Implementation Documentation

Complete technical documentation of API endpoints and frontend screens for the Event Ticket Booking System.

Table of Contents

- Backend API Reference
  - Authentication APIs
  - Event APIs
  - Booking APIs
  - Admin APIs
- Frontend Screens Reference
  - Authentication Screens
  - User Screens
  - Organizer Screens
  - Admin Screens

Backend API Reference

Base URL: `http://localhost:3000/api`

Authentication: Bearer token in Authorization header (except login/register)

Common Response Format:

```
{
  "success": true/false,
  "message": "Descriptive message",
  "data": { /* response data */ }
}
```

---

1. Authentication APIs

1.1 Register User

Endpoint: `POST /api/auth/register`

Description: Create a new user account

Access: Public

Request Body:

```
{
  "name": "John Doe",
  "email": "john@example.com",
  "password": "password123",
  "role": "user"
}
```

Field Validations:

- `name` : Required, string
- `email` : Required, unique, valid email format
- `password` : Required, minimum 6 characters
- `role` : Required, enum: ["user", "organizer", "admin"]

Success Response (201):

```
{
  "success": true,
  "message": "User registered successfully",
  "user": {
    "id": "507f1f77bcf86cd799439011",
    "name": "John Doe",
    "email": "john@example.com",
    "role": "user"
  }
}
```

Error Responses:

400 - Validation Error:

```
{
  "success": false,
  "message": "Please provide all required fields"
}
```

400 - Duplicate Email:

```
{
  "success": false,
  "message": "User already exists with this email"
}
```

1.2 Login

Endpoint: `POST /api/auth/login`

Description: Authenticate user and receive JWT token

Access: Public

Request Body:

```
{
  "email": "john@example.com",
  "password": "password123"
}
```

Success Response (200):

```
{
  "success": true,
  "message": "Login successful",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "user": {
    "id": "507f1f77bcf86cd799439011",
    "name": "John Doe",
    "email": "john@example.com",
    "role": "user"
  }
}
```

Error Responses:

400 - Invalid Credentials:

```
{
  "success": false,
  "message": "Invalid email or password"
}
```

404 - User Not Found:

```
{
  "success": false,
  "message": "User not found"
}
```

---

2. Event APIs

2.1 Get All Events

Endpoint: `GET /api/events`

Description: Retrieve all events (public access)

Access: Public

Request: No body required

Success Response (200):

```
{
  "success": true,
  "count": 2,
  "events": [
    {
      "_id": "507f1f77bcf86cd799439011",
      "title": "Tech Conference 2026",
      "description": "Annual technology conference",
      "date": "2026-06-15T10:00:00.000Z",
      "imageUrl": "https://res.cloudinary.com/.../image.jpg",
      "location": "San Francisco, CA",
      "totalTickets": 500,
      "availableTickets": 450,
      "price": 99.99,
      "organizer": {
        "_id": "507f...",
        "name": "Organizer Name",
        "email": "organizer@example.com"
      },
      "attendees": ["userId1", "userId2"]
    }
  ]
}
```

2.2 Get My Events (Organizer Only)

Endpoint: `GET /api/events/my-events`

Description: Get only the authenticated organizer's events

Access: Organizer only

Headers:

```
Authorization: Bearer <token>
```

Success Response (200):

```
  {
    "success": true,
    "count": 3,
    "events": [
      {
        "_id": "...",
        "title": "My Event"
        // ... same structure as Get All Events
      }
    ]
  }
```

Error Responses:

401 - Unauthorized:

```
  {
    "success": false,
    "message": "Access denied. No token provided."
  }
```

403 - Forbidden:

```
  {
    "success": false,
    "message": "Access denied. Required role: organizer"
  }
```

2.3 Get Single Event

Endpoint: `GET /api/events/:id`

Description: Get details of a specific event

Access: Public

URL Parameters:

- `id` : Event ID (ObjectId)

Success Response (200):

```
  {
    "success": true,
    "event": {
      "_id": "507f1f77bcf86cd799439011",
      "title": "Tech Conference 2026"
      // ... full event details
    }
  }
```

Error Responses:

404 - Not Found:

```
  {
    "success": false,
    "message": "Event not found"
  }
```

2.4 Create Event (Organizer Only)

Endpoint: `POST /api/events`

Description: Create a new event

Access: Organizer only

Headers:

```
Authorization: Bearer <token>
```

Request Body:

```json
{
  "title": "Tech Conference 2026",
  "description": "Annual technology conference with top speakers",
  "date": "2026-06-15T10:00:00",
  "imageUrl": "https://res.cloudinary.com/.../image.jpg",
  "location": "San Francisco, CA",
  "totalTickets": 500,
  "price": 99.99
}
```

