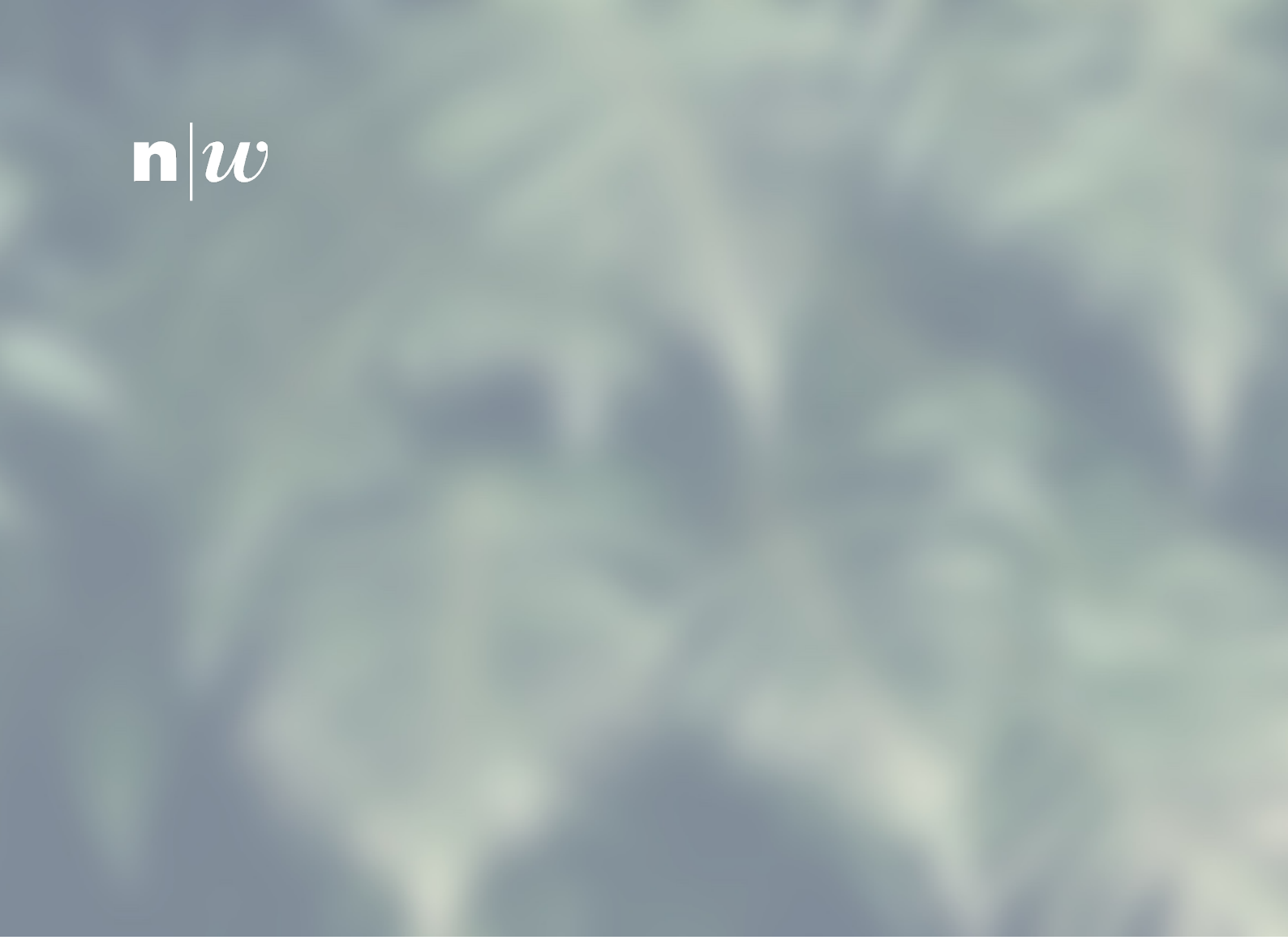

happy flowers

Project Report

Sacha Schmid
Rinesch Murugathas

Daniel Kröni

Windisch, 21 October 2016

1. Summary

The following report describes the ‘happy flowers’ project and its relevant components. It contains information about architectural decisions and technological specialities.

Flowers at the Institute for Mobile and Distributed Systems are often fighting for long dry periods. This project solves that problem by watering the flowers automatically.

This project resulted in a front end application realised using React, a back end application written in Haskell, and a Raspberry Pi device that has been connected to a moisture sensor and a watering pump. Described in this report are the architectural designs of these components and the way they communicate. Apart from this, the report contains recommendations and additional resources that can be used for learning purposes when this project is demonstrated in a course.

1.1. Fact Sheet

Project Name	happy flowers
Project Duration	19 September 2016 – 20 January 2017
Team Members	Sacha Schmid, Rinesch Murugathas
Customer	Institute for Mobile and Distributed Systems
Coach	Daniel Kröni
Repository	https://github.com/RadLikeWhoa/happyflowers/

Table 1 Fact sheet about the ‘happy flowers’ project

Table of Contents

2. Introduction

This report details the process and the findings of the IP5 project ‘happy flowers’ along with its architectural details and design decisions. Rinesch Murugathas and Sacha Schmid have worked on this project over the course of the fall semester 2016 at the University of Applied Sciences and Arts Northwestern Switzerland (*UAS*) in Brugg-Windisch.

The project is motivated by the rise of *Internet of Things* technology, a recent trend that aims at making every-day interactions more intelligent by involving connected devices. While some areas like smart light bulbs have already entered the mass market and become of particular interest to consumers, other areas have not been touched this much.

‘happy flowers’ aims to solve the problem of household plants living in suboptimal conditions. Plants require a certain set of conditions to grow, one of which is the right amount of moisture in the plant’s soil. This requires owners to water their plants on a regular basis which is sometimes not possible or can easily be forgotten. It makes sense to augment this area of every-day life with technology by creating a sensor that waters plants automatically, removing the reliance on human input. Thus, the ideal level of moisture can be guaranteed with a level of precision and reliability that humans could not provide.

Aside from the obvious focus on the product, the project has another goal. A large portion of the code base is written in Haskell, a functional programming language that was established in 1990. *Functional Programming* is a programming methodology that diverges from other approaches, such as *Object Oriented Programming*. While it is still mostly considered an alternative to traditional programming styles, Functional Programming is increasingly important in newly developed technologies and programming languages. Many languages, such as JavaScript or C#, have adapted functional approaches in their recent and upcoming versions and strongly advise programmers to use those features.

The UAS offers a course on Functional Programming — 'fprog' is currently held by Daniel Kröni and Dr Edgar Lederer — and teaches students about the methodology and related concepts. With proper documentation and understandable code, 'happy flowers' is to be used as a demonstration for a real-world application of Haskell during the course in order to prove to students that Haskell is more than just an interesting idea.

A project assignment (https://goo.gl/jO1D13) defined the basic requirements and recommended a process. It mandated that a web front end and a back end be created and documented in a fashion that allows students and interested people to learn about the interesting technologies used throughout the project. In a project agreement (https://goo.gl/mFULmN) the requirements were specified more clearly and agreed upon with the project coach.

From the beginning, the goal of this project was to create the individual components very iteratively so that requirements could be adapted quickly and new features and idea easily incorporated. This also meant that the back and front end and the hardware parts were developed concurrently and integrated step by step. This allowed for splitting the workload inside the team so both team members could work without constantly stepping on each others toes.

All of this project's code is publicly available as part of an open source repository hosted on GitHub (https://goo.gl/n7d4Mo). This allows other people to learn from and contribute to the project.

The main part of this report documents the pieces this project is made of and how they communicate with each other. It uses a combination of text, graphs, illustrations, and code snippets to explain the technologies in use and what is interesting about them.

3. Raspberry Pi

TODO: add image of set up raspberry + hardware

Figure 1 Raspberry Pi setup with connected sensors and hardware

3.1. Requirements

The project requires the following hardware and technologies in order to set up a working environment.

Technology	Minimal Version
Raspberry Pi	3 Model B
Raspbian	8.0 (jessie)
bcm2835	1.50
Motion	3.2.12
Chirp!	2.7.1 (Sensor Mode)
USB Webcam	n/a
USB Water Pump	n/a

Table 2 Requirements for the Raspberry Pi

3.2. Hardware Setup

3.2.1. Setting Up the Raspberry Pi

The Raspberry Pi ('RPi' in following) is based on an ARM architecture. The problem is that Haskell is not supported very well on these systems so a lot of additional configuration is needed. The setup is mostly based on a tutorial (https://goo.gl/6BYqVV). A detailed step-by-step guide for this setup — including specific commands — is available in this project's GitHub repository (https://goo.gl/7vuN7M).

The RPi is not very powerful and SD cards offer very poor I/O performance. It can take a long time to execute the above steps and to install and build

project dependencies. There are ways to guide the tools into using the right amount of memory or improve their performance footprint, but the process still remains time-consuming and error-prone.

Once the device is fully set up, the Haskell code can start communicating with connected devices through a set of ports.

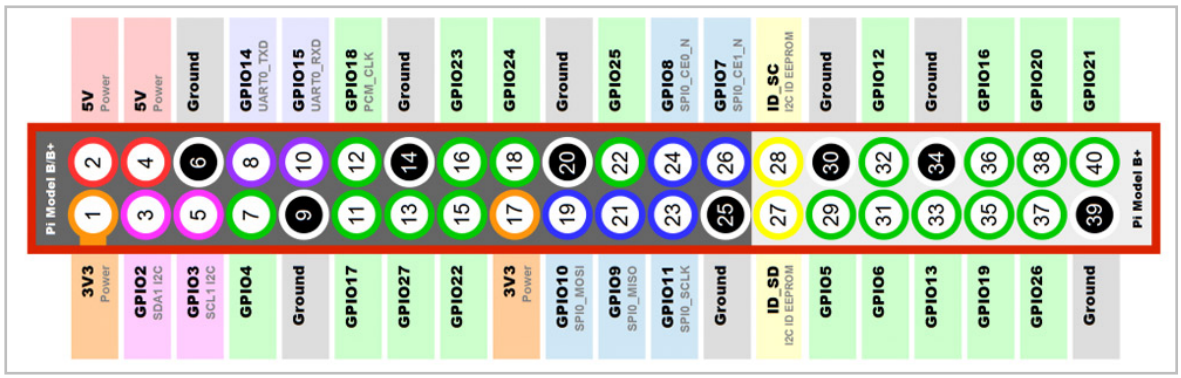

Figure 2 Raspberry Pi 3 pin layout (https://goo.gl/j4Sfp1)

3.2.2. Connecting the Raspberry Pi to the Chirp! Sensor

All data about the plant is retrieved using a 'sensor mode only' Chirp! device (https://goo.gl/mymfPY). The sensor is capable of measuring soil moisture, temperature, and ambient light and transmits data through the I^2C protocol (see *3.3.1*). It requires a supply voltage between 3.3V and 5V, both of which the RPi is capable of producing.

