Pager-Project

Web application, SettingsAction.js

Component Description

Abstract

This document describes the required behaviour and interfaces of SettingsAction, a component part of the Pager-web application. The component is responsible for fetching and sending data to a remote backend which stores user-specific settings and account passwords. A firebase interface is required for the component to function. The internal structure consists of multiple sub-components, each fulfilling a part of the component requirements.

Version History

Date	Version	Author	Description
27/04/2020	1.0.0	Abyel Tesfay	First draft

An Essential Unified Process Document

Table of Contents

1 Introduction

1.1 Document Purpose

The purpose of this document is to describe the behaviour and interfaces of the *SettingsAction* component, a component responsible for letting the system manage users from a remote backend, which keeps track of existing users. The backend also provides an external database which the component uses to fetch or store application-related data. The operations the component provides is triggered by certain actions on the web *application*. The backend is provided by firebase, a cloud-platform service by Google.

This document is intended for web developers and interested readers. The knowledge provided by the document helps with the following:

- Enable the implementation of the component in other systems and products, in a sized and manageable way.
- Establish and clarify the dependencies and requirements the component has with other components.
- Supporting further development of the component though addition of new functions and designs.
- Maintenance and updates through bug fixes and modification in the component design and architecture.
- Reference for development of similar components in the same or different environment and language.
- Provide a basis for testing that the component operates as required.

1.2 Document Scope

The scope of this document is limited to consideration of:

- The specification of the required behaviors for this component
- The specification of the interfaces that this component must implement
- Listing the required interfaces that this component depends upon
- Optionally specifying the internal structure and design of the component.

This scope of this document does not include consideration of:

- The unit tests that is required to verify that this component conforms and performs to its specification
- The specification or implementation of any other components that depend upon this component or that this component depends upon.

1.3 Document Overview

This document contains the following sections:

- **Required Behaviour** – behaviour that is required from the component in terms of the responsibilities that it must satisfy and related requirements
- **Implementation Constraints** – constraints like implementation environment, implementation language and related patterns, guidelines and standards
- **Provided Interfaces** – the interfaces that the component must provide specified in terms of their signatures and constraints
- **Required Interfaces** – lists the interfaces that the component depends upon
- **Internal Structure** – optionally specifies the internal design structure of the component in terms of its constituent components and their relationships
- **Internal Element Designs** – optionally specifies the internal design of the constituent elements.
- **References** – provides full reference details for all documents, white papers and books referenced by this document.

2 Required Behaviour

The component has the following responsibilities in order to fulfil its required behaviour:

1. Act as an interface for data transaction operations between the system and the remote backend, such as fetching or sending data from/to the backend.
2. Provide means for managing users in the remote backend, so that users of the application can manage their accounts as well as remove any personal data should they wish so.

In order to fulfil the above responsibilities, the component is given the following requirements:

- Establishing a link between an active web application and external database, such that data transactions can occur in between.
- Setting a correct navigation path in the external database in the way that the transaction only affects certain areas. Through this, correct data is fetched or data is stored at its intended location.
- Managing the data in a way that it has correct format and semantics when the data is sent or fetched in a transaction.
- Updating the users account and authentication in the backend with correct data, depending on the user.
- Removing all data of the user and therefore their account, should they no longer require the application services.
- Handling exceptions when error occurs such as during communication or processing data.

Upon fulfilling the requirements above, the component will be able to provide the following services for its clients:

Save settings for current user in database	Send a data object containing *app-related* settings to the external database. The object updates existing settings and is stored in a path accessible only for the current user
Fetch user-related settings from database	Fetch the *app-related* settings for the current user from the external database, the app uses the received data to adjust its features according to the current users wish.
Change the password	Updates the password in the backend with the new password
Delete the user	Deletes all data and settings regarding the current user

The component also has several *non-functional* requirements, which is related to each of the functional requirements above and specifies *how* the system should work in each of them:

- The component will not allow unauthorized users of the web application to perform data transactions, write or fetch operations, with the external database.
- Write and fetch operations will only be performed on specific paths in the database, depending on the current user.
- All users must authenticate themselves once before committing operations that may regard account security, such as password change or deletion of account.
- The errors, if they occur, should be visible and understandable in an user friendly way.

There are also requirements which may indirectly affect the component through reuse, maintenance and documentation etc.

- The paths assigned in the component must be changed depending on how the external database is structured, in order to work correctly.
- The structure of the data sent or received during a transaction may require changes depending on how the data is used by the system.
- For reuse, it is recommended for other components to use the same library and language as the component. The reader is therefore recommended to read the provided references for documentation.

3 Implementation Constraints

3.1 Deployment environments

Due to the small scale of the project the component is part of, there are almost none constraints for the deployment environments that the component operates in. The component has however the following minimal requirements:

- A host with the latest runtime environment for the script code of the component to execute in.
- A package manager for the language is required for its *development* and *test environment*. For *staging environments*, the same host can be used, provided that the system with the component is always updated with the latest changes. As for a *live environment*, hosting through servers provided by cloud-services (Google Firebase, IBM cloud etc.), will be sufficient.

