
}
```

Delete Blog Post

Permanently deletes a blog post.

```
DELETE /api/admin/blog/{id}
```

Authentication: Admin required

Success Response (200):

```
{
  "message": "Blog post deleted successfully"
}
```

Toggle Blog Post Publication

Toggles the published/draft status of a blog post.

```
PATCH /api/admin/blog/{id}/toggle-publish
```

Authentication: Admin required

Success Response (200):

```
{
  "message": "Blog post published successfully",
  "published": true
}
```

Get Blog Statistics

Returns blog-related statistics.

```
GET /api/admin/blog/stats
```

Authentication: Admin required

Success Response (200):

```
{
  "totalPosts": 30,
  "publishedPosts": 25,
  "draftPosts": 5,
  "postsThisWeek": 3,
  "postsThisMonth": 12,
  "postsByCategory": [
    { "category": "solo-travel", "count": 10 },
    { "category": "adventure", "count": 8 },
    { "category": "family-trips", "count": 7 },
    { "category": "luxury-travel", "count": 5 }
  ]
}
```

HTTP Status Codes

Code	Description
`200`	OK - Request successful
`201`	Created - Resource created successfully
`400`	Bad Request - Invalid input or validation error
`401`	Unauthorized - Authentication required or failed
`403`	Forbidden - Access denied to resource
`404`	Not Found - Resource does not exist
`500`	Internal Server Error - Server-side error

Rate Limiting

The API does not currently implement rate limiting. For production deployments, consider implementing rate limiting at the reverse proxy level.

CORS

Cross-Origin Resource Sharing (CORS) is configured to allow requests from the frontend development server during development. In production, CORS should be configured to allow only trusted origins.