Frontend Platform documentation

Table of Contents

Technologies

Node and npm

Project it's based on **Node 8** and **Node Package Manager 5.6.0** (`npm`). Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

Npm is a package manager through which one can install various packages (modules) needed for web development. It's a CLI tool which gives access to it's online repository which contains thousands of open-source Node.js projects. You just need to add a package.json file in your project and mention all the dependencies and run npm install from command line and it will install the selected package. See a list of available scripts on this section

Webpack and Babel

The application take advantage of Webpack and Babel:

- **Webpack** bundles all our JavaScript source code files into one bundle (including custom configured build steps)
- **Babel** is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environment

That's why Babel is also needed for React, because JSX -- React's syntax -- and its file extension .jsx, aren't natively supported. React components are mostly written in JavaScript ES6. ES6 is a nice improvement over the language but older browsers cannot understand the new syntax. A webpack loader takes something as the input and produces something else as the output. In our case the `babel-loader` is the Webpack loader responsible for taking in in the ES6 code and making it understandable by the browser of choice.

Available Scripts

Here's a list of available scripts to install new libraries and run the project.

In the project directory, you can run:

`npm install`

Install the dependencies in the local node_modules folder. In global mode (ie, with -g or --global appended to the command), it installs the current package context (ie, the current working directory) as a global package. More information in this section

`npm start`

Runs the app in the development mode.
Open http://localhost:3000 to view it in the browser.

The page will reload if you make edits.
You will also see any lint errors in the console.

`npm run build`

Builds the app for production to the `build` folder.
It correctly bundles React in production mode and optimizes the build for the best performance.

The build is minified and the filenames include the hashes.
Your app is ready to be deployed!

Folder Structure

BOOM WebApp project presents the following folder structure:

```
app/
  README.md
  node_modules/
  package.json
```

```
  public/
    index.html
    favicon.ico
  src/
    api/
    assets/
    components/
    config/
    hoc/
    redux/
    routes/
    service/
    styles/
    tranlsations/
    utils/
```

For the project to build, **these files must exist with exact filenames**:

- `public/index.html` is the page template;
- `src/index.js` is the JavaScript entry point.

You can delete or rename the other files.

You may create subdirectories inside `src`. For faster rebuilds, only files inside `src` are processed by Webpack.
You need to **put any JS and CSS files inside `src`**, otherwise Webpack won't see them.

Only files inside `public` can be used from `public/index.html`.
Read instructions below for using assets from JavaScript and HTML.

api

This folder contains the Axios instances and enpoints used by the application to make REST calls. We will discuss it in details in this section

assets

This folder contains all the assets of the application including application fonts

components

This folder contains all the React Components used in the application. Every new component that it's not representing an application route must be created in this folder.

config

This folder contains three main files

- configurations.js: this file contains methods to elaborate at run time variables like cliend id/secret, google API keys, etc. Variables are extracted taking advantage of the REACT_APP_PROFILE Node Env Variable
- consts.js: this files contains usefull constants and maps used across the application
- utils.js: this files contains usefull utils function used across the application

hoc

This folder contains the requireAuthentication High Order Component. This component wraps the application routes that requires an authentication to be accessed and handle this condition.

redux

This folder contains Redux Actions and Reducers of the application. This is the core of the application logic and where the application state is handled.

routes

This folder contains all the React Components that represents the main views and routes of the application. All this kind of Components are distinguished by the View suffix.

service

This folder contains the OrganizationService file. This is used by the ACL module to handle fetching organizations from the Backend.

styles

As the name suggests this folder containts the main .css files handling the style of components and views of the application

translations

This folder containts the i18next translation file. Currently the application support English and Italian languages.

utils

The utils folder contains functions and classes to handle platform Access Control Levels. They will be discribed in further sections

Supported Browsers

By default, the generated project supports all modern browsers, however the application it's optimized to work with Google Chrome.
Support for Internet Explorer 9, 10, and 11 requires polyfills.

Supported Language Features

This project supports a superset of the latest JavaScript standard.
In addition to ES6 syntax features, it also supports:

- Exponentiation Operator (ES2016).
- Async/await (ES2017).
- Object Rest/Spread Properties (ES2018).
- Dynamic import() (stage 3 proposal)
- Class Fields and Static Properties (part of stage 3 proposal).
- JSX and Flow syntax.

Learn more about different proposal stages.

While we recommend using experimental proposals with some caution, Facebook heavily uses these features in the product code, so we intend to provide codemods if any of these proposals change in the future.

Note that **this project includes no polyfills** by default.

If you use any other ES6+ features that need **runtime support** (such as `Array.from()` or `Symbol`), make sure you are including the appropriate polyfills manually, or that the browsers you are targeting already support them.

