Explication détaillée de l'architecture et du fonctionnement de l'application

1. Structure générale du composant

L'intégralité de l'application est portée par un **composant fonctionnel** `App` (fonction JavaScript) qui se charge de :

- Importer les **bibliothèques React Native** (`Text` , `View` , `ScrollView` , etc.) et le module d'icônes (`MaterialCommunityIcons`).
- Définir les **hooks React** pour gérer l'état local.
- Appeler la fonction **`fetchData`** au montage pour récupérer les données météo initiales.
- Construire et retourner un **arbre JSX** qui représente l'interface utilisateur.

L'ensemble est enveloppé dans un `<SafeAreaView>` pour respecter les zones non couvertes (notch, barre d'état), suivi d'un `<StatusBar>` configuré, d'un **header**, puis du contenu principal dans un `<ScrollView>` .

---

2. Les hooks React utilisés

2.1 **`useState`**

À l'ouverture du composant, on appelle successivement :

```
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);
const [current, setCurrent] = useState(null);
const [dailyData, setDailyData] = useState([]);
const [searchModalVisible, setSearchModalVisible] = useState(false);
const [searchQuery, setSearchQuery] = useState('');
```

- **`loading`** : indicateur de chargement réseau. Tant que `true` , on affiche un loader (`ActivityIndicator`).
- **`error`** : stocke un message d'erreur éventuel.

- `current` : objet décrivant la météo courante (ville, icône, température, description, humidité, vent).
- `dailyData` : tableau d'objets pour chaque jour de la prévision (jour abrégé, icône, température).
- `searchModalVisible` : booléen qui contrôle l'affichage du modal de saisie de la ville.
- `searchQuery` : chaîne de caractères liée au champ `<TextInput>`.

À chaque appel à `setXxx`, React marque le composant comme "dirty" et déclenche un **re-render**.

2.2 `useEffect`

```
useEffect(() => {
  fetchData('Tizi-Ouzou');
}, []);
```

- Le tableau `[]` en second argument signifie « n'exécute ce code **qu'une seule fois** après le premier rendu ».
- On y place l'appel initial à `fetchData` pour charger la météo de Tizi-Ouzou.

---

3. Fonction `fetchData(location)` : récupération et traitement des données

Cette fonction est **asynchrone** (`async`) et suit ce schéma :

1. **Initialisation des états**

```
setLoading(true);
setError(null);
```

2. **Requête sur l'API météo courante**

```
const weatherRes = await fetch(`${BASE_URL}/weather?
q=${location}&units=metric&appid=${API_KEY}`);
const weatherJson = await weatherRes.json();
if (!weatherRes.ok) throw new Error(weatherJson.message);
```

- Retourne un objet JSON avec `name`, `weather[0].icon`, `main.temp`, etc.

3. **Mise à jour de `current`**

```
setCurrent({
 city: weatherJson.name,
 icon: `https://openweathermap.org/img/wn/${...}.png`,
 temp: Math.round(weatherJson.main.temp),
 description: weatherJson.weather[0].description,
 humidity: weatherJson.main.humidity,
 wind: Math.round(weatherJson.wind.speed),
});
```

4. **Requête sur l'API de prévisions 5 jours**

```
const forecastRes = await fetch(`${BASE_URL}/forecast?
q=${location}&units=metric&appid=${API_KEY}`);
const forecastJson = await forecastRes.json();
if (!forecastRes.ok) throw new Error(forecastJson.message);
```

- Renvoie une liste de prévisions toutes les 3 heures.

5. **Extraction des prévisions journalières**

```
const dailyList = forecastJson.list
 .filter(item => item.dt_txt.includes('12:00:00'))
 .slice(0, 7);
setDailyData(
 dailyList.map(item => ({ ... }))
);
```

- On **filtre** pour ne garder que les entrées à midi, une par jour.
- On utilise `.slice(0,7)` pour toujours avoir 7 jours.

6. **Gestion des erreurs et fin de chargement**

```
} catch (err) {
 setError(err.message);
 Alert.alert('Erreur', err.message);
} finally {
 setLoading(false);
}
```

4. Interaction utilisateur et insertion dans l'UI

4.1 Loader et gestion d'erreur

- Si `loading === true`, on retourne un `<ActivityIndicator>` centré.
- Si `error` est non null, on affiche simplement le message d'erreur.