Field Validations:

- `title` : Required, string
- `description` : Required, string
- `date` : Required, valid date
- `imageUrl` : Required, string (Cloudinary URL)
- `location` : Required, string
- `totalTickets` : Required, number > 0
- `price` : Required, number >= 0

Success Response (201):

```json
{
  "success": true,
  "message": "Event created successfully",
  "event": {
    "_id": "507f1f77bcf86cd799439011",
    "title": "Tech Conference 2026",
    "availableTickets": 500,
    "organizer": "organizerId"
    // ... all fields
  }
}
```

Error Responses:

400 - Validation Error:

```
  {
    "success": false,
    "message": "Please provide all required fields"
  }
```

400 - Invalid Data:

```
  {
    "success": false,
    "message": "Total tickets must be positive and price must be non-negati
  }
```

2.5 Update Event (Organizer Only)

Endpoint: `PUT /api/events/:id`

Description: Update an existing event (only event creator can update)

Access: Organizer only (must be event creator)

Headers:

```
Authorization: Bearer <token>
```

URL Parameters:

- `id` : Event ID

Request Body (all fields optional):

```
  {
    "title": "Updated Tech Conference 2026",
    "description": "Updated description",
    "date": "2026-07-15T10:00:00",
    "imageUrl": "https://res.cloudinary.com/.../new-image.jpg",
    "location": "New York, NY",
    "totalTickets": 600,
    "price": 129.99
  }
```

Business Rules:

- Only provided fields are updated
- `totalTickets` cannot be reduced below already sold tickets
- `availableTickets` is automatically recalculated

Success Response (200):

```
  {
    "success": true,
    "message": "Event updated successfully",
    "event": {
      // ... updated event details
    }
  }
```

Error Responses:

403 - Not Event Owner:

```
  {
    "success": false,
    "message": "You can only update your own events"
  }
```

400 - Invalid Ticket Count:

```
  {
    "success": false,
    "message": "Cannot reduce total tickets below 50 (already sold)"
  }
```

404 - Not Found:

```
  {
    "success": false,
    "message": "Event not found"
  }
```

2.6 Delete Event (Organizer Only)

Endpoint: `DELETE /api/events/:id`

Description: Delete an event (only event creator can delete)

Access: Organizer only (must be event creator)

Headers:

```
Authorization: Bearer <token>
```

URL Parameters:

- `id` : Event ID

Success Response (200):

```
{
  "success": true,
  "message": "Event deleted successfully"
}
```

Error Responses:

403 - Not Event Owner:

```
{
  "success": false,
  "message": "You can only delete your own events"
}
```

404 - Not Found:

```
{
  "success": false,
  "message": "Event not found"
}
```

---

3. Booking APIs

3.1 Create Booking (User Only)

Endpoint: `POST /api/bookings`

Description: Book tickets for an event

Access: User only

Headers:

```
Authorization: Bearer <token>
```

Request Body:

```
{
  "eventId": "507f1f77bcf86cd799439011",
  "numberOfTickets": 2
}
```

Field Validations:

- `eventId` : Required, valid ObjectId
- `numberOfTickets` : Required, number > 0

Business Logic:

1. Checks ticket availability
2. Atomically reduces availableTickets
3. Adds user to attendees array
4. Creates booking record
5. Prevents race conditions

Success Response (201):

```
{
  "success": true,
  "message": "Booking successful",
  "booking": {
    "_id": "booking123",
    "user": {
      "_id": "user123",
      "name": "John Doe",
      "email": "john@example.com"
    },
    "event": {
      "_id": "event123",
      "title": "Tech Conference 2026",
      "date": "2026-06-15T10:00:00.000Z",
      "location": "San Francisco, CA",
      "price": 99.99
    },
    "numberOfTickets": 2,
    "bookingDate": "2026-02-12T08:30:00.000Z"
  }
}
```

Error Responses:

400 - Validation Error:

```
  {
    "success": false,
    "message": "Please provide eventId and numberOfTickets"
  }
```

400 - Insufficient Tickets:

```
  {
    "success": false,
    "message": "Insufficient tickets. Only 1 tickets available"
  }
```

400 - Race Condition:

```
  {
    "success": false,
    "message": "Tickets no longer available. Please try again"
  }
```