In order to retrieve data, one must read two bytes from a given I^2C register — register 0 for soil moisture, register 5 for temperature. The data is then returned as a byte string. Processing the string produces a value that represents moisture on a relative scale between 0 and 65535 and temperature on an absolute scale where the value is in tenths-Celsius.

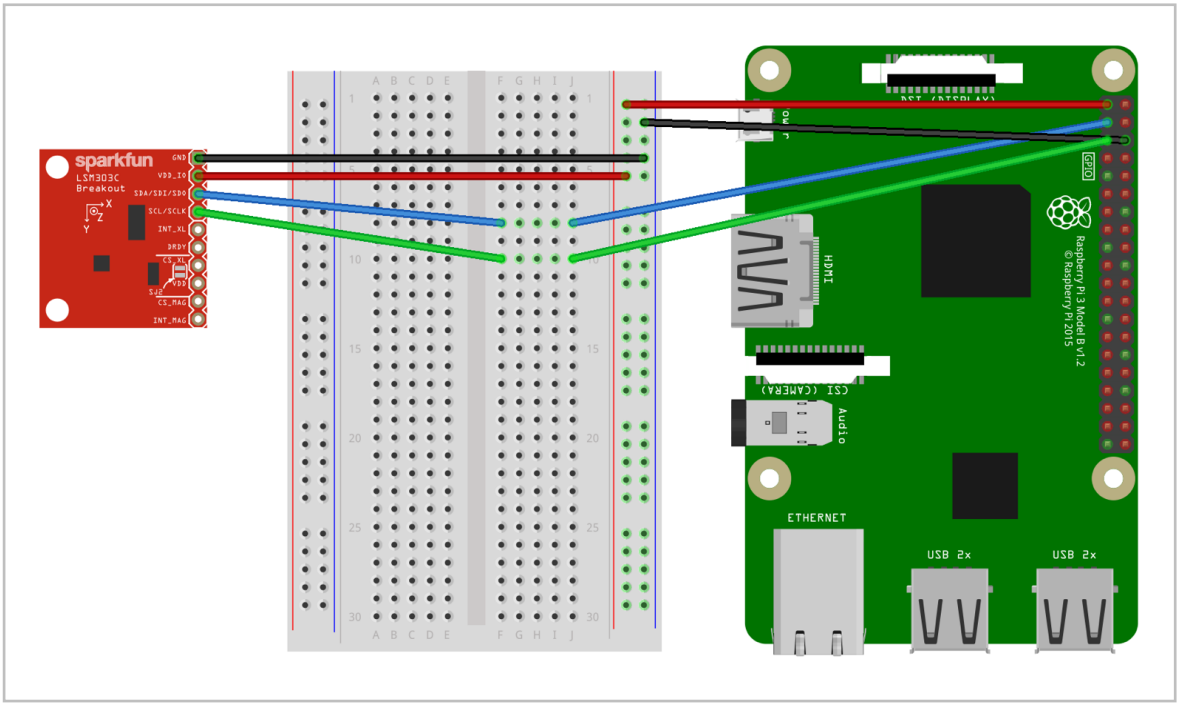

Figure 3 Connection schematics for the Chirp! sensor

3.2.3. Connecting the Raspberry Pi to the Watering Pump

The standard USB water pump is connected through the breadboard. It is activated and deactivated using GPIO (see *3.3.1*).

TODO: add schematics

Figure 4 Connection schematics for the USB water pump

3.2.4. Connecting the Raspberry Pi to a USB Webcam

The camera is simply connected to one of the USB ports of the RPi. To use the livestream, motion is required to be installed (see *3.3.2*).

3.2.5. Connecting to a Headless Raspberry Pi

The way the RPi and its operating system are set up, there is no possibility for a graphical user interface, making it a so-called *headless setup*. This is mostly due to lack of memory on the device. As a result of this, the easiest way is to connect through a terminal via SSH.

For this to work, the RPi needs to be connected to a local network and the accessing device needs to be on the same network. This means, that for de-

velopment purposes, the internet connection of a computer could be shared with the RPi so that they are automatically on the same network.

In order to connect to the device, one needs to find the local IP of the RPi. There are multiple approaches to finding this address, e.g. `arp -a` in a UNIX terminal. The next step is connecting via SSH using `ssh pi@<ip>` with the RPi's password, by default `raspberry`. The device is usually aliased as `raspberrypi.local` so that finding the IP can be circumvented in many cases.

A more detailed version of this tutorial is available on the Raspberry Pi forums (https://goo.gl/LiGQ0D).

3.3. Important Tools and Libraries

3.3.1. I²C, GPIO, and bcm2835

I²C ('Inter-Integrated Circuit', https://goo.gl/lHeXMr) is a serial computer bus that enables communication between peripherals and processors. In the context of the RPi it is used to allow the device to communicate with sensors that are attached to it using a *breadboard*.

- Enter the RPi config using `sudo raspi-config`.
- Select `9 Advanced Options`.
- Select `A7 I2C`.
- Confirm with `Yes` and `OK`.
- Save changes with `Finish`.

The *baudrate* is set too high by default to allow for communication with the Chirp! sensor. This has to be changed by appending `dtparam=i2c1_baudrate=3814` to the `/boot/config.txt` file.

A collection of tools to manage the I²C functionality can be installed with `sudo apt-get install i2c-tools`. This installs a binary called *i2cdetect*, which can be used to list all active I²C connections using `i2cdetect -y 1`.

GPIO ('General Purpose Input and Output', https://goo.gl/ADHVqY) is a user-controlled generic PIN on a circuit board. It is often used to control other

devices using a simple on / off mechanism. In the context of this project it is used to toggle the connected USB water pump.

Usage of these tools in a Haskell environment requires the *bcm2835* C library (https://goo.gl/TJ6OfB) to be installed. It enables the GPIO communication between code and the RPi itself.

```
wget http://www.open.com.au/mikem/bcm2835/
bcm2835-1.50.tar.gz
tar xvfz bcm2835-1.50.tar.gz
cd bcm2835-1.50
./configure
make
sudo make install
```

Listing 1 Steps to install I²C

3.3.2. Motion

Motion (https://goo.gl/zQzRDe) is a command line tool that captures one or more camera feeds off a connected camera device and detects significant motion in the feed. While motion detection itself is not important for this project, the aspect that it can broadcast an MJPEG ('motion JPEG') stream is very useful for establishing a livestream without too much hassle.

A tutorial (https://goo.gl/hT1kgP) was used for the basic setup of the tool, but some alterations had to be made in order to achieve the desired setup.

The command `sudo apt-get install motion` installs motion on the RPi. For the proper setup, the default configuration then needs to be changed slightly:

```
daemon                  on
width                   1280
height                  1024
framerate               100
output_pictures         off
ffmpeg_output_movies    off
stream_maxrate          100
stream_localhost        off
webcontrol_localhost    off
```

Listing 2 Motion configuration

The `width` and `height` properties are depending on the USB camera that is used for the project. The livestream can be started with the command `sudo service motion start`. In this project, motion is always run in *daemon mode* offering an accessible live stream on port 8081 (see *6.5)*.

4. Haskell Back End

4.1. Requirements

The project requires the following technologies in order to compile and run the Haskell source code. All further dependencies are listed in the Cabal configuration file `rpi.cabal`.

Technology	Minimal Version
GHC	7.10.3
Cabal	1.22.9.0
SQLite	3.14.0

Table 3 Requirements for the Haskell project

4.2. Folder Organisation

All Haskell code resides in the `rpi/` directory. Most of the folder organisation is either mandated by the use of Haskell and Cabal or by following established best practices. Files marked as 'local' need to be created separately for each new installation as they should not be tracked by version control.

Name	Local	Description
HappyFlowers/	No	Contains Haskell source code.
happyflowers.db	Yes	SQLite database.
happyflowers.sql	No	Contains SQLite database setup.
cabal.config	No	Locks dependencies.
Main.hs	No	Entrypoint for the Haskell back end application.
rpi.cabal	No	Cabal configuration file listing dependencies and executables.
rpi.cfg	Yes	Internal configuration file for the application.

Table 4 Folder organisation of the Haskell project

4.3. Setting Up the Cabal Sandbox

Ideally, packages are supposed to be developed in an environment where they are unaffected by other developments and in turn do not affect these developments. Most programming languages offer such a setup and Haskell is no exception to this rule.

In Haskell there are two options to create an isolated development for a package. *Cabal* (https://goo.gl/XgyzVT) is a tool originally released in 2005 that aims to facilitate packaging of Haskell software and modules and is the default package manager used by GHC. It uses a global package repository to manage project dependencies, which often leads to version conflicts. It is because of this that later versions of the tool introduced a feature called *sandboxing* (https://goo.gl/u8uv7s) where packages could be developed in a truly isolated environment.

Due to the various problems developers have been facing with Cabal during its lifecycle, a new tool called *Stack* (https://goo.gl/HRqoFZ) has been introduced in 2015. Its aim is to reduce the number of version conflicts during dependency management while maintaining compatibility to the existing Cabal ecosystem. Since its creation, Stack has been the preferred choice for most Haskell developers.

'happy flowers' was designed to use Stack initially, however, there were issues with the setup on the Raspberry Pi. A setup using Cabal and cabal-sandbox worked on local machines and the Raspberry Pi so this was the preferred setup.

Setting up a cabal-sandbox requires a `.cabal` file where the project is configured and its dependencies noted and the command `cabal sandbox init` which creates the sandbox and lets Cabal know where further dependencies should be installed. Dependencies can then be installed using `cabal install` while inside the sandbox directory.

All versions are locked using `cabal freeze`. This prevents dependency version conflicts from happening. The locked versions are noted in the `cabal.config` file.

Cabal offers the following tasks:

- `cabal build rpi`: Installs dependencies and builds the project using the given configuration flags.
- `cabal run rpi`: Installs dependencies, builds the project, and then runs the executable.
- `cabal configure -f`: Sets flags for the build phase. In this project, flags are used to differentiate between development and production environments (see *4.7*).

  Due to using GPIO and I^2C functionality, the executable needs to be run with administrator privileges, which does not work with cabal. It is thus required to first build the project with the desired configuration and then run the executable directly using `sudo ./dist/build/rpi/rpi`.

  A separate executable called `I2CTest` is available that can be used to test the I2C functionality without including any other components of the project.

- `cabal build I2CTest`: Installs dependencies and builds the project.
- `cabal run I2CTest`: Installs dependencies, builds the project, and then runs the executable.

4.4. SQLite Database

SQLite (https://goo.gl/0ziZrB) is a relational database management system that is not a client-server database engine. This is a considerable advantage for a project like this since it reduces the number of processes that need to be managed at any given time. This project uses SQLite version 3, which is incompatible with older versions.

The database is used to store information about the flower the device is connected to, along with historical data about moisture level measurements and waterings. Thus the entities are very much isolated, which in turn keeps the database architecture rather simple.

