

Bachelor's Thesis

Entwicklung einer App für das HappyFace Projekt

Development of an app for the HappyFace project

prepared by

Timon Vogt

from Minden, NRW

at the II. Physikalischen Institut

Thesis number: II.Physik-UniGö-BSc-2018/03
Thesis period: 3rd April 2018 until 10th July 2018
First referee: Prof. Dr. Arnulf Quadt
Second referee: Priv.-Doz. Dr. Jörn Große-Knetter

Zusammenfassung

Das Speichern und Analysieren der aufgenommenen Daten am LHC (Large Hadron Collider) am CERN-Forschungszentrum in der Schweiz ist eine ressourcenintensive Aufgabe, für die das WLCG (Worldwide LHC Computing Grid) aufgebaut wurde. Dessen stabiler und störungsarmer Betrieb ist dabei unerlässlich um eine verlässliche und effiziente Analyse der Daten zu ermöglichen. Zu diesem Zweck wird von den Administratoren des WLCG eine Vielzahl unterschiedlicher Überwachungssoftware eingesetzt um den aktuellen Zustand aller Komponenten zu bestimmen und mögliche Ausfälle so früh wie möglich zu erkennen und zu beheben. Das modulare Meta-Überwachungssystem HappyFace wird dabei eingesetzt, um die unterschiedlichen Zugriffspunkte auf verschiedene Überwachungssysteme zu vereinheitlichen und in einer zentralen Oberfläche darzustellen. Diese Bachelorarbeit beschäftigt sich mit der Entwicklung einer mobilen Version des HappyFace Systems für Smartphones und Tablets um die Arbeit der Administratoren flexibler und ortsunabhängiger zu gestalten.

Stichwörter: Physik, WLCG, ATLAS, Grid Computing, HappyFace, Meta-Monitoring, App, ionic, cordova, Android, iOS

Abstract

The storage and analysis of data recorded at the LHC (Large Hadron Collider) at the CERN research center in Switzerland is a resource intensive task, for which the WLCG (Worldwide LHC Computing Grid) was established. Its stable operation with few interferences is thereby imperative for an efficient and reliable analysis of the data. To this end, its administrators use a variety of monitoring tools to view the current state of all components and to recognize failures in an early state. The modular meta-monitoring tool HappyFace is thereby used to unify the access to different monitoring systems under one common interface. This bachelor thesis deals with the development of a mobile version of HappyFace for smartphones and tablets to make the job of the administrators more flexible and less location dependent.

Keywords: Physics, WLCG, ATLAS, Grid Computing, HappyFace, Meta-Monitoring, App, ionic, cordova, Android, iOS

Contents

1. Introduction	1
2. Background	3
2.1. The LHC	3
2.2. The ATLAS detector	3
2.3. Acquiring data	4
2.4. The WLCG	5
2.5. Monitoring	6
3. The HappyFace Project	7
3.1. Concepts of HappyFace	7
3.2. HappyFace Mobile version 1.0	10
3.2.1. The user interface	10
3.2.2. The applications background	13
4. Smartphone application	15
4.1. Motivation for writing a new app	15
4.1.1. ionic	16
4.1.2. cordova	16
4.2. Background Software	17
4.3. Architecture	18
4.4. Graphical user interface	20
4.4.1. Monitoring Tab	20
4.4.2. Analyzer Tab	21
4.4.3. Controller Tab	23
4.4.4. Visualizer Tab	25
4.4.5. Logs Tab	25
4.4.6. Home Tab	26
4.4.7. Settings page	28

Contents

4.5. Special Functions	30
4.5.1. The instance chooser	30
4.5.2. The ssh terminal	32
4.5.3. The HappyFace classical view	33
4.5.4. The connection error modal	36
4.5.5. Widgets	36
4.5.6. The search function	37
5. Outlook and Conclusion	39
5.1. Outlook	39
5.2. Conclusion	39
Appendix A. List of links	41

1. Introduction

Since mankind has existed, humans have tried to understand the world they live in. Starting with practical questions, for example about the origin and the properties of fire to acquire warmth and protection against predators, the questions evolved to deeper, more complicated ones about the components of atomic nuclei or the origin of the universe itself.

To answer these questions, humans did not limit themselves to their biological senses but invented processes and machines able to perceive much more than any human could. Of all these machines the particle accelerator Large Hadron Collider (LHC) is probably the largest, most expensive, and most complicated one ever build. Its purpose is just as impressive: to find the fundamental particles the universe consists of and the interactions between them, thereby (hopefully) answering two fundamental questions of mankind: "What are we?" and "How did we get here?"

To do that, the LHC collides protons with a very high kinetic energy, and records the particles emerging from these collisions hoping to witness evidences of new fundamental mechanisms responsible for the existence of the universe. To find them, the recordings of the events which each contain a large amount of data then need to be analysed and compared with simulations of the events, a job too enormous to be done by hand or by a single computer. Therefore the WLCG was established, a worldwide network of powerful computer clusters, each containing hundreds or thousands of computing cores, thereby providing in its entirety enough resources to accomplish such a task.

Given the complexity of the WLCG itself and its tasks, keeping such a system up and running is a toilsome task on its own. One of the tools used for this job is called "Happy-Face", a meta-monitoring tool capable of reporting the status of computer systems and finding possible failures.

During this bachelor thesis, the monitoring tool "HappyFace" is brought to mobile devices like smartphones and tablets, which gives the administrators a greater flexibility for their job and improves their working speed.

2. Background

The HappyFace project, and subsequently also a HappyFace smartphone application, is a tool designed to monitor big and inhomogeneous computer clusters. This is of course useful in any situation and for any project which has to deal with a lot of data, and therefore uses big and complex computing systems, even commercial computer clusters. However, currently the LHC is likely the most data intensive physics experiment. Additionally, its data needs to be accessible from all over the world since it is processed in universities and physics institutes in different countries. Therefore such a monitoring tool is essential for the LHC.

2.1. The LHC

The Large Hadron Collider (LHC) is a proton-proton particle collider located on the CERN site near Geneva in Switzerland. Currently it is the most powerful particle collider, able to accelerate protons to an energy of 6.5 TeV. Thereby, it produces a luminosity of $10^{34} \text{ cm}^{-2}\text{s}^{-1}$ [1]. The LHC was constructed in a 27 km long tunnel which was built for its predecessor, LEP (Large Electron Positron Collider). This tunnel lies about 100 m below ground level in a stable rock layer. At this depth, the environment is protected against the synchrotron radiation from the accelerator and the detector is protected against most of the cosmic radiation which would decay in it and would thereby produce fake data.

