



Mobile Application User Manual

Version 1.3

Table of Contents

Installation.....	3
Normal Installation.....	3
Developer Installation.....	3
Create Account Page.....	3
The Login Page.....	4
First Look.....	4
Login Page.....	6
Logging In.....	7
Invalid Login Error.....	9
The Profile Selection Page.....	10
First Look.....	10
Edit Profile.....	11
Profile Passcode Validation.....	12
User Inputs Passcode.....	13
The Schedule Page.....	14
First Look.....	14
Schedule Page Filter: Example 1.....	15
Schedule Page Filter: Example 2.....	16
Schedule Page Filter: Example 3.....	17
Schedule Page: Marking an Event Finished.....	18
Home Care Page.....	23
First Look.....	23
Log Feeding Event.....	25
Log Medicine Event.....	26
Log Therapy Event.....	27
Log Temperature Event.....	28
Log Diaper Event.....	29
Log Emesis Event.....	30
Log Gas Event.....	31
Log G-Tube Misplacement Event.....	32
Log Inhalation and Oxygen Event.....	33
Log Eye Drops Event.....	34
Log Height Event.....	35
Log Weight Event.....	36
The Mobile Data Log.....	37
Offline Functionality.....	38
Interval Scheduling.....	45

Installation

Normal Installation

An .apk file is provided. Transfer the file to your Android device first. Next, trust unsigned applications. This is different by Android device, but will be found in the settings most likely in a security section. This is necessary because the app is not certified for the Google Play store. You will then be able to open the application and use it.

Developer Installation

To develop on the application, the prerequisite is node.js. This can be installed from <https://nodejs.org/en/download>. Install it according to the instructions given. After, find the KARE directory. Navigate into the folder labeled MobileApp, and open a terminal there. To initialize the project and install the packages, run the following command in your terminal:

```
npm install
```

Packages will be installed. After, run the following command to start the project.

```
npm start
```

This will start the development server. A QR code will be provided. In order to connect, you must install the Expo Go app from the Google Play Store. After, scanning the QR code will open the application. Please see EXPO Go documentation for further details.


Create Account Page

The Sign-up page is where the user is navigated to once the user clicks on the Sign-up button from the Login page. The Sign-up page consists of six fields: First Name, Last Name, Email, Password, Confirm Password and Pin. Once the user submits the information by clicking on the Signup button, an error handling function is called to make sure the required fields are filled out correctly and confirms the format of each field. If the information used to create an account is valid then the database API pulls the information passed through by the user and creates an account in the database. If any field has an error, the appropriate error message will tell the user what they need to correct to create an account.


[← Back to log in](#)

Create an account


First Name

 First name



Last Name

 Last name

Email

 example@domain.com

Password

 Password 

Confirm password





 Confirm password 

Figure 1: Create Account

The Login Page

First Look

Upon running the application for the first time, the user will be presented with the option to enter their server's IP address. This requires that the server and database are configured properly and are running.



×

Welcome!

Please enter your server address.

127.0.0.1

Submit

Log in

or



Sign up

Figure 2: Change Server


Login Page

The login page is what the user is greeted with when they first open the application (assuming they have already set their server's IP address). The user is prompted to either log in, sign up, change their server's address with the button on the top right of the screen or navigate to the Forgot Password page by clicking the Forgot Password button. If the user does not already have an account, they would click on the sign-up button which will navigate them to the sign-up page. If the user already has an account, they will enter their valid email address and password.



Once the user clicks Log in, the database API will pull the information passed through by the user and check if it matches with what the database has. If the information passed by the user is invalid, an error handler function is called to display the correct error message to the user. Some examples include, "Please enter your email address in format: example@domain.com" or "Email or password is incorrect." If the information is valid, then the API passes all the user accounts information through to the Profile Selection page. If the user forgot their password, they would click on the Forgot Password button which navigates them to the Forgot Password page (The forgot password page currently has no functionality).



Email

 Enter your email...

Password

 Enter your password... 

[Forgot Password?](#)

Log in

or


Sign up

Figure 3: Login Screen



Logging In

After the server is set, the user can proceed to logging in. In this case, in Figure 5, their email address is test@test.com and their password is “password.” The eye icon can be used to show or hide the password as it is typed.

Email

 test@test.com

Password

 password 

[Forgot Password?](#)

Log in

Figure 4: Credentials Entered

After you have entered your Email and Password, it is time to validate your credentials by pressing the “Log In” button as seen in Figure 6.

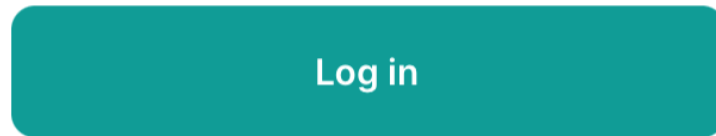


Figure 5: Log in button

Invalid Login Error

Currently, the mobile application has no visual popup that displays the error, but desktop has functionality that would be similarly implemented on mobile – a popup that tells the user what went wrong with their credentials.



Figure 6: Invalid Login

You can re-enter your Email and Password as many times as you like to get the spelling correct. However, if the account has not been set up the user will have to use the create an account page.

The Profile Selection Page

First Look

The Profile Selection page is where the user is navigated to when they successfully login to their account. The Profile Selection page consists of existing profiles in the account, an edit button that navigates the user to the Profile Edit page, and a logout button that logs the user out and navigates them back to the Login page. If the user clicks on an existing profile the database API passes the user account information such as the UUID, AccountUUID, name, and granted accessibility options to the Passcode Page where the user is prompted to enter in the corresponding passcode linked to the specific profile. If the user passes the passcode validation successfully, then the user will be navigated to the Home page. If the user fails the passcode validation, they will not be able to use the application. If the user wants to edit a profile or create a profile, they will click on the edit button, which the database API will pass the account user's UUID to the Profile edit page where the user can edit or create a profile.

← Logout



Select Profile

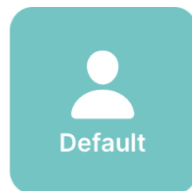


Figure 7: Select Profile

Edit Profile

The Profile Edit page is where the user is navigated to when they click on the edit button from the Profile Selection page. The Profile Edit page prompts the user to either edit an already existing profile or to create a new profile. If the user decides to edit a specific profile, they will click on the existing profile. This process triggers an API call that passes the user accounts information such as the AccountUUID, pin, name, and granted accessibility options to the Profile Update page which is where the user will be navigated to. If the user decides to create a new profile, then the API will pass the user accounts UUID to the Profile Creation page which is where the user will be navigated to. If the user decides to not create a profile or edit a profile, then they may navigate back to the Profile Selection page by clicking on the cancel button.

