Sahib Update

How Qibla Calculation & Compass Work

---

1. Qibla Direction Calculation

Library Used

- adhan — JavaScript library for Islamic calculations

```
export const findNearestCity = async (
  latitude: number,
  longitude: number,
): Promise<NearestCityResult> => {
  const nearest = await db.query.cities.findFirst({
    columns: { id: true, name: true },
    orderBy: sql`abs(${cities.latitude} - ${latitude}) + abs(${cities.longitude} -
${longitude})`,
  });

  return nearest || null;
};
```

How It Works

```
const coordinates = new Coordinates(latitude, longitude);
const qiblaAngle = Qibla(coordinates); // Returns angle in degrees (0-360)
```

2. Device Heading (Compass)

Library Used

- expo-location — Expo’s location services

How It Works

```
Location.watchHeadingAsync((heading) => {
  const headingValue = heading.trueHeading > 0
    ? heading.trueHeading    // True North (GPS-based)
    : heading.magHeading;    // Magnetic North (compass-based)

  setDeviceHeading(headingValue); // 0-360 degrees
});
```

- Listens to the device’s magnetometer/compass
- Returns the device’s heading (0–360°)
- trueHeading: GPS-based (more accurate, requires GPS)
- magHeading: magnetic compass (works without GPS)

3. Compass Rotation Formula

```
targetRotation = -deviceHeading

currentNormalized = normalizeAngle(currentRotation)  // 0-360
targetNormalized = normalizeAngle(targetRotation)    // 0-360

diff = targetNormalized - currentNormalized
if (diff > 180) diff -= 360   // shortening the movement speed
if (diff < -180) diff += 360  // shortening the movement speed

finalTarget = currentRotation + diff
```

Why negative?

- When device points North (0°), compass should show North at top (0° rotation)
- When device rotates 90° East, compass rotates -90° to keep North at top

4. Visual Display

Compass Circle

- Rotates with device movement
- Formula: rotate(-deviceHeading)
- Keeps North at the top visually

Qibla Arrow

- Stays fixed, pointing to qibla direction
- Formula: rotation = qiblaDirection (e.g., 58°)
- Shows the direction to Mecca relative to the compass

Design patterns in prayer-times service

---

1. Template Method Pattern

What it is: The base class defines the algorithm steps, and subclasses fill in the specific parts.
How it works here:

```
export interface PrayerTimeInput {
  coordinates: Coordinates;
  date?: Date;
```

```
  timezone: string;
  format?: string;
  adjustments?: PrayerAdjustments;
}
export interface PrayerTimesResult {
  fajr: Date;
  sunrise: Date;
  dhuhr: Date;
  asr: Date;
  maghrib: Date;
  isha: Date;
  formatted?: FormattedPrayerTimes;
}

export interface FormattedPrayerTimes {
  fajr: string;
  sunrise: string;
  dhuhr: string;
  asr: string;
  maghrib: string;
  isha: string;
}
// Base class defines the template
abstract class PrayerTimeCalculator {
  public calculate(input: PrayerTimeInput): PrayerTimesResult {
    // Step 1: Prepare (subclass implements)
    const params = this.prepareCalculationParameters(input);

    // Step 2: Apply adjustments (subclass implements)
    const adjustedParams = this.applyAdjustments(params, input.adjustments);

    // Step 3: Calculate (subclass implements)
    const rawTimes = this.calculateRawPrayerTimes(input, adjustedParams);

    // Step 4 & 5: Common steps (base class handles)
    const timezoneAware = this.convertToTimezone(rawTimes, input.timezone);
    const formatted = this.formatPrayerTimes(timezoneAware, input.timezone);

    return { ...timezoneAware, formatted };
  }
```

```
}
```

Analogy: A recipe template where steps 1–3 are fixed, and steps vary by method

2. Adapter Pattern

What it is: Translates one interface to another so different systems can work together.

