happy flowers

Project Report

Sacha Schmid
Rinesch Murugathas

Daniel Kröni

Windisch, 28 December 2016

1. Summary

The following report describes the 'happy flowers' project and its relevant components. It contains information about architectural decisions, technological specialities, as well as challenges faced over the course of the project.

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 is 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

1. Summary 1

1.1. Fact Sheet 1

2. Introduction 4

3. Dashboard 7

3.1. Choosing the Right Framework 8

3.2. Front End Architecture 9

3.3. Immutability 10

3.4. Component Architecture 11

3.5. Redux Architecture 14

3.6. Routing 17

3.7. Charts 18

3.8. WebSockets Client 19

3.9. Important Libraries and Tools 20

4. Back End 21

4.1. Back End Architecture 21

4.2. SQLite Database 22

4.3. HTTP Server 24

4.4. WebSockets Server 26

4.5. Hardware Process 26

4.6. Configuration 28

4.7. Important Libraries and Tools 29

5. Hardware 31

5.1. Hardware Setup 32

5.2. Important Tools and Libraries 34

6. Communication 36

6.1. Authentication and Security 36

6.2. REST API 36

6.3. WebSockets 37

6.4. Preventing Race Conditions 38

7. Processes 41

7.1. Preventing Concurrent Actions 41

7.2. Measurement 41

7.3. Watering 42

8. Conclusion 44

8.1. Recommend Applications 45

8.2. Evolution Scenarios 46

9. References 47

9.1. List of Tables 47

9.2. List of Figures 47

9.3. List of Listings 48

10. Appendix A: Common Issues 50

11. Appendix B: Learning Material 51

12. Appendix C: Development Environment 53

13. Honesty Declaration 54

2. Introduction

This report details the process and the findings of the IP5 project 'happy flowers' along with its architectural details and design decisions.

'happy flowers' aims to solve the problem of household plants living in sub-optimal 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.

The result of this project is a Raspberry Pi that is connected to a moisture sensor and a water pump. A Haskell back end is responsible for handling the hardware processes and persisting the data. Users can then access this data through a web front end. The combination of these components (as shown in *figure 1*) creates a reliable foundation that solves the previously described problem.

Figure 1 High-level architecture for the 'happy flowers' project

Aside from the obvious focus on the product, the project has another goal. A large portion of the program is written in Haskell, a functional programming language.

The UAS offers a course on Functional Programming 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.

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.

3. Dashboard

An elegant dashboard, accessible through a web browser, provides users with an up-to-date overview of their flower. Users can quickly check the flower's moisture and temperature levels, as well as statistics about the last measurement and waterings. This data is provided both as part of an at-a-glance snapshot and as a chart including additional historical data.

The dashboard is updated live as soon as new data is collected so users always have all the information they require about their flowers. Additionally, a live stream is available where users can look at their flowers and how they

are doing. Users also have the possibility to manually trigger a watering if they wish to do so.

3.1. 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 mostly functional setup.

- **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 lack a lot of the ecosystem — i.e. third-party libraries, custom components, etc. — that React already offers so there would have been a lot of custom development and additional effort.

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, as shown in *listing 1*. 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.

```
render() {
  return <Loader loading={true} />
}

// is equal to

render() {
  return React.createElement(Loader, { loading: true })
}
```

Listing 1 JSX code and its JavaScript representation

3.2. Front End Architecture

The React project exists in a Node environment. Required technologies and relevant setup information is available in this project's setup guide (https://goo.gl/rIABN6).

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 — shown in *figure 2* — that communicates with the back end through the API and WebSockets.

Figure 2 Components and communication inside the React project

The React setup is based on the scaffolding created by a tool called *create-react-app* (https://goo.gl/QxYuR4, 'CRA' in following). It aimed to simplify the process of setting up and maintaining a React app. Apart from the scaffolding it offers a library called `react-scripts` that makes it easy to keep the environment up-to-date.

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.

3.3. 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 (see *listing 2*). Every mutation leads to the creation of a new structure, a core aspect of immutability.

```
chartOptions = fromJS({ ... })

chartOptions = chartOptions
  .deleteIn([ 'xAxis', 'min' ])
  .deleteIn([ 'xAxis', 'max' ])
  .set('series', [ ... ])
```

Listing 2 Sample code showing manipulation of immutable structures

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.

3.4. 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. The full component architecture is outlined in *figure 3*.