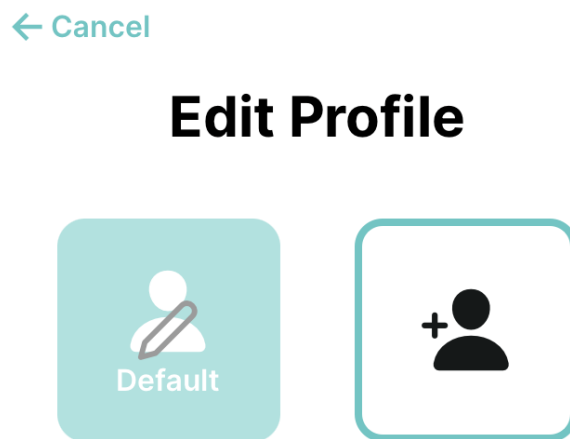


Figure 8: Edit Profile

Profile Passcode Validation

[← Go back](#)

Enter Passcode

Enter

1	2 ABC	3 DEF
4 GHI	5 JKL	6 MNO
7 PQRS	8 TUV	9 WXYZ
	0	⌫

Figure 9: Enter Passcode Dull

User Inputs Passcode

When the user has input all four digits of their passcode, the “Enter” button will become bold to signify that the user can now try their credentials to login to the profile.

[← Go back](#)

Enter Passcode

Enter

Figure 10: Enter Passcode Bold

The Schedule Page

First Look

The Schedule Page is the first page you will see after successful profile validation. All the options that are available for you to interact with can be seen from the Home Page as shown in Figure 11.

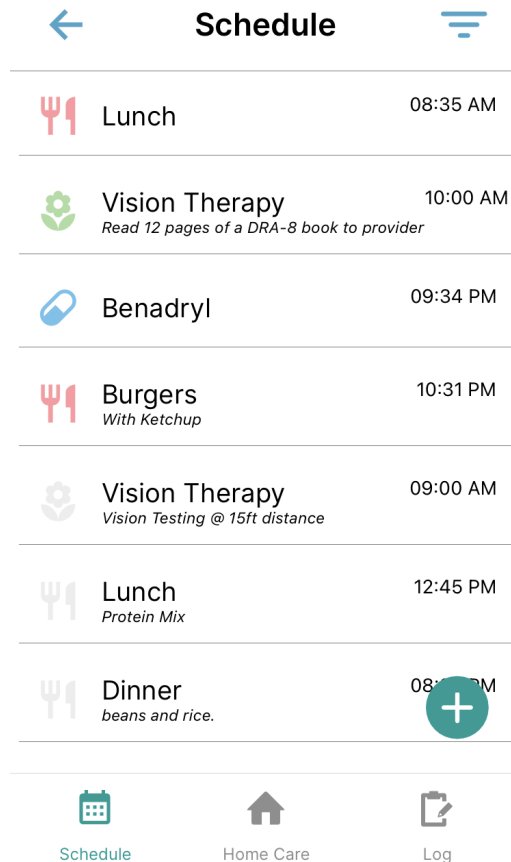


Figure 11: Schedule Page

Schedule Page Filter: Example 1

In this example, the feeding filter is activated, and the feeding events appear on the user's screen as seen in Figure 12.

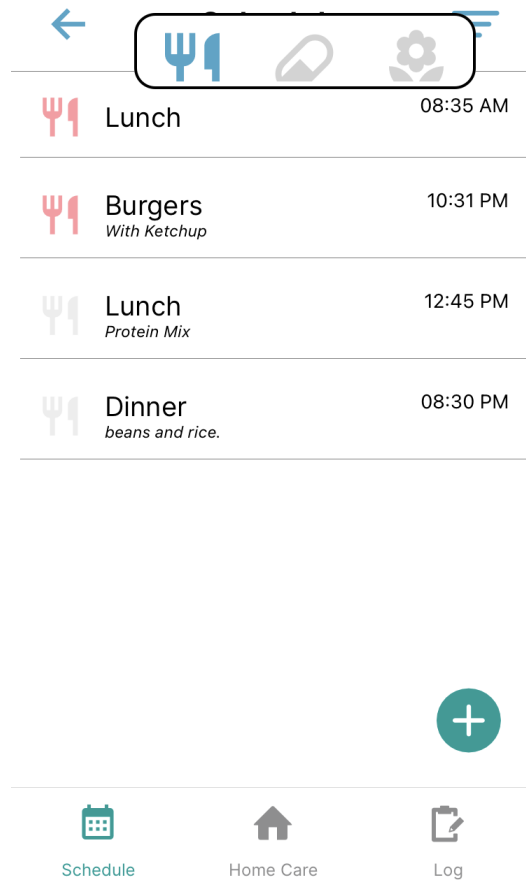


Figure 12: Feeding filter

Schedule Page Filter: Example 2

In this example, the therapy filter is the only filter activated. Two events are displayed, the gray one is already marked complete by the user and the green one is yet to be completed.

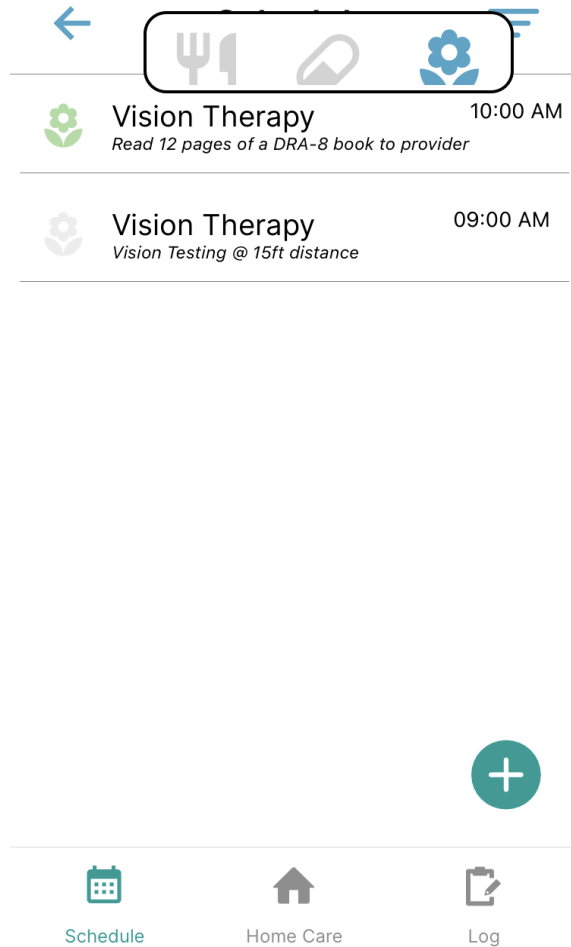


Figure 13: Medicine Filter

Schedule Page Filter: Example 3

In this example two filters are activated: Feeding and Medicine. With these filters activated the feeding event appears.

