React Native Expoで作る 3プラットフォーム対応 アプリ開発ハンズオン

React Native for Webを使ってiOS Androidに加え、
ブラウザでも動くアプリが作れる！

渡邊達明 著

React Native Expoで作る3プラットフォーム対応アプリ開発ハンズオン

渡邊達明（@nabettu）

はじめに

この度は**React Native Expoで作る3プラットフォーム対応アプリ開発ハンズオン**を手にとっていただきありがとうございます。本書はReact NativeとExpoを使って簡単なカメラアプリを作ってみる流れを1冊でまるっと学べる本となっております。

Expoのバージョン33から React Native for Web が利用できるようになりました。本書では2019年9月時点の最新バージョンである34を利用してiOS,Androidに加えてブラウザ対応を行い、3プラットフォームへ対応したアプリを開発していきます。

この本の対象読者・目的

本書では主に次のような方をターゲットとしています。

- React.js触った事あってReact Nativeも興味ある
- React Nativeなんとなく触った事あるけどExpoはない
- Expo使ってるけどWeb用のがどんなもんか知りたい
- FlutterでもIonicでも3プラットフォーム対応できるので比較したい

このような方たちが本書を手に取ることで、日々の開発・運用が楽になる・調査の手間が省けることを目的としています。

また、本書ではReact Native自体の詳細な説明はあまり記載しておりません。もしReact NativeおよびExpoの基礎部分について知りたい方は

- **実践Expo React NativeとFirebaseで、SNSアプリを最速ストアリリース！（https://www.amazon.co.jp/dp/B07L5W41H4/ref=dp-kindle-redirect?_encoding=UTF8&btkr=1 ）**

を読んでいただくと、基本的な知識がつけられると思いますのでおすすめです。

本書のレベル感としては、React.js自体を触ったことがある方ならある程度調べつつ作っていけるような内容となっております。

Expoってなに？

- React Nativeを完全にJSだけで作れるOSSの開発・ビルド環境
- Expo製アプリのビルドやPush通知サービス、OTAサーバーなどを運営しているスタートアップ

です。前者のメンテナを後者のスタートアップメンバーが行っているという状況のようです。

普通のReact Nativeと何が違うの？

Expoを利用すると、自分でReact Nativeプロジェクトを1から立ち上げることなく、足元が整った状態から開始できるので、開発に専念できます。

- iOS,Androidアプリのビルド
- HotReload
- 実機確認用アプリと即座に確認できる環境
- チーム共有機能
- Push通知もOSを気にせず使える

などなど…便利な機能がいっぱいです。

Expo対応のライブラリしか使えないという縛りはありますが、カメラやビデオ再生・各種センサーへのアクセスなどのような大体のアプリで使えるようなライブラリが一通り揃っています。その上今もなお対応しているライブラリが徐々に増えてきています。

また、OS毎の差分をかなり吸収してくれることや、完全にJSだけで書けるためスイッチングコストが低いというのも人気の理由です。

表記関係について

本書に記載されている会社名、製品名などは、一般に各社の登録商標または商標、商品名です。会社名、製品名については、本文中では©、®、TMマークなどは表示していません。

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について著者はいかなる責任も負いません。

また、本書では開発したアプリをWebサイトとして公開するステップは記述していますが、App Store, Google Playストアに公開する手順については記述しておりませんのでご了承ください。

動作確認環境

- Mac OS 10.14.6
- node v10.13.0
- npm 6.4.1
- expo-cli 3.0.10

動作サンプルとサンプルリポジトリ

- **アプリケーションのブラウザ版（https://telopmaker-sample.netlify.com ）**
  ※ iOS Safariでは正常に動作しません。
- **サンプルリポジトリ(ソースコード)**
  （https://github.com/nabettu/telopmaker-sample/ ）

ぜひ感想を聞かせて下さい！

巻末に記載してある著者のTwitterへリプライを送っていただいたり、ハッシュタグ #ExpoWebハンズオン をつけて感想をつぶやいていただけると次回への励みになります。

目次:

第1章　Expoのセットアップ

早速セットアップから開始していきましょう。お使いのPCにnodeとnpmはインストールされている前提で開始します。

ターミナルでのコマンドについては以下の用に$から開始して記述します。

```
$ npm -v
```

1.1　Expo CLIのインストールと初期設定

すでにExpoをお使い方に関しては飛ばしてしまって構いません。

Expoを使っての開発ではExpo CLIを利用します。

Expo CLIの主な機能は次のようになっています。

- プロジェクトの作成
- 開発中のサーバーの起動とシミュレーターとの接続
- バンドルするJSファイルのPublishやリリースマネジメント
- アプリ自体のビルド

未インストールの人は次のコマンドでグローバルにインストールするところから開始しましょう。

```
$ npm install expo-cli --global
```

インストールが終わり、次のコマンドで3.0.10以上のものが入っているか確認出来たらOKです。

```
$  expo -V
```

プロジェクトの作成

```
$  expo init
```

でプロジェクトを作成します。

そこで最初に開発するテンプレートを選べますので、**blank (TypeScript)**を選択しましょう。

次にプロジェクトの名前であるnameと、expo上でのプロジェクトidであるslugを設定しましょう。

もちろんあなたが付けたい名前を自由に付けてもらって構いませんが、今回作るアプリは「テロップメーカー」という名前でslugは「telopmaker」として説明を続けます。

Yarnがインストールされている場合は

```
? Yarn v1.XX.X found. Use Yarn to install dependencies? (Y/n)
```

と聞かれます。好みですが今回はnpmで本書の説明を進めます。

あとはnpm installまで自動で開始されるので、少々待って頂いて、終わったらtelopmakerのディレクトリを開いてください。

1.2 ファイルの内容説明

```
├── .expo-shared
│   └── assets.json
├── .watchmanconfig
├── App.tsx
├── app.json
├── assets
│   ├── icon.png
│   └── splash.png
├── babel.config.js
├── package-lock.json
├── package.json
└── tsconfig.json
```

Expo特有のファイルについてのみ説明します。

app.json

Expoで作るアプリに関する設定はここにまとめます。この本で作成するアプリに関する重要な部分だけ説明します。

- privacy：publicにしておくと、アプリをbuildかpublishした時点で**Expoのサービスサイト（https://expo.io）上の自身のアカウントのアプリとして公開状態となって誰でも触れる**ため注意してください。非公開にする場合は**"unlisted"**を設定してください。
- sdkVersion：利用するExpo自体のバージョン指定です。
- platforms：利用するプラットフォームです。webが入っていることを確認してください。

他詳細や追加で設定できる項目もありますので、

- **公式ドキュメントのapp.jsonについて（https://docs.expo.io/versions/latest/workflow/configuration/）**

を確認してください。

.expo-shared

最初からassets.jsonが入っていますが、Expo CLIでは画像の圧縮等をビルド時に行ってくれる機能があります。どのファイルに圧縮を行ったかのログが記載されています。