Inside the LHC, protons are running in two contrary circles and collide in one of the four main experiments ATLAS, CMS, ALICE, and LHCb. These experiments use the free energy to produce elementary particles which are then measured by the detectors of these experiments [2].

2.2. The Atlas detector

The ATLAS detector is located at collision point 1 of the LHC. Currently, there the proton packages of the LHC are collided with an energy of 6.5 TeV each, resulting in a centre of mass energy of 13 TeV.

2. Background

The ATLAS detector is 44 m long, 25 m high and weighs 7000 tons. It consists of three concentric layers containing different detector types. This innermost layer consists of pixel and micro-strip silicon semiconductor detectors as well as transition radiation trackers. These lay within a magnetic solenoid field, which forces a circular motion upon any charged particles due to the Lorentz force. Based upon the direction of the circular trajectory and the radius, one can conclude the sign of the charge as well as the transverse momentum of the particles. The next layer hosts the electromagnetic and the hadronic calorimeters. In these, some particles (electrons, positrons, and photons in the electromagnetic, and hadrons in the hadronic calorimeter) tend to produce particle showers, which then can be used to measure the energy of the particles. Via the energy-momentum relation

$$E = \sqrt{p^2 + m^2} \quad (2.1)$$

one can then deduct the mass of the particle. The outer layer of the detector is a muon spectrometer to detect muons, which are too heavy to be stopped by the calorimeters. A special feature of the ATLAS detector is that the muon spectrometer is surrounded by a toroidal magnetic field, which forces the charged muons on another circular trajectory. This is again used to calculate the transverse momentum of the muons, resulting in two independent measurements, thereby reducing the errors [3].

2.3. Acquiring data

The bunch crossing frequency at the ATLAS detector is 40 MHz, giving collisions every 25 ns. This, given the amount of detector cells, of course results in a huge amount of data, too much to be saved on the current systems. To reduce the data rate, ATLAS uses a combination of so-called triggers, beginning with custom-made integrated circuits sitting directly on top of the detector cells to determine whether the cells have detected something important, and special algorithms which do the first reconstructions of the occurred event to exclude further uninteresting events. In combination, these triggers are able to reduce the data size to an average of 1.3 Mbyte per event at an average event rate of 200 Hz giving an average output of 300 Mbyte/s from the ATLAS experiment, which can then be stored in the CERN data storage service [3].

2.4. The WLCG

Even though a large number of events is sorted out using the triggers, the data from the LHC still reaches up to 50 PB per year, which requires large efforts to process them [2]. To do this, the WLCG (Worldwide LHC Computing Grid) was established. The WLCG is a decentralized system consisting of 170 computing centres all over the world. Most of them belong to universities or institutes for high energy physics and are categorized into four Groups (so called tiers):

- **Tier-0** The tier-0 centre of the grid system is the CERN data centre located on the CERN site itself. It gets the raw data from the detectors directly after the triggers and also provides computing power for the first reconstructions. For redundancy and security, the CERN data centre is directly connected to a backup tier-0 data centre located in Budapest, Hungary via three fibre optic cables with a transfer rate of 100 GB/s each. Together, these two centres host 20% of the computing power of the WLCG [4].
- **Tier-1** Currently there are 13 national tier-1 computing centres in 12 countries worldwide. They save a large amount of raw data and provide most of the computing power used for reconstructions and simulations of the events. To do that, each tier-1 centre is directly connected to the CERN data centre via a 10 GB/s fibre optic cable. These cables are called the LHC optical private network [4].
- **Tier-2** Tier-2 computing centres are regional and are hosted directly by the institutes and universities working on the LHC. Their main job is to run simulations from the respective institute on the data provided by tier-0 and tier-1 computing centres. Tier-2 computing centres do not have a dedicated connection to the WLCG, but send and receive data via the internet. Currently there are 155 tier-2 computing centres worldwide. The one in Goettingen is called the "GoeGrid" computer cluster.
- **Tier-3** A tier-3 computing centre is a local computing cluster. In contrast to all tiers above a tier-3 center does not have to share its computing resources with the WLCG, but is completely used by its owner[4].

2.5. Monitoring

“The act of performing computer surveillance of computing resources is called *monitoring*. The analysis of monitoring data is responsible for the detection, comprehension, and rectification of failures related to computing resources” [5].

In the WLCG, many different software packages are used to monitor the different computing clusters. These monitoring tools are either central monitoring tools, which provide an overview over the whole WLCG, or large sections of it or local monitoring tools which are intended to monitor single clusters or even single computers. Thereby the local monitoring tools often provide the data for the central monitoring tools by implementing software interfaces which enables central monitoring tools to read out their results. Examples of the central monitoring tools are SSB (Site Status Board) [6], DDM (Distributed Data Management Dashboard) [7], and Big Panda (monitoring dashboard of the ATLAS job submission system) [8]. Examples for local monitoring tools include popular monitoring tools for big computer systems like Nagios [9] and Ganglia [10].

3. The HappyFace Project

3.1. Concepts of HappyFace

The HappyFace project was developed to meet the problems coming from establishing a distributed computing system like the WLCG, described in Chapter 2.4. Because of the structure of the WLCG, it is very likely, especially for the tier-1 and tier-2 computing clusters, to consist out of a large amount of individual processing units, these could even be decentralized and stored in various buildings. Furthermore, the individual computers may not be equally configured, but use various versions of software and even different operating systems. Monitoring such an inhomogeneous system can be a toilsome task for the administrators because of the different ways of accessing the monitoring data from different systems and different monitoring software packages. To solve this issue, the meta-monitoring system HappyFace was developed [11].

Meta-monitoring means that the HappyFace system does not observe the system itself but instead collects the data from another monitoring system. This data is then combined with data from other monitoring systems on the same and other machines and is afterwards presented to the administrator in a compressed way, so that an administrator can check quickly and easily whether a certain computer is working normally or has problems which need a reaction [11].