404 - Event Not Found:

```
  {
    "success": false,
    "message": "Event not found"
  }
```

403 - Wrong Role:

```
  {
    "success": false,
    "message": "Access denied. Required role: user"
  }
```

3.2 Get My Bookings (User Only)

Endpoint: `GET /api/bookings/my`

Description: Get all bookings for authenticated user

Access: User only

Headers:

```
Authorization: Bearer <token>
```

Success Response (200):

```
  {
    "success": true,
    "count": 3,
    "bookings": [
      {
        "_id": "booking123",
        "event": {
          "_id": "event123",
          "title": "Tech Conference 2026",
          "date": "2026-06-15T10:00:00.000Z",
          "location": "San Francisco, CA",
          "price": 99.99,
          "imageUrl": "https://..."
        },
        "numberOfTickets": 2,
        "bookingDate": "2026-02-12T08:30:00.000Z"
      }
    ]
  }
```

3.3 Get Event Bookings (Organizer Only)

Endpoint: `GET /api/bookings/event/:id`

Description: Get all bookings for a specific event (only for event organizer)

Access: Organizer only (must be event creator)

Headers:

```
Authorization: Bearer <token>
```

URL Parameters:

- `id` : Event ID

Success Response (200):

```
{
  "success": true,
  "event": {
    "_id": "event123",
    "title": "Tech Conference 2026"
  },
  "summary": {
    "totalTicketsSold": 50,
    "totalRevenue": 4999.5,
    "totalBookings": 25
  },
  "bookings": [
    {
      "_id": "booking123",
      "user": {
        "_id": "user123",
        "name": "John Doe",
        "email": "john@example.com"
      },
      "numberOfTickets": 2,
      "bookingDate": "2026-02-12T08:30:00.000Z"
    }
  ]
}
```

Error Responses:

403 - Not Event Owner:

```
{
  "success": false,
  "message": "You can only view bookings for your own events"
}
```

---

4. Admin APIs

4.1 Get All Events (Admin Only)

Endpoint: `GET /api/admin/events`

Description: Get all events with organizer information

Access: Admin only

Headers:

```
Authorization: Bearer <token>
```

Success Response (200):

```
{
  "success": true,
  "count": 10,
  "events": [
    {
      "_id": "...",
      "title": "Event Title",
      "organizer": {
        "_id": "org123",
        "name": "Organizer Name",
        "email": "org@example.com"
      }
      // ... all event fields
    }
  ]
}
```

4.2 Get All Bookings (Admin Only)

Endpoint: `GET /api/admin/bookings`

Description: Get all bookings across all events

Access: Admin only

Headers:

```
Authorization: Bearer <token>
```

Success Response (200):

```json
{
  "success": true,
  "count": 150,
  "bookings": [
    {
      "_id": "booking123",
      "user": {
        "_id": "user123",
        "name": "John Doe",
        "email": "john@example.com"
      },
      "event": {
        "_id": "event123",
        "title": "Tech Conference 2026",
        "date": "2026-06-15T10:00:00.000Z",
        "location": "San Francisco, CA",
        "price": 99.99
      },
      "numberOfTickets": 2,
      "bookingDate": "2026-02-12T08:30:00.000Z"
    }
  ]
}
```

4.3 Get Analytics (Admin Only)

Endpoint: `GET /api/admin/stats`

Description: Get comprehensive analytics using aggregation pipelines

Access: Admin only

Headers:

```
Authorization: Bearer <token>
```

Success Response (200):

```
{
  "success": true,
  "analytics": {
    "overview": {
      "totalEvents": 10,
      "totalBookings": 150,
      "totalTicketsSold": 350,
      "totalRevenue": 34995.5
    },
    "eventStats": [
      {
        "eventId": "event123",
        "eventTitle": "Tech Conference 2026",
        "totalTicketsSold": 50,
        "totalBookings": 25,
        "price": 99.99,
        "revenue": 4999.5
      }
    ],
    "topUsers": [
      {
        "userId": "user123",
        "userName": "John Doe",
        "userEmail": "john@example.com",
        "totalBookings": 5,
        "totalTickets": 12
      }
    ]
  }
}
```

Aggregation Pipeline Details:

Pipeline 1 - Event Statistics:

1. Group bookings by event
2. Sum total tickets sold per event

3. Lookup event details
4. Calculate revenue (tickets × price)
5. Sort by revenue descending

Pipeline 2 - Overview:

1. Count total bookings
2. Sum all tickets sold

Pipeline 3 - Top Users:

1. Group bookings by user
2. Count total bookings per user
3. Sum total tickets per user
4. Lookup user details
5. Sort by bookings descending
6. Limit to top 10

4.4 Update Any Event (Admin Only)

Endpoint: `PUT /api/admin/events/:id`

Description: Update any event (admins can update events they don't own)

Access: Admin only

Headers:

```
Authorization: Bearer <token>
```

URL Parameters:

- `id` : Event ID

Request Body (all fields optional):

```
{
  "title": "Updated by Admin",
  "price": 200.0,
  "totalTickets": 700
}
```

Success Response (200):

```
{
  "success": true,
  "message": "Event updated successfully by admin",
  "event": {
    // ... updated event
  }
}
```

4.5 Delete Any Event (Admin Only)

Endpoint: `DELETE /api/admin/events/:id`

Description: Delete any event (admins can delete events they don't own)

Access: Admin only

Headers:

```
Authorization: Bearer <token>
```

URL Parameters:

- `id` : Event ID

Success Response (200):

```
{
  "success": true,
  "message": "Event deleted successfully by admin"
}
```

Frontend Screens Reference

Screen Architecture

The app uses **Provider** pattern for state management with three main providers:

- **AuthProvider**: Authentication state
- **EventProvider**: Event data and operations
- **BookingProvider**: Booking operations

Navigation Structure:

```
AuthWrapper (Initial)
    ↓
Login/Register Screens
    ↓
Role-based Dashboard:
    - User → Event List
    - Organizer → Organizer Dashboard
    - Admin → Admin Dashboard
```

---

1. Authentication Screens

1.1 Login Screen

File: `lib/screens/login_screen.dart`

Purpose: Authenticate users and navigate to role-specific dashboards

UI Components:

- Email input field (TextFormField)
- Password input field (obscured text)
- Login button
- "Register" navigation link
- Error message display (SnackBar)
- Loading indicator during authentication

State Management:

- Uses `AuthProvider`
- Calls `authProvider.login(email, password)`

Validation:

- Email: Required, valid format
- Password: Required, minimum length

User Flow:

1. User enters email and password
2. Taps Login button
3. `AuthProvider` sends request to `/api/auth/login`

4. On success:
    - JWT token stored in SharedPreferences
    - User object stored
    - Navigate to role-based screen:
        - User → EventListScreen
        - Organizer → OrganizerDashboardScreen
        - Admin → AdminDashboardScreen
5. On error: Display error message

Key Functions:

```
void _handleLogin() async {
    if (_formKey.currentState!.validate()) {
        final success = await authProvider.login(
            email: _emailController.text,
            password: _passwordController.text,
        );

        if (success) {
            // Navigation handled by AuthWrapper
        } else {
            // Show error
            ScaffoldMessenger.of(context).showSnackBar(
                SnackBar(content: Text(authProvider.errorMessage))
            );
        }
    }
}
```

1.2 Register Screen

File: `lib/screens/register_screen.dart`

Purpose: Create new user accounts

UI Components:

- Name input field
- Email input field
- Password input field
- Role dropdown (User, Organizer, Admin)
- Register button

- "Login" navigation link
- Success/Error messages

State Management:

- Uses `AuthProvider`
- Calls `authProvider.register()`

Validation:

- Name: Required
- Email: Required, valid format, unique
- Password: Required, minimum 6 characters
- Role: Required selection

User Flow:

1. User fills registration form
2. Selects role from dropdown
3. Taps Register button
4. `AuthProvider` sends request to `/api/auth/register`
5. On success:
    - Shows success message
    - Automatically navigates to login
6. On error: Display error message

Role Selection:

```
DropdownButtonFormField<String>(
    value: _selectedRole,
    items: [
        DropdownMenuItem(value: 'user', child: Text('User')),
        DropdownMenuItem(value: 'organizer', child: Text('Organizer')),
        DropdownMenuItem(value: 'admin', child: Text('Admin')),
    ],
    onChanged: (value) => setState(() => _selectedRole = value),
)
```

---

2. User Screens

2.1 Event List Screen

File: `lib/screens/event_list_screen.dart`

Purpose: Browse all available events

UI Components:

- App bar with title and logout button
- Grid view of event cards
- Each card shows:
  - Event image (from Cloudinary)
  - Event title
  - Date and time
  - Location
  - Price (₹)
  - Available tickets
- Pull-to-refresh
- Loading indicator
- Empty state message
- "My Bookings" navigation button

State Management:

- Uses `EventProvider` and `AuthProvider`
- Calls `eventProvider.fetchAllEvents()`

Data Flow:

1. `initState()` calls `fetchAllEvents()`
2. Events loaded from `/api/events`
3. Grid displays all events
4. User taps event → navigate to EventDetailScreen
5. Returns from detail screen → auto-refresh if booking made

Auto-Refresh Implementation:

```
onTap: () async {
    final result = await Navigator.push(
        context,
        MaterialPageRoute(
            builder: (context) => EventDetailScreen(event: event),
        ),
    );

    // If true returned, booking was made
    if (result == true) {
        eventProvider.fetchAllEvents(); // Refresh
    }
}
```

Pull-to-Refresh:

```
RefreshIndicator(
    onRefresh: () => eventProvider.fetchAllEvents(),
    child: GridView.builder(...)
)
```

2.2 Event Detail Screen

File: `lib/screens/event_detail_screen.dart`

Purpose: View event details and book tickets

UI Components:

- Large event image
- Event title (headline text)
- Date, time, location, organizer info
- Price per ticket
- Available tickets counter
- Description text
- Ticket quantity selector (- and + buttons)
- Total price calculation
- "Book Now" button
- Edit button (only visible to event organizer)
- Confirmation dialog

State Management:

- Uses `BookingProvider` and `AuthProvider`
- Calls `bookingProvider.createBooking()`

Dynamic UI:

- For regular users: Shows booking section
- For event organizer: Hides booking, shows info message with edit option

User Flow (for users):

1. View event details
2. Adjust ticket quantity using +/- buttons
3. See total price update
4. Tap "Book Now"
5. Confirm in dialog
6. `BookingProvider` creates booking via `/api/bookings`
7. On success:
    - Show success message
    - `Navigator.pop(context, true)` to signal parent
    - Parent screen refreshes event list
8. On error: Show error message

User Flow (for organizers):

1. View event details
2. See "You are the organizer" message
3. Tap Edit button in app bar
4. Navigate to EditEventScreen
5. After edit, return and refresh dashboard

Ticket Selector:

```
Row(
    children: [
        IconButton(
            onPressed: _numberOfTickets > 1
                ? () => setState(() => _numberOfTickets--)
                : null,
            icon: Icon(Icons.remove_circle_outline),
        ),
        Text('$_numberOfTickets'),
        IconButton(
            onPressed: _numberOfTickets < availableTickets
                ? () => setState(() => _numberOfTickets++)
                : null,
            icon: Icon(Icons.add_circle_outline),
        ),
    ],
)
```

Total Price Display:

```
Text('₹${(event.price * _numberOfTickets).toStringAsFixed(2)}')
```

2.3 My Bookings Screen

File: `lib/screens/my_bookings_screen.dart`

Purpose: View user's booking history

UI Components:

- App bar with title
- List of booking cards
- Each card shows:
    - Event image (thumbnail)
    - Event title
    - Event date
    - Number of tickets booked
    - Total price paid
    - Booking date
- Empty state if no bookings
- Pull-to-refresh

State Management:

- Uses `BookingProvider`
- Calls `bookingProvider.fetchMyBookings()`

Data Flow:

1. `initState()` calls `fetchMyBookings()`
2. Bookings loaded from `/api/bookings/my`
3. List displays all user's bookings
4. Shows most recent bookings first

Booking Card Layout:

```
ListTile(
    leading: Image.network(booking.event.imageUrl),
    title: Text(booking.event.title),
    subtitle: Column(
        children: [
            Text('Date: ${formatDate(booking.event.date)}'),
            Text('Tickets: ${booking.numberOfTickets}'),
            Text('Total: ₹${totalPrice}'),
        ],
      ),
  )
```

---

3. Organizer Screens

3.1 Organizer Dashboard Screen

File: `lib/screens/organizer_dashboard_screen.dart`

Purpose: View and manage organizer's events

UI Components:

- App bar with "My Events" title and logout button
- List of event cards (organizer's events only)
- Each card shows:
  - Event image
  - Event title
  - Date and location
  - Statistics row:
    - Price (₹)
    - Tickets sold
    - Tickets available
  - Action buttons:
    - Bookings (view bookings)
    - Edit (edit event)
    - Delete (delete event)
- Floating Action Button (+) to create new event
- Pull-to-refresh
- Empty state message

State Management:

- Uses `EventProvider` and `AuthProvider`
- Calls `eventProvider.fetchMyEvents()`

Data Flow:

1. `initState()` calls `fetchMyEvents()`
2. Events loaded from `/api/events/my-events`
3. Only events created by this organizer are shown

4. Pull-to-refresh updates list

Action Buttons:

1. Bookings Button:

```
OutlinedButton.icon(
    onPressed: () {
        Navigator.push(
            context,
            MaterialPageRoute(
                builder: (context) => OrganizerEventBookingsScreen(
                    event: event,
                ),
            ),
        );
    },
    icon: Icon(Icons.people),
    label: Text('Bookings'),
)
```

2. Edit Button:

```
OutlinedButton.icon(
    onPressed: () async {
        final updated = await Navigator.push<bool>(
            context,
            MaterialPageRoute(
                builder: (context) => EditEventScreen(event: event),
            ),
        );

        if (updated == true) {
            eventProvider.fetchMyEvents(); // Refresh
        }
    },
    icon: Icon(Icons.edit),
    label: Text('Edit'),
)
```

3. Delete Button:

```
IconButton(
    onPressed: () => _showDeleteDialog(event.id),
    icon: Icon(Icons.delete),
    color: Colors.red,
)
```

Delete Confirmation:

```
void _showDeleteDialog(String eventId) {
    showDialog(
        context: context,
        builder: (context) => AlertDialog(
            title: Text('Delete Event'),
            content: Text('Are you sure?'),
            actions: [
                TextButton(onPressed: () => Navigator.pop(context)),
                ElevatedButton(
                    onPressed: () async {
                        Navigator.pop(context);
                        final success = await eventProvider.deleteEvent(e
                        // Show success/error message
                    },
                ),
            ],
        ),
    );
}
```

Statistics Display:

```
Row(
    mainAxisAlignment: MainAxisAlignment.spaceAround,
    children: [
        _buildStat('Price', '₹${event.price}', Icons.payments),
        _buildStat('Sold', '${ticketsSold}', Icons.confirmation_number),
        _buildStat('Available', '${event.availableTickets}', Icons.event_
    ],
)
```

3.2 Create Event Screen

File: `lib/screens/create_event_screen.dart`

Purpose: Create new events with image upload

UI Components:

- Form with fields:
  - Title (TextFormField)
  - Description (multiline TextFormField)
  - Location (TextFormField)
  - Date & Time picker (ListTile with date display)
  - Image upload section:
    - Image preview (if selected)
    - "Pick & Upload Image" button
    - Upload progress indicator
    - Success checkmark
  - Price (number input with ₹ symbol)
  - Total Tickets (number input)
- Create Event button
- Form validation
- Loading indicator during creation

State Management:

- Uses `EventProvider`
- Uses `CloudinaryService` for image upload

Image Upload Flow:

1. User taps "Pick & Upload Image"
2. `ImagePicker` opens gallery
3. User selects image
4. Image converted to `Uint8List` (bytes)
5. Preview shown immediately
6. Upload to Cloudinary starts
7. Progress indicator shown
8. On success:
   - Cloudinary URL received
   - Success message shown
   - URL stored for event creation
9. On error: Error message shown

Date & Time Picker:

```
Future<void> _pickDateTime() async {
    final date = await showDatePicker(...);
    if (date != null) {
        final time = await showTimePicker(...);
        if (time != null) {
            setState(() {
                _selectedDate = DateTime(
                    date.year, date.month, date.day,
                    time.hour, time.minute,
                );
            });
        }
    }
}
```

Image Upload Implementation:

```
Future<void> _pickAndUploadImage() async {
    final XFile? image = await _picker.pickImage(
        source: ImageSource.gallery,
        maxWidth: 1920,
        maxHeight: 1080,
        imageQuality: 85,
    );

    if (image == null) return;

    setState(() => _isUploading = true);

    final bytes = await image.readAsBytes();
    setState(() => _imageBytes = bytes);

    final imageUrl = await CloudinaryService.uploadImageBytes(
        bytes,
        'event_${DateTime.now().millisecondsSinceEpoch}.jpg',
    );

    setState(() {
        _uploadedImageUrl = imageUrl;
        _isUploading = false;
    });
}
```

Create Event Function:

```
void _handleCreateEvent() async {
    if (_formKey.currentState!.validate()) {
        if (_uploadedImageUrl == null) {
            // Show error: image required
            return;
        }

        final event = Event(
            id: '',
            title: _titleController.text.trim(),
            description: _descriptionController.text.trim(),
            date: _selectedDate,
            imageUrl: _uploadedImageUrl!,
            location: _locationController.text.trim(),
            totalTickets: int.parse(_totalTicketsController.text),
            price: double.parse(_priceController.text),
            organizerId: '',
        );

        final success = await eventProvider.createEvent(event);

        if (success) {
            // Show success
            Navigator.pop(context);
        } else {
            // Show error
        }
    }
}
```