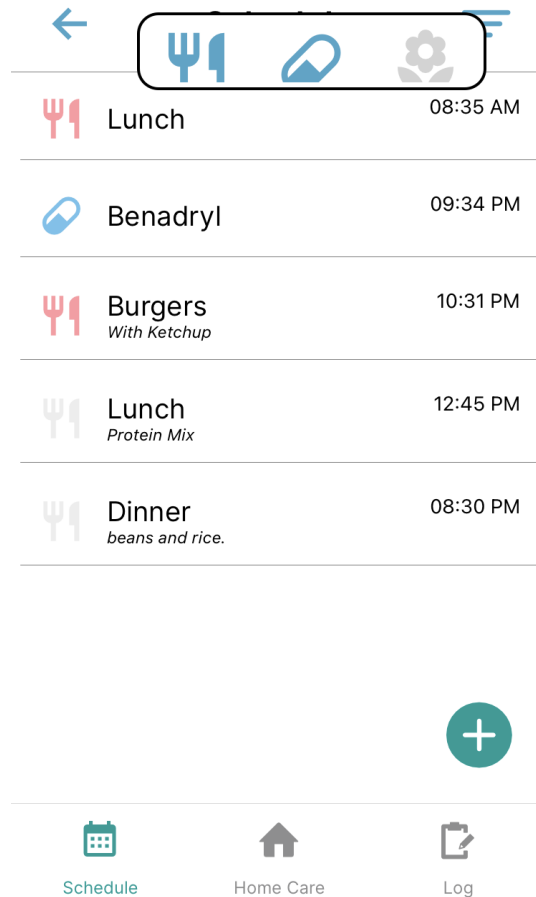


Figure 14: Feeding and Medicine Filters

Schedule Page: Marking an Event Finished

In this example, as seen in Figures 15 and 16, a scheduled event is marked as finished. The event Dinner goes from the top section with the red color to the bottom section with the gray color.

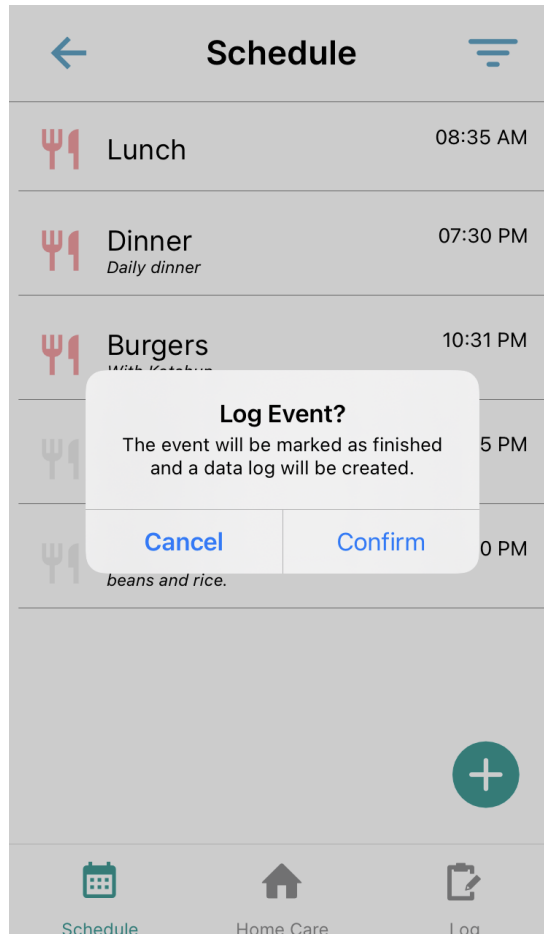


Figure 15: Log Event Pop-up

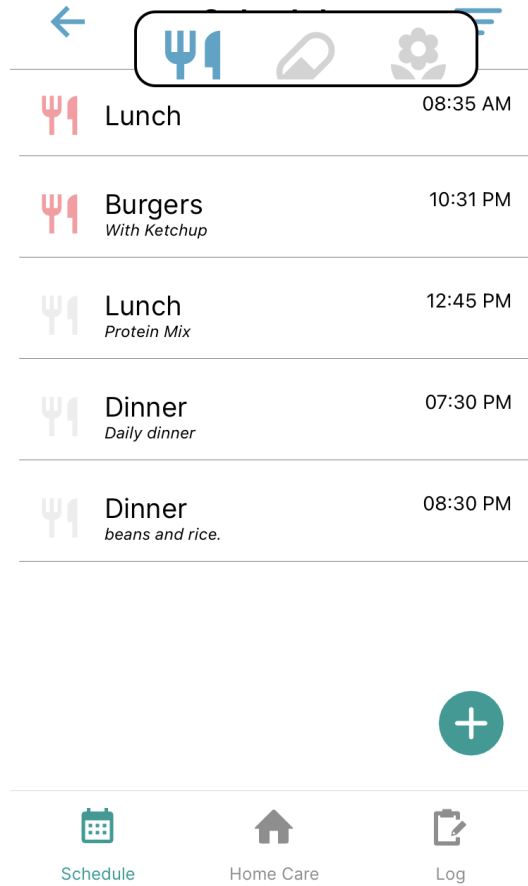


Figure 16: Log Event Complete Result

Schedule Page: Reset Event

In this example, as seen in Figures 17 and 18, a scheduled event is changed from finished to unfinished. The event Dinner is moved from the gray bottom section to the red top section.

