BonJoy Rider API Service

Complete TypeScript API Client with Axios & AsyncStorage

Version: 1.0.0 | **Base URL:** https://bonjoy.in/api/v1

Overview

This is a comprehensive TypeScript API client for the BonJoy Rider application that handles authentication, rider profile management, and emergency contacts. It features:

Key Features:

- **Type Safety:** Full TypeScript support with detailed interfaces
- **Authentication:** JWT-based auth with automatic token management
- **Local Storage:** AsyncStorage for caching and offline support
- **Interceptors:** Request/response interceptors for error handling
- **FormData Support:** Built-in multipart form data handling
- **Error Handling:** Comprehensive error management with fallbacks

Architecture Pattern: Repository Pattern with Service Layer

The service combines API calls with local storage persistence, providing a seamless data layer abstraction.

Installation & Setup

```
import axios, { AxiosError, AxiosInstance } from 'axios'; import AsyncStorage from '@react-native-async-storage/async-storage'; // Configuration const BASE_URL = 'https://bonjoy.in/api/v1'; const TIMEOUT = 15000; // Create Axios instance const api: AxiosInstance = axios.create({ baseURL: BASE_URL, timeout: TIMEOUT, });
```
Copy

Dependencies Required

Package	Version	Purpose
`axios`	^1.3.0	HTTP client for API requests
`@react-native-async-storage/async-storage`	^1.17.0	Persistent local storage
`react-native`	≥0.64.0	React Native framework

Storage Configuration

The service uses AsyncStorage with the following key structure:

Storage Keys

```
const AUTH_TOKEN_KEY = 'AUTH_TOKEN'; // JWT Token const USER_KEY = 'USER_SESSION'; // User session data const RIDER_PROFILE_KEY = 'RIDER_PROFILE'; // Rider profile data const USER_CONTACTS_KEY = 'USER_CONTACTS'; // Emergency contacts
```
Copy

Storage Helper Functions

```
// Save authentication session export const saveSession = async (token: string, user: UserSession) => { await AsyncStorage.multiSet([ [AUTH_TOKEN_KEY, token], [USER_KEY, JSON.stringify(user)], ]); }; // Clear all session data export const clearSession = async () => { await AsyncStorage.multiRemove([ AUTH_TOKEN_KEY, USER_KEY, RIDER_PROFILE_KEY, USER_CONTACTS_KEY, ]); }; // Get authentication token export const getAuthToken = async () => AsyncStorage.getItem(AUTH_TOKEN_KEY); // Get user session export const getUserSession = async (): Promise<UserSession | null> => { const data = await AsyncStorage.getItem(USER_KEY); return data ? JSON.parse(data) : null; };
```
Copy

Data Models (TypeScript Interfaces)

Common Response Interface

```
export interface ApiResponse<T> { success: boolean; message: string; data?: T; }
```
Copy

Authentication Models

```
export interface LoginWithMobileRequest { mobile: string; // 10-digit Indian mobile number } export interface LoginWithMobileResponse { success: boolean; message: string; } export interface VerifyOtpRequest { mobile: string; // Same mobile used for login otp: string; // 4-6 digit OTP } export interface UserSession { id: number; // User ID mobile: string; // User's mobile number userType: 'Rider' | string; // User type } export interface VerifyOtpResponse { success: boolean; message: string; token: string; // JWT token user: UserSession; // User session data }
```
Copy

Rider Profile Models

```
export interface RiderUser { id: number; email: string | null; mobile: string; userType: string; status: string; } export interface RiderProfileResult { profile_id: number; userId: number; fullName: string; gender: string; profileImage: string | null; city: string; preferredPaymentMethod: string | null; date_of_birth: string | null;
```
Copy

```
createdAt: string; updatedAt: string; User: RiderUser; } export interface
RiderProfileDetailData { results: RiderProfileResult[]; userContact: UserContact[]; }
export interface RiderProfileDetailResponse { success: boolean; message: string; data:
RiderProfileDetailData; } export interface RiderProfile { id: number; fullName:
string; gender: string; dob: string; city: string; profileImage?: string; email:
string; mobile: string; userType: string; status: string; remark?: string; createdAt:
string; }
```

User Contact Models

```
export interface UserContact { id: number; userId: number; contactType?: string;
relationship?: string; contactName: string; contactNumber: string; address?: string |
null; is_primary?: number; // 0 or 1 createdAt?: string; updatedAt?: string; } export
interface UserContactResponse { success: boolean; message: string; data: UserContact |
UserContact[]; }
```
Copy

Axios Interceptors

Request Interceptor

```
// Automatically add Authorization header to requests
api.interceptors.request.use(async config => { const token = await getAuthToken(); if
(token) { config.headers.Authorization = `Bearer ${token}`; } return config; });
```
Copy

Response Interceptor

```
// Handle 401 Unauthorized responses api.interceptors.response.use( response =>
response, async (error: AxiosError) => { if (error.response?.status === 401) { await
clearSession(); // Auto-logout on unauthorized } throw error; } );
```
Copy

Authentication APIs

POST `/loginWithMobile`

Initiates OTP-based login by sending OTP to the provided mobile number.