現状はそのログだけですが、今後他にもそういったExpoでの設定やログが格納されていくと思われます。

babel.config.js

babel-preset-expoというExpo用のbabelのpresetが設定されています。

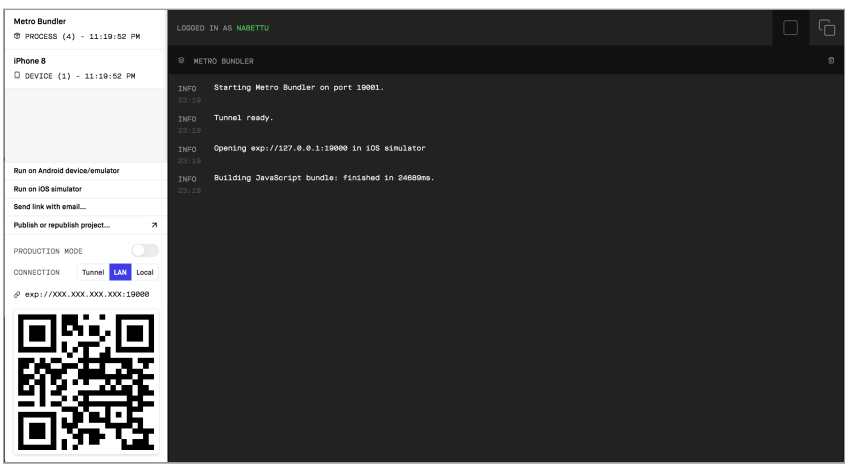
App.tsx

Expoプロジェクトのエントリーポイントです。こちらはプロジェクトルートにApp.tsx(js)を置くのがExpoの規定となっています。他の場所に変えたい場合はpackage.jsonのmainを変更し、元のAppEntryと同じコードを書けば変更出来ます。

1.3 プロジェクトの起動

```
$  npm run start
```

を実行してみましょう。次のような画面が表示されましたでしょうか。



Expo start後表示される画面

「Run on iOS simulator」を押下すると自動的にiOS simulatorが起動し、Expo Clientがインストールされてアプリが起動します。

※実行されない場合はXcodeを開いてiOS simulatorを起動してから再度お試しください。

AndroidでもマシンにAndroid Studioを経由してAVD Managerからインストールしたシミュレータを利用すれば同じように実行出来ます。

Expo Clientのダウンロード

iOS及びAndroidの実機で確認したい場合は以下のリンクからExpo Clientをダウンロードして利用してください。

- **iOSクライアント**（**https://apps.apple.com/jp/app/expo-client/id982107779** ）
- **Androidクライアント**（**https://play.google.com/store/apps/details?id=host.exp.exponent&hl=ja** ）

Expoではこれらのクライアントを利用することで実機を使って素早くデバッグ出来ます。

ダウンロード出来たら、iOSの場合はOSのカメラでPCに表示されているQRコードを読み取ってください。

Androidの場合はアプリを起動して、Projectタブの上の「Scan QR Code」を押下してQRコードリーダーを起動してQRコードを読み取ってください。

今後の画面確認はこれらを利用して進めていきましょう。

これでセットアップは終了です。次の章から早速開発に進んでいきます。

第2章　画面を作ってみる

セットアップが終わったところで、順番に画面を作っていってみましょう。

2.1　アプリの説明画面を作ってみる

まずはこのアプリがどんな動作をするのかを説明する画面を作っていきましょう。

プロジェクトルートにsrcディレクトリを作り、その下にまたscreensというディレクトリを作り、About.tsxというファイルを作成します。

App.tsxをコピー&ペーストして、以下のように文章をちょっと変えてみましょう。

```
import React from "react";
import { StyleSheet, Text, View } from "react-native";

export default function AboutScreen() {
  return (
    <View style={styles.container}>
      <Text>
        このアプリは以下のようなニュース風テロップ付きの画像が作れるアプリです。
      </Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    paddingTop: 32,
    paddingHorizontal: 16,
    backgroundColor: "#fff",
```

```
    alignItems: "center",
    justifyContent: "center"
  }
});
```

作った画面をApp.tsxで読み込んで表示します。

```
import AboutScreen from "./src/screens/About";
/* ～～～省略～～～ */
export default function App() {
  return (
    <View style={styles.container}>
      <AboutScreen />
    </View>
  );
}
```

2.2 画像を追加

ここからサンプルに表示する画像をダウンロードして、画面に表示してみます。

サンプルリポジトリの画像ディレクトリ(https://github.com/nabettu/telopmaker-sample/tree/master/assets)

assetsディレクトリにそれぞれsample.png,telop.pngをダウンロードして同じ名前で保存しておきます。

そのままImageコンポーネントを使って画像を表示しましょう。

```
import React from "react";
import { StyleSheet, Text, Image, View } from "react-native";
const sampleImg = require("../../assets/sample.png");

export default function AboutScreen() {
```

```
  return (
    <View style={styles.container}>
      <Text>
        このアプリは以下のようなニュース風テロップ付きの画像が作れるアプリです。
      </Text>
      <Image source={sampleImg} style={styles.img} />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    paddingTop: 32,
    paddingHorizontal: 16,
    backgroundColor: "#fff",
    alignItems: "center",
    justifyContent: "center"
  },
  img: {
    margin: 16,
    width: 300,
    height: 200
  }
});
```

画像を追加するとアプリはこのような表示になります。

画像を追加した画面

2.3 Webでも起動してみる

画面を増やしたり画像を出したりしてみましたが、次はアプリではなくブラウザでも動作確認しましょう。

```
$ npm run start
```

でプロジェクトを開始していましたが、Control + cで一度止めていただいて、次は

```
$ npm run web
```

を実行してみてください。

同じようにプロジェクトが起動しますが、自動で別なタブに先程作ったアプリのWeb版が表示されます。

それがExpoに追加されたWebサポートでの実行結果です。ワンソースで3プラットフォーム対応するという実感が湧いて来ましたでしょうか。今後はこのようにWeb版でも動作を確認しながら進めて行きましょう。

2.4 絶対パスで書けるように変更する

先程画像を読み込んだ際に "../../assets/sample.png" と記述しました。

しかしこれだと他のファイルで同じファイルを読み込もうとした時や、コンポーネントのファイルの位置が変わってしまった時にすべて書き直さなくてはなりません。

そこで絶対パスでファイル読み込みを記述できるように変更していきます。

まずpackage.jsonにnameという項目を追加しappというデータを入れておきます。

```
{
    "name": "app",
/* ～～～省略～～～ */
```

そして、tsconfig.jsonのcompilerOptions内に次の記述を追加します。

```
/* ～～～省略～～～ */
    "baseUrl": "./",
    "paths": { "app/*": ["*"] }
```

```
    }
}
```

baseUrlはプロジェクトの基準パスの記述で、pathsはソース内で"app/"の記述があったらこのbaseUrl基準のパスに変更するように指定する記述です。

設定ファイルを書き直したらプロジェクトを再起動してみましょう。

そしてApp.tsxとsrc/screens/About.tsxにおいて相対パスで記述していた部分を変更してみましょう。

シミュレータ上のアプリで問題なくアプリが表示されているのを確認できましたでしょうか。

読み込みが失敗している様子

しかし残念なことにブラウザ版を見てみるとエラーになっています。困りましたね。順番に直していきましょう。

2.5 Web版でのエラーを修正する

今回のエラーはWeb版ではファイルの読み込みの仕方がtsconfigを変えただけでは変わらないので、別途設定をする必要があります。

Expo Webでは裏でWebpackを利用してWeb版のビルドを行っていますが、設定ファイルなどは初期状態ではすべて隠蔽された状態になっています。そのままですべてビルドがうまく行けば問題ないのですが、今回のようにカスタマイズしようとする際に困ってしまいます。

そこでExpo CLIではcustomizeコマンドというのを用意していて、その辺りの設定をカスタマイズできるような仕組みがあります。早速一旦プロジェクトを止めてから次のコマンドを実行してみてください。

```
$ expo customize:web
? Which files would you like to generate? … (Use <space> to select,
<return> to submit)
✓ webpack.config.js
✓ web/favicon.ico
✓ web/index.html
✓ web/serve.json
```