This places various requirements on the design of such a system [11]:

- **Flexibility** HappyFace needs to be compatible with a large number of different architectures and setups. Therefore, it needs to be built to general guidelines and support different environments by implementing interfaces which can be accessed by locally customized modules.
- **Simplicity** HappyFace should provide a single output, on which all information from all systems come together. Critical information (such as the current overall status) should be visualized through symbols positioned prominently, so that a quick glance is enough to know if a reaction is necessary. All further (non-critical) information should be quickly accessible too, optimally in less than four mouse

3. The HappyFace Project

clicks.

- **Speed** The interface of HappyFace should provide a fast access to all information and should therefore not have very long loading times. On the contrary, many monitoring systems which HappyFace is intended to observe, have long readout times. Therefore the information gathering and the visual presentation in HappyFace should be separated, for example by a readout system which writes into a database and a user interface reading from there, even though this means that the displayed data is not always the latest one available.
- **History** To be able to find correlations between problems and to predict trends for the future, a history of the data should be kept in the database. This history should be easily accessible through the interface.
- **Scripting** Since the HappyFace system is mainly intended for administrators of big computer clusters and advanced WLCG users to check their batch jobs, it should provide an API to include customized scripts. With that, administrators as well as users have the opportunity to apply custom tests, write their own programs to monitor their jobs or extend the functionality of HappyFace in another way to meet their requirements.

To realize all requested features, HappyFace uses a modular design. The base framework, called HappyCore, provides the requested database, controls the automatic execution of all modules to update the dataset and initiates the composition of all modules output into a single webpage once a user requests it. On top of HappyCore, a variable number of modules realise the monitoring of each individual system. These modules, once executed, read out the data of their monitoring system and store it into the database managed by HappyCore. Also, the modules create PHP scripts and HTML fragments to read out the data from the database and visualize it. HappyCore then collects these fragments and composes them into a single webpage (as shown in Figure 3.1(a)). Once a user opens this webpage, the data from all monitoring systems is visible. Additionally some global values are presented which HappyFace calculated from the result from all modules. This is, for example, an overall rating calculated as a combination of all ratings of the different modules, possibly modified by some weights to realise importance levels between the modules.

To meet the scripting feature request, an XML file is generated, containing the data from all modules. This XML file can be read out using custom scripts, so that custom programs reacting on the data from HappyFace can be realised. An example of this XML file is shown in Figure 3.1(b) [11].

The screenshot shows the HappyFace Project interface. At the top, it displays 'The Happy Face Project Version 3, rev. 926M' and the date '08. Jul 2018 09:25'. Below this are several monitoring icons: Monitoring (red down arrow), DB Webservice (green up arrow), Site Services (red down arrow), dCache (red down arrow), PanDA Info (green up arrow), and DDM Info (green up arrow). The main content area shows 'Apel Accounting Information from Central Apel Server' with a table of data. Below the table is a section for 'BDII Status from GStat'.

Record Start	Record End	Count Database	Count Published	Synchronization Status
2018-07-01	2018-07-07	68097	68097	OK [last published 0 days ago: 2018-07-07]
2018-06-01	2018-06-30	366502	366502	OK
2018-05-01	2018-05-31	560160	560160	OK
2018-04-01	2018-04-30	380416	380416	OK
2018-03-01	2018-03-31	522319	522319	OK
2018-02-01	2018-02-28	374151	374151	OK
2018-01-01	2018-01-31	396958	396958	OK
2017-12-01	2017-12-31	348254	348254	OK
2017-11-01	2017-11-30	640724	640724	OK
2017-10-01	2017-10-31	410875	410875	OK
2017-09-01	2017-09-30	302671	302671	OK
2017-08-01	2017-08-31	331448	331448	OK
2017-07-01	2017-07-31	398961	398961	OK

(a) HTML output of HappyFace.

```

<category>
  <name>data_management</name>
  <title>DDM Info</title>
  <status>1.0</status>
  <type>rated</type>
  <link>
    http://happyface-goeGRID.gwdg.de/category/data_management?date=2018-07-08&time=09:25
  </link>
  <module>
    <name>ddm</name>
    <title>ATLAS Data Management Information for GoeGrid</title>
    <type>rated</type>
    <status>1.0</status>
    <time>2018-07-08 09:25</time>
    <link>
      http://happyface-goeGRID.gwdg.de/category/data_management?date=2018-07-08&time=09:25#ddm
    </link>
  </module>
  <module>
    <name>ddm_deletion</name>
    <title>Dataset Deletion Efficiency for GoeGrid</title>
    <type>rated</type>
    <status>-1.0</status>
    <time>2018-07-08 09:25</time>
    <link>
      http://happyface-goeGRID.gwdg.de/category/data_management?date=2018-07-08&time=09:25#ddm_deletion
    </link>
  </module>
</category>

```

(b) XML output of HappyFace.

Figure 3.1.: Example output from the HappyFace instance *GoeGrid* located at Goettingen, Germany.

3.2. HappyFace Mobile version 1.0

The HappyFace webpage discussed in Chapter 3.1 gives a main entry point to the status of a monitored computer cluster. However, the webpage has one main disadvantage: it is optimised to be used on a desktop computer or a notebook. The screen resolutions and orientations of these devices make it necessary to use font sizes and layouts which are hardly readable from mobile devices. But smartphones and tablets have become so popular in the past decade that nowadays it is fair to say that nearly everybody owns a smartphone and most people tend to carry it in their pocket at all times. This makes the incapability of using the webpage from a mobile device a huge drawback. Just updating the webpage to support mobile devices and low screen resolutions was considered a wasted potential, because mobile applications have different advantages over optimized websites, for example faster startup times (because the source code does not need to be downloaded from a server, but instead resides on the smartphone), better security (the encryption algorithm does not need to be sent over a potentially insecure connection) and the ability to use the smartphones native features (vibration, background tasks, file storage access). Due to these advantages the decision was made to develop a smartphone application (app) for the HappyFace project [12].

The first version of the HappyFace smartphone application was developed by Fabian Kukuck during his bachelor thesis at the university of Goettingen in the year 2014 [12].