Example:

- AdhanAdapter wraps the Adhan library to match the generic interface.

```
/**
 * Adapter for Adhan library
 * This adapter handles all Adhan-specific logic, making the calculator generic
 */
export class AdhanAdapter {
  /**
   * Create Adhan Coordinates from generic coordinates
   */
  static createCoordinates(input: PrayerTimeInput): AdhanCoordinates {
    return new AdhanCoordinates(input.coordinates.latitude, input.coordinates.longitude);
  }



  /**
   * Calculate prayer times using Adhan library
   */
  static calculatePrayerTimes(
    input: PrayerTimeInput,
    params: CalculationParameters,
  ): RawPrayerTimes {
    const coordinates = AdhanAdapter.createCoordinates(input);
    const date = input.date || new Date();
    const prayerTimes = new AdhanPrayerTimes(coordinates, date, params);

    return {
      fajr: prayerTimes.fajr,
      sunrise: prayerTimes.sunrise,
      dhuhr: prayerTimes.dhuhr,
      asr: prayerTimes.asr,
      maghrib: prayerTimes.maghrib,
```

```
      isha: prayerTimes.isha,
    };
  }
}
```

3. Facade Pattern

What it is: A simple interface that hides complexity behind the scenes.

```
/**
 * Factory function to create a calculator instance based on method name
 * @param methodName - Name of the calculation method (enum or 'Dummy' string)
 * @returns Calculator instance
 */
export const createCalculator = (methodName: CalculationMethodName | 'Dummy'):
PrayerTimeCalculator => {
  switch (methodName) {
    case CalculationMethodName.MuslimWorldLeague:
      return new MuslimWorldLeagueCalculator();
    case CalculationMethodName.Egyptian:
      return new EgyptianCalculator();
      //....
    default:
      throw new Error(`Unknown calculation method: ${methodName}`);
  }
};

/**
 * Convenience function to calculate prayer times
 * @param methodName - Name of the calculation method
 * @param input - Prayer time calculation input
 * @returns Prayer times result
 */
export const calculatePrayerTimes = (
  methodName: CalculationMethodName,
  input: PrayerTimeInput,
): PrayerTimesResult => {
  const calculator = createCalculator(methodName);
```

```
  return calculator.calculate(input);
};
```

Usage

```
export enum CalculationMethodName {
  MuslimWorldLeague = 'MuslimWorldLeague',
  Egyptian = 'Egyptian',
  Karachi = 'Karachi',
  UmmAlQura = 'UmmAlQura',
  Dubai = 'Dubai',
  MoonsightingCommittee = 'MoonsightingCommittee',
  NorthAmerica = 'NorthAmerica',
  Kuwait = 'Kuwait',
  Qatar = 'Qatar',
  Singapore = 'Singapore',
  Tehran = 'Tehran',
  Turkey = 'Turkey',
}
const prayerTimesResult = calculatePrayerTimes(CalculationMethodName.MuslimWorldLeague, {
        coordinates: {
          latitude: state.latitude,
          longitude: state.longitude,
        },
        date: new Date(),
        timezone: state.timezone,
        format: 'h:mm A',
        adjustments: state.prayerAdjustments,
      });
```

Data Storage

I migrated from WatermelonDB

migrated from WatermelonDB to Drizzle ORM because:

1. Prepopulated database support: WatermelonDB doesn't support prepopulated databases, which we need for cities data.
2. Migration limitations: No built-in support for renaming or deleting columns, even using raw sql is pain.
3. API extensibility: Hard to extend or customize.
4. Documentation: Poor and outdated.

Drizzle ORM addresses these and works well with Expo SQLite.

Drizzle Features

1. Database instance (drizzle)

Feature: Creates a Drizzle ORM instance wrapping SQLite

```
import { drizzle } from 'drizzle-orm/expo-sqlite';
import { openDatabaseSync } from 'expo-sqlite';
import * as schema from './schemas';

const expoDb = openDatabaseSync('sahib.db', {
  enableChangeListener: true,
});

export const db = drizzle(expoDb, { schema });
```

Usage: Use db for all database operations.

2. Table definition (sqliteTable)

Feature: Defines SQLite tables with type-safe columns

```
import { integer, real, sqliteTable, text } from 'drizzle-orm/sqlite-core';

export const cities = sqliteTable('cities', {
  id: integer('id').primaryKey({ autoIncrement: true }),
  name: text('name').notNull(),
  latitude: real('latitude').notNull(),
```

```
  longitude: real('longitude').notNull(),
});
```

Column types used:

- integer() - Integer numbers
- text() - Text strings
- real() - Floating point numbers

3. Query builder API (db.query)

Feature: Type-safe query builder with relation support

```
// Query
const cityWithCountry = await db.query.cities.findFirst({
  where: (cityTable, { eq }) => eq(cityTable.id, cityId),
  columns: {
    name: true,
    latitude: true,
  },
  with: {
    country: {
      columns: {
        name: true,
        timezones: true,
      },
    },
    state: {
      columns: {
        name: true,
      },
    },
  },
});
```

