FishingBuddy Technical Documentation

About:

Fishing Buddy is a React Native mobile application designed to enhance and simplify the fishing experience for recreational anglers. It provides a complete end-to-end solution from trip planning to catch logging, built with a strong focus on personalization, safety, and ease of use.

The app integrates multiple features including:

- Real-time weather data
- Interactive map-based fishing spots
- Catch history logging
- A smart trip assistant
- Secure user profiles with Firebase Authentication
- Scalable backend storage via Supabase

Tech Stack:

The techstack used includes:

Core Frameworks

- React Native (v0.76.9) for cross-platform mobile app development.

Authentication & Backend

- Firebase Authentication for user authentication.
- Supabase (@supabase/supabase-js) for scalable cloud database and profile storage.
- Formspree for handling email-based OTP verification.

State & Storage

- React Context API for managing global state (profile).
- AsyncStorage for local persistence (favorites, weather cache, checklist items).

Maps & Location

- react-native-maps for displaying interactive maps.
- expo-location for real-time user location tracking and updates.

Weather & Data

- OpenWeatherMap API for real-time weather data.
- Caching weather data locally using AsyncStorage for optimization.

Testing

- Jest, Testing Library, React Test Renderer for unit and component testing.

Folder Structure

```
src/
├── __mocks__          # Test mocks
├── assets             # Static assets like images, icons, and animations
├── common             # Common utilities and reusable logic
├── components         # Reusable UI components
├── constants          # App-wide constants (e.g., screen names)
├── contexts           # React Contexts (e.g., ProfileContext)
├── data               # Static mock data
├── navigation         # Navigation setup (Stacks, Navigators)
├── screens            # Screen folders for each feature
│   ├── AppInformation
│   ├── CatchHistory
│   ├── Dashboard
│   ├── HamburgerMenu
│   ├── HelpfulLinks
│   ├── LicenseUpload
│   ├── LogCatch
│   ├── Login
│   ├── Map
│   ├── OTPVerification
│   ├── Profile
│   ├── ProfileSetup
│   ├── ResetPassword
│   ├── TripAssistant
│   ├── Welcome
├── services           # External service integrations (Firebase,
Supabase, Weather)
├── test-utils         # Testing utilities
├── theme              # Color themes and typography
├── types              # TypeScript types
└── utils              # Helper functions (encryption, distance
calculation, etc.)
```

Authentication:

Firebase is used for managing authentication securely.

Initialization:

```
import { initializeApp, getApps, getApp } from "firebase/app";
import { getAuth } from "firebase/auth";

const firebaseConfig = { /* firebase keys */ };

const app = getApps().length === 0 ? initializeApp(firebaseConfig) :
getApp();
// This ensures only one Firebase instance exists throughout the app
lifecycle.
export const auth = getAuth(app);
```

For 2FA:

Formspree is used to send OTP to the user's email

```
export async function sendEmailWithOtp(email: string, otp: string) {
  const formData = new FormData();
  formData.append("email", email);
  formData.append("message", `Your Fishing Buddy OTP is: ${otp}`);

  try {
    const response = await fetch("https://formspree.io/f/xkgrnygd", {
      method: "POST",
      body: formData,
      headers: {
        Accept: "application/json",
      },
    });
    const result = await response.json();
    if (response.ok) {
      console.log("OTP sent", result);
    } else {
      console.error("Failed to send OTP", result);
    }
  } catch (error) {
    console.error("Error sending OTP", error);
  }
}
```

Database:

Supabase is used as the backend to store encrypted user profile data.
Connection:

```
import { createClient } from "@supabase/supabase-js";

export const supabase = createClient(SUPABASE_URL, SUPABASE_ANON_KEY);
```

Profile Schema:

```
CREATE TABLE profiles (
  id UUID PRIMARY KEY,
  email TEXT,
  encrypted_data TEXT
);
```

Saving a profile with supabase:

```
await supabase.from("profiles").upsert([
  {
    id: user.uid,
    email: profile.email,
    encrypted_data: encryptedData,
  },
]);
```

Loading a Profile:

```
const { data } = await supabase
  .from("profiles")
  .select("*")
  .eq("id", user.uid)
  .maybeSingle();

const decryptedProfile = decryptData(data.encrypted_data);
```

Supabase + Firebase Identity Sync:

```
const user = await waitForAuthUser(); // Ensures auth is ready
await supabase.from("profiles").upsert([{ id: user.uid, ... }])
```

- Ties Firebase Auth identity with Supabase backend.
- Ensures single source of truth for user profile.

