React Native Template

Moove it

Why?

React-Native has been taking the mobile world over step by step since it was first released to the public, and here at Moove-it we are always on the front of technology. We work really hard to create beautiful, scalable and maintainable mobile apps, and React Native is perfect for this.

Since it’s a pretty new technology, there isn’t a consensus on the best architectures to use. That’s why we decided to implement our own!

The main reason why we wanted to define a baseline for any new project was to reduce the number of decisions that a developer has to make before beginning to develop a new project. With that in mind, and based on our previous experience on writing several other applications, we defined a project template that works as a starting point for React Native projects; providing a base architecture, core frameworks, and helpers to jumpstart development.

This doc introduces the fundamental concepts and tools included in the project, as well as examples to make things more clear.

Base tooling

As you may already know, React Native uses Javascript, so we can use the npm ecosystem and all the libraries/frameworks that it provides. The following list includes all the basic packages that come already installed in the project and their purposes. Most of them are further explained afterwards.

- Axios for networking.
- PropTypes to type-check our components exposed properties.
- React-Native-Config to manage environments.
- React-Native-Navigation native navigation library.
- React-Native-Localization for string localization.
- Redux for state management.
- Recompose for utilities.
- Redux-Persist as persistence layer.
- Redux-Thunk to dispatch asynchronous actions.
- Reselect the selector library for redux.

- Jest and Enzyme for testing.

Architecture

Container - Presentation

Even though it’s not defined explicitly in the template and is more of a decision of the team, we prefer to structure our components following the container-presentation pattern.

For those who may not know what it is about, it consists of dividing your components into two types: container and presentation. Container components are the ones that manage logical state and any logic involved (such as data fetching) and the presentation components are the ones that are exclusively responsible for the UI.

This brings a lot of advantages in terms of reusability and separation of responsibilities. If you want to further look into this pattern, you can check this and this posts that explain it with more detail.

Example

The following is a small example to better explain what is a presentation component and a container component. Imagine we have a home screen where you need to fetch the users from the backend and show them in a list.

Following the previously stated rules, we will have two components, one container that has the logic (fetch the users) and one presentation component that has the UI (the actual list)

In this case, we could have a *HomeContainer.js* that may be similar to the following one:

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import PropTypes from 'prop-types';
import { getUsersFromNetwork, actionTypes } from
'../../actions/UserActions';
import getUsers from '../../selectors/UserSelectors';
import loadingSelector from '../../selectors/LoadingSelector';
import { errorsSelector } from '../../selectors/ErrorSelector';
```

```
class HomeContainer extends Component {
  componentDidMount() {
    this.props.fetchUsers();
  }

  render() {
    const { isLoading, errors } = this.props;
    return <Home users={this.state.users} isLoading errors />;
  }
}

HomeContainer.propTypes = {
  login: PropTypes.func.isRequired,
  users: PropTypes.object,
  isLoading: PropTypes.bool.isRequired,
  errors: PropTypes.array,
};

HomeContainer.defaultProps = {
  users: null,
  errors: [],
};

const mapStateToProps = state => ({
  users: getUsers(state),
  isLoading: loadingSelector([actionTypes.GET_USERS])(state),
  errors: errorsSelector([actionTypes.GET_USERS])(state),
});

const mapDispatchToProps = dispatch => ({
  fetchUsers: () => dispatch(getUsersFromNetwork()),
});

export default connect(mapStateToProps, mapDispatchToProps)(HomeContainer);
```

As you may see, the home container has no idea on how the home screen is actually being shown (a list, cells, etc) it just has the logic that is required to show it.

Another **HUGE** benefit from this is that the container is completely reusable in other platforms. In other words, it can be easily reused in a react web application if necessary. Do not underestimate this benefit, React/React Native applications have a lot of potential to reuse most of the logic code.

On the other side, the presentational component (*Home.js*) may look like the following:

```
import React from 'react';
import {
  FlatList,
  Text,
} from 'react-native';
import PropTypes from 'prop-types';
import styles from './styles';

const Home = (props) => {
    const { isLoading, errors, users } = props;
    return (
     <FlatList
       data={users}
       renderItem={({user}) => <Text>{user.name}</Text>}
     />
    );
    // Error and loading management missing here
 }

Home.propTypes = {
  users: PropTypes.object,
  isLoading: PropTypes.bool.isRequired,
  errors: PropTypes.array,
};