Figure 3 Component hierarchy for the React project

3.4.1. 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, as shown in *listing 3*.

```
const Tooltip = ({ text }) => (
  <div className="tooltip">
    <span data-icon="help">
      <span className="tooltip-body">
        {text}
      </span>
    </span>
  </div>
)
```

Listing 3 Sample component definition

Containers (sometimes also called *smart components*) are more complex elements. They are often composed of other components and have an internal state, thus they are written as classes, as shown in *listing 4*. Their usual task is to define the composition of components and to fetch data. In the context of a Redux architecture this often means that containers manage the connection to the application state using `mapStateToProps`.

```
class Dashboard extends React.Component {
  static propTypes = { ... }

  componentDidMount() {
    const { dispatch } = this.props
    dispatch(fetchSettings())
  }

  render() {
    return (
      <main className="site">
        {/* ... */}
        <div data-col="L1-2">
          <Stats socket={this.props.socket} />
        </div>
        {/* ... */}
      </main>
    )
  }
}
```

Listing 4 Sample container definition

3.4.2. 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. An example usage is shown in *listing 5*.

```
<Widget title="Moisture History"
        tooltip="..."
        isLoading={isFetching}>
 {/* ... */}
</Widget>
```

Listing 5 Sample code showing usage of widgets

`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.

3.5. 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. This project's state is shown in *listing 7*. No other component manages the state, except for UI and explicitly internal state.

Other components can access the state using the `connect`, `mapStateToProps` (shown in *listing 6*) and `mapDispatchToProps` functions, which inject data from the application state into components.

```
{
  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 6 Redux state structure

```
const mapStateToProps = state => ({
  snapshot: state.history.snapshot,
})
```

Listing 7 mapDispatchToProps pulls Redux state into React components

In order to change data from the state, components need to `dispatch` an *action*. Actions are a term coined by the *Flux* (https://goo.gl/LDQLOM) architecture, which Redux is 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.

3.5.1. 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.

```
export const login = data => ({
  actions: [ request, success, error ],
  apiCall: () => axios.post('/api/auth', data)
})
```

Listing 8 Sample code showing usage of API actions

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. An action using the API middleware is shown in *listing 8* and the workflow of the middleware in *figure 4*.

Figure 4 Custom Redux middleware for communication with an API

3.6. 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. React 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, as shown in *listing 9*. 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 — of web applications is still a possibility.

```
<Route path="/"
       component={Connector}>
  <Route path="/dashboard"
         component={Dashboard} />
  <Route path="/settings"
         component={Settings}
         onEnter={ ... } />
  <Route path="/login"
         component={Login}
         onEnter={ ... } />
</Route>
```

Listing 9 Routing code

Sub-routes are used to wrap all routes with the WebSockets connector component. This ensures that the a user's connection is kept alive throughout the entire session. The full set of front end routes is outlined in *table 2*.

Route	Description
/	Redirects to the dashboard. Only acts as a wrapper for other routes and components.
/dashboard	Shows the dashboard with all live-updating data.
/settings	Shows the settings page. Requires users to be authenticated.
/login	Shows the login page. Requires users to not be authenticated.

Table 2 Available front end routes

3.7. Charts

Historical data for moisture and temperature should be displayed in an easy-to-use and visually appealing chart. This chart must handle possibly large amounts of data and be performant enough for live updates.

Highcharts (https://goo.gl/PgX1Ra) is just that. It is a charts library offering a wide amount of display options and better performance than most of its competitors. It also offers a zooming functionality which allows users to drag the cursor over a section of the chart in order to display that section in more detail. Using this feature it does not matter if the provided data spans a very large period of time as users can quickly select the data they want.

This project uses react-highcharts (https://goo.gl/KIJiFL) to include charts on the dashboard. The chart configuration is created using a set of default options that is extended with the data and other options, as shown in *listing 10*.