Encryption:

AES Encryption (using `crypto-js`) is used to encrypt all profile data before storage.

```
export const encryptData = (data: any): string => {
  const stringified = JSON.stringify(data);
  return CryptoJS.AES.encrypt(stringified, SECRET_KEY).toString();
};
```

```
export const decryptData = (cipherText: string): any => {
  const bytes = CryptoJS.AES.decrypt(cipherText, SECRET_KEY);
  const decryptedString = bytes.toString(CryptoJS.enc.Utf8);
  return JSON.parse(decryptedString);
};
```

Sensitive Information Encrypted:

- Personal Details (name, email, photo, location)
- Preferences
- Location
- License image reference
- Experience data

```
 profile: {
      name: string;
      email: string;
      password: string;
      age?: number;
      photo?: string | null;
      location?: {
        latitude: number;
        longitude: number;
      } | null;
      licenseImage: string | null;
      preferences: {
        level: string | null;
        fishSpecies: string[];
        fishingTypes: string[];
        gear: string[];
        desiredCatch: string[];
      };
    };
```

Weather caching:

Optimized weather fetching using AsyncStorage:

Weather Data Types:

```
type WeatherData = {
  description: string;
  temperature: number;
  windSpeed: number;
};

type CachedWeather = {
  timestamp: number;
  data: WeatherData;
};
```

Fetch Logic:

- Cache lifespan: 30 minutes.
- On cache hit: use stored data.
- On cache miss: fetch from OpenWeatherMap and update cache.

Fetching with cache:

```
export async function getWeatherForCoordinatesWithCache(lat, lon, spotId) {
... }
```

Log A Catch:

- Allows users to document individual fish captures.
- Capture images via camera or upload from the gallery.
- Catches are saved in AsyncStorage.
- Validation ensures minimum input before saving.

```
type CatchEntry = {
  image: string;
  fishType: string;
  size: string;
  timestamp: string;
  location: string;
};
```

Catch History

- Displays all previously logged catches.
- Fetches catch entries from AsyncStorage.

Map Screen – View Fishing Spots

- Real-time user location and displays nearby fishing spots within a 20km radius.
- Users can mark/unmark spots as favorites (saved to AsyncStorage).

Distance Calculation (Haversine Formula) based on longitude and latitudes of two spots:

```
export const getDistanceInKm = (lat1, lon1, lat2, lon2) => {
  const toRad = (val) => (val * Math.PI) / 180;
  const R = 6371;
  const dLat = toRad(lat2 - lat1);
  const dLon = toRad(lon2 - lon1);
  const a = Math.sin(dLat/2) ** 2 + Math.cos(toRad(lat1)) *
Math.cos(toRad(lat2)) * Math.sin(dLon/2) ** 2;
  const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
  return R * c;
};
```

Trip Assistant

Checklist

- Pre-filled default items (fishing gear, license, safety gear).
- Users can add/remove custom items.
- Local storage via AsyncStorage.
- Gamified with visual progress bar

Select location screen

- Reuses MapScreen logic.
- Highlights nearby spots, favorites, and selected location with custom colors.

Weather Check

- Pulls live weather via OpenWeatherMap API.
- Uses getWeatherForCoordinatesWithCache() to avoid redundant API calls.
- Determines whether weather is "amazing" based on wind and temperature.
- Shows user feedback on whether it's a good day to fish.

End Time Setting

- Optional date/time picker for estimated return time.
- Saves this to AsyncStorage.
- Triggers background polling to check and alert emergency contact if user doesn't return (checkTripEndAndAlert).

- Emergency contact is also stored in the async storage

checkTripEndAndAlert()

- Runs periodically every 60 minutes
- Compares current time with saved end time
- If expired, opens SMS prefilled with message to emergency contact.