3.2 Languages

The component supports both JavaScript ES6 and TypeScript, the languages can be used together with JavaScript libraries such as React, Vue and Angular JS.

3.3 Framework

There is no framework that the component operates in. The functions of the component follow a certain pattern that is used in the Redux library, but Redux is not a framework and the component, if the developer wishes, can be modified slightly to remove the pattern [1].

3.4 Design pattern

A recommended design pattern the component uses from Redux is presenting the component as a list of functions that can be exported to other components. There are no other design patterns or architectural standards that the component must conform to, the component however should be placed along similar components in a folder for better source code structure [1].

3.5 Environmental constraints

As for environmental constraints, the component requires the system implementing it to have network connection and minimal memory in order to provide its services for the system.

4 Provided Interfaces

The component offers, based on its defined requirements and constraints, an interface for the system running the *pager* web application to communicate with the servers containing the remote backend and database, provided by *firebase*. The interface can be divided into several smaller implementations, each of them providing an operation that regards certain parts of the backend or database.

Through this the component restricts the system from accessing the remote backend and database by default, the system can only communicate with the backend through the component. The component will provide its services under the condition that its *required interface* is also provided.

5 Required Interfaces

The component has the following list of required interfaces in order to provide its services:

- An interface for accessing *firebase* services, provided by the npm firebase package which should be included in the npm node modules. The interface should be connected to an existing firebase project (which can be created at the firebase homepage) and be authorized with its configuration data, which the connected project at firebase console provides with [2].
- An interface that connects the component with the system, its implementation can be written in vanilla JavaScript or use existing JS framework or libraries. Through the interface the component should receive the correct parameters in order to operate correctly.

6 Internal Structure

The component consists of a set of functions, where each of them is designed to handle a specific requirement that the system requires from the component. These functions do not collaborate or interact with each other, each of them are individual constituent pieces of the component that separately fulfils parts of the components required behaviour

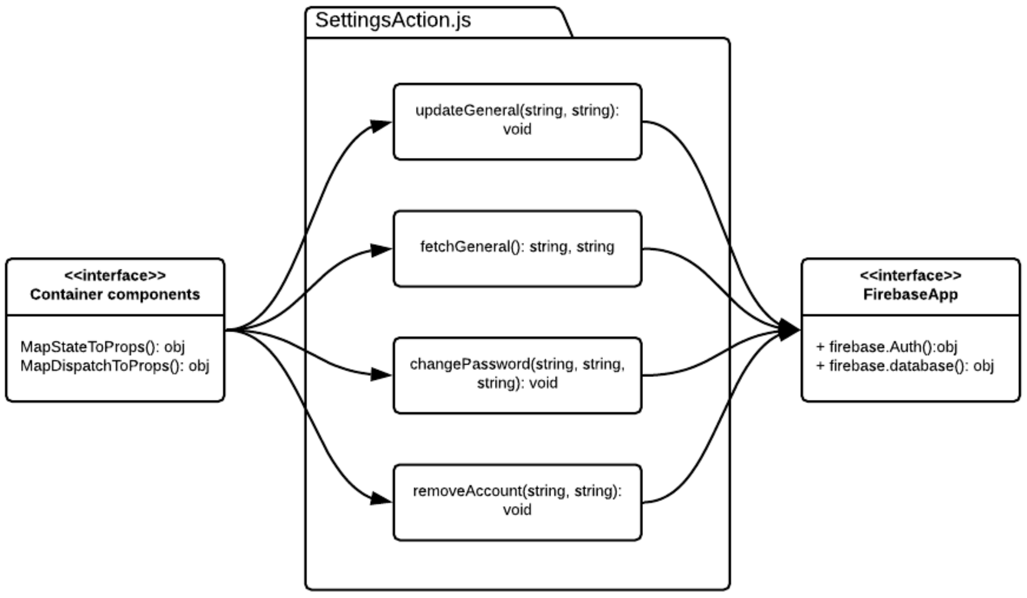

SettingsAction.js
updateGeneral(string, string): void
fetchGeneral(): string, string
changePassword(string, string, string): void
removeAccount(string, string): void
<<interface>>
Container components
MapStateToProps(): obj
MapDispatchToProps(): obj
<<interface>>
FirebaseApp
+ firebase.Auth():obj
+ firebase.database(): obj

Fig 1: UML diagram over the component ***SettingsAction*** *and its constituents*

As seen in *fig 1*: each constituent part is required to fulfil a required operation from the *container component* that connects this component with the rest of the system. Each part uses the interface provided by the firebase/app package to access the firebase cloud services [3].

7 Internal Element Designs

This section specifies the internal design of the constituent parts that together forms the component.