と表示されるので、webpack.config.jsを選択してスペースキーを押してからエンターで実行します。するとプロジェクト直下にwebpack.config.jsが作成されます。

エラーが発生した場合

※2019年9月現在の最新版のExpo CLIではコマンド実行時に

```
Cannot read property 'readConfigJsonAsync' of undefined
```

というエラーが出てしまいます。

その場合は次の **Expo Web Betaのsize limit問題から見えた、実用性への道筋**（**https://qiita.com/Nkzn/items/61b7a1bd9034e63533b9** ）を参考にして作業を進めて下さい。

こうして作成されたwebpack.config.jsを読むと、単純に隠蔽前の@expo/webpack-configを読み込んで出力するだけの内容となっています。

絶対パスでファイルを読み込める様に次のようにファイルを編集してください。

```
const createExpoWebpackConfigAsync = require("@expo/webpack-config");
const path = require("path");

module.exports = async function(env, argv) {
  const config = await createExpoWebpackConfigAsync(env, argv);
```

```
  // Customize the config before returning it.
  config.resolve.alias = {
    app: path.resolve(__dirname, "")
  };
  return config;
};
```

これで再度プロジェクトを実行すれば、ブラウザでもアプリが開けるようになるはずです。おめでとうございます。

このようにExpo Webでは設定を変更するためにwebpack.config.jsを編集していくのは必須と言ってもいいと思います。その分拡張性はありますが、逆にWebpackが全く触れないと凝ったものを作るのが途端に難しくなるというのが著者の印象です。

以上でこの章は終わりです。お疲れ様でした。

2.6 【コラム】マルチプラットフォームアプリをワンソースで書ききる事は本当に可能なのか

今回の編集で、ネイティブアプリの方では大丈夫でしたがブラウザ版ではエラーが出てしまいました。**マルチプラットフォームアプリを開発する際には環境によって動いたり動かなかったりすることはよくあること**です。

「iOSではキレイに動いてもAndroidではpaddingがちょっと違う」などはExpoを使っていても未だに日常茶飯事です。ライブラリのアップデートによって徐々に差分はなくなってきていますが、やはり多少は差が出てきてしまうので**OS毎に処理を変えるようなことは必要**になってきます。

もしプラットフォーム毎の差分がとても多い場合には、同じ階層に

- Button.ios.js
- Button.android.js
- Button.web.js

のようにファイルの名称をプラットフォーム名.jsとして配置して、import側では次のように読み込んでおきます。

```
import Button from 'components/Button';
```

すると自動でプラットフォーム毎に読みこむファイルを分けてくれるので便利です。

今後もこういったことがあってもすぐに気付けるようできるだけそれぞれの環境で確認するようにすると良いでしょう。また、CI等でそれらのチェックを自動化してもいいですね。

ワンソースですべて書ききることができる世界...React NativeにおいてもExpo Webではかなりそれに近づいて来た気がします。他にもFlutterやIonicというフレームワークでもワンソースでiOS,Androidのネイティブアプリとブラウザでの Webアプリを実現出来ます。Unityもブラウザ対応を進めているところであったりしますね。

デバイスサイズやOS差異の部分は仕方がないとはいえ、現状それを無視すれば、**これらフレームワークを利用して概ねワンソースで動くものが作れる**のではないでしょうか。

もちろんそれはライブラリがそれぞれのプラットフォーム上で動くように作ってくれているからであって、ライブラリの開発者の方々には足を向けて寝られませんね。これらライブラリ・フレームワークにバグを見つけたらGitHubにissueを書くだけでも価値がありますし一人ひとりが貢献できることは少しでもやっていけるといいですね。

第3章　ナビゲーションを追加しよう

さて、画面が１つだけでは寂しいので次は画面を増やして、それら画面間を遷移できるようにしていきましょう。

3.1　React Navigationをインストール

画面遷移を制御するライブラリもいくつかありますが、React Navigationが一番メジャーなのでこれを使っていきます。

また、2019年9月頭にv5(alpha)が発表された所ですが、まだリリースされていないのでv4をインストールしていきます。

v4からはモジュールが分割され、StackNavigatorなどは別々でインストールする必要があります。また、React NavigationのReact Nativeでの利用にはreact-native-gesture-handlerを別途インストールが必要なので入れましょう。

今回書籍の進行都合上**expo init**で選択したテンプレートを**blank**としました。もし**tabs**を選択するとその時にリリースされているExpoのバージョンに最適化されたreact-navigationが初期インストールされていますので、そちらを選択している場合は追加のinstallは不要です。

- react-navigation
- react-navigation-stack
- **react-native-gesture-handler@1.3.0**

の3つをまとめてinstallします。

※ React Native v60では解消しているようですが、Expo v34では警告が出てしまうのでgesture-handlerのバージョンを1.3.0とします。

```
$  npm i react-navigation react-navigation-stack react-native-gesture-handler@1.3.0
```

srcディレクトリ内にnavigationディレクトリ、その下にAppNavigator.tsxを作成してひとまず現状の画面がそのまま動くように設定をしていきましょう。

```
import { createAppContainer } from "react-navigation";
import { createStackNavigator } from "react-navigation-stack";

import AboutScreen from "app/src/screens/About";

export default createAppContainer(
  createStackNavigator({
    AboutScreen: { screen: AboutScreen }
  })
);
```

プロジェクトルートのApp.tsxで、AboutScreenとしていた部分をAppNavigatorに変更します。import先も変更しましょう。また、containerにalignItemsが設定されているとAppNavigatorが表示できないのでstyleから削除します。

```
import React from "react";
import { StyleSheet,  View } from "react-native";
import AppNavigator from "app/src/navigation/AppNavigator";

export default function App() {
  return (
    <View style={styles.container}>
      <AppNavigator />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: "#fff",
    justifyContent: "center"
  }
});
```

これでReact Navigationを使っての画面表示は出来ましたので、次は画面を増やして遷移させてみましょう。

3.2 画面を増やす

アプリについての説明を行う画面の次には、テロップを付ける画像を選択する画面を増やします。

screens内にImagePick.tsxを作成し、ひとまず文章を1行表示するようにしておきます。

```
import React from "react";
import { StyleSheet, Text, View } from "react-native";

export default function ImagePickScreen(props) {
  return (
    <View style={styles.container}>
      <Text>テロップを付ける画像を選択してください。</Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    paddingTop: 32,
    paddingHorizontal: 16,
    backgroundColor: "#fff",
    alignItems: "center",
    justifyContent: "center"
  }
});
```

次にAppNavigatorにImagePickScreenとして、画面を登録しておきます。

```
import { createAppContainer } from "react-navigation";
import { createStackNavigator } from "react-navigation-stack";

import AboutScreen from "app/src/screens/About";
import ImagePickScreen from "app/src/screens/ImagePick";

export default createAppContainer(
  createStackNavigator({
    AboutScreen: { screen: AboutScreen },
    ImagePickScreen: { screen: ImagePickScreen }
  })
);
```