Syntax Highlighting in the Editor

To configure the syntax highlighting in your favorite text editor, head to the relevant Babel documentation page and follow the instructions. Some of the most popular editors are covered. We also took advantage of ESLint with AirBnb extension, if you're using VSCode you can find a setup guide here.

Debugging in the Editor

This feature is currently only supported by Visual Studio Code

Visual Studio Code support debugging out of the box with Create React App. This enables you as a developer to write and debug your React code without leaving the editor, and most importantly it enables you to have a continuous development workflow, where context switching is minimal, as you don’t have to switch between tools.

Visual Studio Code

You would need to have the latest version of VS Code and VS Code Chrome Debugger Extension installed.

Then add the block below to your `launch.json` file and put it inside the `.vscode` folder in your app’s root directory.

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Chrome",
      "type": "chrome",
      "request": "launch",
      "url": "http://localhost:3000",
      "webRoot": "${workspaceRoot}/src",
      "sourceMapPathOverrides": {
        "webpack:///src/*": "${webRoot}/*"
      }
    }
  ]
}
```

> Note: the URL may be different if you've made adjustments via the HOST or PORT environment variables.

Start your app by running `npm start`, and start debugging in VS Code by pressing `F5` or by clicking the green debug icon. You can now write code, set breakpoints, make changes to the code, and debug your newly modified code—all from your editor.

Having problems with VS Code Debugging? Please see their troubleshooting guide.

Chrome Debugger Extension and react-logger

Using the library `redux-logger` you'll be able to log all the Redux State changes in the **Console** tab of the Chrome Debugger Extensions. This is extremely helpful to debug Redux Actions and Reducers.

> Note: Using the logger will drastically drop the application performance

In order to avoid poor performances in the Production environment the logger enabled only for other enviroments via the followind snippet in the `ìndex.js` file

```
if (process.env.NODE_ENV !== 'production') {
  middlewares.push(logger);
}
```

You can extends this condition based on your enviroments

Installing a Dependency

The generated project includes React and ReactDOM as dependencies. It also includes a set of scripts used by Create React App as a development dependency. You may install other dependencies (for example, React Router) with `npm`:

```
npm install --save react-router-dom
```

Alternatively you may use `yarn`:

```
yarn add react-router-dom
```

This works for any library, not just `react-router-dom`.

Importing a Component

This project setup supports ES6 modules thanks to Webpack.
While you can still use `require()` and `module.exports`, we encourage you to use `import` and `export` instead.

For example:

`Button.js`

```
import React, { Component } from 'react';

class Button extends Component {
  render() {
    // ...
  }
}

export default Button; // Don't forget to use export default!
```

`DangerButton.js`

```
import React, { Component } from 'react';
import Button from './Button'; // Import a component from another file

class DangerButton extends Component {
  render() {
    return <Button color="red" />;
  }
}

export default DangerButton;
```

Be aware of the difference between default and named exports. It is a common source of mistakes.

We suggest that you stick to using default imports and exports when a module only exports a single thing (for example, a component). That's what you get when you use `export default Button` and `import Button from './Button'`.

Named exports are useful for utility modules that export several functions. A module may have at most one default export and as many named exports as you like.

Learn more about ES6 modules:

- When to use the curly braces?
- Exploring ES6: Modules
- Understanding ES6: Modules

Adding a Stylesheet

This project setup uses Webpack for handling all assets. Webpack offers a custom way of "extending" the concept of `import` beyond JavaScript. To express that a JavaScript file depends on a CSS file, you need to **import the CSS from the JavaScript file**:

`Button.css`

```
```

```
.Button {
  padding: 20px;
}
```

`Button.js`

```
import React, { Component } from 'react';
import './Button.css'; // Tell Webpack that Button.js uses these styles

class Button extends Component {
  render() {
    // You can use them as regular CSS styles
    return <div className="Button" />;
  }
}
```

This is not required for React but many people find this feature convenient. You can read about the benefits of this approach here. However you should be aware that this makes your code less portable to other build tools and environments than Webpack.

In development, expressing dependencies this way allows your styles to be reloaded on the fly as you edit them. In production, all CSS files will be concatenated into a single minified `.css` file in the build output.

If you are concerned about using Webpack-specific semantics, you can put all your CSS right into `src/index.css`. It would still be imported from `src/index.js`, but you could always remove that import if you later migrate to a different build tool.

Material UI

BOOM Platform take massively advantage of the Material UI library for components an styling. The library it's a React extension of the basic Material UI, learn about the React Material UI here. Additionally we use the magnificent Material UI Icons set, learn more about it here

Material UI Inline Style

Material UI introduce a different type of inline style for components. The style it's defined as a `const` in the component scope and than the componets access to it via the `classes` props. This props is provided to the component via the `withStyles` high order component. You can read more about Material UI Style here