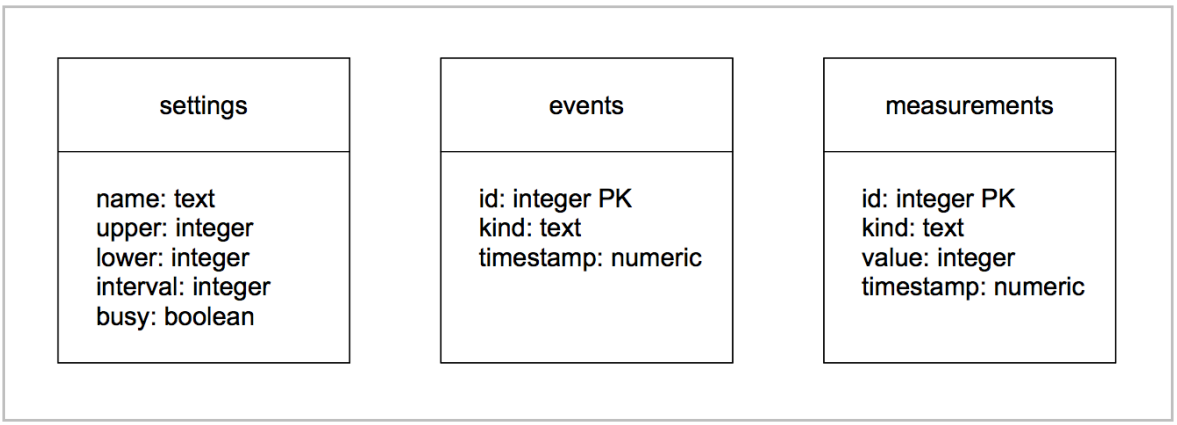

Figure 5 Entity Relationship Model for the SQLite database

Communication with the SQLite database is handled through a single Module called `HappyFlowers.DB`. It encapsulates calls to the `sqlite-simple` library and exports a collection of methods that are used to query, update, and create data. Queries are included as strings (i.e. *query syntax*) instead of relying on a *method syntax* offered by other frameworks.

Error handling is performed using a combination of the `Either` and `Maybe` monads. The database calls are parsed to a `FromRow a => Either SQLError [a]` monad, which is then converted to a `FromRow a => Maybe a` monad. This happens for two reasons:

- The specific error is not of particular importance since error handling in the project only differentiates between success and failure.
- Some functions return records containing data from multiple queries. Instead of handling multiple, sometimes complex combinations of results it is easier to return a `Just` if all queries succeeded or `Nothing` if even one of them failed.

The only required entity is one entry in the settings table to describe all the application settings. It consists of the flower's name, the upper and lower moisture limit, and the measurement interval.

4.5. HTTP Server

Part of the back end is an HTTP server, powered by the *Scotty* library. It offers a simple programming API that can be used to implement routing, re-

quest handling and JSON distribution. It is possible to include any `wai-middleware` in a Scotty server implementation. By default, the server is run on port 5000.

4.5.1. REST API Implementation

The project exposes a fully *RESTful* API that can be used for communication between the front and back end. A set of routes produces either a JSON response containing the requested data or an HTTP error code. By internal convention, the route handlers are always named with their HTTP verb and the data handled by the request, e.g. `getSettings` or `putSettings`.

Since the data returned by the database code is always contained in a `Maybe` monad (see *4.4*), it made sense to write a handler for those situation that returns the Just value as JSON or an HTTP error that is passed as a parameter. The `jsonOrError` function does just this: it accepts an HTTP `Status` code and a `Maybe` value and uses the `maybe` function to produce the appropriate response.

PUT and POST handlers expect a JSON payload in the request body. This payload contains the data submitted by the users. In order to access fields from this payload, Scotty's `jsonData` function is used along with custom records. These types also follow an internal convention and are named after their API endpoint, e.g. `PutSettingsBody`. The value returned by `jsonData` needs to be typecast to `FromJSON a => ActionM a`. From this point on, all values that are part of the record are accessible using their named accessors.

Some endpoints of the API require authentication. These endpoints expect a `token` field to be part of the request body (see *6.1*).

4.5.2. Static File Server

The HTTP server is also used to serve static files, in this case the web front end, using a piece of middleware called `wai-middleware-static`. With the Static middleware Scotty recognises all requests that don't target one of its routes and attempts to serve files from a local directory. Static files are generated as part of the front end build process, defined in section (see *5.4*).

All requests that do not target the API are rewritten to target the root of the server where `index.html` is served. From this point on the JavaScript code takes over further routing (see *5.6*). This is done using the Rewrite middleware, which is part of the `wai-extra` package.

4.6. WebSockets Server

Real-time communication is used throughout the project to allow constant live-updates to all connected clients. This is realised with a WebSockets implementation, provided by the *websockets* library. The server acts as an intermediary and simply broadcasts messages from one client to all other connected clients (see *6.3*).

Since WebSockets require an internal state to keep track of connected clients and their current state, the code to realise this is slightly more complicated than other parts of the codebase. The code makes use of many concurrency features like `MVar` and various functions to read and modify those variables.

4.7. Hardware Process

It is the responsibility of the back end to spawn and maintain the processes of taking a measurement of the flower's moisture level and reacting to the retrieved data, i.e. activate the pump. The process runs inside a WebSockets client so they can inform other connected clients about any new information gained from the sensors (see *6.3*).

Since local development environments do not have access to I2C and GPIO devices and ports, they need mocking functions that simulate the data. The environment can be changed between production and development using `cabal configure -f Development` for development mode and `cabal configure -f -Development` for production mode. A production system is considered to be the RPi, thus access to sensors is considered possible. Other systems use the mocking functions.

This differentiation is done using *CPP* ('C preprocessor') and `{-# LANGUAGE CPP #-}` statements. Code can then be enclosed in a `#if #else #endif`

structure so that certain parts of the code are only included in the compiled result if certain criteria match.

4.8. Configuration

A configuration file is used to customise some of the properties of the application. It is designed to offer a high degree of flexibility without introducing unnecessary complexity.

```
password=test
frame=140
```

Listing 3 Default project configuration

In the current state of the project, the configuration determines the password that is used for user authentication (see *6.1*) and the time frame for the display of historical data on the web front end.

All communication with the configuration is handled through a single module called `HappyFlowers.Config`. The location of the configuration file is also defined in this module. By default, a file called `rpi.cfg` in the Haskell root directory is expected.

- The file contains one entry per line. An entry is terminated with a newline (`\n`) character.
- Each entry consists of a name and a value in the format `name=value`.

  Since the existence of the configuration file and its formatting is closely connected to this project, error handling was intentionally left out of the module.

4.9. Back End Architecture

The back end is a Haskell application consisting of an API Server, a static file server, a WebSockets server, and a WebSockets client representing the

hardware processes. These services communicate with each other and are used to access the database from the front end.

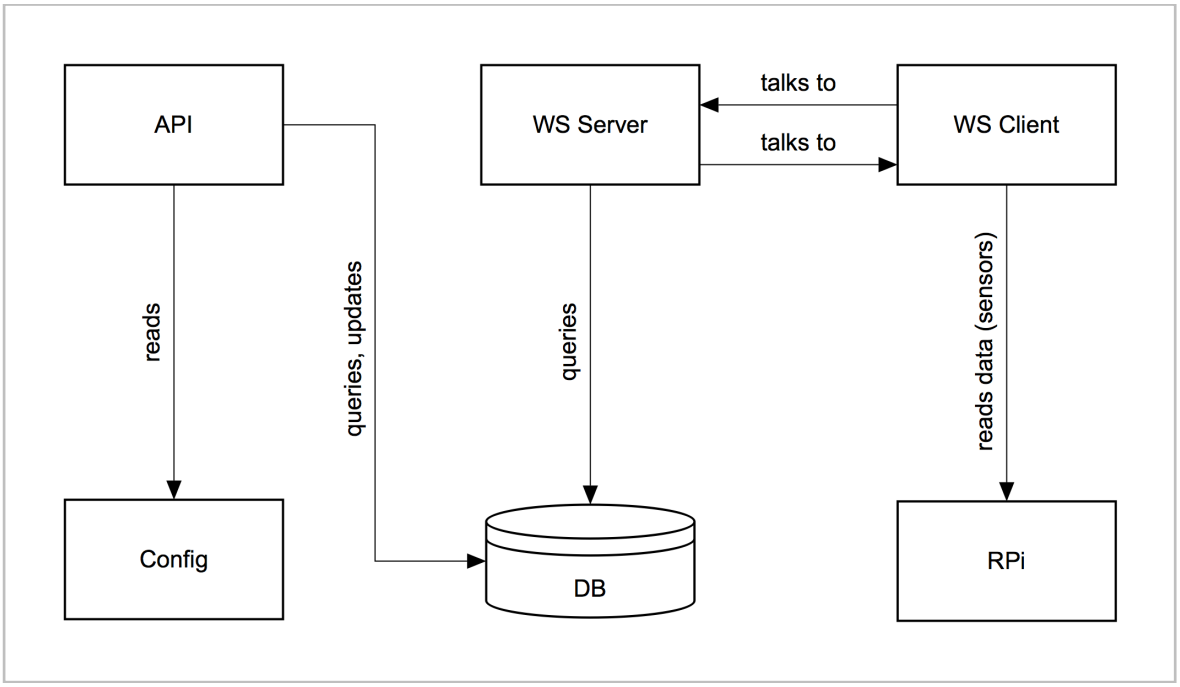

Figure 6 Components and communication inside the Haskell project

4.10. Important Libraries and Tools

One of the primary goals of this project was not to *reinvent the wheel*. That means using existing libraries whenever it was deemed appropriate was essential in planning the development of the project. The use of external libraries needs to be understandable and transparent as the project will be used for teaching purposes.

Library	Description
scotty	An easy-to-use, flexible web server. Scotty is extended using multiple pieces of middleware.
sqlite-simple	SQLite database bindings. Allows statements to be executed against a local database.
websockets	A WebSockets client and server library.
aeson	A Library for encoding and decoding JSON values.
HPi	Enables communication between Haskell code and the Raspberry Pi's hardware and ports. HPi depends on the bcm2835 C library (see *3.3.1*).
jose	A JSON Web Token library. Used to encode and decode the token string.

Table 5 Important libraries for the Haskell project

Using tools to enforce commonly accepted best practices helped create code that not only runs without any apparent errors, but also follows certain coding conventions that improve ease of reading and understandability.

Tool	Description
cabal	Used as the central dependency manager and build system throughout the project. A cabal sandbox is used to create an isolated environment.
haddock	Automatically generates documentation based on standardised code comments (see *4.11.1*).
hlint	Validates code against a set of conventions (see *4.11.2*).

Table 6 Important tools for the Haskell project

4.11. Coding Practices and Guidelines

4.11.1. Documentation

All Haskell code in this project is documented following the *haddock* documentation standard (https://goo.gl/uypVFR). It requires all modules to contain a documentation header and mandates that publicly available, i.e. exported functions are documented. It is possible to document function parameters, however, this is reserved for cases where the type of the parameter is not sufficiently expressive.