3.2.1. The user interface






The focus of this version was to display the data from the HappyFace web instance on smartphones. To do that, the data from the HappyFace web instance is separated into a logical structure. Every level of this structure is then displayed in another so-called *view*. A *view* is a logical unit of a mobile app, displayed fullscreen to the user. A view displays information and provides buttons to move to the next view. The first view, and therefore the first level, in the HappyFace mobile application is the categories view shown in Figure 3.2(a). It lists every category available on the HappyFace web instance in a box with its name and an icon giving the current overall status of the category. The icons are:  for *ok*,  for *warning*,  for *critical* and  for *error*. Over the boxes, a time stamp is displayed showing when the information was last updated. To manually update the information, a refresh button is placed on the top right of the screen using the icon . Since the data displayed is not live, but the status in the moment of the last update, it outdates after a while and should be refreshed. To make sure this is done, 20 minutes after the last successful refresh, an overlay is placed on top of the view stating "Outdated" as shown in

Figure 3.2(b). This reminds the user not to use old data when checking the status of the HappyFace instance. Once a user wants detailed information about the category, a tap on its box on the screen redirects to the *modules view* [12].

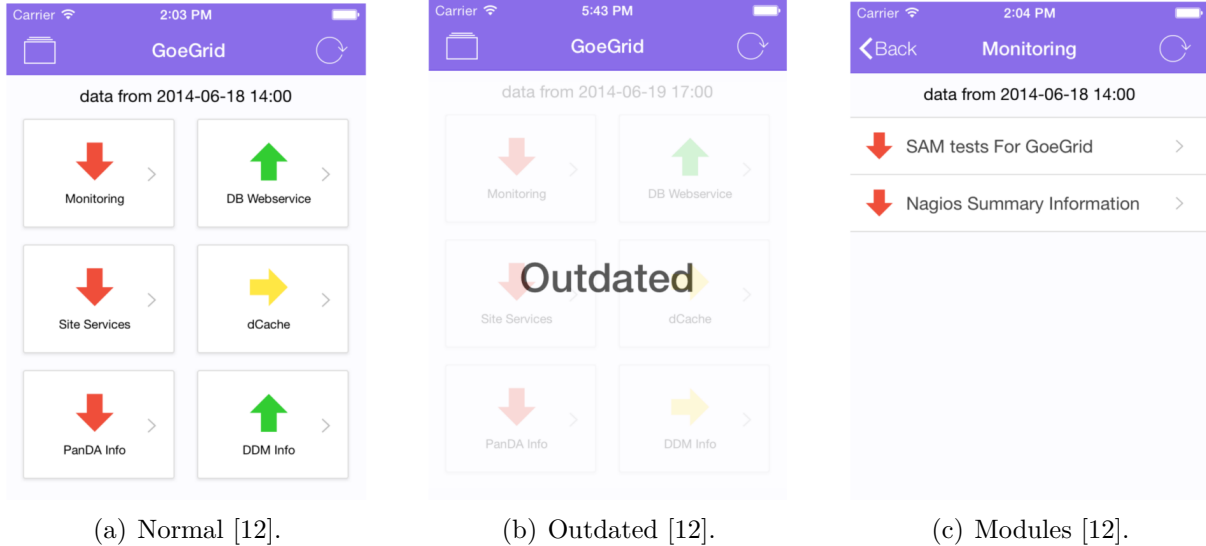




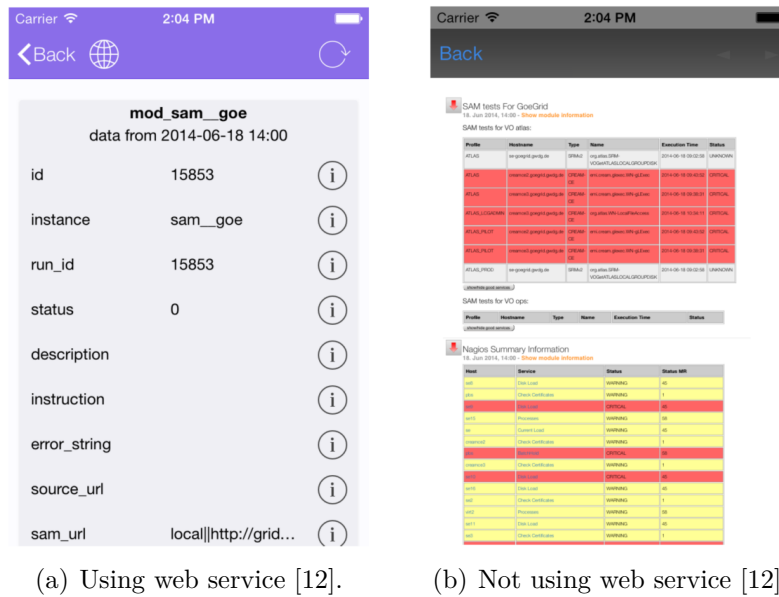
Figure 3.2.: Categories (normal and outdated) and modules view.

The modules view is displayed as a list of the modules contained in a category, as shown in Figure 3.2(c). Every module is again shown with its name and an arrow icon indicating the status of the module. The status icons are the same as in the categories view and also the timestamp and the refresh button are visible on the top of the view. However in the top left corner of the view, there is now a **back** button, directing back to the categories view. By tapping on an entry from the module list, the user is directed to the *detailed module view* [12].

The detailed module view displays either the HTML fragment produced by the module (Figure 3.3(b)) or a parsed version of the data of the module (Figure 3.3(a)), which are both stored in the database managed by HappyCore. Which of the two versions is shown depends on whether the active HappyFace instance uses the *DB web service*, as explained in detail in 3.2.2, since it provides the raw data. If the parsed version is available and shown, the  button displays the HTML fragment of the module. As in the modules view, the top left corner of the detailed module view contains a back button bringing the user back to the modules view [12].

In the categories view, instead of the back button, the  opens the side menu, which swipes in from the left of the screen. This menu contains the settings for the application. Most of this view is a list containing all known instances of HappyFace which can be