Request Body:

```
{ "mobile": "9876543210" // 10-digit Indian mobile }
```

Copy

```
export const loginWithMobile = (mobile: string) =>
api.post<LoginWithMobileResponse>('/loginWithMobile', { mobile });
```

Copy

POST `/verifyOtpAndLogin`

Verifies OTP and logs in the user. Automatically saves session to storage.

Request Body:

```
{ "mobile": "9876543210", "otp": "123456" // Received OTP }
```

Copy

```
export const verifyOtpAndLogin = async (mobile: string, otp: string):
Promise<UserSession> => { const response = await api.post<VerifyOtpResponse>
('/verifyOtpAndLogin', { mobile, otp }); const { token, user } = response.data;
await saveSession(token, user); // Auto-save session return user; };
```

Copy

LOGOUT

Local Session Clear

Clears all authentication and user data from local storage.

```
export const logout = async () => { await clearSession(); };
```

Copy

Rider Profile APIs

POST `/createRiderProfile`

Creates a new rider profile with multipart form data support for image upload.

> **Content-Type:** multipart/form-data

```
export const createRiderProfile = async (formData: FormData): Promise<RiderProfile> => { const response = await api.post<CreateRiderProfileResponse>('/createRiderProfile', formData, { headers: { 'Content-Type': 'multipart/form-data' }, }); if (!response.data.success || !response.data.data?.[0]) { throw new Error(response.data.message || 'Failed to create profile'); } const profile = response.data.data[0]; await saveRiderProfile(profile); // Auto-save to storage return profile; };
```

Copy

PUT `/updateRiderProfile/{userId}`

Updates an existing rider profile. Includes comprehensive error handling and fallback mechanisms.

```
export const updateRiderProfile = async (userId: number, formData: FormData): Promise<RiderProfile> => { const response = await api.put<CreateRiderProfileResponse>( `/updateRiderProfile/${userId}`, formData, { headers: { 'Content-Type': 'multipart/form-data' }, } ); if (!response.data.success) { throw new Error(response.data.message || 'Failed to update profile'); } // Fallback: If no data returned, fetch fresh profile if (!response.data.data || response.data.data.length === 0) { const freshResponse = await getRiderProfileById(userId); if (freshResponse.data.success && freshResponse.data.data.results?.length > 0) { const transformedProfile = transformRiderProfileResult(freshResponse.data.data.results[0]); await saveRiderProfile(transformedProfile); return transformedProfile; } } const updatedProfile = response.data.data[0]; await saveRiderProfile(updatedProfile); return updatedProfile; };
```

Copy

GET `/getRiderProfileById/{userId}`

Retrieves rider profile by user ID.

```
export const getRiderProfileById = (userId: number) => api.get<RiderProfileDetailResponse>(`/getRiderProfileById/${userId}`);
```

Copy

GET `/getAllRiderProfiles`

Retrieves paginated list of all rider profiles (admin function).

```
export const getAllRiderProfiles = (page: number, limit: number) => api.get<GetAllRiderProfilesResponse>('/getAllRiderProfiles', { params: { page, limit } });
```

Copy

Profile Transformation Helper

```
export const transformRiderProfileResult = (result: RiderProfileResult): RiderProfile
=> ({ profile_id: result.id, fullName: result.fullName || '', gender: result.gender ||
'', dob: result.date_of_birth || '', city: result.city || '', profileImage:
result.profileImage || undefined, email: result.User?.email || '', mobile:
result.User?.mobile || '', userType: result.User?.userType || '', status:
result.User?.status || '', createdAt: result.createdAt || '', });
```

Copy

Local Storage Helpers

```
// Save rider profile to local storage export const saveRiderProfile = async (profile:
RiderProfile) => { if (!profile) { await AsyncStorage.removeItem(RIDER_PROFILE_KEY);
return; } await AsyncStorage.setItem(RIDER_PROFILE_KEY, JSON.stringify(profile)); };
// Get rider profile from local storage export const getRiderProfile = async ():
Promise<RiderProfile | null> => { const data = await
AsyncStorage.getItem(RIDER_PROFILE_KEY); if (!data) return null; try { return
JSON.parse(data) as RiderProfile; } catch { return null; } }; // Business rule: Check
mandatory profile data export const hasMandatoryProfileData = (profile: RiderProfile |
null) => { if (!profile) return false; return !!profile.fullName && !!profile.gender
&& !!profile.city && !!profile.mobile; };
```

Copy

User Contacts APIs

Emergency contact management with automatic local storage synchronization.