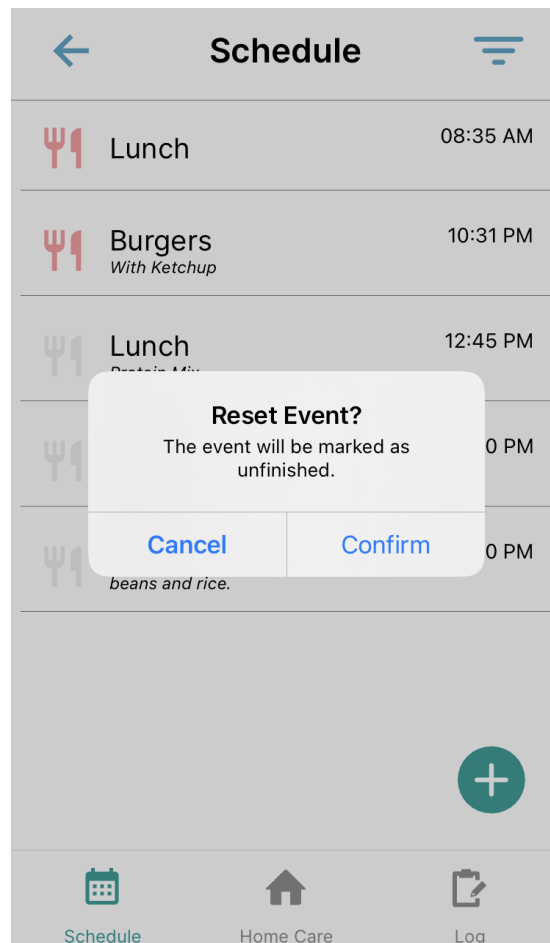


Figure 17: Log Event Reset Pop-up

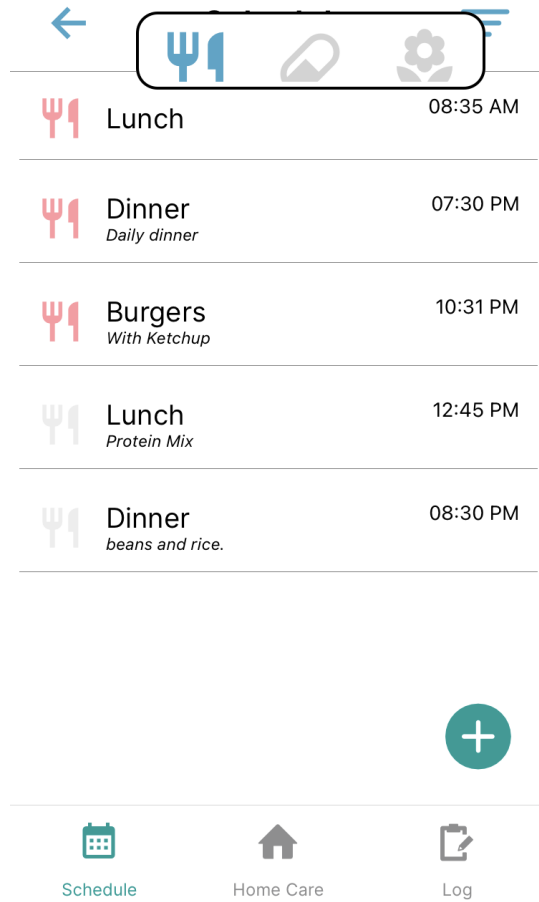


Figure 18: Log Event Reset Result

The schedule event page appears when the green plus icon is pressed on the previous schedule page (Seen in Figure 19).




Figure 19: Schedule Event Button

The schedule event page helps a user to keep track of frequently repeated events. For example, a user's child always takes a certain medicine at a consistent time, this would be a great candidate for a scheduled event.

← Cancel

Schedule an Event

 Feeding >

 Medicine >


 Therapy >

Figure 20: Schedule Event Page

← Cancel

Schedule an Event

Schedule Feeding

Start Date

June 3, 2023

Time

7:30 PM

Repeat Interval *

1

minutes

hours

days

weeks

months

Name *

Dinner

Dose *

15

mL

Rate *

30

mL/h

Figure 21: Schedule Event Feeding

Home Care Page

First Look

The Home Care page is one of the main pages included in the tab bar navigator. This page consists of nine buttons: Temperature, Diaper, Emesis, Gas, G-Tube misplacement,

Inhalation/Oxygen, Eye drops, Height, and Weight. All these buttons prompt the user to a pop-up modal that allows the user to log data about the button that they pressed. Each modal logs information based on what the modal is about. For example, if the Temperature button was pressed, then the Temperature pop-up modal is presented to the user and allows the user to enter in information about the date, time, degree in Celsius or Fahrenheit, and notes. Once the user submits the information through the submit button, the database API pulls all the information and pushes it to the database. This submission process is done for all pop-up modals in home care. Seen below are each of the event submission pages.

Home Care

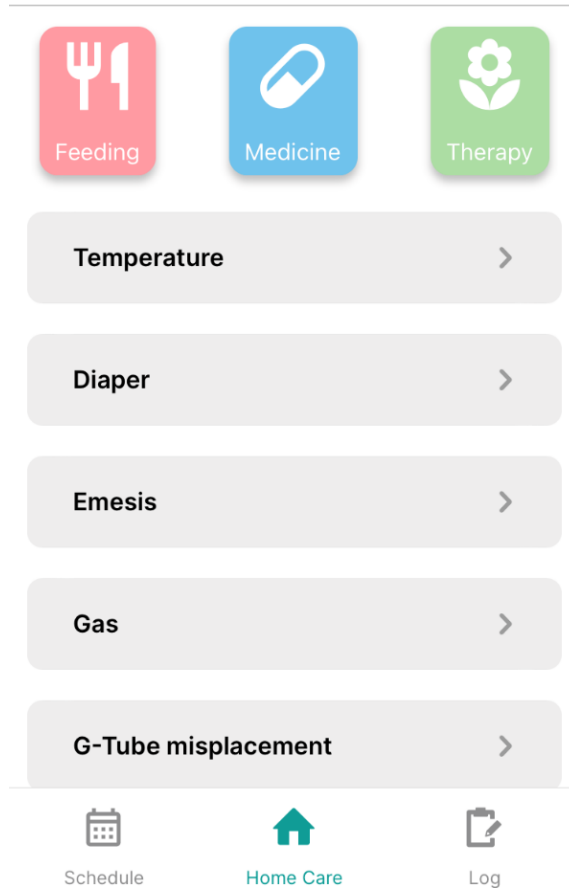


Figure 22: Home Care Page

Log Feeding Event

Home Care

Log Feeding

Date

May 24, 2023

Time

11:24 AM

Name *

e.g. Apple

Dose *

0 mL

Rate *

0 mL/h

Notes

Add a note

Figure 23: Log Feeding Event

Log Medicine Event

The screenshot shows a mobile application interface with a grey header bar labeled "Home Care". Below the header is a white card titled "Log Medicine" with a teal underline. The form contains several input fields: "Date" (May 24, 2023), "Time" (11:24 AM), "Name *" (e.g. Tylenol), "Prescriber *" (Prescriber name), "Dose *" (0 mL), and "Note" (Add a note).