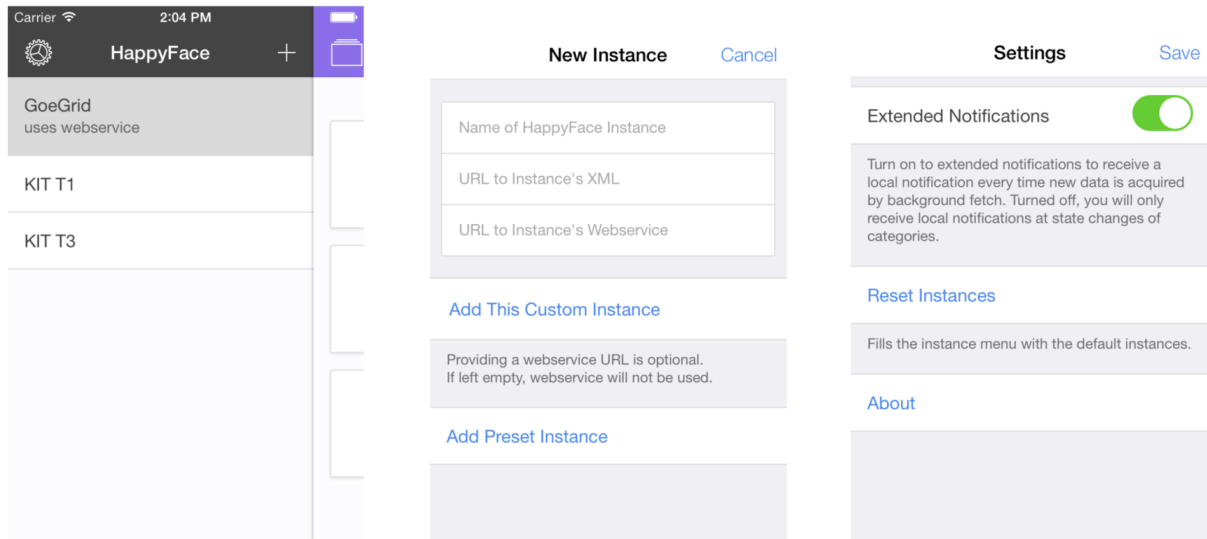
3. The HappyFace Project



(a) Using web service [12].

(b) Not using web service [12].

Figure 3.3.: The detailed module view.



(a) Site menu [12].

(b) Add instance [12].

(c) Settings [12].

Figure 3.4.: The site menu of the app.

monitored (Figure 3.4(a)). A tap on one of them loads their data into the categories and modules views. In the top left corner the button opens the settings view (Figure 3.4(c)), in which the user can enable “extended notifications”, which will notify the user every time the background service updates the data, reset the instances to predefined ones and open the legal notice. In the top right corner is the + button, which opens a view where the user can add a new instance by entering its name, the URL of its XML file and the URL

of its DB web service if that is available (Figure 3.4(b)). When entered, the user has to tap on “Add This Custom Instance” and the instance will be saved in the list of the site menu view [12].

3.2.2. The applications background

The HappyFace mobile application gets its data in two different ways: the data for the categories view and the modules view are parsed from the XML file that the HappyFace web instance generated. Also the module details view shows HTML fragments using links from the XML file. However, using the DB webservice the module details view gets its data from a special HappyFace module called *database web service*. This module is not assigned to any monitoring system, instead it is its purpose to provide the data stored in the internal HappyFace database. To get the data, the DB webservice uses compound URLs: The URL starts with the path of the script (e.g. `http://happyface-goegrid.gwdg.de/webservice/db_backend.py`) followed by `?data=` as an identifier. The rest of the URL specifies the desired data using the SQL language (modified to be usable as an URL, following [13]). The called script (`db_backend.py`) then queries the database and returns the results in the form of a JSON file stored on the server (e.g. `http://happyface-goegrid.gwdg.de/webservice/query_result.json`), which the application downloads and parses into the module details view [12].

The HappyFace smartphone application was intended for the two major smartphone operating systems *Android* and *iOS*. Since these two operating systems require different programming languages and frameworks to be used, a variety of frameworks tried to fill that gap and produce *Android* and *iOS* applications based on one source code. For the HappyFace project the *ionic* framework was chosen, a framework which uses HTML and JavaScript.

4. Smartphone application

4.1. Motivation for writing a new app

Compared to recent developments, the HappyFace mobile version described in Section 3.2 has some major drawbacks.

Since the WLCG grows and evolves, more and more new HappyFace instances have been created. The HappyFace smartphone application uses a list of known instances to choose from, with the ability to add new ones to the bottom of the list. With a growing WLCG and the desire of users to supervise many different instances, the list tends to become confusing and hard to manage. Also the addition of a new instance is done by entering hard coded URLs. That means once a URL changes or is disabled, all users of the smartphone application need to be notified about the change. In the new smartphone application the list of active instances is downloaded as a JSON file from a server, which could easily be modified once any change in the URLs is necessary. Furthermore the instance choosing ability was extended to a “tree-like” structure, giving the ability to group HappyFace instances inside parent instances and give them child instance themselves. This organisation increases the readability significantly.

Another drawback of the “old” HappyFace mobile application is the strict focus on the categories/modules structure of the HappyFace web instance. By simply parsing the XML data from the HappyFace instance and displaying it as a menu, the HappyFace mobile application follows the strict rule that every module has the same importance, even though some display generalised, summarised information about the whole managed computer centre while others display very specific data from one monitoring system supervising one particular aspect of one machine. This tends to be complicated to read and results in many unnecessary clicks. In the new smartphone application, the modules are grouped by their importance and an overall status is shown, which gives a quick view of the whole system. Furthermore the interdependency of the different modules is displayed to help determine the significance of an error occurring in a particular module.

Additionally, as stated in the HappyFace development article [14], HappyFace should contain a history function to compare performances over time to detect failures in an

4. Smartphone application

early stage. This function was missing in the HappyFace mobile application, instead once the data was updated via the refresh button, the old data was gone. In the new HappyFace smartphone application a history function was implemented.

For the development of these functions, the question arose whether to extend the existing application or to write a completely new one. Even though extending the existing app would be faster, it was decided that a complete rewrite is the better option. The reason is that the used frameworks `ionic` and `cordova` were updated in the recent years, so a complete rewrite gives the opportunity to integrate those updates and make use of the new functions they bring. Additionally, in the fast changing world of development frameworks, these new versions of `ionic` and `cordova` are more future-safe; whereas the old versions carry the risk of being discontinued and unsupported in the future.

4.1.1. ionic

The user interface (UI) of the mobile application is developed with the `ionic` framework, which was updated to version 3.20.0. In `ionic`, apps are built using website technologies like HTML, CSS and Typescript (an extended version of JavaScript supporting typed variables and object-oriented programming) instead of native code to develop applications, which are then packed into native applications for both the *Android* and *iOS* operating system. The `ionic` framework is an open source project under the MIT-licence (although a paid version also exists) developed by the company “Drifty Co”. Its focus lies on very “natively” feeling user interfaces, which fit into the "look and feel" of native smartphone applications. The structure and controlling of the application is done by `angular`, an open-source framework for dynamic HTML maintained by Google. This framework manages the control of the application by choosing the view to be displayed (so called routing), reacting to button taps and updating the displayed view according to the user input or remote events. The default choice in `ionic` is `angular` which is shipped with it [15] [16].