```
let options = defaultOptions
  .setIn([ 'yAxis', 'min' ], 15)
  .setIn([ 'yAxis', 'max' ], 30)
  .setIn([ 'yAxis', 'labels', 'formatter' ], function () {
    return this.value + ' °C'
  })
```

Listing 10 Highcharts configuration

3.8. 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, as shown in *table 3*. 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.

The `Connector` container is responsible for setting up the WebSockets connection and handling messages. It listens to messages transmitted over WebSockets and dispatches Redux actions to update other components.

Message	Payload	Redux Action
MeasurementReceived	`{ measurement: Object }`	`measurementReceived`
EventReceived	`{ event: Object }`	`eventReceived`
HistoryReceived	`{ res: Object }`	`fetchHistorySuccess`
SettingsChanged	`{ res: Object }`	`fetchSettingsSuccess`
BusyChanged	`{ busy: Bool }`	`busy`

Table 3 WebSockets messages

3.9. 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, as shown in *table 4*.

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 4 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, as shown in *table 5*. By starting the project using *create-react-app* (see *3.2*) it was clear that technologies like *babel*, *eslint* and *webpack* would be involved.

Tool	Description
yarn	A package manager for Node.js and JavaScript code.
create-react-app	Sets up a React project and offers a build toolchain for such a project.
babel	Transpiles ECMAScript 6 code to ES5 code.
eslint	Validates JavaScript code against a set of conventions.
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.
postcss	Transforms CSS code with a set of plugins such as autoprefixer, a tool that adds vendor prefixes to certain CSS properties.

Table 5 Important tools for the React project

4. Back End

```
[API] Starting server on port 5000...
Setting phasers to stun... (port 5000) (ctrl-c to quit)
[WS]  Starting server on port 9160...
[HW]  Starting processes...
```

The back end is responsible for multiple tasks. First, it contains a web server that serves the dashboard (see *4.3*). This web server consists of a static file server (used for the React dashboard) and a REST API (used for communication between the dashboard and the back end). In order to provide live communication between connected clients and the hardware, the back end also runs a WebSockets server.

Perhaps most importantly, the back end controls all processes that deal with the flower. This includes the regular measurements as well as the waterings. All communication with the sensors and connected devices run through this back end.

All of the functionality of the back end is realised using Haskell. A very modular approach is used in order to achieve a well-documented and easy-to-understand program.

4.1. Back End Architecture

The back end project exists in a Haskell environment. Required technologies and relevant setup information is available in this project's setup guide (https://goo.gl/IEQhUi).

The back end is a Haskell application — shown in *figure 5* — 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 5 Components and communication inside the Haskell project

The project is based on a standard Cabal sandbox (https://goo.gl/u8uv7s). This allows the project to be developed without any side-effects from other applications installed on the system.

4.2. SQLite Database

Historical data from the sensor should be persisted and stored in a database so that it can be accessed by multiple clients at any given time. For this project, storing data in a server-centric database, for example, would lead to quite a lot of overhead as these processes would have to be managed in addition to all the other running processes.

SQLite (https://goo.gl/0ziZrB) is a relational database management system that is based on an entirely serverless (i.e. *embedded*) architecture.

The database is used to store information about the flower the device is connected to (i.e. configuration for the processes), along with historical data about moisture level measurements and waterings. Thus the entities are very much isolated, which in turn keeps the database architecture quite simple, as shown in *figure 6*.

Figure 6 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. Since the included statements do not contain overly complex SQL logic, the most logical approach was to include the queries directly as strings.

```
conn <- S.open dbName
rows <- try (S.query_ conn "SELECT * FROM settings") :: IO
(Either S.SQLError [Settings])
S.close conn
```

Listing 11 Database querying code

Error handling is performed using `Either` and `Maybe`. All database calls — an example given in *listing 11* — are parsed to `FromRow a => Either SQLError [a]`, which is then converted to `FromRow a => Maybe a`. 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.

Most database queries are supposed to return only one value. Thus it makes sense to either return `Nothing` if no value is available or the head of the query result, wrapped in a `Just`. This differentiation is shown in *listing 12*.

```
either (const Nothing) (\r -> Just $ head r)
```

Listing 12 Error handling for single-row query results

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.

4.3. 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, request 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.

4.3.1. 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, as shown in *listing 13*. By internal convention, the route handlers are always named with their HTTP verb and the data handled by the request, such as `getSettings`.

```
getHistory = get "/api/history/" $ liftIO DB.queryHistory
>>= jsonOrError status500
```

Listing 13 Sample REST API route

Since the data returned by the database code is always contained in `Maybe` (see *4.2*), 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, as shown in *listing 14*. 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 parsed to `FromJSON a => ActionM a`. From this point on, all values that are part of the record are accessible using their named accessors.