```
import { withStyles } from '@material-ui/core/styles';

const styles = theme => ({
  boxContainer: {
    marginRight: 20,
    color: '#80888d',
  },
  textStyle: {
```

```
    fontSize: 20,
    color: 'red',
  }
)}

const Box = ({ classes}) => (
  <div className={classes.divStyle}>
    <p className={classes.textStyle}>Material UI</p>
  </div>
);
export default withStyles(styles)(Box);
```

Adding a CSS Modules Stylesheet

This project supports CSS Modules alongside regular stylesheets using the `[name].module.css` file naming convention. CSS Modules allows the scoping of CSS by automatically creating a unique classname of the format `[filename]\_[classname]\_\_[hash]`.

> **Tip:** Should you want to preprocess a stylesheet with Sass then make sure to follow the installation instructions and then change the stylesheet file extension as follows: `[name].module.scss` or `[name].module.sass`.

CSS Modules let you use the same CSS class name in different files without worrying about naming clashes. Learn more about CSS Modules here.

`Button.module.css`

```
.error {
  background-color: red;
}
```

`another-stylesheet.css`

```
.error {
  color: red;
}
```

`Button.js`

```
import React, { Component } from 'react';
import styles from './Button.module.css'; // Import css modules stylesheet
as styles
import './another-stylesheet.css'; // Import regular stylesheet

class Button extends Component {
  render() {
```

```
    // reference as a js object
    return <button className={styles.error}>Error Button</button>;
  }
}
```

Result

No clashes from other `.error` class names

```
<!-- This button has red background but not red text -->
<button class="Button_error_ax7yz"></div>
```

This is an optional feature. Regular `<link>` stylesheets and CSS files are fully supported. CSS Modules are turned on for files ending with the `.module.css` extension.

Adding Images, Fonts, and Files

With Webpack, using static assets like images and fonts works similarly to CSS.

You can **`import` a file right in a JavaScript module**. This tells Webpack to include that file in the bundle. Unlike CSS imports, importing a file gives you a string value. This value is the final path you can reference in your code, e.g. as the `src` attribute of an image or the `href` of a link to a PDF.

To reduce the number of requests to the server, importing images that are less than 10,000 bytes returns a data URI instead of a path. This applies to the following file extensions: bmp, gif, jpg, jpeg, and png. SVG files are excluded due to #1153.

Here is an example:

```
import React from 'react';
import logo from './logo.png'; // Tell Webpack this JS file uses this
image

function Header() {
  // Import result is the URL of your image
  return <img src={logo} alt="Logo" />;
}

export default Header;
```

This ensures that when the project is built, Webpack will correctly move the images into the build folder, and provide us with correct paths.

This works in CSS too:

```
```

```
.Logo {
  background-image: url(./logo.png);
}
```

Webpack finds all relative module references in CSS (they start with `./`) and replaces them with the final paths from the compiled bundle. If you make a typo or accidentally delete an important file, you will see a compilation error, just like when you import a non-existent JavaScript module. The final filenames in the compiled bundle are generated by Webpack from content hashes. If the file content changes in the future, Webpack will give it a different name in production so you don't need to worry about long-term caching of assets.

Please be advised that this is also a custom feature of Webpack.

It is not required for React but many people enjoy it (and React Native uses a similar mechanism for images). An alternative way of handling static assets is described in the next section.

Adding SVGs

> Note: this feature is available with `react-scripts@2.0.0` and higher.

One way to add SVG files was described in the section above. You can also import SVGs directly as React components. You can use either of the two approaches. In your code it would look like this:

```
import { ReactComponent as Logo } from './logo.svg';
const App = () => (
  <div>
    {/* Logo is an actual React component */}
    <Logo />
  </div>
);
```

This is handy if you don't want to load SVG as a separate file. Don't forget the curly braces in the import! The `ReactComponent` import name is special and tells Create React App that you want a React component that renders an SVG, rather than its filename.

Using the `public` Folder

Changing the Page `<title>`

You can find the source HTML file in the `public` folder of the generated project. You may edit the `<title>` tag in it to change the title from "React App" to anything else.

Note that normally you wouldn't edit files in the `public` folder very often. For example, adding a stylesheet is done without touching the HTML.

If you need to dynamically update the page title based on the content, you can use the browser `document.title` API. For more complex scenarios when you want to change the title from React components, you can use React Helmet, a third party library.

If you use a custom server for your app in production and want to modify the title before it gets sent to the browser, you can follow advice in this section. Alternatively, you can pre-build each page as a static HTML file which then loads the JavaScript bundle, which is covered here.

Changing the HTML

The `public` folder contains the HTML file so you can tweak it, for example, to set the page title. The `<script>` tag with the compiled code will be added to it automatically during the build process.

Adding Assets Outside of the Module System

You can also add other assets to the `public` folder.

Note that we normally encourage you to `import` assets in JavaScript files instead. For example, see the sections on adding a stylesheet and adding images and fonts. This mechanism provides a number of benefits:

- Scripts and stylesheets get minified and bundled together to avoid extra network requests.
- Missing files cause compilation errors instead of 404 errors for your users.
- Result filenames include content hashes so you don't need to worry about browsers caching their old versions.

However there is an **escape hatch** that you can use to add an asset outside of the module system.

If you put a file into the `public` folder, it will **not** be processed by Webpack. Instead it will be copied into the build folder untouched. To reference assets in the `public` folder, you need to use a special variable called `PUBLIC_URL`.

Inside `index.html`, you can use it like this:

```
<link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
```

Only files inside the `public` folder will be accessible by `%PUBLIC_URL%` prefix. If you need to use a file from `src` or `node_modules`, you'll have to copy it there to explicitly specify your intention to make this file a part of the build.

When you run `npm run build`, Create React App will substitute `%PUBLIC_URL%` with a correct absolute path so your project works even if you use client-side routing or host it at a non-root URL.

In JavaScript code, you can use `process.env.PUBLIC_URL` for similar purposes:

```
render() {
  // Note: this is an escape hatch and should be used sparingly!
  // Normally we recommend using `import` for getting asset URLs
  // as described in "Adding Images and Fonts" above this section.
  return <img src={process.env.PUBLIC_URL + '/img/logo.png'} />;
}
```

Keep in mind the downsides of this approach:

- None of the files in `public` folder get post-processed or minified.
- Missing files will not be called at compilation time, and will cause 404 errors for your users.
- Result filenames won't include content hashes so you'll need to add query arguments or rename them every time they change.

When to Use the `public` Folder

Normally we recommend importing stylesheets, images, and fonts from JavaScript. The `public` folder is useful as a workaround for a number of less common cases:

- You need a file with a specific name in the build output, such as `manifest.webmanifest`.
- You have thousands of images and need to dynamically reference their paths.
- You want to include a small script like `pace.js` outside of the bundled code.
- Some library may be incompatible with Webpack and you have no other option but to include it as a `<script>` tag.

Note that if you add a `<script>` that declares global variables, you also need to read the next section on using them.

Adding Bootstrap

The BOOM platform doesn't use any bootstrap extension. However we recommend to use reactstrap together with React. It is a popular library for integrating Bootstrap with React apps. If you need it, you can integrate it with Create React App by following these steps:

Install reactstrap and Bootstrap from npm. reactstrap does not include Bootstrap CSS so this needs to be installed as well:

```
npm install --save reactstrap bootstrap@4
```

Alternatively you may use `yarn`:

```
yarn add bootstrap@4 reactstrap
```

Import Bootstrap CSS and optionally Bootstrap theme CSS in the beginning of your `src/index.js` file:

```
import 'bootstrap/dist/css/bootstrap.css';
// Put any other imports below so that CSS from your
// components takes precedence over default styles.
```

Import required reactstrap components within `src/App.js` file or your custom component files:

```
import { Button } from 'reactstrap';
```

Now you are ready to use the imported reactstrap components within your component hierarchy defined in the render method. Here is an example `App.js` redone using reactstrap.

Adding Custom Environment Variables

As stated before your project can consume variables declared in your environment as if they were declared locally in your JS files. By default you will have `NODE_ENV` defined for you, and any other environment variables starting with `REACT_APP_`.

The environment variables are embedded during the build time. Since Create React App produces a static HTML/CSS/JS bundle, it can't possibly read them at runtime. To read them at runtime, you would need to load HTML into memory on the server and replace placeholders in runtime, just like described here. Alternatively you can rebuild the app on the server anytime you change them.

> Note: You must create custom environment variables beginning with `REACT_APP_`. Any other variables except `NODE_ENV` will be ignored to avoid accidentally exposing a private key on the machine that could have the same name. Changing any environment variables will require you to restart the development server if it is running.

These environment variables will be defined for you on `process.env`. For example, having an environment variable named `REACT_APP_SECRET_CODE` will be exposed in your JS as `process.env.REACT_APP_SECRET_CODE`.

There is also a special built-in environment variable called `NODE_ENV`. You can read it from `process.env.NODE_ENV`. When you run `npm start`, it is always equal to `'development'`, when you run `npm test` it is always equal to `'test'`, and when you run `npm run build` to make a production bundle, it is always equal to `'production'`. **You cannot override `NODE_ENV` manually.** This prevents developers from accidentally deploying a slow development build to production.

These environment variables can be useful for displaying information conditionally based on where the project is deployed or consuming sensitive data that lives outside of version control.

First, you need to have environment variables defined. For example, let's say you wanted to consume a secret defined in the environment inside a `<form>`:

```
render() {
  return (
    <div>
      <small>You are running this application in <b>{process.env.NODE_ENV}
</b> mode.</small>
      <form>
        <input type="hidden" defaultValue=
{process.env.REACT_APP_SECRET_CODE} />
      </form>
    </div>
  );
}
```

During the build, `process.env.REACT_APP_SECRET_CODE` will be replaced with the current value of the `REACT_APP_SECRET_CODE` environment variable. Remember that the `NODE_ENV` variable will be set for you automatically.

When you load the app in the browser and inspect the `<input>`, you will see its value set to `abcdef`, and the bold text will show the environment provided when using `npm start`:

```
<div>
  <small>You are running this application in <b>development</b> mode.
</small>
  <form>
    <input type="hidden" value="abcdef" />
  </form>
</div>
```

The above form is looking for a variable called `REACT_APP_SECRET_CODE` from the environment. In order to consume this value, we need to have it defined in the environment. This can be done using two ways: either in your shell or in a `.env` file. Both of these ways are described in the next few sections.

Having access to the `NODE_ENV` is also useful for performing actions conditionally, as we see before for activate the redux-logger in BOOM platform:

```
if (process.env.NODE_ENV !== 'production') {
  middlewares.push(logger);
}
```

When you compile the app with `npm run build`, the minification step will strip out this condition, and the resulting bundle will be smaller.

Redux Actions and Reducers

Reducers

Redux it's the perfect tool to handle your application state. The whole state of your app is stored in an object tree inside a single store. The only way to change the state tree is to emit an action, an object describing what happened. To specify how the actions transform the state tree, you write pure reducers like the one in this example

```
/* user.reducers.js (simplified) */

import Immutable from 'seamless-immutable';

import {
  SAVE_USER_DATA,
} from '../actions/actionTypes/user';

const initialState = Immutable({
```

```
  userData: {},
});

export default function (state = initialState, action) {
  switch (action.type) {
    case SAVE_USER_DATA:
      return state.set('data', action.userData);
    default:
      return state;
  }
}
```

This is a reducer, a pure function with (state, action) => state signature. It describes how an action transforms the state into the next state. The shape of the state is up to you: it can be a primitive, an array, an object or even an Immutable.js data structure like in the example. The only important part is that you should not mutate the state object, but return a new object if the state changes. In this example, we use a `switch` statement and strings to determine how to react to an external action and wich part of the state should mutate and return. So instead of mutating the state directly, you specify the mutations you want to happen with plain objects called `actions`. Then you write a special function called a `reducer` to decide how every action transforms the entire application's state.

In a typical Redux app, there is just a single store with a single root reducing function. As your app grows, you split the root reducer into smaller reducers independently operating on the different parts of the state tree, and this is exactly like how we handle the state of the BOOM application.

You can find all the reducers under the `src/redux/reducers` folder.

Actions

To change something in the state, you need to `dispatch` an `action`. An action can be triggered for example by pressing a button SAVE in the application and enforcing that every change is described as an action lets us have a clear understanding of what's going on in the app. If something changed, we know why it changed. Actions are like breadcrumbs of what has happened. Finally, to tie state and actions together, we write a function called a reducer.

As best practice, for every reducer there's a set of corresponding actions that operate only in the scope of that reducer. For example the set actions that operates on the `user.reducer`, so on the user portion of state, are collected in the `user.actions` file on the `src/redux/actions`. Here are an example of a user action:

```
export function saveUserData(userData) {
  return {
    type: SAVE_USER_DATA,
    userData,
  };
}
```

As you can see an action is just plain JavaScript object that describes what happened. It has a `type` field (VERY IMPORTANT) and the other custom field that will be read by the reducer.

> Note: All the action types of the application are defined in ordered separated files under the `src/redux/actions/actionTypes` folder.

So what it's happening here it's the following:

- We are `dispatching` and action of the `type` SAVE_USER_DATA
- This action will arrive to the `User Reducers`
- The reducer pure function will switch on the action `type`
- If the specific `action type` case it's handled, the reducer will return the mutated state

Asynchronous Actions with redux-thunk

The Action presented in the example it's a `synchronous action` meaning that the every time this action it's dispatched, the state will update immediatly. When you call an asynchronous API for example, there are two crucial moments in time: the moment you start the call, and the moment when you receive an answer (or a timeout).

`Redux Thunk` middleware allows you to write action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met. The inner function receives the store methods `dispatch` and `getState` as parameters. Here's a version of the previous action using Redux Thunk