3.3 画面遷移してみる

登録しただけでは何も始まらないので、次はAbout.tsxへボタンを追加して画面遷移するようにしましょう。

TouchableOpacityを追加でimportして、画像の下にボタンを追加します。styleを書いてボタンらしい見た目にしましょう。

そしてAboutの引数としてpropsを定義してナビゲーションのメソッドを実行するように関数を追加し、ボタンに設定します。

```
import React from "react";
import { TouchableOpacity, StyleSheet, Text, Image, View } from "react-native";
const sampleImg = require("app/assets/sample.png");

export default function AboutScreen(props) {
  const gotoImagePickScreen = () => {
    props.navigation.navigate("ImagePickScreen");
  };
```

```
  return (
    <View style={styles.container}>
      <Text>
        このアプリは以下のようなニュース風テロップ付きの画像が作れるアプリです。
      </Text>
      <Image source={sampleImg} style={styles.img} />
      <TouchableOpacity onPress={gotoImagePickScreen} style={styles.btn}>
        <Text style={styles.btnText}>遊んでみる</Text>
      </TouchableOpacity>
    </View>
  );
}

const styles = StyleSheet.create({
  /*
  省略
  */
  btn: {
    marginTop: 32,
    backgroundColor: "#099",
    borderRadius: 4,
    paddingVertical: 8,
    paddingHorizontal: 16
  },
  btnText: {
    color: "#fff"
  }
});
```

ボタンを押したら画像選択を行う画面へ遷移するようになりましたでしょうか。次は何も表示されていないヘッダーへ画面の名称を表示させます。

ボタンを追加した画面

3.4 ヘッダーにタイトルを追加

React Navigationではデフォルトで画面上にヘッダーが表示されています。

navigationOptionsというオプションを、React Navigationへ読み込ませる画面に設定すると自動でヘッダーのコンテンツを設定出来ます。

今回はAbout画面に「本アプリについて」というタイトルを表示させます。

export defaultをAboutScreenから除き、navigationOptionsを設定した後にexportします。

```
/* 省略 */
function AboutScreen(props) {
/* 省略 */
}
/* 省略 */
AboutScreen.navigationOptions = { title: "本アプリについて" };
export default AboutScreen;
/* 省略 */
```

ImagePick画面にも同じ様に「画像選択」というタイトルを設定しておきましょう。

ブラウザ版でも同様にヘッダーにタイトルが表示されるようになりましたでしょうか。

タイトルが追加された画面

タイトルのスタイルを変えたいときなどは、詳細な使用方法は公式ドキュメントを参照してください。**React Navigation**（https://reactnavigation.org ）

以上でこの章は終わりです。

画面遷移は右にStackしているのがデフォルトです。もしモーダルを利用したい場合はStackNavigatorの設定で**mode:"modal"**とすること実現出来ますのでぜひ試してみてください。

第4章　カメラとカメラロールにアクセスして画像を使う

この章では画像をアプリに取り込む部分の実装を行っていきます。

4.1　カメラとカメラロールへのアクセス権限の取得

OS毎にパーミッション（権限管理）の違いはありますが、今回利用するカメラとカメラロールへのアクセス権限はiOS,Android双方で別々に取得が必要です。また、ブラウザに関してはカメラを使わずファイルアップロードだけ対応するという方針にします。

アプリを開いているのが「ブラウザかどうか」をチェックするには次のような記述で判定します。

```
import Constants from "expo-constants";
if(Constants.platform.web){
    // ブラウザで開かれている場合の処理
}
```

また、アプリではカメラロールの権限が無いとカメラのアプリ内利用・カメラロール双方が使えません。そのためその場合にはエラーメッセージを表示するようにしましょう（カメラを権限を拒否されてもカメラロールが利用できればそのままアプリの利用が出来ます）。

次のように、画面を開いたタイミングでPermissionsを使ってパーミッションの確認をして、Hooks APIを利用して権限管理を行って行きます。

```
import React, { useState, useEffect } from "react";
import { StyleSheet, Linking, Text, View, Alert } from "react-native";
import * as Permissions from "expo-permissions";
import Constants from "expo-constants";

function ImagePickScreen(props) {
  const [hasPermissionCameraRoll, setHasPermissionCameraRoll] =
```

```
useState(false);
  const [hasPermissionCamera, setHasPermissionCamera] =
useState(false);

  const checkCameraAndImagePermission = async () => {
    if (!Constants.platform.web) {
      const { status: cameraRollPermissionStatus } = await
Permissions.askAsync(
        Permissions.CAMERA_ROLL
      );
      if (cameraRollPermissionStatus === "granted") {
        setHasPermissionCameraRoll(true);
      }
      const { status: cameraPermissionStatus } = await
Permissions.askAsync(
        Permissions.CAMERA
      );
      if (cameraPermissionStatus === "granted") {
        setHasPermissionCamera(true);
      }
      if (cameraRollPermissionStatus !== "granted") {
        Alert.alert("アプリの利用にはカメラロールへのアクセス許可が必要で
す。");
      }
    } else {
      setHasPermissionCamera(true);
      setHasPermissionCameraRoll(true);
    }
  };

  useEffect(() => {
    checkCameraAndImagePermission();
  }, []);

  return (
```

```
    <View style={styles.container}>
      <Text>テロップを付ける画像を選択してください。</Text>
      {!hasPermissionCameraRoll && (
          <Text style={styles.notice}>写真にアクセスする権限がありません
</Text>
      )}
    </View>
  );
}
/* styles */
  notice: {
    marginTop: 32,
    color: "#f66"
  },
/* 省略 */
```

この状態でアプリを開くと、権限の確認画面が表示され、カメラロールを拒否するとエラーメッセージが出るようになりました。

4.2 権限がない場合はアプリの設定画面を開くように

エラーメッセージを出すだけでは不親切なので、次はアプリの設定画面へ遷移するようなボタンを用意してあげましょう。

ブラウザ以外の場合にはメッセージの下にボタンを配置し、openAppSettingという関数を作ってアプリの設定画面へ遷移するようにしましょう。

iOSの場合にはReact NativeのLinkingが利用できるのですが、Androidの場合は別途expo-intent-launcherを利用します。IntentLauncherでは色々な機能がありますがアプリのpackage名を指定して開く機能を使います。

```
$ npm i expo-intent-launcher
```

でインストールした後、次のようにコードを変更します。

Expo Clientで開発している場合にはpackage名は"host.exp.exponent"で固定となり、ビルドしたアプリの場合には個別のパッケージ名を指定します。app.jsonに記載したpackage名も、Expo Clientかどうかの判定もConstantsから取得できるので、それらを利用していきましょう。

```
/* 省略 */
import { TouchableOpacity } from "react-native"; // 追加
import * as IntentLauncher from "expo-intent-launcher";
/* 省略 */

/* ImagePickScreen内 */
  const openOsSetting = () => {
    if (Constants.platform.android) {
      IntentLauncher.startActivityAsync(
        IntentLauncher.ACTION_APPLICATION_DETAILS_SETTINGS,
        {
          data:
            "package:" + Constants.appOwnership === "standalone"
              ? Constants.manifest.android.package
              : "host.exp.exponent"
        }
      );
    } else {
      Linking.openURL("app-settings:");
    }
  };

/* render内 */
      {!hasPermissionCameraRoll && (
        <>
          <Text style={styles.notice}>写真にアクセスする権限がありません。アプリの設定画面でカメラロールへのアクセスを許可してください。</Text>
          {!Constants.platform.web && (
            <TouchableOpacity onPress={openAppSetting} style={styles.btn}>
              <Text style={styles.btnText}>アプリの設定画面を開く</Text>
```

```
            </TouchableOpacity>
          )}
        </>
      )}

/* 省略：StyleにはAboutと同じbtn,btnTextを追加 */
```