POST `/createUserContact`

Creates a new emergency contact for the user.

Request Parameters:

`number`	userId	REQUIRED	User ID
`string`	contactType	REQUIRED	Type of contact

`string`	contactName	REQUIRED	Contact person name
`string`	contactNumber	REQUIRED	10-digit mobile number
`number`	is_primary	REQUIRED	0 or 1
`string`	relationship	OPTIONAL	Relationship type

```
export const createUserContact = async ( userId: number, contactType: string, contactName: string, contactNumber: string, is_primary: number, relationship: string ): Promise<UserContact> => { const payload = { userId, contactType, contactName, contactNumber, is_primary, relationship }; const response = await api.post('/createUserContact', payload); if (response.data.success && response.data.data) { const newContact = response.data.data; // Update local storage const existingContacts = await getUserContacts(); const updatedContacts = [...existingContacts, newContact]; await saveUserContacts(updatedContacts); return newContact; } else { throw new Error(response.data.message || 'Failed to create contact'); } };
```

Copy

GET `/getAllUserContacts`

Retrieves all emergency contacts for the current user.

> **Feature:** Automatic fallback to local storage if API fails

```
export const getAllUserContacts = async (): Promise<UserContact[]> => { try { const response = await api.get('/getAllUserContacts'); if (response.data.success && Array.isArray(response.data.data)) { const contacts = response.data.data; // Save to local storage await saveUserContacts(contacts); return contacts; } else { throw new Error(response.data.message || 'Failed to fetch contacts'); } } catch (error: any) { // Return local contacts as fallback try { const
```

Copy

```
localContacts = await getUserContacts(); return localContacts; } catch (localError) { throw error; // Re-throw original error if local fails } } };
```

PUT `/updateUserContact/{id}`

Updates an existing emergency contact.

```
export const updateUserContact = async ( id: number, data: { relationship?: string; address?: string; is_primary?: number; contactName?: string; contactNumber?: string; } ): Promise<UserContact> => { const response = await api.put(`/updateUserContact/${id}`, data); if (response.data.success && response.data.data) { const updatedContact = response.data.data; // Update local storage const existingContacts = await getUserContacts(); const updatedContacts = existingContacts.map(contact => contact.id === id ? updatedContact : contact ); await saveUserContacts(updatedContacts); return updatedContact; } else { throw new Error(response.data.message || 'Failed to update contact'); } };
```
Copy

DELETE `/deleteUserContact/{id}`

Deletes an emergency contact.

```
export const deleteUserContact = async (id: number): Promise<{ success: boolean; message: string }> => { const response = await api.delete(`/deleteUserContact/${id}`); if (response.data.success) { // Update local storage const existingContacts = await getUserContacts(); const updatedContacts = existingContacts.filter(contact => contact.id !== id); await saveUserContacts(updatedContacts); return { success: true, message: 'Contact deleted successfully' }; } else { throw new Error(response.data.message || 'Failed to delete contact'); } };
```
Copy

Specialized Contact Functions

```
// Sync contacts from server to local storage export const syncUserContacts = async (): Promise<UserContact[]> => { try { const contacts = await getAllUserContacts(); await saveUserContacts(contacts); return contacts; } catch (error) { // Return local contacts as fallback return await getUserContacts(); } }; // Get only emergency contacts export const getEmergencyContacts = async (): Promise<UserContact[]> => { const allContacts = await getAllUserContacts(); return allContacts.filter(contact => contact.contactType === 'emergency' ); }; // Get primary contact export const getPrimaryContact = async (): Promise<UserContact | null> => { const allContacts = await getAllUserContacts(); const primary = allContacts.find(contact => contact.is_primary === 1); return primary || null; };
```

Copy

Error Handling Strategy

1. **API Request**
   All API calls include proper error boundaries
2. **Response Validation**
   Check `response.data.success` flag
3. **Error Throwing**
   Throw meaningful error messages from API response
4. **Local Fallback**
   Use local storage data when API fails
5. **401 Handling**
   Auto-logout on unauthorized responses

```
// Example of comprehensive error handling try { const profile = await updateRiderProfile(userId, formData); return profile; } catch (error: any) { console.error('Profile update failed:', { userId, error: error.message, response: error.response?.data }); // Provide user-friendly error message if (error.response?.data?.message) { throw new Error(error.response.data.message); } throw new Error('Failed to update profile. Please try again.'); }
```