Home.defaultProps = {
  users: null,
  errors: [],
};

export default Home;
```

This component has no logic whatsoever, it just receive props and build the UI accordingly. That’s why it’s a stateless component (no state, just a function). It’s not even aware of which state management, networking, or any library we are using, so it’s easily reusable even if big changes happen in the architecture of the application.

Folder structure

As you may know, having an organized folder structure is of utmost importance in a JS based application, so we defined this basic guidelines of how to structure the files and folders.

- **src**: This folder is the main container of all the code inside your application.
    - **actions**: This folder contains all actions that can be dispatched to redux.
    - **assets**: Asset folder to store all images, vectors, etc.
    - **components**: Folder that contains all your application components.
        - **Common**: Folder to store any common component that you use through your app (such as a generic button, textfields, etc).
        - **MyComponent**: Each component should be stored inside it's own folder, and inside it a file for its code and a separate one for the styles. Then, the index.js is only used to export the final component that will be used on the app.
            - MyComponent.js
            - styles.js
            - Index.js
    - **helpers**: Folder to store any kind of helper that you have.
    - **reducers**: This folder should have all your reducers, and expose the combined result using its index.js
    - **selectors**: Folder to store your selectors for each reducer.
    - **controllers**: Folder to store all your network and storage logic (you should have one controller per resource).
    - **App.js**: Main component that starts your whole app.
- **index.js**: Entry point of your application as per React-Native standards.

State Management

As most of the community agrees, **Redux** is a great option to handle the state of a React (native or not) application, so it was an easy decision. It’s widely used, has a lot of support, and provides an elegant way to store the in-memory data.

Nevertheless, like every other framework, it has it’s caveats. If not used correctly, you can end up having way too much boilerplate code that is confusing and makes the codebase really messy.

This happens particularly on situations where we need to save the status (and error) of an asynchronous operation (such as a network request). Having to repeat the same code (create a reducer for the status of the request and the error) again and again is never a good choice.

That's why we were set to find a solution for this problem, and came up with the following mechanism.

Async handling

We came up with the idea of separating everything related to the status of an asynchronous operation from the actual data that we'll save in the store.

If you are familiar with **Redux**, you are familiar with reducers. We created two separated ones, one for saving the current status of every asynchronous operation (if it's loading or not) and one reducer for saving all the errors that may have happened in the mentioned operations.

By defining a convention for the type of the actions, we are able to use this reducers for expressing the status of every asynchronous operation in the application without having to write any more code!

The conventions we defined are the following:

- For the loading reducer, we'll use as the action type *{ACTION_NAME}_REQUEST*
- For the errors reducer, we'll use as the action type *{ACTION_NAME}_ERROR*

For example, the following is the reducer that receives the errors.

```
export default (state = {}, action) => {
  const { type, error } = action;
  const matches = /(.*)_(REQUEST|ERROR)/.exec(type);

  if (!matches) return state;
  const [, requestName, requestState] = matches;
  return {
    ...state,
    [requestName]: requestState === 'ERROR' ? error : null,
  };
};
```

Here, we are using regular expression (good old friends) to check if the action is fact an error. If this is the case, then we save the error in the store, with the request name. Easy!

In the root reducer, we have a loading key and an error key, which are separated from the other entities

```
import { combineReducers } from 'redux';
import loading from './LoadingReducer';
import error from './ErrorReducer';
import user from './UserReducer';

const rootReducer = combineReducers({
  loading,
  error,
  user,
});

export default rootReducer;
```

That enough for the reducers, now we'll see how this is managed on the components side. By using the library Reselect, we are able to access the state errors in a very efficient way.

```
export const errorsSelector = actions => state => actions.reduce(
  (prevState, value) => {
    const error = state.error[`${value}`];
    if (error) {
      prevState.push(error);
    }
    return prevState;
  },
  [],
);
```

Then, when you need a specific error, you only need to call this selector with the correct type:

```
const mapStateToProps = state => ({
  errors: errorsSelector([actionTypes.ACTION])(state),
});
```

Example:

To explain this better, let’s look at a small example. Imagine you need to login a user. This would need a network request to log in, so it’s an ideal case to show how this works:

When the requests starts, we need to save in the store that the request has started, so we dispatch a LOGIN_REQUEST action

```
const loginRequest = () => ({
  type: actionTypes.LOGIN_REQUEST,
});

... Other actions ....

export const login = (email, password) => async (dispatch) => {
  dispatch(loginRequest());
  ... Rest of the code ....
};
```

This would be received by the loading reducer, and save the status of the request.
Then, when the request is finished, we dispatch a success, or an error in case the request failed.

```
const loginError = error => ({
  type: actionTypes.LOGIN_ERROR,
  error,
});

const loginSuccess = user => ({
  type: actionTypes.LOGIN_SUCCESS,
  user,
});

export const login = (email, password) => async (dispatch) => {
  dispatch(loginRequest());
  try {
    const user = await UserController.login(email, password);
    dispatch(loginSuccess(user));
  } catch (error) {
    dispatch(loginError(error.message));
  }
};
```

The errors (or loading reducer) would receive this actions, and will set the corresponding result in the store.

Then, how do we get the status of the request in the component? In the mapStateToProps function that connects the component to **Redux's** store, we use the mentioned selector to get the errors and the status.