では権限の準備が出来たので次は画像を取得してきます。

4.3 ImagePickerで画像を選択

まずは画像選択をトリガーするボタンを設置します。

横並びになるようにbtnAreaというViewで囲ってflexDirection: "row"を設定しておきましょう。

また、ブラウザ版の場合は画像アップロードだけなのでカメラから取得するボタンを非表示としておきます。

※今回は使いませんがExpo Cameraというライブラリを利用すれば、ブラウザでもWebカメラを使えるように出来ます。

```
/* render内 */
      <Text>テロップを付ける画像を選択してください。</Text>
      <View style={styles.btnArea}>
        {hasPermissionCamera && !Constants.platform.web && (
          <TouchableOpacity
            onPress={() => openImagePicker({ type: "camera" })}
            style={styles.btn}
          >
            <Text style={styles.btnText}>カメラを開く</Text>
          </TouchableOpacity>
        )}
        {hasPermissionCameraRoll && (
          <TouchableOpacity
            onPress={() => openImagePicker({ type: "library" })}
```

```
            style={styles.btn}
          >
            <Text style={styles.btnText}>ライブラリを開く</Text>
          </TouchableOpacity>
        )}
      </View>
      {!hasPermissionCameraRoll && (
/* Style内 */
  btnArea: {
    flexDirection: "row"
  },
/* 省略 */
```

次にopenImagePicker関数の中身を書いていきます。まずは画像選択のためのライブラリのExpo Image Pickerをnpm installしてください。

```
$ npm i expo-image-picker
```

Expo Image Pickerは画像を取得するためのライブラリで、カメラとカメラロールが同じインタフェースで画像を取得出来ます。画像を取得したら、そのuriをHooksのphotoUriで取得できるようにしておきましょう。

また、Expo Image Pickerでは画像を取得時にアスペクト比を調整する画面を挟んで任意のアスペクト比に指定する機能がAndroidでのみ利用出来ます。※iOSでも指定する画面は出ますが正方形以外の調整が出来ません。

そのため、allowsEditingをConstants.platform.androidの場合のみtrueにしておきましょう。

先程配置したボタンを押してからphotoUriに画像を格納するまでは次のようなコードで実現します。

```
/* 省略 */
import * as ImagePicker from "expo-image-picker";
/* ImagePickScreen()内 */
    const [photoUri, setPhotoUri] = useState(null);
  const openImagePicker = async ({ type = "library" }) => {
```

```
    const imageOption: ImagePicker.ImagePickerOptions = {
      allowsEditing: Boolean(Constants.platform.android),
      aspect: [3, 2]
    };

    const photo: ImagePicker.ImagePickerResult =
      type === "camera"
        ? await ImagePicker.launchCameraAsync(imageOption)
        : await ImagePicker.launchImageLibraryAsync(imageOption);
    if (!photo.cancelled) {
      setPhotoUri(photo.uri);
    }
  };
```

4.4 画像をプレビュー

画像のURIはphotoUriに格納されているので、プレビューしてみましょう。ついでにやり直しボタンでもう一度アップロードできるような機能も一気に作っていきます。

```
import { Image } from "react-native"; // 追加
/* 省略 */
/* ImagePickScreen()内 */
  const removePhoto = () => {
    setPhotoUri(null);
  };
  const gotoEditScreen = () => {};
/* render()内 */
      {photoUri ? (
        <>
          <Image source={{ uri: photoUri }} style={styles.img} />
          <View style={styles.btnArea}>
            <TouchableOpacity style={styles.btn} onPress={removePhoto}>
              <Text style={styles.btnText}>やり直す</Text>
```

```
            </TouchableOpacity>
            <TouchableOpacity style={styles.btn} onPress=
{gotoEditScreen}>
              <Text style={styles.btnText}>これでOK</Text>
            </TouchableOpacity>
          </View>
        </>
      ) : (
        <View style={styles.btnArea}>
                    ...
        </View>
      )}
/* style内 */
  img: {
    marginTop: 32,
    width: 300,
    height: 200
  }
```

4.5 画像を次の画面に渡す

次は取得した画像をテロップを載せて編集する画面に渡します。画面をまたいだデータの引き渡しはnavigationにデータを乗せます。

まずは引き渡し先の新しいEditScreenを作成します。About.tsxをコピーして、テキストを変えつつImageのSourceは一旦placehold.jpの仮画像を入れておきます。

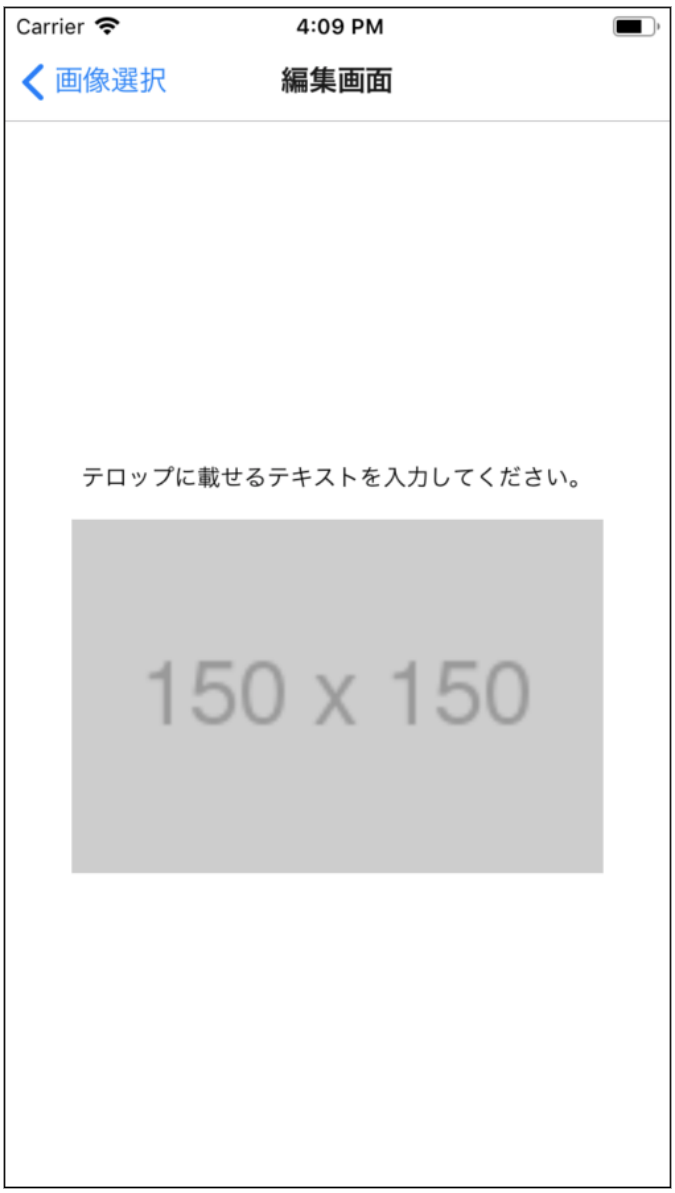
```
import React from "react";
import { StyleSheet, Image, Text, View } from "react-native";

function EditScreen(props) {
  return (
    <View style={styles.container}>
```

```
      <Text>テロップに載せるテキストを入力してください。</Text>
      <Image
        source={{ uri: "https://placehold.jp/150x150.png" }}
        style={styles.img}
      />
    </View>
  );
}

EditScreen.navigationOptions = { title: "編集画面" };
export default EditScreen;

/* StyleはAbout.tsxと同じ */
```