```
data PostAuthBody = PostAuthBody
    { password :: !Text
    } deriving Generic

instance FromJSON PostAuthBody

body <- jsonData :: ActionM PostAuthBody
```

Listing 14 Parsing JSON data from a request body

Some endpoints of the API require authentication. These endpoints expect a `token` field to be part of the request body (see *6.2*).

4.3.2. 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 (see *3.2*).

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 *3.6*). This is done using the Rewrite middleware, which is part of the `wai-extra` package.

4.4. 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 the hardware to all connected clients.

Since WebSockets require a variable to keep track of connected clients and their current state, the code to realise this is more complicated than other parts of the program. The code uses many concurrency features like `MVar` and functions to read and modify those variables, as shown in *listing 15*.

```
b <- modifyMVar busy $ \s -> return (state, state)
```

Listing 15 Modifying an MVar to reflect the new application state

4.5. 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. These processes run independently from all other back end tasks and only communicate with the servers and the database through a callback function. This callback function expects a command and its respective parameters, as shown in *table 6*.

Command Name	Parameters	Description
SaveEvent	`EventKind`	Stores event data in the database and notifies connected clients.
SaveMeasurement	`MeasurementKind, Int`	Stores measurement data in the database and notifies connected clients.
UpdateBusy	`BusyState`	Changes the busy state of the flower and informs connected clients.
PumpRequired	`Int`	Starts the automatic pump procedure. Stops (at latest) when the parameter is 0.
CheckRequired	`-`	Starts the interval measurement.

Table 6 Hardware Commands

This allows for separation of concerns between the WebSockets server, the database and the hardware processes. The sensor code is responsible only for reading the data and triggering the GPIO port. It does not know the database or the WebSockets server itself.

4.5.1. Reading Data

In order to read 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 data leads to either a relative or an absolute value. The interpretation of this data is explained in the following section.

4.5.2. Interpretation of Data

The project reads data on soil moisture and room temperature. Both of these can be retrieved and interpreted — shown in *table 7* — through the I^2C protocol. Both sets of data range from 0 to 65535.

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

Table 7 Data interpretation and conversion

4.5.3. Mock Environment

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 `-Development` for production. A production system is considered to be the RPi, thus access to sensors is 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 CPP structure — as shown in *listing 16* — so that certain parts of the code are only included in the compiled result if certain criteria match.