Generating the documentation is included in the cabal setup and can be performed using `cabal haddock --executables`. This generates documentation for all packages that are part of an executable package.

haddock can include documentation of dependencies so that internal documentation may reference other libraries. This requires passing the `--enable-documentation` flag during the `cabal install` step.

4.11.2. Style Guide

This project follows a set of style guides that have been widely accepted by the Haskell community as best practices:

- Haskell Style Guide (https://goo.gl/TXVTY6)
- UPENN Style Guide (https://goo.gl/Y7wvdo)

There is no automatic enforcement of these practices, instead the code is reviewed for its conformance on a regular basis.

4.11.3. Linting

All Haskell code is linted using *hlint* and its available IDE integrations. All available options (`Default`, `Dollar`, `Generalise`) are used in order to find possible improvements throughout the codebase.

5. JavaScript Front End

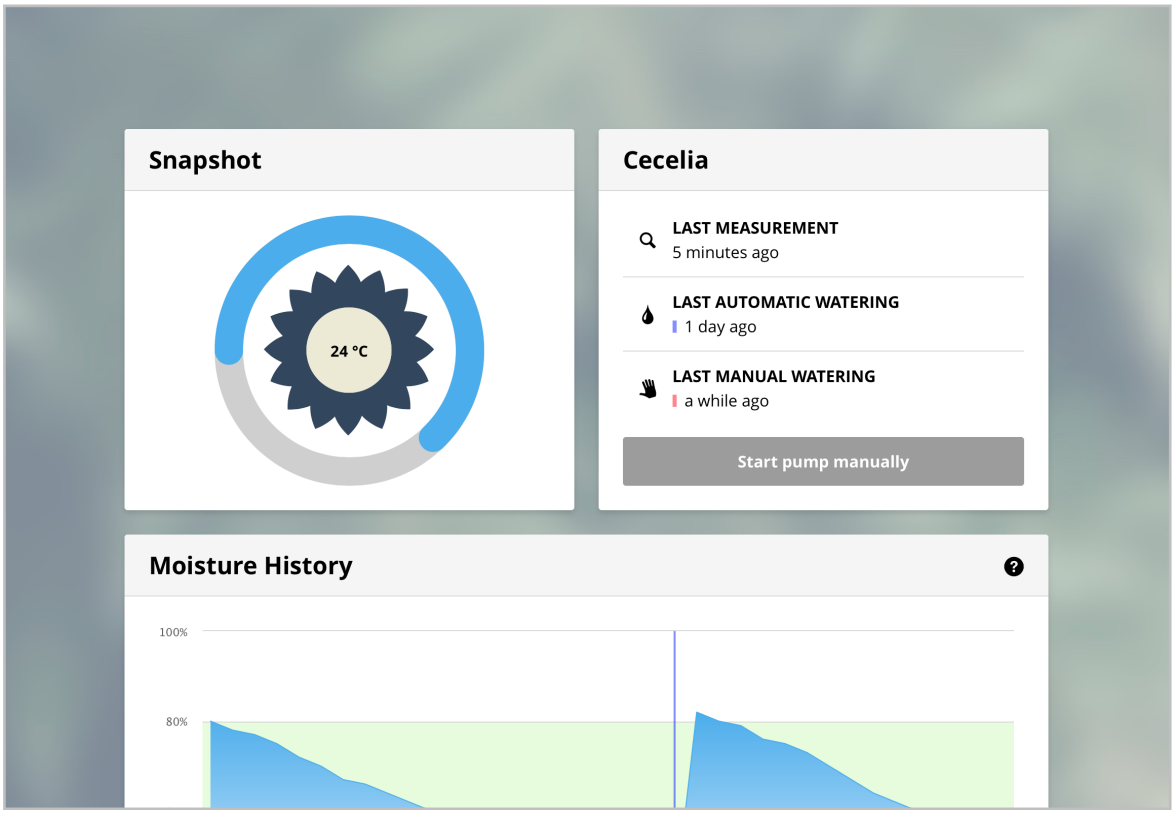

Figure 7 Web front end showing the live dashboard

5.1. Choosing the Right Framework

The project description offered the choice between various frameworks that could be used to implement the web front end.

- **React / Redux** (https://goo.gl/B76Bmi): *React* is a declarative interface library that has become one of the leading frameworks for creating a modern front end application. It is written in JavaScript — usually with the *ECMAScript 6* syntax — and can be developed to follow a very functional approach. *Redux* can be used to augment React with sensible state management and event handling, which leads to functional purity, further increasing the similarity to a functional language.

- **Elm** (https://goo.gl/FnoyJ4): *Elm* is a new language that is very similar in syntax to Haskell that compiles to JavaScript. It boasts its type safety as one of

the most important benefits, along with its speed. It is very similar to React in its underlying motivations.

- **PureScript** (https://goo.gl/zCtkLx): *PureScript* is another strongly typed language that compiles to JavaScript. Just like with Elm, its syntax is very similar to Haskell. It is, however, less focused on creating a separate environment and instead offers bindings for React as well.

- **ReflexDOM** (https://goo.gl/0GZxq2): *ReflexDOM* is a Haskell library that allows developers to create web applications in Haskell. It does not compile to JavaScript, which sets it apart from all the other entries.

  React is, of course, currently the big player in the world of front end development. It is because of this that it was the favourite from early on. The other tools, especially ReflexDOM require a lot of additional code to get the same result as with a traditional JavaScript setup. Elm and PureScript on the other hand lack a lot of the ecosystem that React already offers.

  While it would have been interesting to fully commit to Haskell and develop the front end using one of the alternatives, React still seemed like the reasonable choice. It would save time that could be used to create a performant, well documented front end, as well as an all-Haskell back end that can actually be used for teaching purposes, as was the goal of the project.

  Using various conventions and external tools, the React setup was shaped into a form that is very similar to a fully functional setup. A strong reliance on React's inherent immutability, along with Redux's pure state management and *immutable.js's* data structures helped create a JavaScript setup that follows many of the best practices of functional languages and thus still kept with the spirit of the project.

  React apps are often written in ECMAScript 6 (*ES6*), the upcoming version of JavaScript. It brings many new features like classes and modules. Part of React is its JSX syntax, which is used to include HTML templates in a component's `render` method. Since most of ES6's and all JSX features are not supported in current browsers, the code needs to be transpiled to working ES5 code (see *5.4)*.

5.2. Requirements

The project requires the following technologies in order to compile and run the JavaScript source code. All further dependencies are listed in the yarn configuration file `package.json`.

Technology	Minimal Version
Node	7.2.0
yarn	0.16.0

Table 7 Requirements for the React project

5.3. Folder Organisation

All JavaScript code resides in the `web/` directory. Most of the folder organisation is either mandated by the use of React and yarn (https://goo.gl/1bi0UH) or by following established best practices.

Name	Description
public/	Static files accessible on the web front end.
src/actions/	Redux action creators.
src/components/	React components with per-component styling.
src/containers/	React / Redux containers with per-container styling.
src/middleware/	Custom Redux middleware.
src/reducers/	Redux reducers.
src/index.css	Global application styles.
src/index.js	Entrypoint for the React front end application.
package.json	yarn configuration file listing dependencies and executable scripts.
yarn.lock	Locks dependencies.

Table 8 Folder organisation for the React project

5.4. Setting Up the React Environment

There are many ways to create React apps, most of which rely on some sort of build system to transpile and bundle the *ES6* and *JSX* code. In 2016, a tool

called *create-react-app* (https://goo.gl/QxYuR4, 'CRA' in following) was released that aimed to simplify the process of setting up and maintaining a React app. It provides scaffolding for the directory structure, as well as for build and test tasks. Apart from the scaffolding it offers a library called `react-scripts` that makes it easy to keep the environment up-to-date.

Setting up the project is as easy as calling `create-react-app my-app`. This command creates a new project called my-app with all the setup and tools included. This is a major benefit over handpicking all the components of the build chain and a great timesaver.

Once the setup is done there are two important scripts that can be run inside the project directory:

- `yarn start`: Runs the application in the development environment. This creates a `webpack-dev-server` that updates the application in browsers on the fly. Errors and warnings are printed to the console.
- `yarn run build`: Builds the application in the production environment. This creates an optimised build for deployment.

Besides making the project setup easier and more transparent due to conforming to widely accepted standards, CRA also removes the need for a dedicated build system like Gulp. It used to be that developers were faced with the choice of using a build system or rolling their own system using yarn scripts. CRA uses the yarn scripts approach, but takes away the complexity of setting up the toolchain. In the end, this leads to a more understandable setup so more time can be spent developing the app, rather than fixing the project environment.

5.5. Component Architecture

Components are a core feature of React. Most of its architecture revolves around the use and creation of custom components and their composition to create an application. Components then have *props* and *state*, which manage the data contained in a single component.

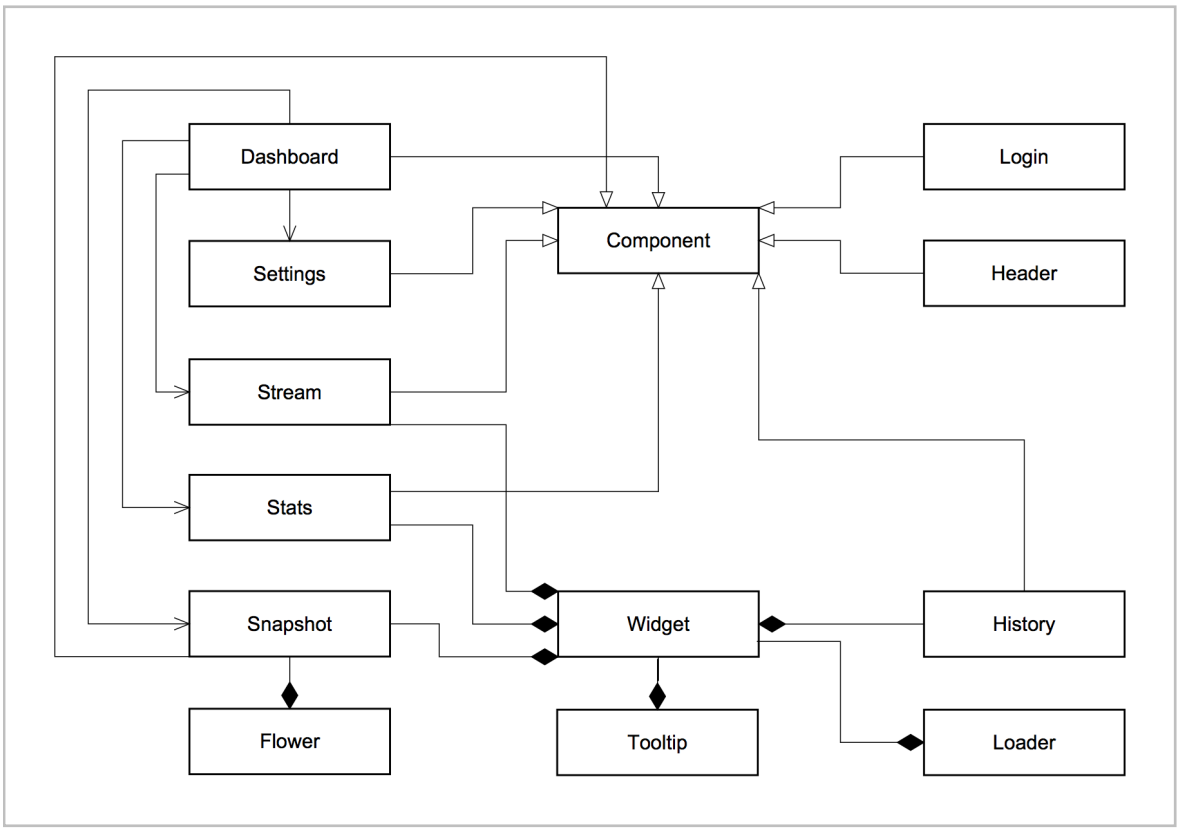

Figure 8 Component hierarchy for the React project

5.5.1. Component vs Container

While React itself does not differentiate components and containers, it has become an accepted best practice to separate these two kinds of elements in order to improve the separation of concerns and possibly improve performance due to less unnecessary re-rendering, among other benefits (see https://goo.gl/sMTpeI).