3.3 Edit Event Screen

File: `lib/screens/edit_event_screen.dart`

Purpose: Edit existing event details

UI Components:

- Pre-filled form with existing event data
- Same fields as Create Event
- Current image displayed

- Option to upload new image
- Helper text showing tickets sold constraint
- Update Event button

State Management:

- Uses `EventProvider`
- Receives `Event` object as parameter

Pre-fill Logic:

```
@override
void initState() {
    super.initState();
    _titleController = TextEditingController(text: widget.event.title);
    _descriptionController = TextEditingController(text: widget.event.des
    _locationController = TextEditingController(text: widget.event.locati
    _priceController = TextEditingController(text: widget.event.price.toS
    _totalTicketsController = TextEditingController(text: widget.event.tc
    _selectedDate = widget.event.date;
    _uploadedImageUrl = widget.event.imageUrl; // Existing image
}
```

Ticket Constraint Validation:

```
TextFormField(
    controller: _totalTicketsController,
    decoration: InputDecoration(
        labelText: 'Total Tickets',
        helperText: 'Current: ${event.totalTickets} (${ticketsSold} sold)
    ),
    validator: (value) {
        final tickets = int.tryParse(value ?? '');
        final ticketsSold = event.totalTickets - event.availableTickets;

        if (tickets == null || tickets < ticketsSold) {
            return 'Cannot be less than $ticketsSold (already sold)';
        }
        return null;
    },
)
```

Update Function:

```
void _handleUpdateEvent() async {
    if (_formKey.currentState!.validate()) {
        final updates = {
            'title': _titleController.text.trim(),
            'description': _descriptionController.text.trim(),
            'date': DateFormat('yyyy-MM-ddTHH:mm:ss').format(_selectedDat
            'imageUrl': _uploadedImageUrl,
            'location': _locationController.text.trim(),
            'totalTickets': int.parse(_totalTicketsController.text),
            'price': double.parse(_priceController.text),
        };

        final success = await eventProvider.updateEvent(
            widget.event.id,
            updates,
        );

        if (success) {
            Navigator.pop(context, true); // Return true to refresh parer
        }
    }
}
```

3.4 Organizer Event Bookings Screen

File: `lib/screens/organizer_event_bookings_screen.dart`

Purpose: View all bookings for a specific event

UI Components:

- App bar with event title
- Summary section:
  - Total tickets sold
  - Total revenue (₹)
  - Number of bookings
- List of booking cards
- Each card shows:
  - Attendee name
  - Attendee email
  - Number of tickets
  - Total price

    - Booking date
- Empty state if no bookings

State Management:

- Uses `BookingProvider`
- Receives `Event` object as parameter

Data Flow:

1. `initState()` calls `fetchEventBookings(eventId)`
2. Bookings loaded from `/api/bookings/event/:id`
3. Summary calculated from booking list
4. List displays all bookings for this event

Summary Calculation:

```
int totalTicketsSold = bookings.fold(
    0,
    (sum, booking) => sum + booking.numberOfTickets
);

double totalRevenue = totalTicketsSold * widget.event.price;
```

Booking Card:

```
ListTile(
    leading: CircleAvatar(
        child: Icon(Icons.person),
    ),
    title: Text(booking.userName ?? 'N/A'),
    subtitle: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
            Text('Email: ${booking.userEmail}'),
            Text('Tickets: ${booking.numberOfTickets}'),
            Text('Price: ₹${booking.numberOfTickets * widget.event.price}
            Text('Booked: ${formatDate(booking.bookingDate)}'),
        ],
    ),
)
```

---

4. Admin Screens

4.1 Admin Dashboard Screen

File: `lib/screens/admin_dashboard_screen.dart`

Purpose: View analytics and system-wide statistics

UI Components:

1. **Overview Cards:**
    - Total Events (with event icon)
    - Total Bookings (with ticket icon)
    - Total Revenue (₹, with money icon)
2. **Revenue Chart:**
    - Bar chart showing revenue per event
    - X-axis: Event titles (abbreviated)
    - Y-axis: Revenue (₹)
    - Interactive tooltips
    - Color-coded bars
3. **Event Stats List:**
    - Event title
    - Tickets sold
    - Revenue
    - Number of bookings
    - Sorted by revenue (highest first)
4. **Top Users Section:**
    - User name
    - Total bookings
    - Total tickets purchased
    - Top 10 users