```
export function saveUserData(userData, saveToLocalStorage = false) {
  return (dispatch, getState) => {
    /* PERFORM SOME HACKING HERE AND THEN DISPATCH */
    dispatch({
      type: SAVE_USER_DATA,
      userData,
    });
  };
}
```

This is fundamental when we need to make an asynchronous API call, we will cover this topic in the next section so stay tuned!

> Note: All this information are deeply covered in the beautiful official Redux Guide

Fetching Data with API Requests

Axios instance and interceptors

In BOOM project axios allows you to make REST Call to the backend. Under the /api folder you can find the initialization of the Axios Instance in the `boomInstance.js` file. An Axios Instance it's normally defined by its **base URL**, this is the URL that will be used by the instance. In our case the URL it's dynamically generated based on the NODE ENVIRONMENT VARIABLE. Another important Axios component are **interceptors**. A request or response insterceptor are functions that literally intercepts Axios API before they are handled by the

calling Promise. An example of interceptor usage in the BOOM projects it's when setting the authorization header token for our API calls.

```
export function setRequestInterceptor(accessToken) {
  return axiosBoomInstance.interceptors.request.use(
    (config) => {
      config['headers']['authorization'] = `Bearer ${accessToken}`;
      return config;
    });
}
```

> Note: since the axiosBoomInstance it's a global constant interceptors, base URL and other config properties will be globally set across the application

You can also remove one or more interceptors if you need, for example after user logout.

```
export function interceptorEjectRequest() {
  _.each(
    _.keys(axiosBoomInstance.interceptors.request.handlers), (key) => {
      axiosBoomInstance.interceptors.request.eject(key);
    }
  );
}
```

Making a request

This is an example of a GET HTTP request using the `boomInstance` previously instantiated.

```
export function fetchUser(organizationId) {
  return
axiosBoomInstance.get(`/api/${API_VERSION}/organizations/${organizationId}
/users/me`);
}
```

Every axios call always return a `Promise`, so in order to handle it we wrap whe api call inside a Redux Action. This way If the Promise resolve we can user the results and for example, save it in the Application State, otherwise we handle the returned error. Here's an example of the wrapping Asynchronous Redux Action.

```
export function fetchUser() {
  return async (dispatch, getState) => {
    try {
      /* THE FETCHING! */
      const response = await UserAPI.fetchUser(organization);
      if (response && response.data) {
        const { data: userData } = response;
```

```
        /* THE ACTION!! */
        dispatch({
          type: SAVE_USER_DATA,
          userData,
        });
        return userData;
      }
      throw new Error();
    } catch (error) {
      throw error;
    }
  };
}
```

In BOOM application we follow the `async/await` Promise syntax and wrap it inside a `try/catch` block as common best practice. As in the example, we are **awaiting** the Promise resolve and save it's results in the `response` variable, then we extract the content, save it and return it. If the response it's `undefined` or not contains the data an `Error` it's throwned. Eventually, if the Axios Promise it's rejected the code will fall on the `catch` block, throwing the returned error.

> Note: In the Action in example, we are returning an `async` function. This is because the `await` operator can only be used inside an async function

> Note: If you don't `await` a promise, the code execution will continue without wait for the Promise resolution leading to unexpected results. NEVER forget to `await` your Promises!

Handling Translations

Translations are handled by the i18next framework. The translations file can be found on the `/src/translations/i18next`. The file contains the `i18next.init`Promise where the framework it's initialized along with all the translations per language.

```
i18next
  .init({
    interpolation: {
      escapeValue: false,
    },
    fallbackLng: 'en',
    resources: {
      en: { translation: {/* english translations */}},
      it: { translation: {/* italian translations */}},
      ...other languages
    }
  });
```

The language used by i18next in the BOOM Web Application it's derived by the user default language stored in its data. After fetching the user data the framework language it's set this way:

```
i18next.changeLanguage(language);
```

To use the translations inside the application you just need to import the file and access the correct translation index. We use the t function provided by the i18next API. Let's see an example

```
/* /src/translations/i18next.js */
....,
en: {
  translation: {
    login: {
      welcome: 'Welcome!',
    },
  }
},
it: {
  translation: {
    login: {
      welcome: 'Benvenuto!',
    },
  }
}

/* In any Application Component */

import translations from '../../translations/i18next';

/* then in the render function */
<h4>{translation.t('login.welcome')}</h4> // Welcome! or Benvenuto!
```

The framework it's very flexible and provides a huge amount of features, like handling the gender and the plurals. For more information please consult the main online documentation.

Routing with react-router-dom

Application Routing its managed with react-router-dom library.

The entry point of the whole application it's `index.js` file, in particular