Home Care

Log Medicine

Date
May 24, 2023

Time
11:24 AM

Name *
e.g. Tylenol

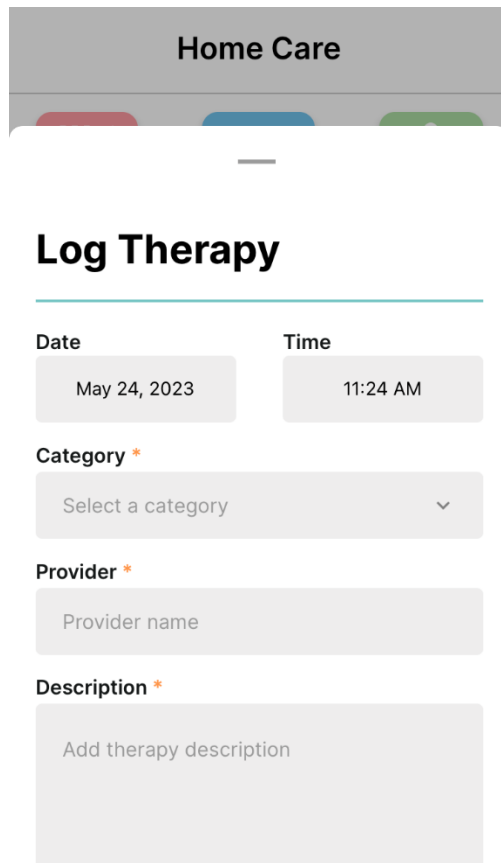
Prescriber *
Prescriber name

Dose *
0 mL

Note
Add a note

Figure 24: Log Medicine Event

Log Therapy Event



The screenshot shows a mobile application interface for logging a therapy event. At the top, there is a grey header bar with the text "Home Care". Below the header, there is a white background with a blue horizontal line. The title "Log Therapy" is displayed in a large, bold, black font. Below the title, there are two input fields for "Date" and "Time". The "Date" field contains "May 24, 2023" and the "Time" field contains "11:24 AM". Below these fields, there is a "Category" field with a dropdown menu showing "Select a category" and a downward arrow. Below the "Category" field, there is a "Provider" field with a text input area showing "Provider name". Below the "Provider" field, there is a "Description" field with a text input area showing "Add therapy description".

Home Care

Log Therapy

Date: May 24, 2023 Time: 11:24 AM

Category *
Select a category

Provider *
Provider name

Description *
Add therapy description

Figure 25: Log Therapy Event

Log Temperature Event

The screenshot shows a mobile application interface for logging a temperature event. At the top is a grey header bar with the text 'Home Care'. Below the header is a white card with rounded corners. Inside the card, the title 'Temperature' is displayed in bold black text, followed by a thin teal horizontal line. Below this line are two input fields: 'Date' with the value 'May 24, 2023' and 'Time' with the value '11:24 AM'. Another thin teal horizontal line follows. Below this is a 'Degrees' section with a small orange asterisk. It contains a numeric input field with the value '0', a unit selector with 'C' selected, and two radio buttons for temperature units: '°C' (which is selected) and '°F'. Below the 'Degrees' section is a 'Note' section with a large, light grey text area containing the placeholder text 'Add a note'. A final thin teal horizontal line is at the bottom of the card.

Home Care

Temperature

Date: May 24, 2023 Time: 11:24 AM

Degrees * 0 C °C °F

Note: Add a note

Figure 26: Log Temperature Event

Log Diaper Event

Home Care

Diaper

Date
May 24, 2023

Time
11:24 AM

Stool size *
Size ▾

Stool color *
Color ▾

Water/Urine *
Weight ▾

Notes
Add a note

Figure 27: Log Diaper Event

Log Emesis Event

The screenshot shows a mobile application interface for 'Home Care'. At the top, there is a grey header bar with the text 'Home Care'. Below this header, there is a row of three colored tabs: 'Feeding' (red), 'Medicine' (blue), and 'Therapy' (green). The 'Medicine' tab is currently selected. Below the tabs, there is a large white area with the title 'Emesis' in bold black text. Underneath the title, there are two input fields: 'Date' with the value 'May 24, 2023' and 'Time' with the value '11:24 AM'. Below these fields, there is a section titled 'Emesis volume *' with a dropdown menu showing 'Volume'. At the bottom, there is a section titled 'Notes' with a large text area containing the placeholder text 'Add a note'.

Home Care

Feeding Medicine Therapy

Emesis

Date
May 24, 2023

Time
11:24 AM

Emesis volume *
Volume

Notes
Add a note

Figure 28: Log Emesis Event

Log Gas Event

Home Care

Feeding Medicine Therapy

Gas

Date May 24, 2023 **Time** 11:24 AM

Time of day *

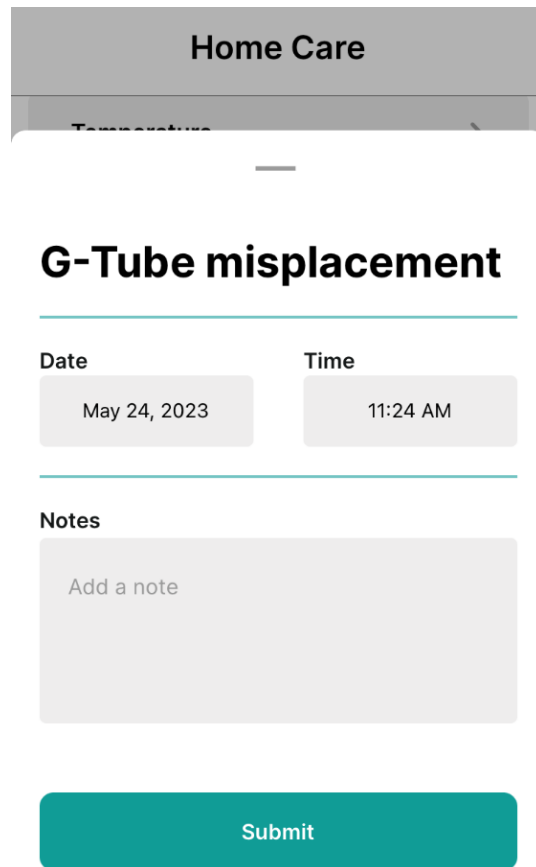
Time ▼

Notes

Add a note

Figure 29: Log Gas Event

Log G-Tube Misplacement Event



The screenshot shows a mobile application interface for 'Home Care'. At the top, there is a grey header bar with the text 'Home Care'. Below this, a section titled 'G-Tube misplacement' is highlighted with a teal underline. This section contains two input fields: 'Date' with the value 'May 24, 2023' and 'Time' with the value '11:24 AM'. Below these fields is a 'Notes' section with a large text area containing the placeholder text 'Add a note'. At the bottom of the form is a teal 'Submit' button.