A component (sometimes also called *dumb component*) is an element that receives a collection of props and displays it in a given way. There is not much logic involved and oftentimes these elements are written as stateless functional components.

Containers (sometimes also called *smart components*) are more complex elements. They are usually composed of multiple components and have an internal state, thus they are written as classes. Their usual task is to define the composition of components and to fetch data as part of the lifecycle methods. In the context of a Redux architecture this often means that containers manage the connection to the application state using `mapStateToProps`.

5.5.2. Widgets

Most of the front end application is build using widgets that represent some kind of data. They can be gathered in a grid and contain any kind of data. In addition to this they can show a loading indicator and a tooltip showing help text.

`Widget` itself is a presentational component, i.e. a dumb component. It is only through the use of containers that they gain meaning as containers can set the widget body to anything they want, usually some representation of part of the Redux application state.

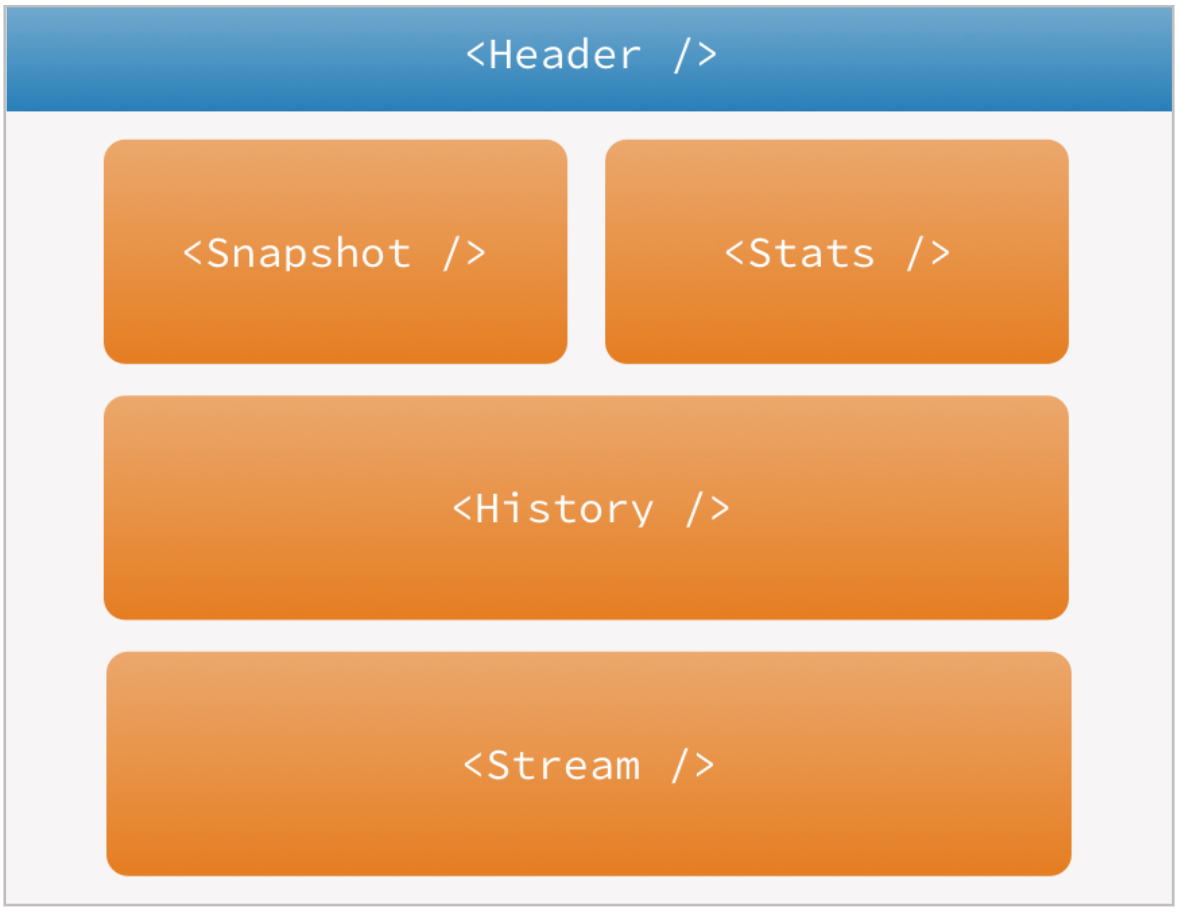

Figure 9 Usage of widgets as demonstrated by the dashboard page

5.6. Single-Page Application Routing

Routing URLs to the appropriate part of a single-page application is a crucial task in such applications. Most modern front end frameworks offer some sort of router, either natively or through the use of third-party libraries. Re-

act is no exception to this rule with `react-router` providing a flexible routing mechanism that connects a route to a component.

A router expects that all requests to the application are rewritten to target the router itself. In this project this means that all non-API requests are redirected to `index.html`, which then starts the internal routing that manages the pages of the application. By using the native browser history and JavaScript APIs that allow modifications of the history, routers can rewrite URLs so that deep linking — i.e. linking to a specific part of an application, oftentimes several layers deep — of web applications is still a possibility.

The front end application only offers three routes. One for the homepage that acts as a dashboard and displays widgets containing the application data, one for the settings section, where users can change things like the moisture limits, and one for the login sections that is used to authenticate users.

5.7. Redux Architecture

Data flow is explicitly one-way in React. Containers pass their data down to other containers, which eventually pass the data to their components. Due to this, React can optimise performance by eliminating unnecessary re-renders and only rendering the parts of the application that actually changed. It also makes the separation of concerns very strict and shows the purpose of each component.

Some confusion often arises when developers wonder which component is responsible for updating a certain part of the data. Quickly, the code turns into the so-called *callback hell*, which Node.js developers have been trying to avoid for many years now. Sub-components have to be passed a callback that they can execute whenever they update any data.

This is where Redux comes into play. It is responsible for managing the global application state. No other component manages the state, except for UI and explicitly internal state. Other components can access the state using the `connect`, `mapStateToProps` and `mapDispatchToProps` functions, which inject data from the application state into components.

In order to change data from the state, components need to `dispatch` an *action*. Actions are a terminology coined by the *Flux* (https://goo.gl/LDQLOM) architecture, which Redux is very much based on. They have a type by which they are differentiated and a payload which represents the data that should be changed in the state. Actions are then caught by *reducers*, pieces of code that actually modify parts of the state and return the new representation. After a change, the new state is then provided back to the components and React takes charge of re-rendering the parts that have changed.

```
{
  auth: {
    jwt: String,
    isLoggingIn: Bool
  },
  history: {
    snapshot: Int,
    events: List<Map>,
    measurements: List<Map>,
    isFetching: Bool
  },
  notifications: List<Map>,
  settings: {
    data: Map,
    busy: Bool,
    isFetching: Bool,
    isSubmitting: Bool,
    isErroneous: Bool
  }
}
```

Listing 4 Redux state structure

5.7.1. Middleware

Middleware form an essential part of Redux architectures. They can be used to transform an action while it is dispatched and they can be combined so that they form a very flexible toolchain. While there are many existing pieces of middleware available on yarn, it is also quite easy to create custom middleware code.

This project uses the following middleware:

- **redux-thunk**: Allows action creators to return functions instead of actions. These inner functions can then access the `dispatch` and `getState` functions to dispatch further actions. This is especially useful for asynchronous tasks like communicating with an API.
- **redux-logger**: Logs all Redux actions to the console. This allows tracking a fully reconstructible evolution of the internal state. Every log entry lists the previous state, the dispatched action, and the newly created state. This piece of middleware is only used in the development environment so that it does not affect the performance or security of a production application.
- **api**: Calling an API endpoint and dispatching actions based on the state of the call is a repeatedly occurring process in this project, thus it made sense to create a custom piece of middleware that facilitates this process.

  The API middleware expects three actions: an action that is dispatched before the request is started and one that is dispatched if the request was successful or erroneous, respectively. It also expects a function that calls an API and allows appending custom data to the action payload, as well as callbacks to execute based on the result of the call.

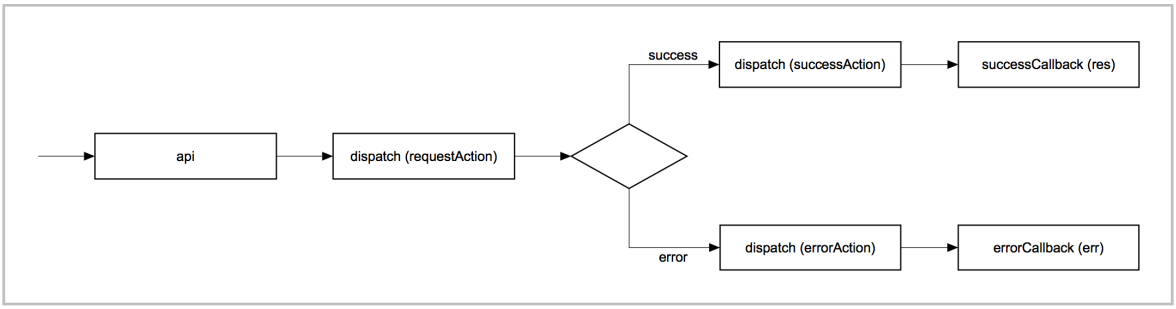

Figure 10 Custom Redux middleware for communication with an API

5.8. Immutability

One of the core principles of both React and Redux is the immutability of the internal state. While these two tools somewhat enforce this convention, other JavaScript code is unaffected by this and can still use mutable data structures.