```
ReactDOM.render(<AppRouter store={store} />,
document.getElementById('root'));
```

We call this a “root” DOM node because everything inside it will be managed by React DOM. Applications built with just React usually have a single root DOM node. To render a React element into a root DOM node you need pass both to ReactDOM.render(), in our example we are passing the `AppRouter` component inside the root DOM node.

All the components responsible for app navigation can be found under the folder `src/routes/NavigationComponents`.

The `AppRouter.js` component contains all the main routes of the application. It contains a set of external views like Login and Set Password views, and a `Switch` that handle the inner navigation. A `<Switch>` looks through all its children elements and renders the first one whose path matches the current URL. Use a any time you have multiple routes, but you want only one of them to render at a time.

The Route component is perhaps the most important component in React Router to understand and learn to use well. Its most basic responsibility is to render some UI when its path matches the current URL. In the BOOM application every Route (we call them View) it's defined either as `AppRoute` or `AuthRoute`.

```
/* AppRoute.jsx */
const AppRoute = ({ component: Component, layout: Layout, ...rest }) => (
  <Route
    {...rest}
    render={props => (
      <Layout>
        <Component {...props} />
      </Layout>
    )}
  />
);
```

While the first consists in a simple layout, the second it's a View only accesible from authenticated users. In order to do that we wrapped the simple AppRoute component with the `requireAuthentication` **High Order Component (HOC)** that checks the validity of user token along with its permissions every time the wrappe View it's visited.

```
/* AuthRoute.jsx */
// It's just an HOC for AppRoute
export default withRouter(requireAuthentication(checkAuth)(AppRoute));
```

A simple AuthRoute presents this props:

- path: it's the URL path that will match this view
- layout: it's the layout that will wrap the component
- component: it's the View that will be displayed for this Route.

```
/* AppRouter */
<AuthRoute path="/calendar" layout={HeaderLayout} component={CalendarView}
/>
```

You can access the different layouts and route components from the `src/routes` folder.

Changing route

You can change route, get access to the history properties and the closest match via the `withRouter` **Higher-order Component**. The HOC withRouter will pass updated match, location, and history props to the wrapped component whenever it renders. Then you can use the history props to push a new Route in the navigation stack, an in the example

```
import { withRouter } from 'react-router-dom';

MyComponent = () => {
  const { history } = this.props;
  history.push('/otherView');
}
export default (withRouter(MyComponent));
```

Forms with redux-form

All the application forms are managed via the redux-form library. As the name suggests, this library allows to handle forms via your Redux State Store.

The first step it's to add a special Reducer to your reducers list, this way you can access to every form property from your application State.

```
/* index.js in /src/redux/reducers */
import { reducer as formReducer } from 'redux-form';

const rootReducers = combineReducers({
  form: formReducer,
  ....
});
```

Fields

The `Field` component its the building block of every Redux Form. This component is how you connect each individual input to the Redux store. There are three fundamental things that you need to understand in order to use Field correctly:

- The name prop is required. It is a string path, in dot-and-bracket notation, corresponding to a value in the form values. It may be as simple as 'firstName' or as complicated as 'contact.billing.address[2].phones[1].areaCode'.
- The component prop is required. It may be a Component, a stateless function, or a string name of one of the default supported DOM inputs (input, textarea or select).
- All other props will be passed along to the element generated by the component prop.

Let's see a simple example of text input.

```
  <Field
    name="username"
    component={MDTextInputField}
  />
```

All the Form Components defined in the application, like text input, text area, checkbox, radio buttons can be found under the folder `/src/components/Forms/FormComponents`. Usually for every form component we need we define it using two different components with a very specific naming convention:

- a simple **[componentName]View** Component that it's a responsible for rendering the component UI
- a **[componentName]Field** Component that it's rensposible to handle the Redux Form logic of the component.

So referring to the previous example, in the folder we have listed you will find:

- MDTextInputView: the text input UI
- MDTextInputField: the text input logic

> Note: the MD prefix it's a naming convention indicating that the component it's defined using Material Design

We will not go deeply in details on all the props that can be defined for Field since the redux-form documentation it's very specific and complete. However we want to stress one of the main props defined for the Field component that is the `onChange` prop. This is a callback that will be called whenever an onChange event is fired from the underlying input. If you don't invoke this callback the Redux Form will never be aware that the input value has changed.

```
input.onChange(value);
```

Form definition and validation

Now that we have defined our Field Components, we can combine them to create a Form. Let's see and example of a simplified Login Form.