```
const mapStateToProps = state => ({
  user: getUser(state),
  isLoading: loadingSelector([actionTypes.LOGIN])(state),
  errors: errorsSelector([actionTypes.LOGIN])(state),
});
```

As you may have already seen, we only have to pass the actionType to both the loading and errors selector since we rely on the previously mentioned conventions to fetch them. Sweet!

Navigation

Once a really complicated issue in React Native applications, navigation between screens is an easier task than it used to be. The community has decided and given support to two main libraries: **react-navigation** and **react-native-navigation**.

Even Though they provide a solution for the same problem, they do it in completely different ways. **react-navigation** is a pure Javascript solution, with all it's pros and cons, such as performance (Native solutions are often more performant) and look & feel (Not exactly the same UI as native components)

On the other hand, **react-native-navigation** performs all the navigation natively, so it doesn't have the problems previously mentioned. Furthermore, it has an amazing API that's really intuitive and is amazingly integrated with **Redux**. In spite of the fact that creating custom navigations may be a little trickier, **react-native-navigation** was the best and most elegant solution we found.

Navigation structure

Even Though the library’s API is dead simple, things can get pretty messy if we do not follow a specific structure for performing and setting the navigation. That’s why we defined several “patterns” or guidelines on how to set up everything.

Registering screens

Inside a file called Navigation.js under the components folder, is where you must register all the screen that are used in the application. For example, if you had a screen Named A and a screen Named B, the Navigation.js file would be:

```
import { Navigation } from 'react-native-navigation';
import A from './A';
import B from './B';

export const Screens = {
  A: 'A',
  B: 'B',
};

export const registerScreens = (store, provider) => {
  // Register all screens of the app
  Navigation.registerComponent(Screens.A, () => A, store, provider);
  Navigation.registerComponent(Screens.B, () => A, store, provider);
};
```

The screens object is used to avoid using directly strings when using the screen’s identifier (As a rule of thumb, strings that are used in several places should be always set as constants somewhere). We export it in order to be able to navigate between screens using this object afterwards.

The *registerScreens* method is the one that actually registers all of our components to be used in the application. This will be imported and called when the application starts, passing the store and provider from **Redux.**

This is were react-native-navigation shines, its integration with Redux is just what you are seeing, pass the store and the provider and it’s ready!

App

In previous experiences, we found that it's common to have several "big navigations" in your app. By "big navigations" we mean flows in the application that correspond to several steps or parts of the same functionality.

For example, it's really common to have an *authentication* flow (login, register, forgot password) and after the user is logged in, have the *home* flow that is entirely separated from it.

The mechanism that we found that's extremely useful for this is to have an *App* class that manages all this flows. This class has a method to start each of the flows with their corresponding navigations and a method to start the entire application.

Continuing with the previous example, a really simple App class may look like the following:

```
import { Navigation } from 'react-native-navigation';
import { Provider } from 'react-redux';
import { Screens, registerScreens } from './components/Navigation';
import { store, persist } from './reducers';

class App {
  constructor(rootStore, provider) {
    this.store = rootStore;
    this.provider = provider;
  }

  startLoggedInApp = () => {
    Navigation.startSingleScreenApp({
      screen: {
        screen: Screens.Home,
      },
      animationType: 'fade',
    });
  }

  startAuth = () => {
    Navigation.startSingleScreenApp({
      screen: {
        screen: Screens.Login,
      },
      animationType: 'fade',
    });
  }

  startApp = () => {
    registerScreens(this.store, this.provider);
    persist(() => {
      this.startLoggedOutApp();
    });
  }
}

export default new App(store, Provider);
```

In this example, we have a startApp method that calls the *registerScreens* method that was mentioned in the previous part. Then we have two methods, one for each big flow in the app.

Whenever you want to start the new app flow (like when the user has logged in) you just import the app class and call the corresponding method.

Networking

There's almost no app that doesn't have to make a network request. That's why finding a correct library for making HTTP calls is important. We are used to using **Axios** (and happy with it), since it has huge support and popularity in the community. It also provides a lot of cool helpers & features that makes life easier.

Regarding the structure of the networking layer, we went with defining controller that perform the network requests.

Controllers

A controller is a class that has the responsibility of performing the network requests of a specific resource. For example, a *UsersController* would be responsible of doing any network request related to a user (fetch all users, delete a user, update a user, login, etc). This controllers are all placed under the controllers folder.

This is done to avoid having network requests all out through the code, which makes it less maintainable. Imagine having several parts of the application that perform the fetch of several users and you realize that you have to add one more header. Yes, it's pretty messy.

Following the mentioned example, a typical controller could be the following:

```
import httpClient from './HttpClient';

class UserController {
  constructor() {
    this.basePath = '/users';
  }

  login = async (email, password) => {
    try {
      const result = await httpClient.post({
        url: `${this.basePath}/session`,
        method: 'POST',
        data: {
          email,
          password,
        },
      });
      return this.user;
    } catch (error) {
      return error;
    }
  }
}

export default new UserController();
```

Why do we find this useful? Well, apart from the previously mentioned benefits, this is an extra layer between our application and the networking solution we are using.

It's not really common to change the entire networking library in an application, but is does happen that for a specific problem you may need a specific solution. This provides an unified interface for the rest of the project, despite what you are using as a solution beneath.

Persistence

It's really common to have to store some data, going from a small token to having many entities with complex relationships between them. That's why we had to contemplate a solution for this situations.

For simple cases, such as saving a token or small pieces of information, we use **Redux Persist**. It's a library that's built on top of Redux, and provides a fantastic and almost transparent way of saving the information that's inside **Redux's** store. Behind the curtains, it uses **AsyncStorage** to save the information, so no magic there.

Whenever we need a more powerful tool, maybe because we need to express complex relationships or we need to do transactional operations over the data, we are in favor of using **Realm**. It's a robust and powerful database that provides a simple SDK to create, maintain and operate on databases in mobile devices.

Standards

As in any big project, following a consistent coding standards helps improve the quality of the overall software. React Native applications are not the exception to this. The most important part of an standard is the consistency in its rules. The end goal is to have a source code that looks as if a single developer wrote it all in single sit down.

The Javascript ecosystem has some wonderful tools to enforce the guidelines. In order to enforce them, we went with **ESLint**, the default library in the JS ecosystem to do this.

Regarding the rules, we didn't want to reinvent the wheel, so we went **ESlint** rules made by **Airbnb** based on their style guide. Some exceptions were made to this rules, which you can find in the .eslintrc.json file.

We recommend installing the corresponding plugin for the code editor or IDE you are using, since seeing the warnings and errors directly there is really helpful. This is the plugin for Atom and this is the plugin for Visual Studio Code. Suit yourself!

Testing

No app should be coded without testing! That's why we defined from the start that we must have a testing environment set up. There were tons of possibilities, but we liked the combination of **Jest** and **Enzyme**.

Jest provides a really complete testing platform for React that requires a really small amount of configuration. It's already integrated whenever a RN project is created, so it was an obvious decision.

Since applications have a strong dependency on the UI, it's always a good idea to test that it's being correctly rendered and it's showing what's expected. That's where **Enzyme** enters. It allows to perform tests over the DOM elements that are rendered and ensure that certain aspects of the UI are carried out.

Configuration

In order to manage the configuration the best way possible, we are using **react-native-config** to handle all the important configuration in one central place. This also provides mechanisms to manage the configuration for different environments (staging, production, etc).

You just need to provide a .env file in the root of the project for it to work. For further information you can check out the library's doc. Remember to **NEVER** push this file to the repository!

How to use it

There's two options on how to set up a new project using the template:

Option 1: Copy the files

The easier option, you just have to copy everything under the /src and the index.js folder into a new react-native project. Of course, dependencies are defined on the package.json file, so you should copy them to the dependencies of the newly created package.json.

Option 2: Use **react-native-rename**

1. Clone this repository to your machine.
2. Change the git project remote URL to one of your own with git remote set-url origin https://github.com/USERNAME/REPOSITORY.git.
3. Install the npm package react-native name (link)
4. Run the following command in the root of the project: `react-native-rename <newName>`
5. Bingo! The project's been renamed with the new name
6. Push it to the repository

Wrapping Up

We have defined a starting point for any React-Native project that we could imagine. Having a standard for any project is always an advantage, both for developers starting work on a project from scratch, as well as for newcomers who diving into a pre-existing project, since React-Native can be really tricky for those guys that have no prior experience. Remember, there’s always an example that puts all of what we’ve been talking to practice

It’s always easier to step into a project knowing where everything should be, which frameworks are being used, and other basic development information. New technologies like this ones have a hectic start where there a lot changes and nothing is specially defined, that’s why this template provides a big advantage and a lot of stability!

One question that we asked ourselves while building this was: Can we use this for any type of software project. The answer is yes! One of our main goals was to keep this project as small and flexible as possible so that it could be used in as many cases as possible – and we succeeded. We hope it helps you succeed too!