Home Care

G-Tube misplacement

Date May 24, 2023 Time 11:24 AM

Notes

Add a note

Submit

Figure 30: Log G-Tube Misplacement

Log Inhalation and Oxygen Event

The screenshot shows a mobile application interface. At the top is a grey header bar with the text 'Home Care'. Below this is a white card with rounded corners. Inside the card, the title 'Inhalation and Oxygen' is displayed in bold black text, followed by a thin blue horizontal line. Below the line are two input fields: 'Date' with the value 'May 24, 2023' and 'Time' with the value '11:24 AM'. Another thin blue horizontal line follows. Below this is a 'Notes' section with a light grey text area containing the placeholder 'Add a note'. At the bottom of the card is a teal-colored button with the text 'Submit' in white.

Home Care

Inhalation and Oxygen

Date May 24, 2023 Time 11:24 AM

Notes

Add a note

Submit

Figure 31: Log Inhalation and Oxygen

Log Eye Drops Event

The screenshot shows a mobile application interface for logging an eye drop event. At the top is a grey header bar with the text 'Home Care'. Below this is a white card with rounded corners. Inside the card, the title 'Eye Drops' is displayed in bold black text. Underneath the title is a horizontal line. Below the line are two input fields: 'Date' with the value 'May 24, 2023' and 'Time' with the value '11:24 AM'. Another horizontal line follows. Below this is a 'Notes' section with a large text area containing the placeholder 'Add a note'. At the bottom of the card is a teal button labeled 'Submit'.

Home Care

Eye Drops

Date May 24, 2023 Time 11:24 AM

Notes

Add a note

Submit

Figure 32: Log Eye Drops

Log Height Event

Home Care

Height

Date

May 24, 2023

Time

11:24 AM

Height *

0

cm

☒ cm ☐ in

Notes

Add a note

Figure 33: Log Height

Log Weight Event

Home Care

Weight

Date

May 24, 2023

Time

11:24 AM

Weight *

0

kg

☒ kg ☐ lb

Note

Add a note

Figure 34: Log Weight

The Mobile Data Log

This page shows the user what has logged recently.

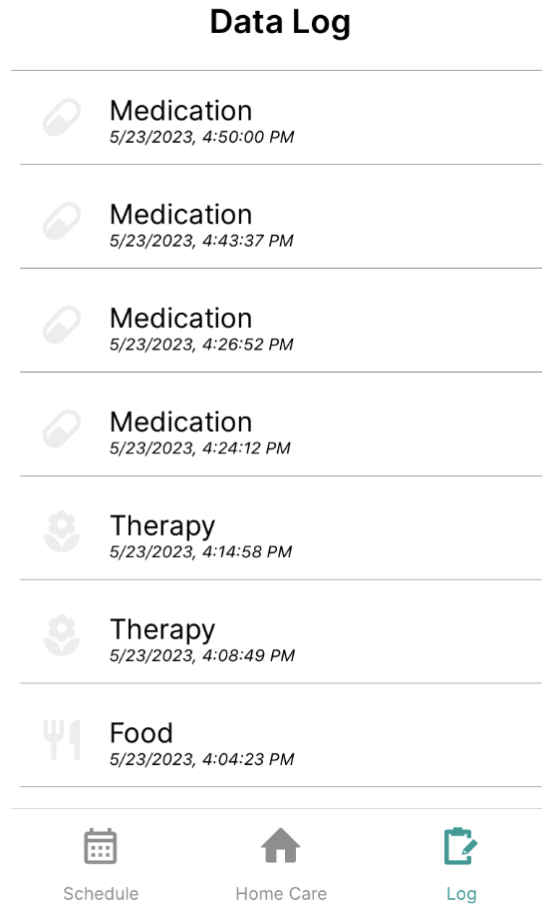


Figure 36: Data Logging

Offline Functionality

This section is intended for developers of the future KARE mobile application. Most of this section is written about the api.js file in the API folder of the MobileApp.

This version's implementation of the project uses Async Storage from react native that allows storing of values in a key value pair on the user's mobile device as seen in Figure 32.

```
import AsyncStorage from "@react-native-async-storage/async-storage";
```

Figure 37: AsyncStorage

The intention of this is to be able to store user credentials when the user successfully passes the authentication stage with the server. For instance, in Figure 33, this function checks to see if the user's credentials are already stored in the async storage and if they are already stored, it is a returning user. However, if they are not already stored, this function will set the async storage at the current user count with the newly input credentials. Afterwards, it will increment the user count so the next new user can be placed correctly.

```
/**
 * Find if the this is the first time the user has logged in
 *
 * @returns 1 if the user has logged in before
 * @returns 0 if the user is new and puts the email and password in storage
 */
export const isNewUser = async (email, password) => {
  try {
    var userCount = Number(await getUserCount());
    for (var i = 0; i < userCount; i++) {
      if (
        email === (await getEncryptedEmail("@encrypted_email" + i.toString())) &&
        password === (await getEncryptedPassword("@encrypted_password" + i.toString()))
      ) {
        console.log("/isNewUser/: returning user");
        return 1;
      }
    }
    console.log("/isNewUser/: new user");
    await setEncryptedEmail(email);
    await setEncryptedPassword(password);
    await incrementUserCount();
    return 0;
  } catch (e) {
    // error reading value
    console.log("ERROR /isNewUser/: ", e);
    return null;
  }
}
```