```
/* The validation function */
import { submit } from 'redux-form';

const validate = (values) => {
  const errors = {};
  if (!values.username) {
    errors.username = translations.t('forms.required');
  }
  if (!values.password) {
    errors.password = translations.t('forms.required');
  }
  return errors;
};
```

```
/* The form UI */
const LoginForm = ({ dispatch}) => (
  <div>
    <Field
      name="username"
      component={MDTextInputField}
    />
    <Field
      name="password"
      component={MDTextInputField}
    />
    <Button
      onPress={dispatch(submit('LoginForm'))}
    />
  </div>
);

/* The export */
export default connect()(reduxForm({
  form: 'LoginForm',
  validate,
})(LoginForm));
```

This simple form it's composed by three main blocks:

- The `validate` function will be invoked when the form is submitted. You can create your own custom validation criterias. You can read more on this topic here
- The form UI it's formed by different Fields component and a button. Note that the button will `dispatch` a submit Redux Action that will trigger the form validation. If the validation function return no error the form is submitted successfully.
- The export part is where you define the form name, LoginForm in this case and the validate function that will be used.

Every Redux Form need to define a specific callback called `onSubmit`. This is the callback that will be invoked if the validate function returns no errors after a submit action has been dispatched. To handle this callback we use the Form component inside a specific view that will handle the submit function.

```
class LoginView extends React.Component {

  onLoginSubmit(loginData) {
    /* do some hack here */
  }

  render() {
    return (
      <div>
        <h4> This is the view that contains the form </h4>
        <LoginForm
```

```
            onSubmit={loginData => this.onLoginSubmit(loginData)}
          />
        </div>
      );
    }
  }
```

Redux Form it's a very powerfull library, with a lot of options, properties and flows. Please refer to the documentation to handle more specific of difficult cases.

Permissions and ACL

In this section we will cover how to handle permissions and ACL in the BOOM web application. Please use the provided **external documentation** to see in details how the ACL logic and roles are defined in the BOOM application. A User belongs to an Organization and have visibility and different privileges over a subset (or all) of the Companies of his Organization. Since Users are not tightly bound to a Company, access policies can be defined with high flexibility at any level of the Organization hierarchy.

Access rules

A User belongs to an Organization and have visibility and different privileges over a subset (or all) of the Companies of his Organization. Since Users are not tightly bound to a Company, access policies can be defined with high flexibility at any level of the Organization hierarchy.

The actions a User can do over entities are defined by his **Roles**. Roles define a set of **Permissions**, logical aggregation of actions that can be performed on items. They also have a level, which allows to have a priority order over Roles. The effective binding between Roles and Companies (and consequently items) is expressed through an Access Rule, which is a set of User, Company and Role.

Every Permission it's defined by:

- a PERMISSION ACTION (ABILITIES): that is a specific action that a user can perfom, like read, update, delete, etc..
- an ENITITY: that is the object on which an action will be performed.

You can find the set of permission actions and entities on the `conts.js` file. To check if a user in a specific organization or company has the right to perform a specific action on a certain entity we defined an `Ability Helper`. This `interface` provides a set of usefull functions and usually get as input a set of permissions , the user abilities, a user role and one or more entities. This interface is exposed via the `Ability Provider` instantiated in the `user.actions`, that is when the user role and permission is fetched from the Backend.

```
/* user.actions.js */
export function setUserCompanyAbilityProvider(companyId) {
  return (dispatch) => {
    const rolePermissions =
dispatch(UtilsActions.getUserRoleAndPermissionsWithinCompany(companyId));
    const abilityProviderHelper =
AbilityProvider.getCompanyAbilityHelper();
    abilityProviderHelper.updateAbilities(rolePermissions.permissions);
```

```
      abilityProviderHelper.setUserRole(rolePermissions.role);
    };
  }
```

Once the Provider it's instantiated, we use it all across the application everytime we need to check if a user can perform an action or see a specific section of the application. To do that we created the `Permission` component.

This HOC component get as input the following properties:

- **do**: The set of abilities that has to be checked
- **on**: The set of entities against we are control the permission
- **abilityHelper**: The ability helper we are using

If the ability helper detects that the user has not the provided ability for the selected entity the Permission component will not render the inner component that its wrapping. Otherwise the inner component will be rendered correctly.

```
/* Permission.jsx */

<Permission do={[PERMISSIONS.CANCEL]} on={PERMISSION_ENTITIES.SHOOTING}
abilityHelper=    {AbilityProvider.getOrganizationAbilityHelper()}>
    <Button
      title="DELETE SHOOTING"
      onClick={() => onCancelShooting()}
    />
  </Permission>
```

In the example below the Button component will be rendered only if the current user has the ability to cancel a shooting within the organization.

For more coarse grain actions inside the application we take advantage of two conditions that can be easily deduced while fetching the user data on login.

- `isBOOM`: The first condition is if a user belongs to the BOOM organization, this condition is verified if the user's organization id it's equal to 1
- `isPhotographer`: This condition is verified by decoding the user JWT token

While is not recommended to use these conditions, there are different cases where is necessary to distinguish if the user is a photographer or a boom user before using the `<Permission>` component and take advantage of the defined ACL.