```
#ifdef Development
import qualified HappyFlowers.Hardware.DeviceMock as HW
#else
import qualified HappyFlowers.Hardware.Device     as HW
#endif
```

Listing 16 CPP statement to include different dependencies

4.6. Configuration

A configuration file — shown in *listing 17* — 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 17 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.

4.7. Important Libraries and Tools

One of the primary goals of this project was not to *reinvent the wheel*. That means using existing libraries — listed in *table 8* — 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 *5.2.1*).
jose	A JSON Web Token library. Used to encode and decode the token string.

Table 8 Important libraries for the Haskell project

Using the tools shown in *table 9* 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.
hlint	Validates code against a set of conventions.

Table 9 Important tools for the Haskell project

5. Hardware

The hardware is the part of the system where data about the flower is actually collected. A Raspberry Pi ('RPi') is used to run the program and communicate to sensors and other devices like a water pump and a USB web camera.

The hardware project is based on a Raspberry Pi setup. Required technologies and relevant setup information is available in this project's setup guide (https://goo.gl/WLsxFL).

5.1. Hardware Setup

5.1.1. Setting Up the Raspberry Pi

The Raspberry Pi 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/WLsxFL).

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, as shown in *figure 7*.

Figure 7 Raspberry Pi 3 pin layout (https://goo.gl/j4Sfp1)

5.1.2. 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), connected directly to the RPi, as shown in *figure 8*. The sensor is capable of measuring soil moisture, temperature, and ambient light and transmits data through the I^2C protocol (see *5.2.1*). It requires a supply voltage between 3.3V and 5V.

Figure 8 Connection schematics for the Chirp! sensor

5.1.3. Connecting the Raspberry Pi to the Watering Pump

The standard USB water pump is connected to the RPi (see *figure 9*). It is activated and deactivated using GPIO (see *5.2.1*).

Figure 9 Connection schematics for the USB water pump

5.1.4. 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 *5.2.2*). The livestream, once started, can be accessed through port `8081`.

5.1.5. 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. The required steps to achieve this connection are outlined in this project's setup guide (https://goo.gl/WLsxFL).

5.2. Important Tools and Libraries

5.2.1. I^2C, GPIO, and bcm2835

I^2C ('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.

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.

The required steps for installing these components are outlined in this project's setup guide (https://goo.gl/WLsxFL).

5.2.2. 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 full setup is outlined in this project's setup guide (https://goo.gl/WLsxFL).

6. Communication

6.1. Authentication and Security

This project uses a token-based form of authentication (shown in *figure 10*) 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 10 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, as shown in *table 10*.

Endpoint	Method	Request Body	Response Body
/api/auth/	POST	`{ password: String }`	`{ token: String }`
	Validates a user-provided password and creates a token that can be used for authenticating follow-up requests.		
/api/history/	GET	–	`{ events: Array<Object>, measurements: Array<Object> }`
	Retrieves historical data about events and measurements.		
/api/settings/	GET	–	`{ name: String, upper: Int, lower: Int, interval: Int }`
	Retrieves application settings, mainly configuration for the hardware processes (e.g. measurement interval or moisture limits).		
/api/settings/	PUT	`{ token: String, name: String, upper: Int, lower: Int, interval: Int }`	`{ name: String, upper: Int, lower: Int, interval: Int }`
	Updates application settings to change the configuration for the hardware processes. Requires authentication using a token.		

Table 10 REST API endpoints with request and response bodies

6.3. WebSockets

A WebSockets server is used for real-time updates of connected clients. This allows clients to keep track of changes to historical data and application settings. This works by passing JSON objects, as shown in *listing 18*.

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

Listing 18 Standard WebSockets events payload

6.4. Preventing Race Conditions

There are situations — such as in *figure 11* — 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 11 Race condition with API + WebSockets fetching data

1. A GET request attempts to retrieve historical data. This is an asynchronous operation so other actions may run at the same time.

2. The WebSockets connection is opened. Again, this is an asynchronous operation.

3. The server receives a new reading from the sensor. This happens independently from all other processes so the server does not wait with processing the input.

4. The WebSockets connection is established. Since the reading in step 3 happened before the connection was established it is lost and the client is not notified about it.

5. Another reading is received.

6. The server notifies the client about the new reading. This is now the first reading that is present in the chart.

7. The GET request from step 1 is finished and data is sent to the client. The newly received data replaces all previous entries, this means the reading from step 5 is lost.

This would lead to the state for events shown in *listing 19*, where the historical data would overwrite the new data and events would go missing.

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

Listing 19 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 approach shown in *figure 12* was selected where initial history data is sent by the WebSockets server instead of relying on the REST API.

Figure 12 Prevention of race conditions

1. The WebSockets connection is opened.
2. The server receives a new reading from the sensor. The new reading is added to the database.
3. The WebSockets connection is established.
4. The server sends historical data to the client as a WebSockets message. This data now includes the reading from step 2.
5. Another reading is received.
6. The server notifies the client about the new reading. This reading is appended to the list of readings that have been sent as part of step 4.

With this fix in place, the events are fetched correctly and stored in the correct order, as shown in *listing 20*. New events cannot go missing.

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

Listing 20 Correct events state

7. Processes

7.1. 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. A working process needs to ensure that such conflicts can be prevented without affecting the communication between sensor and clients and the ability to connect new clients.

