TuneIn Project Coding Standards

This document outlines the coding standards and best practices for maintaining code quality, consistency, and scalability across the TuneIn project. By adhering to these standards, we ensure that the project remains readable, maintainable, and efficient.

Table of Contents

- Project Structure
- Coding Standards
  - File Naming
  - Component Structure
  - Hooks and Utility Functions
  - TypeScript Best Practices
  - React Native Best Practices
  - Styling
- Code Quality
  - Linting
  - Prettier Formatting
- Version Control
- Testing
- Resources

Project Structure

The following structure ensures that files and components are well organized and easy to locate:

```
frontend/
├── app/
│   ├── components/
│   ├── hooks/
│   ├── models/
│   ├── screens/
│   │   ├── rooms/
│   │   ├── auth/
│   └── index.tsx
├── my-app.d.ts
├── babel.config.js
├── package.json
```

Folder Descriptions

- **components/**: Reusable UI components such as buttons, cards, and navigation bars.
- **hooks/**: Custom hooks for shared logic.
- **models/**: TypeScript models and data interfaces.
- **screens/**: Different pages of the app, categorized by feature (e.g., rooms, auth).
- **index.tsx**: Entry point for the application.

Coding Standards

File Naming

- Use **PascalCase** for component and screen filenames: `RoomCardWidget.tsx`, `ProfileScreen.tsx`.
- Use **camelCase** for non-component files such as hooks, utility functions, and styles: `useFetchData.ts`, `formatDate.ts`.
- Avoid abbreviations and ensure filenames are descriptive of their purpose.

Component Structure

- **Function Components**: Use functional components over class components.

```
const RoomCard: React.FC = () => {
  return <View>{/* content here */}</View>;
};
```

- **Props Destructuring**: Destructure props at the function level for better readability and performance.

```
const RoomCard = ({ roomName, roomDescription }: RoomCardProps) => {
  return <Text>{roomName}</Text>;
};
```

- **Component Modularity**: Each file should export only one component unless it's a collection of related utilities or types. Keep components focused on a single responsibility.

Hooks and Utility Functions

- **Reusable Hooks**: Place all custom hooks in the `hooks` folder. Follow this pattern:

```
// hooks/useFetchData.ts
import { useState, useEffect } from 'react';

const useFetchData = (url: string) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch(url);
      const result = await response.json();
      setData(result);
      setLoading(false);
    };

    fetchData();
  }, [url]);

  return { data, loading };
};

export default useFetchData;
```

- **Avoid Overuse of Inline Functions**: Avoid inline functions for frequently re-rendering components, as it may degrade performance.

Context Management

- **What is Context?**: In React, context is a way to create global state that can be shared across components without having to pass props manually at every level. It provides a mechanism for components to communicate with each other without the need for complex prop drilling.

- **Purpose of Context**: Context is useful for managing state that needs to be accessible by many components throughout the application, such as user authentication data, themes, or global settings. It simplifies state management in larger applications by allowing data to flow through the component tree seamlessly.

- **Using Context**:

    1. **Create a Context**: Use `createContext()` to create a context object that holds the state and functions.
    2. **Provide the Context**: Wrap components that need access to the context with a `Provider`, which makes the context values available to those components.
    3. **Consume the Context**: Use the `useContext()` hook in functional components to access and update the context values as needed.

    Here's a basic example:

```
import React, { createContext, useContext, useState } from "react";

// Creating a context
const MyContext = createContext<{ count: number; setCount: React.Dispatch<React.SetStateAction<number>> } | undefined>(undefine

const MyProvider: React.FC<{ children: React.ReactNode }> = ({ children }) => {
    const [count, setCount] = useState(0);

    return (
        <MyContext.Provider value={{ count, setCount }}>
            {children}
        </MyContext.Provider>
    );
};

const MyComponent: React.FC = () => {
    const context = useContext(MyContext);
    if (!context) {
        return null; // Handle missing context
    }

    const { count, setCount } = context;

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count + 1)}>Increment</button>
        </div>
    );
};
```