AppNavigator.tsxにEditSceenの定義を行ったらImagePickScreenで空の関数だったgotoEditScreenを次のように変更しましょう。navigateの第2引数にはパラメータでデータの引き渡しが行なえます。今回はphotoUriというパラメータ名でそのまま引き渡し形にしましょう。

```
  const gotoEditScreen = () => {
    props.navigation.navigate("EditScreen", { photoUri });
  };
```

一旦、これで画像選択画面から遷移だけが行える事を確認してください。



仮画像の表示を行った画面

パラメータを受け取って表示する

先程追加したImageタグのsourceを次のように変更してみましょう。

パラメータの取得には**props.navigation.getParam**を利用します。ImagPickScreenで指定したphotoUriを取得します。

もしデータが入っていなかった場合には第2引数のデータが代わりに入るので、デバッグなどでこの画面を直接開いた際にも仮画像が表示されるようになります。

```
<Image
  source={{
    uri: props.navigation.getParam(
      "photoUri",
      "https://placehold.jp/150x150.png"
    )
  }}
  style={styles.img}
/>
```

前の画面で取得した画像を表示した状態

画像の表示が行えましたでしょうか。次の章ではいよいよその画像に他の画像及び文章を合成する部分を作っていきます。

第5章　画像を合成する

いよいよ画像にテロップを合成して、保存できるところまで行ったら完成です。あと少し頑張りましょう!

ステップとしては次の4ステップとなっております。

1. 画像の上にテロップの枠と文字を載せて表示する
2. 文字を自由入力して枠内の文字を動的に変更する
3. 画像を保存できる形式に合成する
4. 合成した画像を保存する

5.1　画像の上にテロップの枠と文字を載せて表示する

React NativeではWebと同じ様にポジション指定が可能で、いわゆる **"position: absolute;"** のような親要素からの相対位置の指定が出来ます。

それを利用してテロップを画像の上に乗せて行きましょう。

- 画像として保存する領域をcaptureAreaとしてViewで囲む
- 2章でダウンロードしたtelop.pngを画像の上に表示
- テキストを画像の上に表示

の3点を実装します。

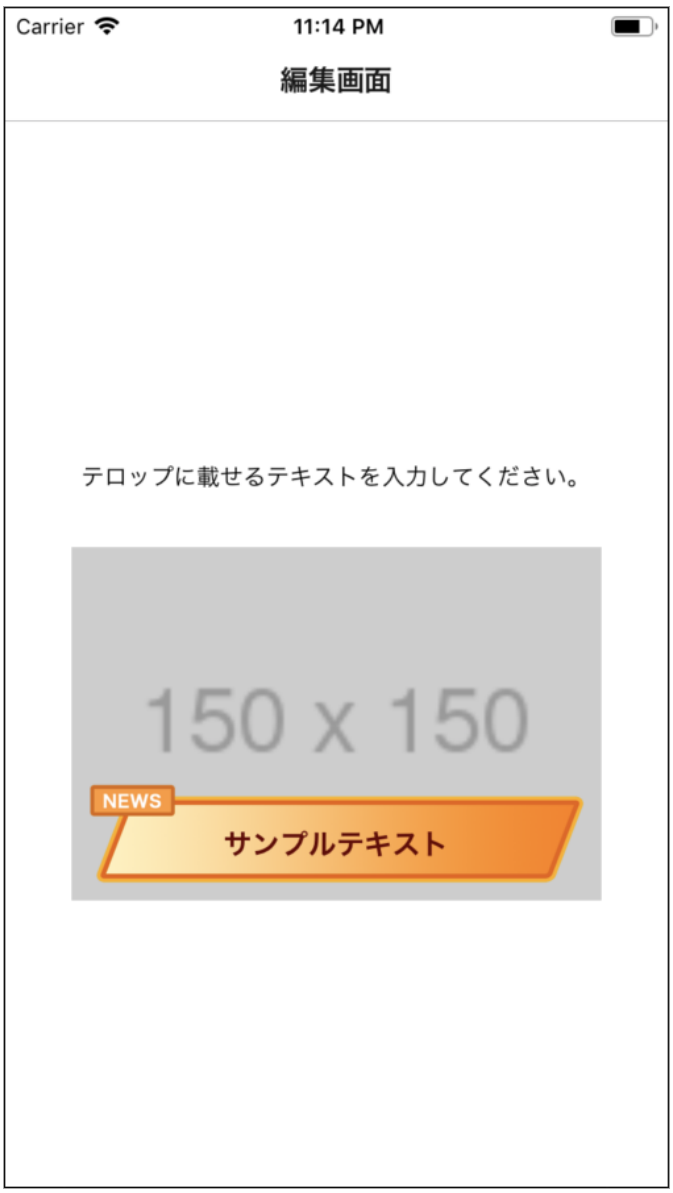
また、この画面の編集の際にはリロード時にEditScreenが表示されていると都合がいいので、AppNavigatorでの順番を変えておくと便利です。特にしていしなければ1番上に読み込んだ画面が初期表示する画面になります。

```
import React, { useState } from "react";
import { StyleSheet, Image, Text, View } from "react-native";
const telopImg = require("app/assets/telop.png");

function EditScreen(props) {
  const [text, setText] = useState("サンプルテキスト");
```

```
  return (
    <View style={styles.container}>
      <Text>テロップに載せるテキストを入力してください。</Text>
      <View style={styles.captureArea}>
        <Image
          source={{
            uri: props.navigation.getParam(
              "photoUri",
              "https://placehold.jp/150x150.png"
            )
          }}
          style={styles.img}
        />
        <Image source={telopImg} style={styles.telop} />
        <Text style={styles.telopText}>{text}</Text>
      </View>
    </View>
  );
}

EditScreen.navigationOptions = { title: "編集画面" };
export default EditScreen;

const styles = StyleSheet.create({
  captureArea: {
    position: "relative",
    overflow: "hidden",
    marginTop: 32,
    width: 300,
    height: 200
  },
  img: {
    // marginは削除
    width: 300,
    height: 200
```

```
  },
  telop: {
    position: "absolute",
    bottom: 10,
    left: 10,
    width: 280,
    height: 280 / 5,
    resizeMode: "cover"
  },
  telopText: {
    position: "absolute",
    bottom: 10,
    height: 45,
    lineHeight: 45,
    width: "100%",
    textAlign: "center",
    fontSize: 17,
    fontWeight: "bold",
    color: "#660000",
    left: 0
  },
/*  container, btn, btnTextはそのまま */
});
```

実装してこのような表示になりましたでしょうか。サンプルテキストを変更して動的な表示の確認をしてみてください。



テロップを入れた状態

5.2 文字を自由入力して枠内の文字を動的に変更する

captureAreaの次にTextInputを配置して、onChangeTextへsetTextを入れて入力を反映できるようにしましょう。

```
import { StyleSheet, TextInput, Image, Text, View } from "react-
native";
/* render内 */
      <TextInput
        defaultValue={text}
        style={styles.textInput}
        onChangeText={inputText => setText(inputText)}
      />
/* style内 */
  textInput: {
    backgroundColor: "#ccc",
    borderWidth: 1,
    borderColor: "#666",
    borderRadius: 4,
    marginTop: 16,
    padding: 8,
    fontSize: 20,
    width: 300
  },
```

