Creating a **production-grade, highly complex React Native project** involves a thoughtful folder structure for scalability, maintainability, and organization. Below is a recommended folder structure tailored for a high-complexity project:

Root Folder Structure

plaintext
Copy code

```
project-root/
├── android/                  # Android native code
├── ios/                      # iOS native code
├── src/                      # Main source code for the app
│   ├── assets/               # Static assets (images, fonts, icons,
etc.)
│   ├── components/           # Reusable UI components
│   ├── config/               # App configuration files (e.g., themes,
environment variables)
│   ├── constants/            # App-wide constants
│   ├── hooks/                # Custom hooks
│   ├── modules/              # Feature-specific modules
│   │   └── [FeatureName]/    # Individual feature modules
│   │       ├── components/   # Feature-specific components
│   │       ├── screens/      # Screens for the feature
│   │       ├── services/     # API services for the feature
│   │       └── styles/       # Styles for the feature
│   ├── navigation/           # All navigation-related code
│   ├── redux/                # Redux store and slices
│   ├── screens/              # Main app screens (high-level
navigation)
│   ├── services/             # Shared API calls or integrations
(e.g., auth, analytics)
│   ├── utils/                # Utility functions/helpers
│   └── types/                # TypeScript definitions and interfaces
├── scripts/                  # Custom scripts for automation
├── .env                      # Environment variables
├── .gitignore                # Files to be ignored by git
├── app.json                  # Expo/React Native app configuration
├── babel.config.js           # Babel configuration
├── index.js                  # App entry point
├── metro.config.js           # Metro bundler configuration
├── package.json              # Node dependencies and scripts
└── tsconfig.json             # TypeScript configuration
```

Detailed Explanation

1. `src/`

Contains all your app logic, broken down into reusable and modular components.

`assets/`: Store static files like images, fonts, or videos. For example:
plaintext
Copy code
```
assets/
├── images/
├── fonts/
└── icons/
```

-

`components/`: Houses reusable UI components like buttons, cards, and modal dialogs.
plaintext
Copy code
```
components/
├── Button/
│   ├── Button.tsx
│   ├── Button.styles.ts
│   └── index.ts
```

-

`config/`: Stores configuration-related files, like themes, environment variables, or app constants.
plaintext
Copy code
```
config/
├── theme.ts
└── env.ts
```

-
- **`constants/`**: Contains app-wide constants like API URLs or static text strings.
- **`hooks/`**: Custom React hooks to encapsulate logic (e.g., `useAuth`, `useFetch`).

`modules/`: Feature-specific directories to encapsulate everything a feature requires (components, screens, services, etc.). For instance:
plaintext
Copy code
```
modules/
```

```
├── Authentication/
│   ├── components/
│   ├── screens/
│   ├── services/
│   └── styles/
├── Dashboard/
└── Profile/
```

-

`navigation/`: All navigation code, including stacks, tabs, and navigators.
plaintext
Copy code
```
navigation/
├── AppNavigator.tsx
├── AuthNavigator.tsx
├── RootNavigator.tsx
└── index.ts
```

-

`redux/`: Redux setup, including slices and the store.
plaintext
Copy code
```
redux/
├── slices/
│   ├── authSlice.ts
│   ├── userSlice.ts
│   └── index.ts
├── store.ts
└── middlewares/
```

-
- **`screens/`**: Top-level screens that represent routes in your app.
- **`services/`**: For managing API calls, analytics, or third-party integrations like Firebase.
- **`utils/`**: Helper functions like debouncing, validation, and formatting.
- **`types/`**: All TypeScript type definitions.

---

2. Root-Level Configuration

- **`android/` and `ios/`**: Native codebases for Android and iOS.

- **`scripts/`**: Custom Node.js scripts for automating tasks (e.g., cleaning builds, generating assets).
- **`.env`**: Store sensitive environment variables like API keys (use `react-native-dotenv`).
- **`metro.config.js`**: Customize the Metro bundler, e.g., for resolving custom paths.

Best Practices

1. **Code Splitting**: Keep your code modular to avoid bloated files.
2. **Feature-Based Architecture**: Group related files by feature for maintainability.
3. **Environment Configurations**: Use `.env` files for different environments (dev, staging, production).

Testing: Add unit and integration tests using Jest and React Native Testing Library.
plaintext
Copy code

```
src/
├── __tests__/          # Test files
├── jest.config.js      # Jest configuration
```

4.
5. **Linting**: Use ESLint and Prettier to enforce code style and consistency.
6. **Version Control**: Follow Git workflows with proper commit messages and branching strategies.
7. **Documentation**: Add comments and maintain a `README.md` file.