By implementing context, you can effectively manage shared state in your React application, leading to cleaner and more maintainable code

TypeScript Best Practices

- **Explicit Types**: Always use explicit types for function parameters, return values, and hooks.

```
interface RoomProps {
  id: string;
  name: string;
  description: string;
}

const RoomCard: React.FC<RoomProps> = ({ id, name, description }) => {
  return <Text>{name}</Text>;
};
```

- **Enums and Interfaces**: Use interfaces for data structures and enums for predefined sets of values. Avoid using `any`.

```
interface User {
  id: string;
  username: string;
  profilePicture: string;
}

enum Status {
  ACTIVE = 'active',
  INACTIVE = 'inactive'
}
```

React Native Best Practices

- **State Management**: Keep components pure and avoid using local state if unnecessary. Manage state globally with libraries like Redux or React Context when needed.

- **Performance Optimization**: Memoize expensive operations or components with `React.memo` or `useMemo`.

```
const expensiveComputation = useMemo(() => {
  return computeValue();
}, [dependency]);
```

- **Key Prop in Lists**: Always provide a unique `key` prop when rendering lists to improve rendering performance and avoid warnings.

```
return rooms.map((room) => <RoomCard key={room.id} {...room} />);
```

Styling

- **Centralized Styling**: Define theme variables (colors, fonts) in a single file to ensure consistency throughout the app. Use the `styles/colors.ts` file for defining and managing colors across the app to keep styling consistent.

```
// styles/colors.ts
export const colors = {
  primary: '#08BDBD',
  secondary: '#8B8FA8',
  background: '#FFFFFF',
  text: '#000000',
};

// styles/themes.ts
import { StyleSheet } from 'react-native';
import { colors } from './colors';

export const globalStyles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 20,
    backgroundColor: colors.background,
  },
  text: {
    color: colors.text,
  },
});
```

- **Use of Colors**: All color definitions should be imported from the `colors.ts` file to maintain uniformity across the app.

```
import { colors } from '../styles/colors';

const styles = StyleSheet.create({
  button: {
    backgroundColor: colors.primary,
    padding: 10,
    borderRadius: 5,
  },
});
```

- **StyleSheet Creation**: Use `StyleSheet.create()` for all styling, avoid inline styles, and prefer Flexbox for layout.

```
const styles = StyleSheet.create({
  button: {
    backgroundColor: colors.primary,
    padding: 10,
    borderRadius: 5,
  },
});
```

Code Quality

Linting

- **ESLint**: Use ESLint to catch potential errors and enforce code quality. Run linting before any commits.

```
npx eslint --fix 'app/**/*.tsx'
```

Prettier Formatting

- **Prettier**: Use Prettier to enforce consistent code formatting. Configure it to run automatically on save in your code editor.

```
npx prettier --write 'app/**/*.tsx'
```

Version Control

- **Branching**: Use descriptive branch names like `feature/room-card-component` or `fix/user-auth-bug`.
- **Commits**: Write concise and meaningful commit messages that describe the changes clearly.

```
git commit -m "Fix: resolve room page navigation issue"
```

Testing

- **Component Tests**: Ensure all components have tests using Jest and React Testing Library.

```
import { render } from '@testing-library/react-native';
import RoomCard from '../components/RoomCardWidget';

test('renders room card correctly', () => {
  const { getByText } = render(<RoomCard roomName="Music Room" />);
  expect(getByText('Music Room')).toBeTruthy();
});
```

- **Automated Testing**: Set up automated testing to ensure that critical components are always functioning.

Resources

- React Native Documentation (https://reactnative.dev/docs/getting-started)
- Expo Documentation (https://docs.expo.dev/)
- TypeScript Documentation (https://www.typescriptlang.org/docs/)
- ESLint Documentation (https://eslint.org/docs/user-guide/getting-started)
- Prettier Documentation (https://prettier.io/docs/en/)