A library called `immutable.js` (https://goo.gl/F85RRu) is used to add immutable data structures to JavaScript. It exposes `List` and `Map`, which can be used to replace arrays and objects, respectively. These new structures can be created directly with a constructor or with the `fromJS` method when a deep copy is desired. Using the `set`, `get`, `setIn`, and `getIn` methods, structures can be queried and mutated. Every mutation leads to the creation of a new structure, a core aspect of immutability.

The case for immutability is clear. A structure that never changes does not need to be kept in sync, since it is always the same. It also makes comparison of two objects easier as they can be compared in a shallow fashion. This leads to many benefits, especially in a React application, where the state is always passed down the component hierarchy. Immutable data structures allow for a simple comparison of the previous state and the new state, which can be used to prevent unnecessary re-renders from happening.

With the help of these immutable structures, JavaScript code can become more pure and thus more similar to a functional language like Haskell. It made sense to use immutable.js so that many concepts could be shared between the front and back end code.

5.9. WebSockets Client

The front end is informed about new measurements and waterings through a WebSockets client. It is a standard implementation using the browsers built-in WebSockets integration that does not rely on any libraries like *socket.io*. This is a consequence of attempting to keep the back end code simple and understandable.

The connected socket is then enhanced with custom handlers. The handlers act on certain events, like the start and end of the connection, and standard messages. Messages are parsed as a JSON object containing a type property that is used to differentiate the kinds of messages. The actual information is then included in a payload and dispatched as part of each respective action. The connection is opened whenever users access a site of the application and closed whenever the user then leaves the site.

5.10. Front End Architecture

The front end is a JavaScript application consisting of a router, reducers and actions, a HTTP client, a WebSockets client and a set of components. These parts form a React application that communicates with the back end through the API and WebSockets.

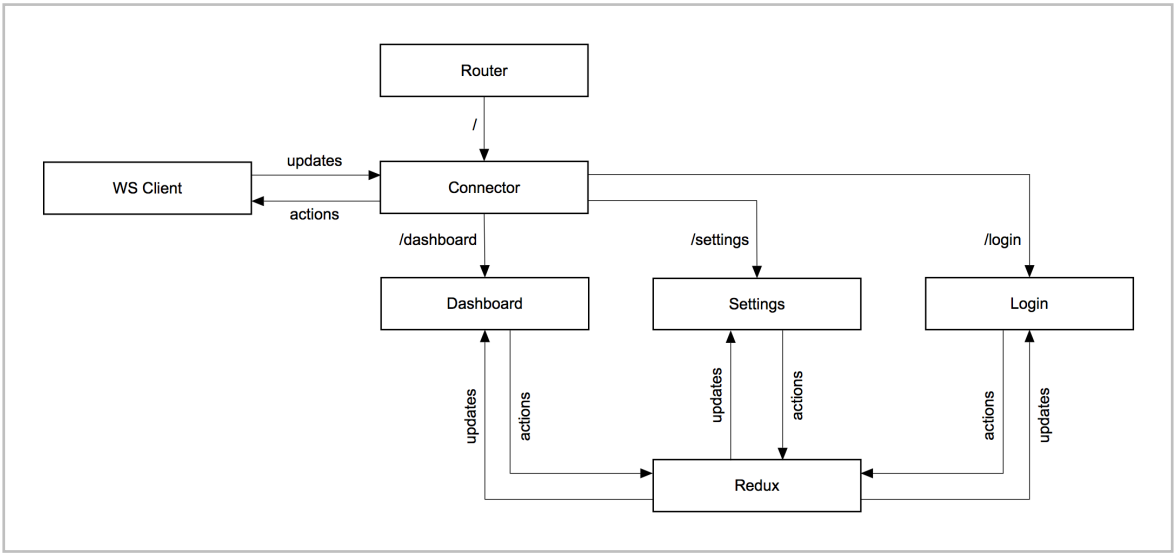

Figure 11 Components and communication inside the React project

5.11. Important Libraries and Tools

Much of the technology stack was determined when React was chosen as the framework for the front end application. It is because of this that the project uses libraries that are very common for a React-Redux setup, extended only by a select few other libraries. All this leads to a highly understandable and easy-to-document codebase.

Library	Description
react	A declarative view library for modern single-page JavaScript applications.
highcharts	Generates SVG-based charts that are used to display the historical application data. Integrated using react-highcharts.
redux	Facilitates state management in JavaScript applications. Allows integration of various pieces of middleware, e.g. for automatic logging.
immutable.js	Adds support for immutable data structures to JavaScript.
axios	A promise-based HTTP client for JavaScript applications.
SSGS	A flexible grid system for CSS and SCSS.

Table 9 Important libraries for the React project

Just like with the choice of third-party libraries, the choice of tools was fixed very early on in the project. By starting the project using *create-react-app* (see 5.4) it was clear that technologies like *babel*, *eslint* and *webpack* would be involved. These tools made it easy to create a maintainable framework for the process of developing the product, but they also allowed for rapid iteration in the design and usability areas.

Tool	Description
yarn	A package manager for Node.js and JavaScript code. Also acts as a build system.
create-react-app	Sets up a React project and offers an efficient build toolchain for such a project.
babel	Transpiles ECMAScript 6 code to ES5 code so current browsers fully understand the code. While support for newer language features is constantly improving, this step is still needed.
eslint	Validates JavaScript code against a set of conventions (see *5.12.2*).
webpack	Bundles front end assets (JavaScript, CSS, images) so a single bundle containing application code and its dependencies can be published.
stylelint	Validates CSS code against a set of conventions (see *5.12.2*).
postcss	Transforms CSS code with a set of plugins. In this project the code is extended using autoprefixer, a tool that adds vendor prefixes to certain CSS properties.

Table 10 Important tools for the React project

5.12. Coding Practices and Guidelines

5.12.1. Documentation

All JavaScript code in this project is documented following the *JSDoc* documentation standard. It requires classes and methods to contain proper documentation of their parameters and return values.

Part of the codebase is somewhat self-documenting. Components can have so called *prop type definitions*, by which the type of their props is defined and warnings are produced when there is a type mismatch.

There is currently no viable option to automatically generate documentation based on JSDoc comments as support for some ES6 and JSX syntax is still missing.

There is no particular style of commenting the CSS code as its syntax is mostly self-explanatory. Comments are used to structure longer files into more easily understandable sections.

5.12.2. Style Guide

This project follows a set of style guides that have been widely accepted by the JavaScript and CSS community as best practices:

- Airbnb JavaScript Style Guide (https://goo.gl/LaiwvB)
- Airbnb CSS Style Guide (https://goo.gl/l44URs)
- Code Guide (https://goo.gl/NfZlBK)

There is no automatic enforcement of these practices, however, linters for both JavaScript and CSS use these style guides as a base for their configuration.

5.12.3. Linting

All JavaScript code is linted using *eslint*. The `react-app` configuration that is part of the `create-react-app` project is used as a base for the ruleset. This

configuration is tailored to a modern React app with ES6, JSX, and React features.

All CSS code is linted using *stylelint*. The standard `stylelint-config` that is the combination of multiple popular style guides is used as a base for the ruleset. The configuration in this project does not enforce rules like a fixed order of properties in order to keep the complexity low when authoring CSS code.

6. Communication

6.1. Authentication and Security

This project uses a token-based form of authentication since it only supports a single user so that session handling and user management is not necessary. The token allows users to log in easily and keep their privileges for the rest of their local browser session.

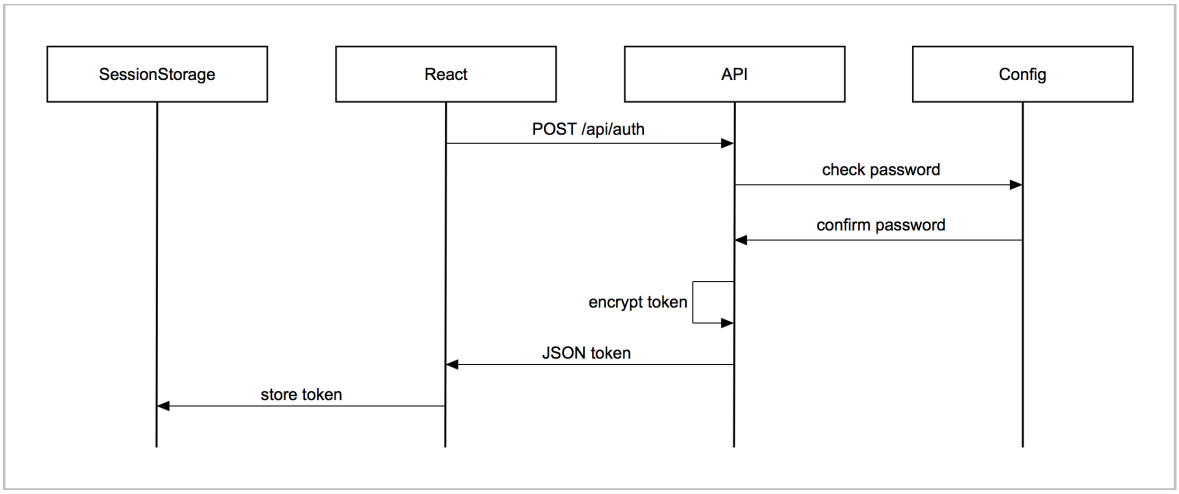

Figure 12 Authentication flow

The generated token is encrypted and decrypted using the HS384 ('HMAC using SHA-384') hashing algorithm. The shared secret is defined as part of the code in the `HappyFlowers.API.Routes` module.

Tokens are stored in the browser's `sessionStorage`. This means that users remain logged in for the duration of their browser session or until they manually log out. They are automatically logged out once they close the browser, upon which the token is removed.

6.2. REST API

A REST API is used for communication between the front and back end. The API is fully compliant to the REST standard and uses the appropriate HTTP verbs and naming conventions for its endpoints.