4.1.2. cordova

To export the HTML, CSS and Typescript-written source code to a binary format which the smartphones can accept and execute (`.apk` for Android and `.ipa` for iOS) `ionic` uses the framework `cordova`. It bundles the source code, which is de facto a webpage, with a natively written webbrowser to display it in. This bundled code is then compiled using the specific producing software for each Android and iOS. Plugins for the webpage are also provided by `cordova`, which enable smartphone specific features like file-storage-access, vibration, bluetooth, etc. The compiled smartphone applications can then be distributed

using the regular app markets [16] [17].

4.2. Background Software

To gather the data for the smartphone application, two new modules were integrated into the HappyFace web instance: the modules *MadAnalyzer* and *MadBrowser*. The name prefix **MAD** is an abbreviation of the steps these modules are performing: **M**onitoring, **A**nalysing and **D**eveloping.

The module *MadBrowser* is responsible for generating images from the overview pages of the different monitoring systems supervised by HappyFace. To do that, it gets the links to the monitoring systems from *MadAnalyzer* and then uses the open-source Firefox browser to open them and capture screenshots, which are transferred back to the *MadAnalyzer* module.

MadAnalyzer then works as the main entry point for the HappyFace mobile application to the data managed in the HappyCore database. It extracts the data from the database, calls *MadBrowser* to produce images, plots the stability of modules over time, and places it all into JSON files, together with some configuration informations. These output JSON files are:

- **config.json** This file contains the basic configuration information necessary for the smartphone application, such as the name of the current HappyFace instance, port numbers and, most importantly, the link component with the data directory in which the images are stored.
- **monitoring-urls.json** The monitoring URLs file contains one entry for every module inside HappyFace web. For each module it holds the name, a direct link to the web interface of the corresponding monitoring software and the file prefix which is used to generate the link to the different plots. *MadAnalyzer* uses a specific link structure to store every plot in a location accessible via the image directory, the file prefix, the capturing time, and the type of plot.
- **summary.json** A file containing the current status (as a level of criticality and a short status text explaining it) as well as the list of link components for the time. Thereby the summary file gives the information how long the history is saved and which previous files are still available.
- **systems.json** This file lists the different systems in the cluster that are monitored as well as information on how to access them. Using this file the HappyFace mobile

4. Smartphone application

application can connect to the cluster and perform rudimentary repairs. This is explained in more detail in section 4.5.2

- **visualizers.json** Because the new HappyFace mobile application may not only be used by administrators looking for specific information for active supervision but also by the responsible people who want to have an overview over their cluster over a longer period of time, the visualizer file hosts links to overview plots for the whole cluster.
- **logs.json** To more easily find the origin of any error occurring, usually it is helpful to have a look into the log files. The logs JSON file therefore hold their links.
- **meta-meta.json** Following the tree-like structure of HappyFace instances explained in Chapter 4.1, each instance can have many sub-instances. These are stored in the meta-meta JSON file of the instance.

4.3. Architecture

Using the latest version of `ionics` provides the opportunity to use the HTML/Typescript written source code of this application, not only as a smartphone application but also as a so-called progressive web application. This is basically a website in the design of a smartphone application. The advantage compared to the classical HappyFace web instance is a unified design. It also increases the support for the HappyFace mobile application, since progressive web apps are supported by chrome books and are very similar to the Universal Apps for Windows [16].

The application is structured using the model-view-controller architecture. The key idea of this software architecture is to separate the data and its loading and process algorithms from its visualization in the user interface. The **model** part thereby loads and stores the data and provides functions for its manipulation. The **view** part is responsible for visualising the data and presenting it to the user. Any input the view receives (e.g. the tap of a button) is not processed directly by the view, but is sent to the **controller** of the view. This part acts as an interface between model and view, it receives interactions with the view, activates the responsible functions in the model to manipulate the data and then updates the view according to the result. The controller also listens for external data changes in the model and updates the model and the view accordingly. The advantage of such an architecture is that all data is stored in a central place and only needs to be loaded once [18].

In the HappyFace smartphone application, the different parts of the architecture reside

in different files in the source code directory.

The `src/data/DataModel.ts` file marks the model part of the architecture. It contains the cumulative data for all views.

The `src/pages/` directory contains the different views for the application. Each subdirectory (except `modals/`) each contain a `.html` file with the same name as the directory, which is displayed by the system. They represent the view part of the architecture. Apart from the `.html` files the subdirectories each contain a `.ts` file with the name of the directory, which act as a controller for their respective view. The controllers load the data from the `DataModel.ts` file and react to user input in the views.

To give the controller of each view access to the data, the `DataModel.ts` file is implemented as an angular service using the `@Injectable()` decorator. Using `angulars dependency injection` (DI), the `DataModel` service will be included into any view controller, making the data accessible in all views and giving all controllers the ability to initiate a reload of the data once a user presses a reload button. If any error occurs while downloading the files, `DataModel` can also initiate the display of a connection error modal (described in more detail in Subsection 4.5.4). After downloading all necessary files, `DataModel` is also responsible for generating the plot links from the file prefix, the chosen time and the plot type. Apart from downloading the necessary data on startup and on reloading, the `DataModel` service manages the storing and loading of the configurations from the settings view. To do that, `DataModel` uses the new `ionic` storage system, which can save key-value pairs in a local database, available in mobile applications as well as in progressive web applications. Furthermore, `DataModel` is also responsible for managing the update loop, which uses the built-in `setInterval()` function from TypeScript to update the data periodically and start regular status message readouts, in a time interval chosen by the user in the settings view. Finally, `DataModel` holds the `isCordova()` function, which is able to determine whether the code is executed as a smartphone application or a progressive web application. Depending on this determination, various design and functional changes are applied, because progressive web applications are limited in multiple ways compared to fully functional smartphone applications (for example by the available computing power).