- changePassword(string, string, string): changes the password for the user, given their current email, password and the new password as *string*

```
export const changePassword = (authUser, authPsw, password) => {
    return dispatch => {
        var user = firebase.auth().currentUser;
        var credential = firebase.auth.EmailAuthProvider.credential
       (authUser, authPsw)
        user.reauthenticateWithCredential(credential).then(
            user.updatePassword(password).then(
                alert("password changed!")
            ).catch(function (error) {
                alert("error: please choose another password")
            })
        ).catch(function (error) {
            console.log("an unexpected error has occured", error)
        })
    }
}
```

- removeAccount(string, string): removes the account of the current user and all data related to the user, given the email and password of the user.

```
export const removeAccount = (authUser, authPsw) => {
    return dispatch => {
        var user = firebase.auth().currentUser;
        var credential = firebase.auth.EmailAuthProvider.credential(aut
                        hUser, authPsw)
        user.reauthenticateWithCredential(credential).then(
            user.delete().then(
                alert("user deleted -
                you will now return to the home page")
            ).catch(function(error) {
                alert("error: try deleting your account later")
            })
        ).catch(function (error) {
            console.log("an unexpected error has occured", error)
        })
    }
}
```

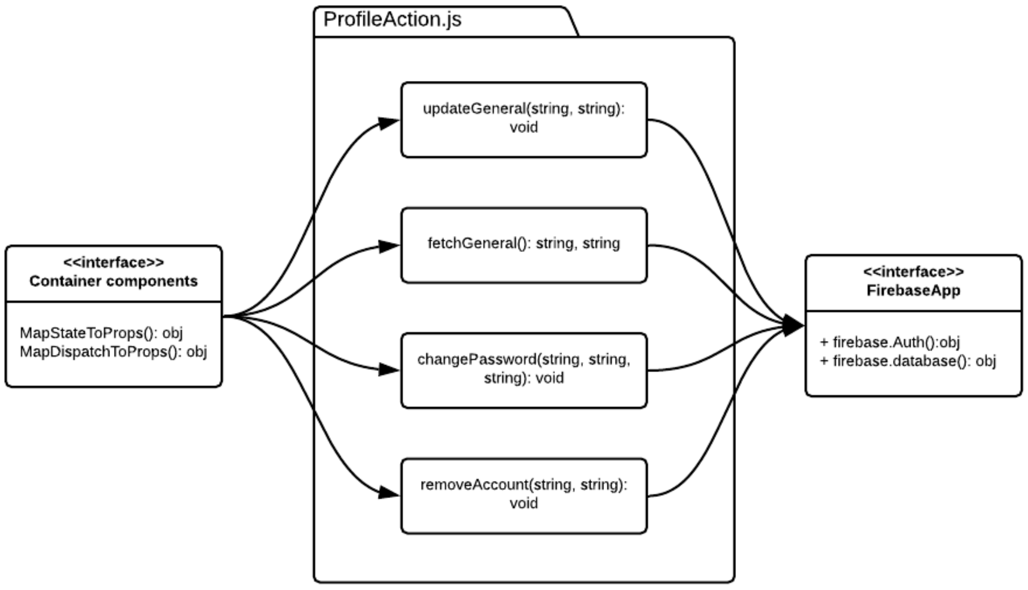
- fetchGeneral(): fetches data objects which represent in-app user settings from a certain path in the database, the path differs depending on the current user.

```
export const fetchGeneral = () => {
    return dispatch => {
        var userId = firebase.auth().currentUser.uid
        return firebase.database().ref('/users/' + userId).once('value'
       ).then(function(snapshot) {
            return snapshot.val()
        });
    }
}
```

- updateGeneral(string, string): updates the in-app settings stored in the remote database for the current user.

```
export const updateGeneral = (colour, amount) => {
    return dispatch => {
        let user = firebase.auth().currentUser.uid
            let options = {
                colour: colour,
                amount_msg: parseInt(amount)
        }
        var updates = {}
        updates['/users/' + user+ '/general/'] = options;
        firebase.database().ref().update(updates)
        .then(function() {
            alert("settings saved!")
        })
        .catch(function(error) {
            alert("Error saving settings: ", error);
        })
    }
}
```

All the above functions share a similar design with the core functionality being encapsulated by a *dispatch* function, which is a callback function used by the Redux library. The important functions will also alert the user a


ProfileAction.js
<<interface>>
Container components
MapStateToProps(): obj
MapDispatchToProps(): obj
updateGeneral(string, string): void
fetchGeneral(): string, string
changePassword(string, string, string): void
removeAccount(string, string): void
<<interface>>
FirebaseApp
+ firebase.Auth():obj
+ firebase.database(): obj

bout changes and if they pass.

Appendix A - References

1. Redux - actions: https://redux.js.org/basics/actions
2. Firebase – managing users: https://firebase.google.com/docs/auth/web/manage-users
3. SettingsAction component: https://github.com/adamliliemark/ii1302-webapp/blob/master/src/data/actions/profileAction.js