Endpoint	Method	Request Body	Response Body
/api/auth/	POST	`{ password: String }`	`{ token: String }`
/api/history/	GET	–	`{ events: Array, measurements: Array }`
/api/settings/	GET	–	`{ name: String, upper: Int, lower: Int, interval: Int, busy: Bool }`
/api/settings/	PUT	`{ token: String, name: String, upper: Int, lower: Int, interval: Int, }`	`{ name: String, upper: Int, lower: Int, interval: Int, busy: Bool }`

Table 11 REST API endpoints with request and response bodies

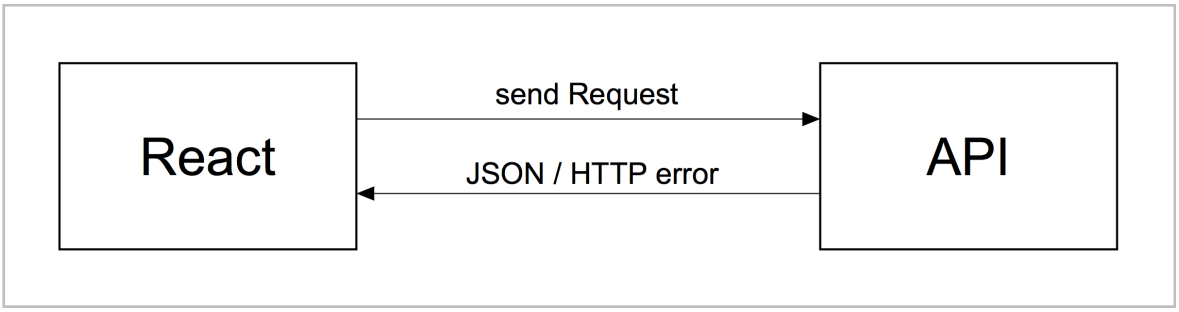

Figure 13 Communication between front and back end through a REST API

6.3. WebSockets

A WebSockets server is used for real-time updates of connected clients. This allows clients to always keep track of changes to historical data and application settings. The communication works by passing JSON objects.

```
{
  type: String,
  payload: Object
}
```

Listing 5 Standard WebSockets events payload

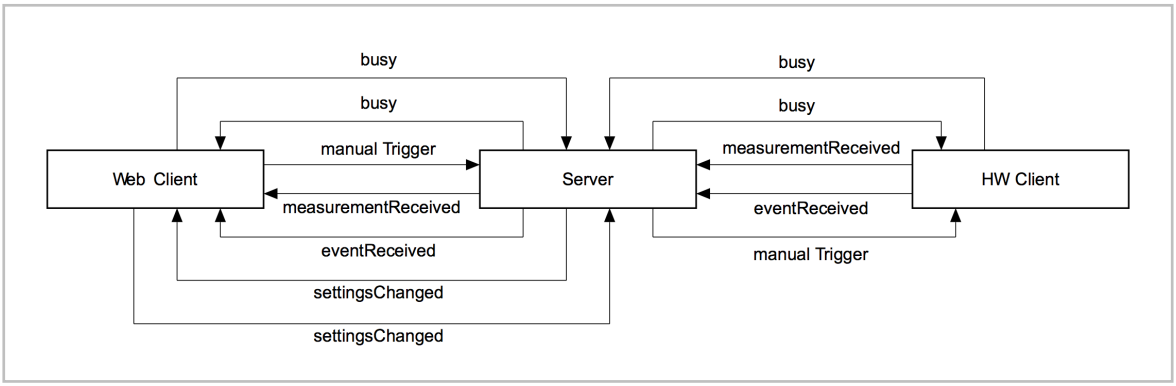

Figure 14 Communication between clients using a WebSockets server

6.4. Preventing Race Conditions

There are situations where handling historical data and real-time data could potentially cause race conditions. This is a rather rare issue in this particular project, but it could lead to problems in applications where data integrity is more important.

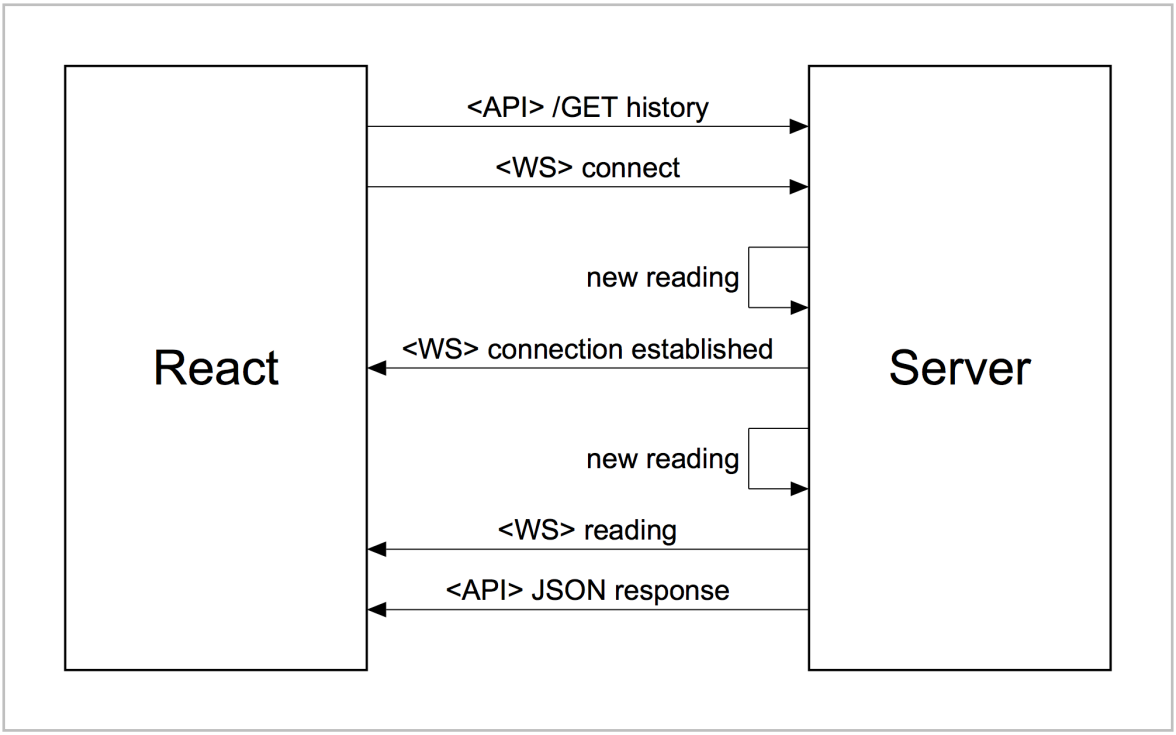

Figure 15 Race condition with API + WebSockets fetching data

This would lead to the following state for events where the historical data would overwrite the new data and events would go missing.

```
{
  events: [...history]
}
```

Listing 6 Erroneous events state due to race conditions

While there are multiple ways that would prevent this situation, most of them would be beyond the scope of this project. For this reason the following approach was selected where initial history data is sent by the WebSockets server instead of relying on the REST API.

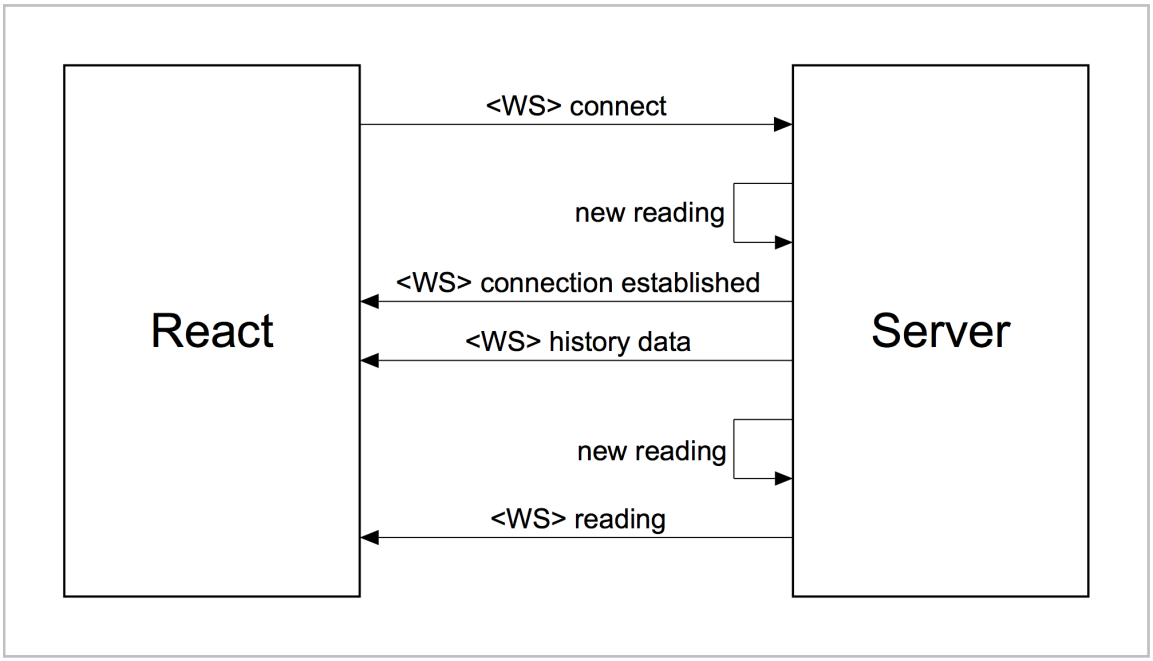

Figure 16 Prevention of race conditions

With this fix in place, the events are fetched correctly and stored in the correct order. New events cannot go missing.

```
{
  events: [...history, event]
}
```

Listing 7 Correct events state

6.5. Livestream

Users can view a livestream of the flower. This is achieved using an MJPEG stream provided by the *motion* library, as described in *3.3.2*.

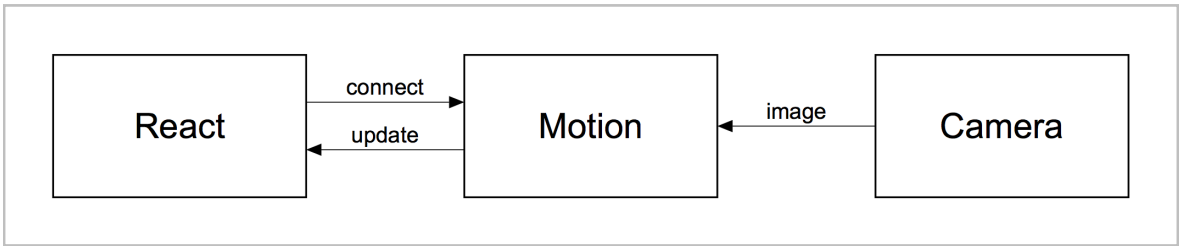

Figure 17 Livestream image communication

7. Processes

7.1. Interpretation of Data

The project reads data on soil moisture and room temperature. Both of these can be retrieved through the I^2C protocol. Both data ranges from 0 to 65535.

	Type	Expected Range	Conversion
Soil Moisture	relative	100 – 800	`round((val - 100) / 700 * 100)`
Temperature	absolute	15 – 30	`round(val / 10)`

Table 12 Data interpretation and conversion

7.2. Measurement

A measurement is the process of measuring the moisture level of the flower the sensor is connected to. The value is measured by the *Chirp!* sensor, as described in *3.2.2*.