Copy

Usage Examples

Complete Authentication Flow

```
import { loginWithMobile, verifyOtpAndLogin, logout, getUserSession } from './api'; // 1. Send OTP const sendOtp = async () => { try { const response = await loginWithMobile('9876543210'); if (response.data.success) { console.log('OTP sent successfully'); } } catch (error) { console.error('Failed to send OTP:', error); } }; // 2. Verify OTP and login const login = async () => { try { const user = await verifyOtpAndLogin('9876543210', '123456'); console.log('Logged in as:', user); return user; } catch (error) { console.error('Login failed:', error); throw error; } }; // 3. Check session const checkAuth = async () => { const session = await getUserSession(); if (session) { console.log('User is logged in:', session); return true; } return false; }; // 4. Logout const handleLogout = async () => { await logout(); console.log('User logged out'); };
```

Copy

Profile Management Example

```
import { getRiderProfile, updateRiderProfile, hasMandatoryProfileData } from './api'; // Load and check profile const loadProfile = async () => { try { const profile = await getRiderProfile(); if (!profile) { console.log('No profile found'); return; } if (hasMandatoryProfileData(profile)) { console.log('Profile is complete'); } else { console.log('Profile missing mandatory data'); } return profile; } catch (error) { console.error('Failed to load profile:', error); } }; // Update profile with image const updateProfile = async (userId: number) => { const formData = new FormData(); formData.append('fullName', 'John Doe'); formData.append('gender', 'Male'); formData.append('city', 'Mumbai'); formData.append('date_of_birth', '1990-01-01');
```

Copy

```
formData.append('email', 'john@example.com'); // Add profile image if available if (profileImage) { formData.append('profileImage', { uri: profileImage.uri, type: 'image/jpeg', name: 'profile.jpg', }); } try { const updatedProfile = await updateRiderProfile(userId, formData); console.log('Profile updated:', updatedProfile); return updatedProfile; } catch (error) { console.error('Update failed:', error); throw error; } };
```

Contacts Management Example

Copy

```
import { getAllUserContacts, createUserContact, getEmergencyContacts, getPrimaryContact, syncUserContacts } from './api'; // Load all contacts const loadContacts = async () => { try { const contacts = await getAllUserContacts(); console.log('Total contacts:', contacts.length); return contacts; } catch (error) { console.error('Failed to load contacts:', error); return []; } }; // Create emergency contact const addEmergencyContact = async (userId: number) => { try { const contact = await createUserContact( userId, 'emergency', 'Jane Doe', '9876543211', 1, // is_primary 'Spouse' ); console.log('Contact added:', contact); return contact; } catch (error) { console.error('Failed to add contact:', error); throw error; } }; // Get only emergency contacts const loadEmergencyContacts = async () => { const emergencyContacts = await getEmergencyContacts(); console.log('Emergency contacts:', emergencyContacts); return emergencyContacts; }; // Sync and refresh contacts const refreshContacts = async () => { try { const contacts = await syncUserContacts(); console.log('Contacts synced:', contacts.length); return contacts; } catch (error) { console.error('Sync failed:', error); // Return local contacts as fallback return await getAllUserContacts(); } };
```

Best Practices & Guidelines

Error Handling

- Always wrap API calls in try-catch blocks
- Use the built-in error messages from API responses
- Implement fallback to local storage where appropriate
- Log errors with context for debugging

Performance Optimization

- Use local storage caching to reduce API calls
- Implement debouncing for search operations
- Use FormData for file uploads to avoid JSON parsing issues
- Batch operations where possible

Security Considerations

- Never store sensitive data in AsyncStorage
- Validate all user inputs before sending to API
- Implement proper session timeout handling
- Use HTTPS for all API communications

Testing Guidelines

```
// Example test structure describe('API Service', () => { beforeEach(() => { // Clear storage before each test AsyncStorage.clear(); }); test('should save and retrieve session', async () => { const mockUser = { id: 1, mobile: '9876543210', userType: 'Rider' }; await saveSession('mock-token', mockUser); const session = await getUserSession(); expect(session).toEqual(mockUser); }); test('should handle API errors gracefully', async () => { // Mock axios to return error jest.spyOn(api, 'post').mockRejectedValue(new Error('Network Error')); await expect(loginWithMobile('invalid')).rejects.toThrow(); }); });
```

Copy

Troubleshooting

Issue	Possible Cause	Solution
401 Unauthorized errors	Expired or invalid JWT token	Call `logout()` and re-authenticate

Issue	Possible Cause	Solution
FormData upload failures	Incorrect Content-Type header	Ensure `multipart/form-data` is set
Slow API responses	Network issues or server load	Implement timeout handling and retries
Storage data corruption	Invalid JSON in AsyncStorage	Clear storage with `clearSession()`
Profile image not updating	Image too large or wrong format	Compress image before upload