In the unlikely case that the `HappyFace` modules `MadAnalyzer` and `MadBrowser` have failed and the JSON files the `HappyFace` mobile application depends on were not produced, the “old” `HappyFace` acquiring mechanism (which depends on the XML file produced by the `HappyFace` web instance) works as a backup. Due to the very module specific nature of the XML file, it is only displayed using a rebuild of the original `HappyFace` interface from the “old” smartphone application (more in Subsection 4.5.3). For

4. Smartphone application

this, a new model is implemented, the `ClassicalDataModel.ts`. It is responsible for downloading the XML file and parsing it for the use in the HappyFace classical view (described in Subsection 4.5.3). However, the `ClassicalDataModel` does not provide the additional functions of `DataModel`, such as the automatic update or the voice readout.

4.4. Graphical user interface

The Graphical user interface consists of six full-sized views sorted in tabs. Below the views, at the bottom of the display, lays the tab bar, which is blue for the tab currently selected. At the top of the display, above the view, is the so-called “action bar”, a small white stripe which displays the name of the current view as well as buttons corresponding to functions for the view.

4.4.1. Monitoring Tab

The first tab, which is selected by default, is called monitoring. The purpose of this tab is to both give an overview over the current overall status of the monitored instance, and show the detailed reports from all modules. Therefore the view starts, on the top, with the so-called status card. This card is a segment which indicates the current overall status. The status level is written on top of the card with a background color fitted to the different levels. Currently the possible levels are Normal (green), Warning (yellow), Critical (orange) and Error (red). Below the top line, the card contains an icon which also displays the current status level (a smiley which gets angrier on higher levels) and the status text. This text describes the current status and, if the status is not normal, says which modules have problems. If the volume of the smartphone is high enough and voice readout is activated in the settings, a tap on the status card will activate the text-to-speech function of the application, which will then read out the status text. This function especially comes in handy when the smartphone application is used to monitor the cluster on-the-side while the user is performing another task simultaneously. By using the voice

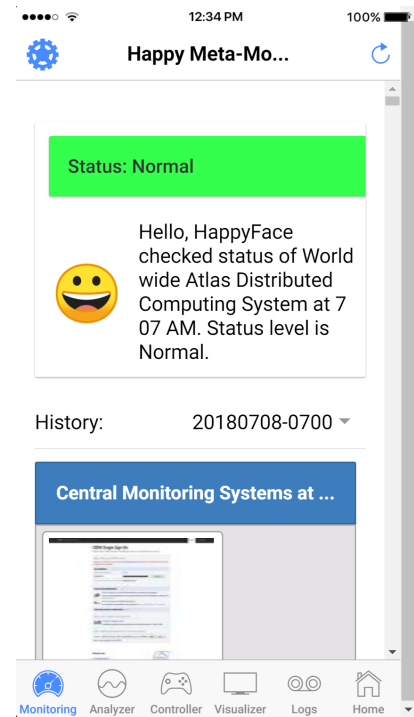


Figure 4.1.: The monitoring tab

readout system, one does not have to look to the screen, but can focus on the other things. This help can be extended by using the automatic readout, described in Subsection 4.4.7. Beneath the status card, the history function is implemented as a dropdown list, holding every time in the past from which the data is still available. Once a new time is selected, the status card and the modules are updated back to that time.

Underneath the history menu, the different modules are displayed. They are divided up into several groups according to their individual level of severity, ordered downwards from the most to the least severe groups. Each group starts with a blue bar above presenting the name of the group. Each module in each group is placed in a small card stating the name and a thumbnail image. The thumbnail is a minimized version of the screenshot MadBrowser took from the overview page of the corresponding monitoring system. This image can be enlarged for better visibility by clicking on the thumbnail, which opens a new view containing a bigger version of the image. Below the thumbnail, the name of the corresponding monitoring system is placed as a link, which redirects to the original page of the system. Therefore, the user can easily choose whether just to have a quick glance at the extended image, or open the page in an external browser to directly interact with the monitoring system.

The action bar of the monitoring tab contains on the left side a gear icon, which opens the settings view, and on the right side a circular arrow which reloads all the data inside DataModel and afterwards rebuilds the user interface of the tab.

4.4.2. Analyzer Tab

The second tab from the left of the smartphone application is the analyzer tab. Following the working path of the modules (as in Section 4.2) this tab is intended to help the user to analyse and delimit a problem once it has been found in the monitoring tab. To delimit and hopefully identify the problem, the analyzer tab contains many different tools.

At the top, the view contains the same status card as the monitoring tab which displays the status level and status text as well as the smiley icon, which changes according to the status. A tap on the status card will also start the readout of the status text.

Instead of the history dropdown menu below the status card, the dropdown menu in the analyzer tab contains the various functions to identify the problem:

- **Status analysis** The first function in the dropdown menu is called “Status Analysis”, visible in Figure 4.2(a). In the status analysis function, for each module from the monitoring tab, a plot is displayed which shows the stability of the module over time. This plot can be enlarged by a tap, while a tap on the name opens the cor-

4. Smartphone application

responding page in an external browser. The modules are separated into the same groups as in the monitoring tab to be easy to find.

- **Info pathway** The second function in the dropdown menu is called “Info Pathway”, shown in Figure 4.2(b). Again, individually for each module, it shows on which other modules this module depends. Since many errors in certain modules tend to stop the output of the module and many modules are interdependent on the output of others, a single error in the right (or wrong) module can lead to a cascade of errors in other modules. This function helps to backtrace the origin of errors to the module it originated in. Also the dependency image can be enlarged by a tap on the thumbnail.
- **Overall info pathway** To further identify the origin of the error, especially in modules which are dependent on a lot of other modules, the dependency plots may become confusing. They also become laborious once the user wants to backtrace an error over a lot of stations. To solve these issues, the overall info pathway contains the complete diagram of the dependencies, so that the path of failed dependencies can easily be found. An example of the interface is visible in Figure 4.2(c)
- **HappyFace classical rating** Since this new HappyFace mobile application breaks with the standard HappyFace design to implement new functions, many administrators may at first find it unfamiliar and confusing. To help those with their transfer process, a new implementation of the user interface of the old HappyFace mobile application was inserted in the analyzer tab under the name “HappyFace classical rating” as shown in Figure 4.2(d). This interface also serves as a backup system in case the HappyFace modules MadAnalyzer and MadBrowser have failed and no JSON files were produced, since the old interface does not depend on them. Further details on the functions and implementation of HappyFace classical rating can be found in Section 4.5.3.
- **Happy Forecast** The last function in the analyzer tab is called Happy Forecast. This system shows plots from the HappyFace MadAnalyzer module which displays the response time of various systems in the instance, as shown in Figure 4.2(e). As it was shown in [19], an increasing response time indicates the beginning of a failure, therefore these plots can help to predict errors in the future. Additionally, using the prediction system described in [19], the plots even show a predicted response time for the future (the blue part of the graph), to indicate possible failures. However these are assumptions, so there is no guarantee for them to predict the future correctly.