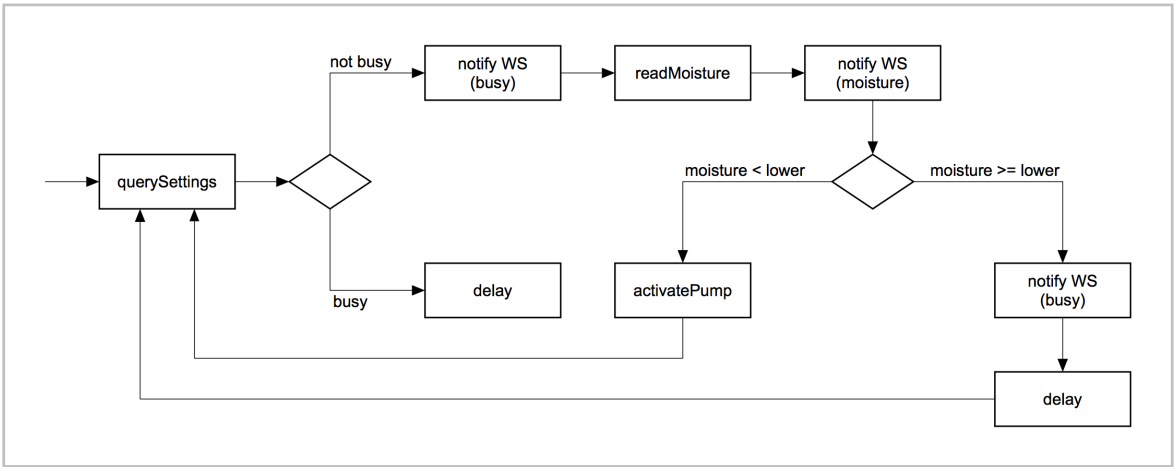

Figure 18 Measurement process

7.3. Watering

A watering is the process of activating the pump for a set period of time, by default for 5 seconds. This step is repeated until the moisture of the flower is higher than the upper threshold defined as part of the application settings.

7.3.1. Automatic Watering

An automatic watering is the result of a measurement that returns a value lower than the lower threshold defined as part of the application settings.-

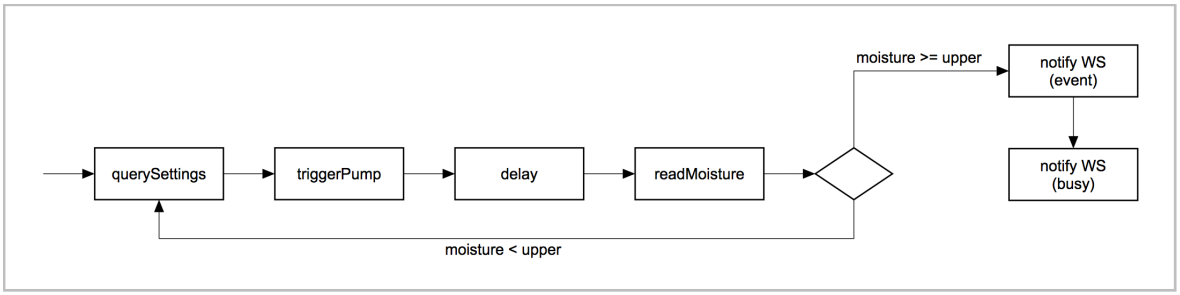

Figure 19 Automatic watering process

7.3.2. Manual Watering

A manual watering is the result of a user interaction on the web front end.

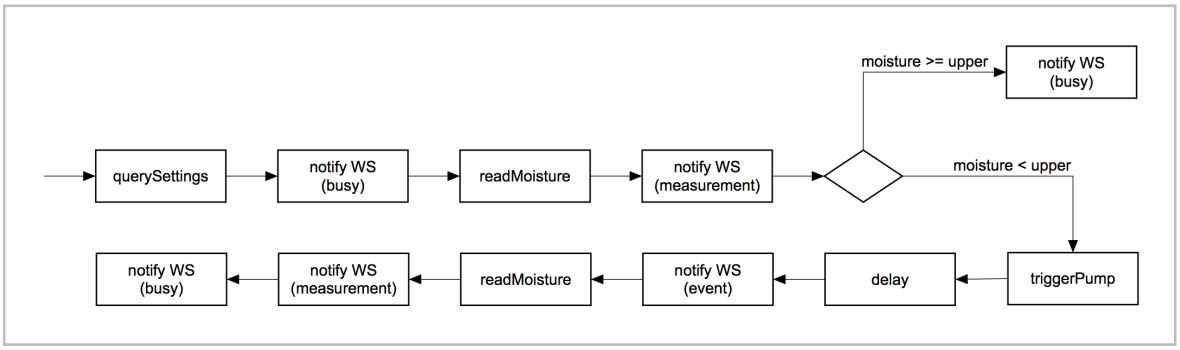

Figure 20 Manual watering process

7.4. Preventing Concurrent Actions

The measurement process and even more so the pump process must always run through without any interruption. This means that users must not be able to activate pump while the sensor is measuring or while the pump is running, otherwise the actions could conflict and produce an erroneous state.

This is ensured using a combination of REST and WebSockets. When a process is started, it updates an entry in the database marking the flower as busy. It also notifies all connected WebSockets clients about the busy state. The web front end prevents activation of the pump while the flower is in the

busy state. By combining REST and WebSockets, it can be guaranteed that the pump is never triggered when it should not be. Also, the schedule checks do not run if the flower is busy and are just delayed by the given interval.

8. Conclusion

One goal of this project was to prove that Haskell could be used to create applications that solve real-world problems. This means that there needs to be a solid ecosystem with a collection of libraries and proper documentation that aids developers in conducting a successful project.

While package repositories like *Hackage* (https://goo.gl/h1bG5y) or *Stackage* (https://goo.gl/bSjdQo) provide a vast collection of useful libraries that solve common problems, many of them are poorly documented and lacking sample code. It is thus up to the developer to figure out the inner workings of these libraries by reading the source code. This leads to a very thorough understanding of the code that is used and has a didactic effect in that developers learn about various concepts they would otherwise not encounter.

Haskell does require some base level of experience to write code efficiently. That means that the barrier of entry is higher than with other languages, e.g. with JavaScript. While this can be off-putting for newer and inexperienced developers, it does pay off in the end. Quite quickly, one develops an eye for recurring patterns, potential for refactoring, and shortcuts that stem from the pure and functional nature of the language.

Internet of Things is a very accessible area of technology so hobbyists enjoy partaking and sharing their experiences. This leads to a wide array of inspiration for starting new projects, but also means that many of the shared projects are either incomplete or poorly documented. Due to this, creators need to be willing to experiment and spend a lot of time on a sometimes fruitless trial and error process. However, especially to a software engineer, the task of creating a tangible product, in this case a piece of hardware, is very rewarding. A lack of personal knowledge caused some delays in working with the hardware, but eventually this part of the project could also be completed successfully.

The front end architecture was the most straightforward part of the project, mostly due to previous personal experience. Without this experience, learning about React and Redux very much depends on choosing the ideal tutorials. Once a developer knows about the best practices for authoring components and how to manage the state, creating a React app becomes a very understandable process (see *10.1*).

Due to the nature of the requirements, it was easy to structure the project without many overlapping tasks. The individual components could be developed in an isolated environment and only communicate with each other once they were ready for integration into the whole system. This also meant that both team members could work in an efficient manner.

All in all the project was an interesting learning experience. The combination of familiar topics with the exploration of unknown areas made 'happy flowers' a project that combined learning and research with hands-on experience. Focusing on keeping the project understandable forced the team members to understand the code at a deeper level and to gain knowledge of advanced topics so that a product could be created that other people can explore and enjoy.

8.1. Recommendation

While the finished product can be used as designed, the much more interesting aspect is its possible usage in a course. All parts of the project — be it the front or back end application or the Raspberry Pi setup — can be used in a course on functional programming, web application development or even internet of things.

As the project is maintained in an open source environment, students and other interested people may read about or contribute to the project and thus learn even more from it. This is a huge benefit since the whole open source community, its related processes and communication skills are of ever increasing importance for modern software developers.

8.2. Evolution Scenarios

Probably the most obvious evolution scenario is the inclusion of additional sensors, e.g. a sensor for brightness, air humidity, etc. These sensors could then be integrated into the calculation of the flower's healthiness and displayed as widgets on the front end.

In the future, multiple devices could be connected to a central server. The web front end could then display an overview of all connected flowers and display statistics about each one. This would likely mean that a centralised user management has to be put in place. It would mostly include the complexity of the front end code, less so for the back end.

A possible option could be to move away from the React setup on the front end and move to an implementation that sticks closer to Haskell with one of the alternatives described in *5.1*. This would necessitate the alternatives to further develop their ecosystem so that the change would not cause unnecessary complexity. The benefits of this would be that an even larger share of code could be used to teach students about Haskell.

Currently, there is no possibility to measure the remaining water inside the tank. This could mean that all watering processes run through without knowing that they do not actually water the flower. With a measurement of the tank's water level the front end could display a warning message of some sort.

In the same vein of thinking it would make sense to include a notification system, either via email or via push notifications. This way, users could stay updated on changes to the flower's status.

9. References

9.1. List of Tables

9.2. List of Figures

9.3. List of Listings

10. Appendix

10.1. Recommended Learning Material

- **Learn You A Haskell** (https://goo.gl/0tqBzz)

  A freely available online book (also available as print) about some of the fundamentals of functional programming and Haskell in particular. This book is easily one of the most engaging ways to learn Haskell and has been useful over the course of this project.

- **Real World Haskell** (https://goo.gl/fdaEXh)

  Just like Learn You A Haskell, this book is also freely available online or acquirable as print. It goes into far more detail and covers a lot more advanced topics than the previously mentioned book. For this project, the chapters about error handling and use of monads and monad transformers have been especially useful.

- **JS Stack From Scratch** (https://goo.gl/HQqWpG)

  This guide explains the technology stack of a modern web application and the tools that are involved in creating such an application. It also covers some more advanced topics like testing and type-safety.

- **React Fundamentals** (https://goo.gl/xY8adJ)

  A 17-part video course about the fundamentals of React, its concepts, and the toolchain that forms the React ecosystem. Towards the end it delves into some more complex applications and explains some of the implementation details of React.

- **Getting Started With Redux** (https://goo.gl/3yf89a)

  A 30-part video course about Redux and its concepts, terminologies, and implementation details. This course by the creator of Redux helps with taking away the magic from reducers, actions, and other concepts.

- **Building React Applications With Idiomatic Redux** (https://goo.gl/Ey6X69)

  This follow-up course to Getting Started With Redux shifts the focus to integrating React and Redux and covers various best practices when authoring modern web applications.

10.2. Recommended Development Environment

While all parts of the project can be developed regardless of the chosen environment, some products and tools were very helpful over the course of this project.

- **Atom**: An open source text editor that is extensible using a myriad of plugins. These include support for syntax highlighting, configuration management, and linting. Since this project uses both Haskell and web languages, an environment offering a well-working integration for all these languages is crucial and Atom is just that.

- **editorconfig**: This tool allows developers to share a common configuration for their editors. This includes settings for indentation (specified on a per-language basis) and use of whitespace. editorconfig is available natively in many editors and as a plugin in many others, including Atom.

- **Atom Linter**: Integration for various linting frameworks, directly in Atom. There are community-created packages for *eslint* (JavaScript), *hlint* (Haskell), and *stylelint* (CSS). These tools allow developers to catch errors before compilation, right in their editor.

11. Honesty Declaration

It is hereby declared that the contents of this report, unless otherwise stated, have been authored by Sacha Schmid and Rinesch Murugathas. All external sources have been named and quoted material has been attributed appropriately.

Date and Location

Sacha Schmid

Rinesch Murugathas