4.2 Header et modal de recherche

- **Header**

```
<View style={styles.header}>
  <TouchableOpacity onPress={() => setSearchModalVisible(true)}>
    <MaterialCommunityIcons name="magnify" ... />
  </TouchableOpacity>
  <Text style={styles.headerTitle}>Météo</Text>
</View>
```

- **Modal**
  - `visible={searchModalVisible}`
  - Contient un `<TextInput>` contrôlé par `searchQuery`.
  - Bouton `<TouchableOpacity>` appelle `handleSearch()`, qui :
    1. Valide la saisie
    2. Lance `fetchData(searchQuery.trim())`
    3. Masque le modal et vide le champ.

4.3 Affichage des données

```
<Text style={styles.city}>{current.city}</Text>
<Image style={styles.weatherIcon} source={{ uri: current.icon }} />
<Text style={styles.temp}>{current.temp} C</Text>
<Text style={styles.description}>{current.description}</Text>
```

- Les **détails** (humidité, vent) sont deux colonnes côte-à-côte.

4.4 Prévisions 7 jours (Scroll horizontal)

```
<ScrollView horizontal showsHorizontalScrollIndicator={false} contentContainerStyle=
{styles.forecastContainer}>
  {dailyData.map((item, idx) => (
    <View key={idx} style={styles.forecastItem}>
      <Text style={styles.forecastDay}>{item.day}</Text>
      <Image source={{uri: item.icon}} style={styles.forecastIcon} />
      <Text style={styles.forecastTemp}>{item.temp}</Text>
    </View>
  ))}
</ScrollView>
```

- L'attribut `horizontal` permet le défilement latéral.
- `contentContainerStyle` positionne chaque carte en `row`.

4.5 Footer date et heure

Affiche la date complète (`jour, jour numérique mois`) et l'heure courante formatées en `fr-FR`.

---

5. Code complet du StyleSheet

```
const styles = StyleSheet.create({
  safeContainer: { flex: 1, backgroundColor: '#e0f7fa' },
  container: { flex: 1, alignItems: 'center', justifyContent: 'center' },
  center: { flex: 1, justifyContent: 'center', alignItems: 'center' },

  header: {
    flexDirection: 'row',
    alignItems: 'center',
    justifyContent: 'center',
    padding: 15,
    backgroundColor: '#b2dfdb',
  },
  headerTitle: { marginLeft: 10, fontSize: 18, fontWeight: 'bold', color: '#00796b' },

  modalOverlay: { flex: 1, backgroundColor: 'rgba(0,121,107,0.5)', justifyContent:
'center', alignItems: 'center' },
  searchModal: { width: '80%', backgroundColor: '#fff', borderRadius: 8, padding: 15
},
  searchInput: { borderWidth: 1, borderColor: '#ddd', borderRadius: 6, padding: 10,
marginBottom: 10 },
  searchButton: { backgroundColor: '#00796b', padding: 12, borderRadius: 6,
alignItems: 'center' },
  searchButtonText: { color: '#fff', fontWeight: '600' },

  scrollContainer: { flexGrow: 1, alignItems: 'center', paddingBottom: 50 },

  city: { fontSize: 32, fontWeight: 'bold', color: '#00796b', marginVertical: 10 },
  weatherIcon: { width: 120, height: 120 },
  temp: { fontSize: 48, fontWeight: 'bold', color: '#d32f2f', marginVertical: 5 },
  description: { fontSize: 24, color: '#616161', marginBottom: 20, textTransform:
'capitalize' },

  additionalInfo: { flexDirection: 'row', justifyContent: 'space-around', width:
'100%', marginVertical: 20 },
  infoBox: { alignItems: 'center', flex: 1 },
  infoLabel: { fontSize: 16, color: '#424242' },
  infoValue: { fontSize: 20, fontWeight: 'bold', color: '#000' },

  forecastTitle: { fontSize: 20, fontWeight: 'bold', color: '#00796b', marginVertical:
10 },
  forecastContainer: { flexDirection: 'row', paddingHorizontal: 10 },
  forecastItem: {
    width: 100,
    height: 140,
    backgroundColor: '#fff',
    borderRadius: 10,
    padding: 10,
    marginHorizontal: 5,
    alignItems: 'center',
    justifyContent: 'center',
```

```
    shadowColor: '#000',
    shadowOffset: { width: 0, height: 2 },
    shadowOpacity: 0.1,
    shadowRadius: 2,
    elevation: 2,
  },
  forecastDay: { fontSize: 18, fontWeight: 'bold', color: '#00796b' },
  forecastIcon: { width: 50, height: 50, marginVertical: 5 },
  forecastTemp: { fontSize: 16, color: '#d32f2f' },

  footer: { alignItems: 'center', marginTop: 20 },
  date: { fontSize: 16, color: '#424242' },
  time: { fontSize: 16, color: '#424242' },
});
```