7. Raw SQL (sql template tag)

```
import { sql } from 'drizzle-orm';


const results = await db.query.cities.findMany({
```

```
  where: sql`lower(${cities.name}) LIKE lower(${`%${searchQuery}%`})`,
  limit: 40,
});
```

8. Reactive queries (useLiveQuery)

```
import { useLiveQuery } from 'drizzle-orm/expo-sqlite';

function CityList() {
  const { data, updatedAt } = useLiveQuery(
    getCitiesQuery(searchQuery, 40),
    [searchQuery], // Dependencies - re-runs when these change
  );

  // data automatically updates when database changes!
  return (
    <FlatList
      data={data ?? []}
      renderItem={({ item }) => <CityItem city={item} />}
    />
  );
}
```

Features:

- Auto-updates when database changes
- Returns data and updatedAt
- Re-runs when dependencies change

10. Migrations (useMigrations)

```
import { useMigrations } from 'drizzle-orm/expo-sqlite/migrator';
import migrations from '@/src/db/migrations/migrations';

function MigratorComponent() {
  const sqlite = useSQLiteContext();
  const db = drizzle(sqlite, { schema });
```

```
  // Automatically runs pending migrations on app start
  useMigrations(db, migrations);

  return null;
}
```

11. Migration generation (drizzle-kit)

```
# Generate migrations from schema changes
npx drizzle-kit generate

# This creates SQL migration files in src/db/migrations/
```

12. Database studio (useDrizzleStudio)

Feature: Visual database browser (development only)

Expo SQLite Key-Value Storage

---

1. Basic setup

```
import Storage from 'expo-sqlite/kv-store';

Storage.setItemSync('key', 'value');
const value = Storage.getItemSync('key');

// Asynchronous API
await Storage.setItem('key', 'value');
const value = await Storage.getItem('key');
```

2. Wrapper implementation

```ts
import Storage from 'expo-sqlite/kv-store';

export const storage = {
  // ==================== Synchronous API ====================

  setString: (key: string, value: string): void => {
    Storage.setItemSync(key, value);
  },

  getString: (key: string): string | null => {
    return Storage.getItemSync(key);
  },

  setBoolean: (key: string, value: boolean): void => {
    Storage.setItemSync(key, String(value));
  },

  getBoolean: (key: string): boolean | null => {
    const value = Storage.getItemSync(key);
    if (value === null) return null;
    return value === 'true';
  },

  setNumber: (key: string, value: number): void => {
    Storage.setItemSync(key, String(value));
  },

  getNumber: (key: string): number | null => {
    const value = Storage.getItemSync(key);
    if (value === null) return null;
    const num = Number(value);
    return isNaN(num) ? null : num;
  },

  remove: (key: string): void => {
    Storage.removeItemSync(key);
  },

  clear: (): void => {
```

```
    Storage.clearSync();
  },

  // ==================== Asynchronous API ====================

  setStringAsync: async (key: string, value: string): Promise<void> => {
    await Storage.setItem(key, value);
  },

  getStringAsync: async (key: string): Promise<string | null> => {
    return await Storage.getItem(key);
  },

  // ... similar async methods for boolean, number, remove, clear
};
```

How Cities Are Handled

Prepopulated database structure

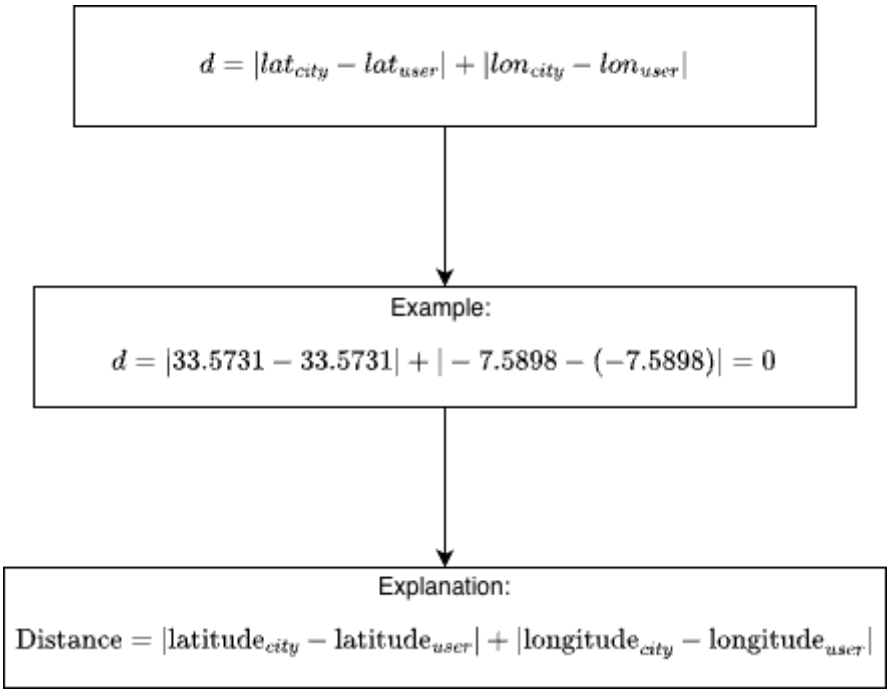
What's prepopulated:

- Cities with coordinates (latitude/longitude)
- Countries with timezones and translations
- States, regions, and subregions
- Relationships between all entities

id	name	state_id	state_code	country_id	country_code	latitude	longitude
1	Andorra la Vella	488	07	6	AD	42.50779	1.52109
2	Arinsal	493	04	6	AD	42.57205	1.48453
3	Canillo	489	02	6	AD	42.5676	1.59756
4	El Tarter	489	02	6	AD	42.57952	1.65362
5	Encamp	487	03	6	AD	42.53474	1.58014
6	Ordino	491	05	6	AD	42.55623	1.53319
7	Pas de la Casa	487	03	6	AD	42.54277	1.73361
8	Sant Julià de Lòr	490	06	6	AD	42.46372	1.49129
9	la Massana	493	04	6	AD	42.54499	1.51483
10	les Escaldes	492	08	6	AD	42.50729	1.53414
11	Ghayathi	3,396	AZ	231	AE	23.9045342	52.5871026
12	Abu Dhabi	3,396	AZ	231	AE	24.41361	54.43295
13	Al Dhaid	3,390	SH	231	AE	25.28812	55.88157
14	Ajman	3,395	AJ	231	AE	25.40328	55.52341
15	Ajman City	3,395	AJ	231	AE	25.40177	55.47878
16	Al Ain City	3,396	AZ	231	AE	24.19167	55.76056
17	Liwa Oasis	3,396	AZ	231	AE	22.8697255	53.2408608
18	Al Batayih	3,390	SH	231	AE	25.22317	55.74272
19	Al Dhafra	3,396	AZ	231	AE	23.65745	53.72225
20	Fujairah	3,393	FU	231	AE	25.11641	56.34141
22	Al Hamriyah	3,390	SH	231	AE	25.46121	55.54813
23	Al Madam	3,390	SH	231	AE	24.95536	55.7682
25	Ruwais	3,396	AZ	231	AE	24.11028	52.73056
26	Bani Yas City	3,396	AZ	231	AE	24.30978	54.62944
28	Dibba Al Fujairah	3,393	FU	231	AE	25.5858	56.24792
30	Dibba Al-Fujairah	3,393	FU	231	AE	25.59246	56.26176
31	Dibba Al-Hisn	3,393	FU	231	AE	25.61955	56.27291
32	Dubai	3,391	DU	231	AE	25.0657	55.17128
33	Kalba	3,390	SH	231	AE	24.99816	56.27207
34	Khalifa City	3,396	AZ	231	AE	24.42588	54.605
36	Khor Fakkan	3,390	SH	231	AE	25.33966	56.3028
37	Manama	3,395	AJ	231	AE	25.32568	56.00259
38	Masfut	3,395	AJ	231	AE	24.83982	56.05158
39	Milehah	3,390	SH	231	AE	25.10097	55.91282
40	Murbaḩ	3,390	SH	231	AE	25.27623	56.36256
41	Mussafah	3,396	AZ	231	AE	24.35893	54.48267
42	Muzayri‘	3,396	AZ	231	AE	23.14355	53.7881
43	Ras Al Khaimah	3,394	RK	231	AE	25.46116	56.04058
46	Sharjah	3,390	SH	231	AE	25.33737	55.41206

Nearest city detection

Formula:




```
export const findNearestCity = async (
  latitude: number,
  longitude: number,
): Promise<NearestCityResult> => {
  const nearest = await db.query.cities.findFirst({
    columns: { id: true, name: true },
    orderBy: sql`abs(${cities.latitude} - ${latitude}) + abs(${cities.longitude} -
${longitude})`,
  });

  return nearest || null;
};
```

Sahib Data Model

not finished

Revision #14
Created 2026-01-23 20:20:32 UTC by EL MAHDI
Updated 2026-01-24 14:00:11 UTC by EL MAHDI