Figure 38: isNewUser

These credentials can then be used to check if the same credentials have been used to login when the server is unavailable due to the necessity of the app to be used in environments when no internet connection is available. This is implemented in the loginValid function as shown in Figure 34 where the function checks if the response was 200 meaning that the authentication with the server was correctly passed, or if the response is 404 or 406 where the server could not handle the request correctly, or if there was a network error in the catch statement where the server could not be reached at all. In this case, the function checks if the error is specifically “TypeError: NetworkRequestFailed.” In these cases, the function will check if the input credentials match what is retrieved from what is stored in the async storage. When the input credentials are found to match, the UUID that corresponds with the input email is returned along with true.

```

MobileApp > API > JS api.js > loginValid
291
292 if (response.status === 200) {
293   const text = await response.text();
294   const row = JSON.parse(text);
295   const returnUUID = parseInt(row["UUID"]);
296   await setUUID(email, returnUUID);
297   return [true, returnUUID];
298 } else if (response.status === 404 || response.status === 406) {
299   const userCount = Number(await getUserCount());
300   if (userCount > 0) {
301     for (let i = 0; i < userCount; i++) {
302       const encryptedEmail = await getEncryptedEmail("@encrypted_email" + i.toString());
303       const encryptedPassword = await getEncryptedPassword("@encrypted_password" + i.toString());
304       if (email === encryptedEmail && password === encryptedPassword) {
305         const returnUUID = await getUUID(email);
306         return [true, returnUUID];
307       }
308     }
309   }
310 } else {
311   console.log("Unsuccessful login validation: " + response.status);
312 }
313 } catch (error) {
314   console.log("loginValid():", error);
315
316   if (error.toString() === "TypeError: Network request failed") {
317     const userCount = Number(await getUserCount());
318     if (userCount > 0) {
319       for (let i = 0; i < userCount; i++) {
320         const encryptedEmail = await getEncryptedEmail("@encrypted_email" + i.toString());
321         const encryptedPassword = await getEncryptedPassword("@encrypted_password" + i.toString());
322         if (email === encryptedEmail && password === encryptedPassword) {
323           const returnUUID = await getUUID(email);
324           return [true, returnUUID];
325         }
326       }
327     }
328   }
329 }

```

Figure 39: loginValid

The next step is correctly storing the user's profiles. When the user has correctly logged in, the application will then load the profile selection page (ProfileSelection.js) as seen in figure 7. When this page is loaded it calls the getProfiles function from api.js. This function first checks if the server responds to the post request with the UUID being sent. If the request is successfully handled, the server will send back the profiles with a response status of 200. When this occurs, the function will first set the async storage of that UUID to contain the profiles as a String. After this it will return the profiles as a JSON to the profile selection page as shown in Figure 35.

```
export async function getProfiles(UUID) {
  let returnProfiles = null;
  if (UUID) {
    let response = await fetch(server_address + "getProfiles", {
      method: "POST",
      headers: {
        Accept: "application/json",
        "Content-Type": "application/json",
      },
      body: `{
        "UUID": "${UUID}"
      }`,
    });
  }
  .then(async (response) => {
    if (response.status == 200) {
      await response.text().then(async function (text) {
        await setProfiles(UUID, text);
        let profiles = JSON.parse(text);
        returnProfiles = profiles;
        return returnProfiles;
      });
    }
  });
}
```

Figure 40: getProfilesOnline

Additionally, for the offline portion of the `getProfiles` function, when the response status is 404 or 406 the async storage tied with the UUID is checked to see if it contains any profiles. If it does the profiles are returned to the profile selection page, if what is returned from the async storage is null then the Unsuccessful get profile request message is logged along with the response status which will most likely be 404 or 406. Furthermore, if an error is caught and this error is specifically, "TypeError: Network request failed" the function will check like it did above where if the async storage contains profiles they will be returned or if there are no profiles stored then the specific error will be displayed and false will be returned.

```
    } else if (response.status == 404 || response.status == 406) {
      let profiles = await getOfflineProfiles(UUID);
      if (profiles != null) {
        returnProfiles = JSON.parse(profiles);
      }
    } else {
      console.log("Unsuccessful get profile request: " + response.status);
    }
  })
  .catch(async (error) => {
    if (error.toString() === "TypeError: Network request failed") {
      let profiles = await getOfflineProfiles(UUID);
      if (profiles != null) {
        returnProfiles = JSON.parse(profiles);
        return returnProfiles;
      }
    }
    console.log("Error getting profiles: " + error);
    return false;
  });
} else {
  console.log("failed to get profiles; UUID is empty");
}
return returnProfiles;
```

Figure 41: `getProfilesOffline`

The next goal was to store each profile's passcode, but this proved to be tricky and time constraints forced the development team to leave the offline access here. With this version, when the user tries to login offline, they will get up to the point where they enter the passcode for their profile, but their passcode will never be authenticated so they will be stuck there.

The following code is used for the implementation of offline API call storage. Offline API call storage is performed by two functions, saveLog and emptyStorage, and a helper function, checkConnection.

First off is the saveLog function. The purpose of saveLog is to save an API call to storage if no connection to the server is found. This function is called from other API functions, such as addEvent. When called from these functions, saveLog requires two variables to be passed, 'type' and 'attributes'. The 'attributes' object is the 'Attributes' object passed into the function that would be calling saveLog. 'Attributes' is a JSON with all necessary attributes connected to an event. The 'type' object is a String stating what type of API call is being stored. For the sake of simplicity, the type is the name of the function calling saveLog in String form.

The saveLog function works using AsyncStorage. It starts by generating a key to match with the new API call being stored – this is done using numbers for easy access in the emptyStorage function. The type and attributes of the API call are then converted into a JSON object and stored via AsyncStorage with the generated key.

```
export async function saveLog(type, attributes) {
  let num = "0";
  try {
    let x = await AsyncStorage.getAllKeys();
    if (!(x === undefined)) {
      num = JSON.stringify(x.length);
    }
  } catch (e) {
    console.log("There was an error getting keys:" + e);
  }

  let value = {
    call_type: type,
    attributes: JSON.stringify(attributes),
  };

  try {
    await AsyncStorage.setItem(num, JSON.stringify(value));
  } catch (e) {
    console.log("There was an error setting item:" + e);
  }
}
```

Figure 42. saveLog

Next is the `emptyStorage` function. This function is called by other API functions, such as `addEvent`, whenever a connection to the server is active. It should be called before that function executes its own code to preserve API call order, as it is unclear if changing the order API calls reach the server would cause issues. The current implementation of `emptyStorage` assumes that `saveLog` stores API calls with numerical keys and no other function stores data with numerical keys. Changing from numerical keys to some other method of creating and reading dynamically created keys would be recommended, but this approach works as long as numerical keys are avoided elsewhere.