入力欄を実装したら早速入力してみましょう。

入力欄が見えなくなってしまった状態

せっかく作った入力欄ですが、iPhone8 サイズの端末で見るとキーボードで入力欄が完全に隠れてしまいます。

React NativeにはそのためのKeyboardAvoidingViewという、入力中にだけ自動でサイズを変更してくれるViewを利用します。また、その中身をScrollViewにしておく必要もあります。

KeyboardAvoidingViewには次の2つのpropsを設定出来ます。これらを設定するとサイズを調整してくれます。

- `behavior="padding"`
- `keyboardVerticalOffset={Header.HEIGHT}`

Headerはreact-navigation-stackで表示しているヘッダーの高さを取得するために追加でimportします。

これらを実装するとキーボード表示中も入力欄が隠れることなく入力できるようになります。

```
import React, { useState } from "react";
import { Header } from "react-navigation-stack";
import {
  KeyboardAvoidingView,
  ScrollView,
  StyleSheet,
  TextInput,
  Image,
  Text,
  View
} from "react-native";
const telopImg = require("app/assets/telop.png");

function EditScreen(props) {
  const [text, setText] = useState("サンプルテキスト");
  return (
    <KeyboardAvoidingView
      style={styles.container}
      behavior="padding"
      keyboardVerticalOffset={Header.HEIGHT}
    >
      <ScrollView contentContainerStyle={styles.contentContainer}>
        <Text>テロップに載せるテキストを入力してください。</Text>
        <View style={styles.captureArea}>
                    ...
        </View>
        <TextInput
          defaultValue={text}
```

```
          style={styles.textInput}
          onChangeText={inputText => setText(inputText)}
        />
      </ScrollView>
    </KeyboardAvoidingView>
  );
}

EditScreen.navigationOptions = { title: "編集画面" };
export default EditScreen;

const styles = StyleSheet.create({
  container: {
    flex: 1
  },
  contentContainer: {
    paddingVertical: 32,
    backgroundColor: "#fff",
    alignItems: "center",
    justifyContent: "center"
  },
/* 省略 */
```

キーボードが隠れなくなった状態

これで入力内容が動的に反映されるようになりました。

次は画像を保存できる形式に合成してみましょう。

5.3 画像を保存できる形式に合成する

今回はreact-native-view-shotというライブラリ使って画像合成を行いますので、npm installしていきます。

```
$ npm add react-native-view-shot
```

このライブラリの使い方は簡単で、画像に変換したいコンポーネントのrefを渡してcaptureRefを実行すると画像のuriが返ってくる形になります。

以下の順序で画像の生成を実装していきます。

- captureAreaのStyleを指定したViewのrefを保存
- テキストの入力が終わったら押すボタンを設置
- ボタンの押下でreact-native-view-shotのcaptureRefを実行
- captureRefの結果を画面に表示

```
/* 省略 */
import { captureRef } from "react-native-view-shot";
/* 省略 */
function EditScreen(props) {
  const [text, setText] = useState("サンプルテキスト");
  const [captureContainerRef, setCaptureContainerRef] = useState(null);
  const [captureImageUri, setcaptureImageUri] = useState(null);

  const captureImage = async () => {
    if (!captureContainerRef) {
      return;
    }
    const result = await captureRef(captureContainerRef, {
      result: "tmpfile",
      width: 600,
      height: 400,
      quality: 1,
      format: "png"
    });
    if (result) {
      setcaptureImageUri(result);
```

```
    }
  };

  return (
    <KeyboardAvoidingView
      style={styles.container}
      behavior="padding"
      keyboardVerticalOffset={Header.HEIGHT}
    >
      <ScrollView contentContainerStyle={styles.contentContainer}>
        <Text>テロップに載せるテキストを入力してください。</Text>
        <View
          style={styles.captureArea}
          ref={ref => setCaptureContainerRef(ref)}
        >
          ...
        </View>
        <TextInput
          defaultValue={text}
          style={styles.textInput}
          onChangeText={inputText => setText(inputText)}
        />
        <TouchableOpacity style={styles.btn} onPress={captureImage}>
          <Text style={styles.btnText}>画像を生成する</Text>
        </TouchableOpacity>
        {captureImageUri && (
          <Image
            source={{ uri: captureImageUri }}
            style={styles.captureImage}
          />
        )}
      </ScrollView>
    </KeyboardAvoidingView>
  );
}
```

```
/* Style内 */
  captureImage: {
    marginTop: 32,
    width: 300,
    height: 200
  },
/*省略 */
```

「画像を生成する」ボタンを押下して、ボタンの下に画像が表示されたら完了です。

さて、ここでお気づきの方もいらっしゃるかもしれませんが、この機能ブラウザ上では正しく動作しません（2019年9月12日現在）。

ということでブラウザ版を動くようにしていきます。

Webで動かない対策

Expoの各種モジュールはv33,34のタイミングでかなりの部分がブラウザに対応しておりますが、react-native-view-shotはまだブラウザには対応していません。

「3プラットフォーム対応と言ったもののどうしよう...」

と、リポジトリを見ているとWebに対応したPRが出ています！

ということで、内容を確認して問題なさそうなのでこちらをnpm installしてみましょう（一旦そのコードが変更される可能性もあるので私がforkしたものをinstallします）。

```
$ npm add react-native-view-shot@https://github.com/nabettu/react-native-view-shot
```

こちらに変更して無事にブラウザでも画像生成が動作しました！

ブラウザで動作した状態

※こちらのライブラリで利用しているhtml2canvasとreact-native-webの組み合わせではiOSのSafariで動作しない場合があります。また、ブラウザではplacehold.jpのような外部リソースを読み込んでの画像化は出来ません。

