Task 1

Dynamic Localization Based on User Device Settings in a React Native Spotify Clone

We have already created a spotify app using react native with some functionality, and also in running status.

Based on a given task we have to understand the problems and according to those problems we have to think of basically two types of approach that is given below-

First Approach: In the first method we have to try to solve this with a direct API call.

How It Works -

When using an API for translation, we send a request containing:

- The text to translate (`q` parameter)
- The source and target languages (`langpair` parameter, e.g., `en|gr`)

Example API Request :

`https://api.example.com/translate?q=Hello&langpair=en|hi`

Response(JSON format)

```
{
  "translatedText": "नमस्ते"
}
```

Then display `"नमस्ते"` instead of `"Hello"`.

Why This Approach Has Major Drawbacks

- ❖ API Calls for Every Text = Slow Performance
  - ➢ Every hardcoded text requires a separate API request.
  - ➢ If our app has hundreds of UI elements, making hundreds of API calls will increase loading time.
- ❖ Increased Network Usage & Costs
  - ➢ API calls consume internet bandwidth.
  - ➢ Some translation APIs have rate limits or charge per request, making it expensive for large-scale apps.
- ❖ Offline Mode Not Possible
  - ➢ If the device is offline, translation won't work, causing UI issues.

- ❖ Security Risks
  - ➢ If our API key is exposed, attackers can misuse it, leading to unauthorized usage or higher API bills.

Second Approach: In the second approach we have to use the i18Next and React i18Next library.

We have to use **i18Next** and **react-i18next** to manage translations locally in the app. This approach is efficient for predefined UI text and improves performance by avoiding network requests.

Inspired by localized software development practices (e.g., Windows, macOS, and mobile apps storing language files), i18Next allows applications to load translations from local JSON files, ensuring **offline support** and **faster performance**.

How It Works

1. Store translations in JSON files (e.g., `en.json`, `hi.json`).
2. Detects the user's device language.
3. Use i18Next to load the correct language and update UI text dynamically.

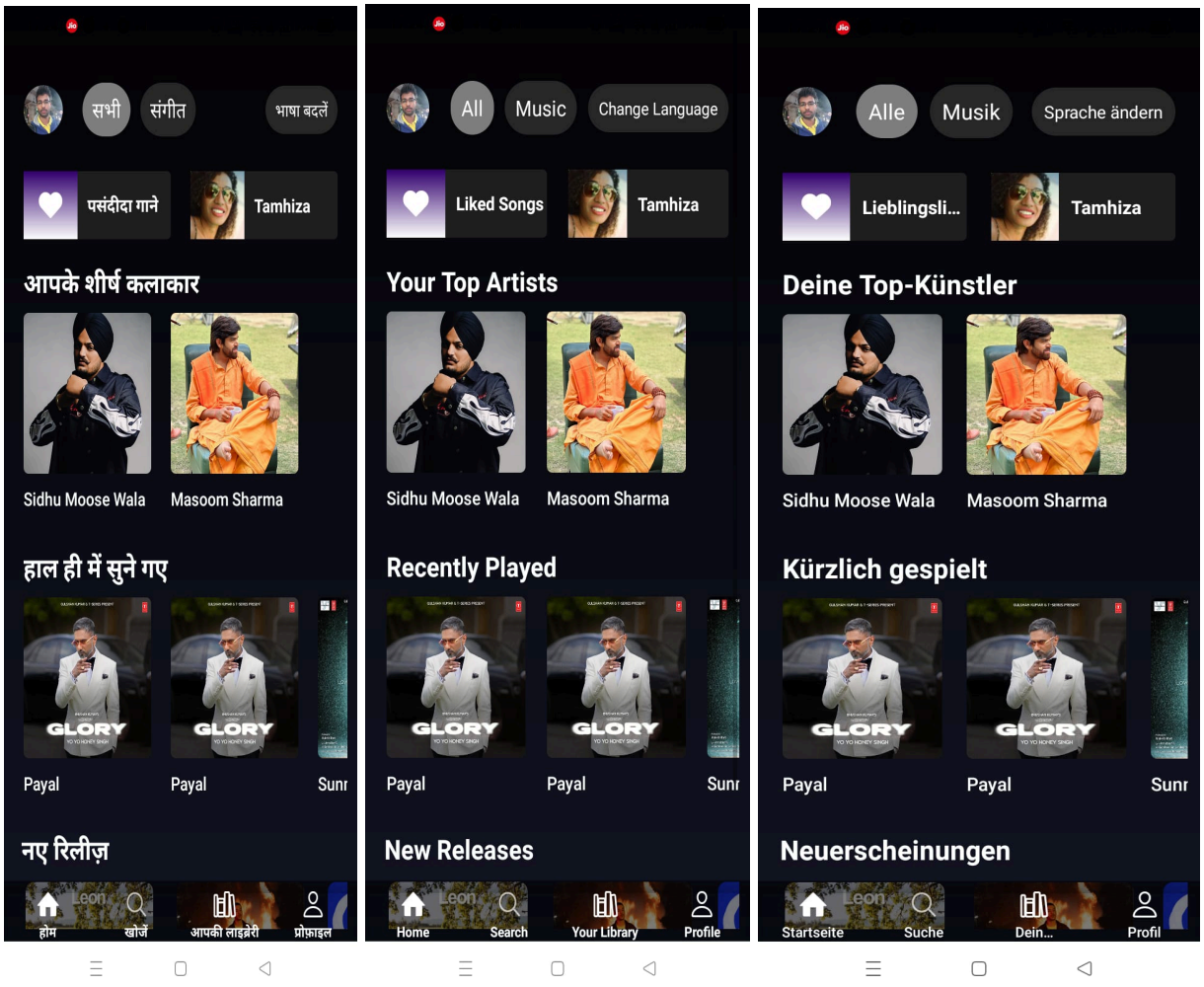
Limitations of Using i18Next for Localization

While i18Next is a great solution for managing translations locally, there are a few limitations to keep in mind:

1. **Limited to Predefined Text** – Since translations are stored in JSON files, it works well for UI text but is not ideal for dynamic content (e.g., user-generated text).
2. **Manual Translation Updates** – Whenever new text needs to be added or modified, translations must be updated manually in all language files.
3. **App Size Increases** – Storing many translations locally can slightly increase the app's size, especially if it supports multiple languages.
4. **No Real-time Updates** – If translations need frequent updates, users must update the app or refresh the cache to see changes.
5. **Pluralization & Formatting Complexity** – Some languages have different rules for plurals and sentence structures. For example:
   - In English: **"1 song"** and **"5 songs"** (just adding "s").
   - In Arabic or Russian: Different words may be used based on the number.

Final Thought

This is **not a bad approach**—in fact, it's **highly efficient** for UI text and offline support. However, if an app requires real-time translations for dynamic content, combining i18Next with an API-based approach can be a good strategy.



Date : 27 Feb 2025

Today, I explored the Spotify project, removed the Context API, and implemented **Redux Toolkit with Redux Persist** for efficient global state management and persistence. Additionally, I improved some functionalities, including language switching using a picker.