The `emptyStorage` function starts by getting all keys currently in use by `AsyncStorage`. For every key received, `emptyStorage` will get the attached item and check if the key is an integer. If the key is not an integer, the current loop will be skipped as that object is not an API call stored by `saveLog`. After passing the integer check, the stored item will be parsed from JSON format so the data can be passed back to their respective functions.

```
export async function emptyStorage() {
  let keys = [];
  try {
    keys = await AsyncStorage.getAllKeys();
  } catch (e) {
    console.log("There was an error getting keys:" + e);
  }

  for (let i = 0; i < keys.length; i++) {
    let log;
    try {
      log = await AsyncStorage.getItem(keys[i]);
    } catch (e) {
      console.log("There was an error getting item:" + e);
    }

    if (log === null) {
      console.log("Log was null");
      continue;
    }
    // all keys relating to offline storage use numbers
    // this block stops other keys from being used
    if (parseInt(keys[i]) !== keys[i]) {
      continue;
    }

    try {
      log = JSON.parse(log);
    } catch (e) {
      console.log("There was an error parsing:" + log);
      break;
    }
  }
}
```

Figure 43. emptyStorage

After parsing the JSON, emptyStorage will determine the 'call_type' of the data, known to saveLog as the 'type' object. The corresponding API function will then be called with the original attributes passed to it. Once the data has been passed back to its function and the API call has been made, emptyStorage will delete the stored item using its key.

```
switch (log.call_type) {
  case "addEvent":
    addEvent(JSON.parse(log.attributes));
    break;
  case "createScheduledEvent":
    createScheduledEvent(JSON.parse(log.attributes));
    break;
  case "removeScheduledEvent":
    removeScheduledEvent(JSON.parse(log.attributes));
    break;
  case "setScheduledEventComplete":
    setScheduledEventComplete(JSON.parse(log.attributes));
    break;
  default:
    console.log("How did you get here?");
}

try {
  await AsyncStorage.removeItem(keys[i]);
} catch (e) {
  console.log("There was an error removing item:" + e);
}
}
```

Figure 44. emptyStorage

This approach can surely be improved upon by changing how dynamic keys are created and detected by saveLog and emptyStorage. The emptyStorage function could also be better implemented if called by a function that does not rely on a new API call to be made first, as the current implementation will not make the stored API calls until a connection to the server is established, and an API call has been made.

The final function made during the development of offline API call storage was `checkConnection`. This is a helper function used to determine if there is an active connection to the server. This function is currently used to determine if `saveLog` should be called or if `emptyStorage` should be called and API calls can be made normally.

Other API functions also currently require a check for server connection, so this function might be useful there with little or no alterations made.

```
export async function checkConnection() {
  try {
    const response = await fetch(server_address);
    if (response.status === 200) {
      return true;
    } else {
      return false;
    }
  } catch (e) {
    console.log("Error fetching server.");
    return false;
  }
}
```

Figure 45. checkConnection

Interval Scheduling

Interval scheduling allows for events to be marked as ready after a specified amount of time passes. This is useful for events that repeat, such as breakfast each day, or longer intervals such as speech therapy every week. This is implemented using asynchronous storage, facilitated by the `AsyncStorage` package.

Expiration dates are stored as time-from-epoch integer values in a hashmap. The hashmap is stored in asynchronous storage with the key `@expiration_dates`. The key is the EventID, such as 219. The value is the expiration time, such as 1685829185050.

```
import AsyncStorage from "@react-native-async-storage/async-storage";
```

The code for this functionality is mainly stored in `./api/checkExpiration.js`.


```

/**
 * An API handler that marks events as ready to be completed when their specified time interval passes.
 * Called in Login.js and on refresh of the scheduler.
 */
export const checkExpiration = async () => {
  const asyncEvents = await getData();
  if (asyncEvents) {
    processKeys(asyncEvents);
  }
};

```

Figure 46. *checkExpiration*

The exported function. It retrieves the data using `getData()`, then if it is not null, attempts to process it.

```

const getData = async () => {
  try {
    const data = await AsyncStorage.getItem("@expiration_dates");
    if (data !== null) {
      // value previously stored
      return JSON.parse(data);
    } else {
      return null;
    }
  } catch (e) {
    // error reading value
    console.log("checkExpiration.js: Error retrieving @expiration_dates from asynchronous storage.");
  }
};

```

Figure 47. *getData*

Abstracts the process of retrieving and parsing the data from asynchronous storage.

```

/**
 * Find and process all scheduled events.
 *
 * @param {dict} data - Async storage events object. {219: 16...458, 220: 16...985, 221: 16...339}.
 */
const processKeys = (data) => {
  let expiredKeys = [];

  for (let key in data) {
    if (isEventExpired(data[key])) {
      setScheduledEventComplete(key, "0");
      expiredKeys.push(key);
    }
  }

  processExpiredKeys(data, expiredKeys);
};

```

Figure 48. *processKeys*

Processes the keys retrieved from `getData`. Checks if an event has expired. If it has, change the event's finished status to "o" and push it to an array. This array is passed to `processExpiredKeys()`.

```
const isEventExpired = (timestamp) => {
  const currentTime = new Date().getTime();
  // console.log(`${currentTime} > ${parseInt(timestamp)}?`);
  return currentTime > parseInt(timestamp) ? true : false;
};
```

Figure 49. *isEventExpired*

Check the expiration date against the current date. Returns a boolean telling if the date has passed or not.

```
/**
 * Remove expired keys from async storage.
 *
 * @param {dict} asyncKeys - Timestamps stored in asynchronous storage.
 * @param {dict} expiredKeys - A list of keys to remove from the JSON that have been used.
 */
const processExpiredKeys = async (asyncKeys, expiredKeys) => {
  for (let key of expiredKeys) {
    delete asyncKeys[key];
  }
  // at the end, setItem in async storage again
  try {
    await AsyncStorage.setItem("@expiration_dates", JSON.stringify(asyncKeys));
  } catch (e) {
    // saving error
    console.log("checkExpiration.js: Error setting @expiration_dates into storage.");
  }
};
```

Figure 50. *processExpiredKeys*

Takes the current `asyncKeys` dictionary and removes all of the events that have been marked as unfinished. Then, store this new modified dictionary in asynchronous storage.