State Management:

- Uses Provider (can add AdminProvider if needed)
- Fetches data from `/api/admin/stats`

Data Flow:

1. `initState()` calls fetchAnalytics()
2. Data loaded from `/api/admin/stats`
3. Overview metrics displayed in cards
4. Chart data processed for fl_chart
5. Lists rendered with statistics

Overview Cards:

```
Card(
    child: Column(
        children: [
            Icon(Icons.event, size: 48, color: Colors.blue),
            SizedBox(height: 8),
            Text(
                '${analytics.overview.totalEvents}',
                style: TextStyle(fontSize: 32, fontWeight: FontWeight.bol
            ),
            Text('Total Events'),
        ],
    ),
)
```

Bar Chart Implementation:

```
BarChart(
    BarChartData(
        barGroups: eventStats.map((stat) => BarChartGroupData(
            x: index,
            barRods: [
                BarChartRodData(
                    toY: stat.revenue,
                    color: Colors.blue,
                    width: 16,
                ),
            ],
        )).toList(),
        titlesData: FlTitlesData(
            bottomTitles: AxisTitles(
                sideTitles: SideTitles(
                    showTitles: true,
                    getTitlesWidget: (value, meta) {
                        final eventTitle = eventStats[value.toInt()].ever
                        return Text(
                            eventTitle.length > 10
                                ? '${eventTitle.substring(0, 10)}...'
                                : eventTitle,
                        );
                    },
                ),
            ),
        ),
    ),
)
```

Event Stats List:

```
ListView.builder(
    itemCount: eventStats.length,
    itemBuilder: (context, index) {
        final stat = eventStats[index];
        return ListTile(
            leading: CircleAvatar(child: Text('${index + 1}')),
            title: Text(stat.eventTitle),
            subtitle: Text('${stat.totalTicketsSold} tickets sold'),
            trailing: Column(
                children: [
                    Text(
                        '₹${stat.revenue.toStringAsFixed(2)}',
                        style: TextStyle(
                            fontSize: 16,
                            fontWeight: FontWeight.bold,
                        ),
                    ),
                    Text('${stat.totalBookings} bookings'),
                ],
            ),
        );
    },
)
```

Top Users Display:

```
ListView.builder(
    itemCount: topUsers.length,
    itemBuilder: (context, index) {
        final user = topUsers[index];
        return ListTile(
            leading: CircleAvatar(
                child: Text('${index + 1}'),
                backgroundColor: index < 3
                    ? Colors.amber
                    : Colors.grey,
            ),
            title: Text(user.userName),
            subtitle: Text(user.userEmail),
            trailing: Column(
                children: [
                    Text('${user.totalBookings} bookings'),
                    Text('${user.totalTickets} tickets'),
                ],
            ),
        );
    },
)
```

Common Screen Patterns

Loading State

```
if (provider.isLoading) {
    return Center(child: CircularProgressIndicator());
}
```

Error State

```
if (provider.errorMessage != null) {
    return Center(
        child: Column(
            children: [
                Text(provider.errorMessage!),
                ElevatedButton(
                    onPressed: () => provider.retryFunction(),
                    child: Text('Retry'),
                ),
            ],
        ),
    );
}
```

Empty State

```
if (items.isEmpty) {
    return Center(
        child: Column(
            children: [
                Icon(Icons.event_busy, size: 80),
                Text('No items found'),
            ],
        ),
    );
}
```

SnackBar Messages

```
ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(
        content: Text('Operation successful'),
        backgroundColor: Colors.green,
    ),
);
```

Navigation Flow Summary

```
App Start
    ↓
AuthWrapper checks login status
    ↓
┌─────────────────┬─────────────────┬─────────────────┐
│ Not Logged In   │ User Role       │ Organizer Role  │ Admin Role
│ LoginScreen     │ EventListScreen │ OrganizerDash   │ AdminDash
└─────────────────┴─────────────────┴─────────────────┘
        ↓                ↓                 ↓             ↓
   RegisterScreen   EventDetail       CreateEvent    View Stats
```

```
                    ↓                  ↓
               MyBookings         EditEvent
                                     ↓
                               EventBookings
```

This implementation guide covers all API endpoints and frontend screens in detail. Use this as a reference for understanding data flow, business logic, and UI patterns throughout the application.