This is ensured using a combination of `MVar` and WebSockets. When a process is started, it updates the `MVar` 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 `MVar` 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.

7.2. Measurement

A measurement is the process of measuring the moisture level of the flower the sensor is connected to, described in detail in *figure 13*. The value is measured by the *Chirp!* sensor, as described in *3.2.2*.

Figure 13 Measurement process

The sensor is notified about the necessity for a check, i.e. a measurement. It only continues if the flower is currently idle, otherwise it just schedules another check. The flower is then marked as busy and data is read, persisted, and sent to all connected clients. Depending on the flower's status, it is either marked as idle again or the sensor is notified about the necessity for an automatic pump action. Finally, once all actions are finished, another check is scheduled after the defined interval. The exact events and their meaning are described in *4.5*.

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, described in detail in *figure 14*.

Figure 14 Automatic watering process

The sensor is notified about the necessity of a pump activation. It then triggers the pump for a set duration. After this, the moisture is read. Depending on the reading, the pump is either triggered again or the measurements — moisture and temperature — are saved and clients are notified. Finally, an `Automatic` event is saved and the flower is marked as idle.

7.3.2. Manual Watering

A manual watering is the result of a user interaction on the web front end, described in detail in *figure 15*.

Figure 15 Manual watering process

The server receives a TriggerPump event through a WebSockets message. It then starts the manual pump process, which only runs if the flower is idle. The process first marks the flower as busy, the proceeds to read moisture and temperature. This data is persisted and sent to all connected clients. If the moisture is not already above the defined upper limit, the pump is triggered for a set duration. After this, a `Manual` event is saved and the flower is marked as idle.

8. Conclusion

One goal of this project was to show students 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.

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. Recommend Applications

The product is ready to be used and can be installed on multiple Raspberry Pi devices in order to enjoy the benefits created by the project with multiple flowers.

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 *3.1*. This would necessitate the alternatives to further develop their ecosystem so that the change would not cause unnecessary complexity. The community around these languages and tools has not yet reached the levels of participation and wide range of choices that the React community — and the JavaScript community in general — have offered for a long time. The benefits of using one of the alternatives would be that an even larger share of the project's code covering additional interesting topics 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

Table 1 Fact sheet about the 'happy flowers' project 1

Table 2 Available front end routes 18

Table 3 WebSockets messages 19

Table 4 Important libraries for the React project 20

Table 5 Important tools for the React project 20

Table 6 Hardware Commands 26

Table 7 Data interpretation and conversion 27

Table 8 Important libraries for the Haskell project 29

Table 9 Important tools for the Haskell project 29

Table 10 REST API endpoints with request and response bodies 36

9.2. List of Figures

Figure 1 High-level architecture for the 'happy flowers' project 5

Figure 2 Components and communication inside the React project 10

Figure 3 Component hierarchy for the React project 12

Figure 4 Custom Redux middleware for communication with an API 17

Figure 5 Components and communication inside the Haskell project 22

Figure 6 Entity Relationship Model for the SQLite database 23

Figure 7 Raspberry Pi 3 pin layout (https://goo.gl/j4Sfp1) 31

Figure 8 Connection schematics for the Chirp! sensor 32

Figure 9 Connection schematics for the USB water pump 32

Figure 10 Authentication flow 35

Figure 11 Race condition with API + WebSockets fetching data 37

Figure 12 Prevention of race conditions 38

Figure 13 Measurement process 41

Figure 14 Automatic watering process 42

Figure 15 Manual watering process 42

9.3. List of Listings

Listing 1 JSX code and its JavaScript representation 9

Listing 2 Sample code showing manipulation of immutable structures 11

Listing 3 Sample component definition 13

Listing 4 Sample container definition 13

Listing 5 Sample code showing usage of widgets 14

Listing 6 Redux state structure 15

Listing 7 mapDispatchToProps pulls Redux state into React components 15

Listing 8 Sample code showing usage of API actions 16

Listing 9 Routing code 17

Listing 10 Highcharts configuration 18

Listing 11 Database querying code 23

Listing 12 Error handling for single-row query results 24

Listing 13 Sample REST API route 24

Listing 14 Parsing JSON data from a request body 25

Listing 15 Modifying an MVar to reflect the new application state 26

Listing 16 CPP statement to include different dependencies 28

Listing 17 Default project configuration 29

Listing 18 Standard WebSockets events payload 36

Listing 19 Erroneous events state due to race conditions 38

Listing 20 Correct events state 39

10. Appendix A: Common Issues

During testing with a real flower, some common problems have occurred.

- **Sensor**: While the sensor mostly works very reliably, it sometimes produces excessively high readings, such as a room temperature of 3280°. This is most likely due to an issue with the sensor, however, the exact reason behind this problem could not be discovered. As a temporary fix, these impossible readings — i.e. readings outside of our defined range — are simply ignored and another readings is taken.
- **Moisture**: Most flowers dry out at a rate which is much slower than initially expected. This produces readings that seem to stagnate for long periods of time.
- **Pump**: Simply leading the tube from the pump to the flower creates waterings that are not very useful as they only cover a small part of the soil. It is much better to lead the tube around the flower's pot and poke holes into it so that the water can drip out in multiple places. This creates a more evenly distributed watering, which in turn leads to more precise and more truthful readings.

11. Appendix B: Learning Material

The following guides have been very useful throughout this project.

- **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.

12. Appendix C: 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.

13. 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