5.4 合成した画像を保存する

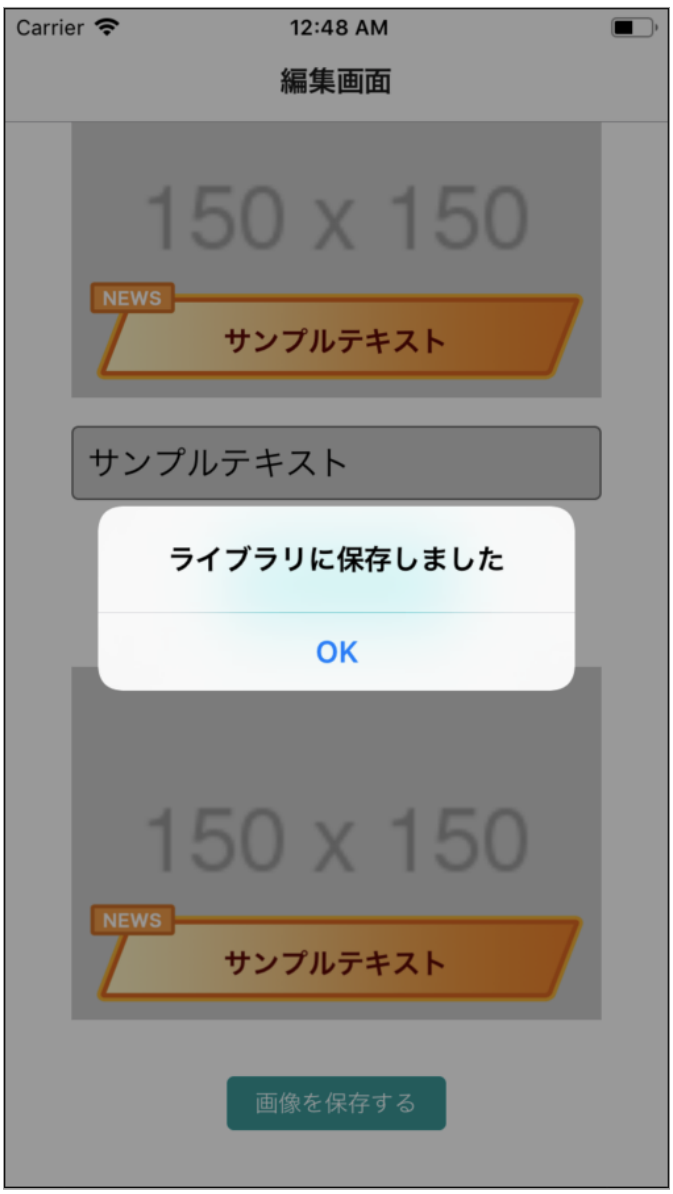
最後に合成した画像をカメラロールへ保存する機能を作って実装は完了とします。

ブラウザでは画像として表示してあれば、画像を長押しすれば保存できますので、その旨を表示することにしましょう。

React Native自体のCameraRollを利用して、画像のURIを指定して保存します。前画面でカメラロールへのパーミッションはもうとってあるはずなのでそのまま保存が可能です。保存したらアラートでメッセージを表示しましょう。

```
import {
  Alert,
  CameraRoll,
    ...
} from "react-native";
import Constants from "expo-constants";

function EditScreen(props) {
  const saveImage = () => {
    CameraRoll.saveToCameraRoll(captureImageUri);
    Alert.alert("ライブラリに保存しました");
  };

/*  render内 */
        {captureImageUri && (
          <>
            <Image
              source={{ uri: captureImageUri }}
              style={styles.captureImage}
            />
            {Constants.platform.web ? (
              <Text>画像を長押しして保存してください</Text>
            ) : (
              <TouchableOpacity style={styles.btn} onPress={saveImage}>
                <Text style={styles.btnText}>画像を保存する</Text>
              </TouchableOpacity>
            )}
```

```
        </>
      )}
```



保存ボタンを押下した状態

以上で実装は完了です。お疲れ様でした!!!

5.5 まとめ

ExpoをつかってのReact Nativeでの開発と、ブラウザ対応についてなんとなく肌感をつかめていただけたでしょうか。

バージョンによっては出ないかもしれませんが、Expo CLIを利用していて次のメッセージが表示されている事に気づいた方もいらっしゃると思います。

> 「Web support in Expo is experimental and subject to breaking changes. Do not use in production yet.」

まだ「実際のプロダクトで使うにはちょっと待って〜」ということらしいですが、個人開発レベルや小規模プロダクト、なんならバグを引き当てても自分で解決していける方なら使えなくはないのかなと思います。

実装についてはこの章で終わりですので、そこだけ気になっていた方はこれで本書を閉じて頂いてかまいません。次の章では実際にブラウザ版としてビルドしてWebサイトとして公開する手順を説明していきます。

アプリとしてビルドしてストアに公開する手順については、

- **Expo公式ドキュメント（https://docs.expo.io/versions/latest/distribution/building-standalone-apps/ ）**
- **実践Expo（https://www.amazon.co.jp/dp/B07L5W41H4/ ）**

などを確認していただければ、必要な情報は得られると思いますので、本書では割愛させていただきます。

次の章ではブラウザ版を公開するための手順を説明していきます。

第6章　ブラウザ版をビルド&サイトとして公開する

本章ではブラウザ版のビルドしてから実際にサイトとして公開するまでを行います。

事前準備として **Netlify**（**https://www.netlify.com** ）に登録しておいてください。Netlifyは静的サイトを手軽に運用できるサービスです。登録方法については本書では割愛させていただきます。

6.1　ブラウザ用にビルドする

Expo CLIを使ってビルドする際にはアプリと同じくbuildコマンドを利用します。

また、2019年9月現在、Expo Webのビルドを利用する際には標準でPWA対応が入っていますが、ビルドエラーの原因となるためPWA対応はOFFにして進めます。

次のコマンドを実行するとPWA対応を抜いた状態で、開発中に見ていたブラウザ版と同じ状態になるhtmlなどのファイルを生成してくれます。

```
$ expo build:web --no-pwa
```

実行後エラーがなければ **web-build** というディレクトリが生成され、必要なファイルが諸々格納されています。ローカルサーバーを起動してブラウザで実行して動作確認をしてみてください。

次はこれをNetlifyのCI上で実行します。その前にCIでもExpo CLIが利用できるようにExpo CLIをnpm installしておきましょう。

```
$ npm i --save-dev expo-cli
```

6.2　Netlifyに登録してビルド設定を行う。

GitLabやBitbucketでも問題ないですが、今回はGitHubを使います。予め今回開発したソースをpushしておきます。

Netlify管理画面→「New site from Git」→「GitHub」→今回のリポジトリを選択します。

Basic build settings内の項目を編集する必要があります。

- Build command `expo build:web --no-pwa`
- Publish directory `web-build`

以上2点をそれぞれ指定して、Deployしましょう。

デプロイ後は自動でURLが発行されるので、Overviewに表示されるURLをクリックしてみましょう。URLを変更したければSettingsから変更が可能です。

作ったサイトがブラウザ上で正しく動作しましたでしょうか。

これでブラウザ版のビルド&公開については完了で本書は以上となります。お疲れ様でした。

あとがき

最後まで読んでいただきありがとうございます。

React Native Expoで作る3プラットフォーム対応アプリ開発ハンズオンはいかがでしたでしょうか?本書を読むことでExpoを使いこなすステップとなり、みなさんの日々の業務や個人開発の効率がアップすれば幸いです。

Expoを使ったことがなかった方も、便利な機能が沢山あったことに気づいていただけたでしょうか。

ぜひこれからも進化するExpoを使いこなして素敵なエンジニアライフを送ってください!

謝辞

渡邊雄さん（ハムカツおじさん @hmktsu ）

レビューありがとうございました!!!

また、日頃React Native周りの情報を発信してくださり非常に勉強になります。本書の読者様も、React Nativeについての最新情報を知りたい方はぜひフォローするといいと思います。

サポート

本書でなにかお気づきのことがございましたら、こちらのフォームからご連絡ください。

https://form.run/@shimesabuzz（https://form.run/@shimesabuzz ）

著者

渡邊達明

株式会社クリモ 取締役副社長 / CTO。1988年宮城県生まれ。仙台高専専攻科を卒業後、富士通株式会社にてWindowsOSのカスタマイズ業務に従事する。

その後面白法人カヤックにて受託開発部門を経験後、ブロガーの妻と二人で株式会社クリモを設立。WebとReact Nativeを使ってのフロントエンド中心の受託開発や保育園問題の解決のためのメディアを運営。

「三度の飯よりものづくり」と言っていたらBMIが17になり健康診断で毎回ひっかかるのが悩み。一番好きな寿司ネタは「えんがわ」。

- twitter: **@nabettu**（**http://twitter.com/nabettu** ）
- blog: **http://nabettu.hatenablog.com/**
  （**http://nabettu.hatenablog.com/** ）

React Native Expoで作る3プラットフォーム対応アプリ開発ハンズオン

2019年9月12日 初版

著者 渡邊達明（@nabettu）

発行所 Flight Books（フライトブックス）
https://flightbooks.pub/

印刷

ビルド buildId: 70699549-7ef7-4d7b-a388-513f461744fb
converter:v1.10
pdf renderer:v69

本書はFlight Booksを用いて制作されました。