The action bar of the analyzer tab contains the name “Happy Monitoring Analyzer” and a circular arrow button which starts the reload and update process. Since the (re-)loading of data is centrally in `DataModel.ts`, a tap on the button will reload all data and rebuild the complete user interface, not only for this tab.

4.4.3. Controller Tab

Once a problem is identified, the user needs to react and execute repairs. To give him/her the opportunity to do that without having to start an external program on a computer, the controller tab provides an entry point for repairs on the system, as one can see in Figure 4.3(a). The view in the tab is a list containing an entry for every accessible system. The systems are stated in the list via their names, a short description, and a thumbnail icon.

Apart from the list, the controller tab holds a circular arrow button in the actionbar, which initiates a reload in `DataModel.ts`, therefore reloading the list of available systems, and an `ssh` button which will open the ssh modal, an overlay page to connect to any given instance via ssh and perform more complex repairs. A detailed description of the ssh modal and its abilities can be found in Subsection 4.5.2.

Once a list entry is tapped on, it opens the controller details page, illustrated for the `AGIS` system visible in Figure 4.3(b), on which certain repairs can be executed. The controller details page therefore lists the name of the system as well showing the associated icon and a list of buttons to perform simple tasks. Even though these tasks are defined by the `systems.json` file and can be easily changed, common tasks are (as displayed in Figure 4.3(b)):

- **E-mail** This function opens the smartphone default email program to write a new email to the administrator responsible for this system. Everyone is highly encouraged to use this function first, before trying to repair anything.
- **Ticket** As an alternative to sending an email, one can also open a ticket in the ticket system by using this button. Depending on the administrator and the system, this might be answered faster or slower than the email. Again, before trying any repairs, the user is **strongly** encouraged to notify the administrators about the problem via ticket or email, and wait for their response.
- **Start** This button will send a command to the instance via ssh to start the corresponding system if it is not working. For security reasons the user will have to authenticate with a username and a password.

4. Smartphone application

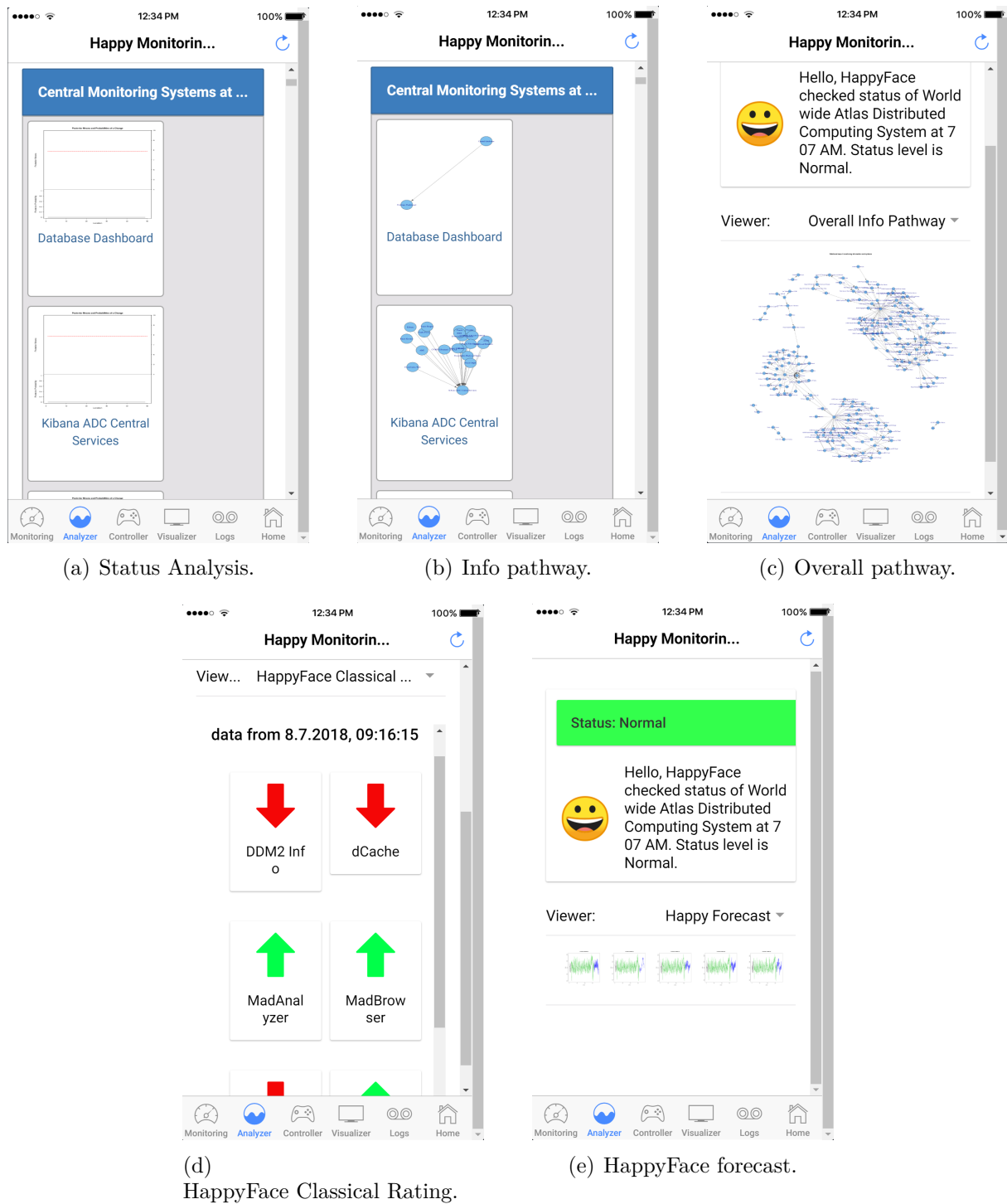


Figure 4.2.: The analyzer tab.

- **Restart** If the system has failed, the user can send a shutdown and reboot command via ssh by using this button. This function also requires an authentication with username and password.

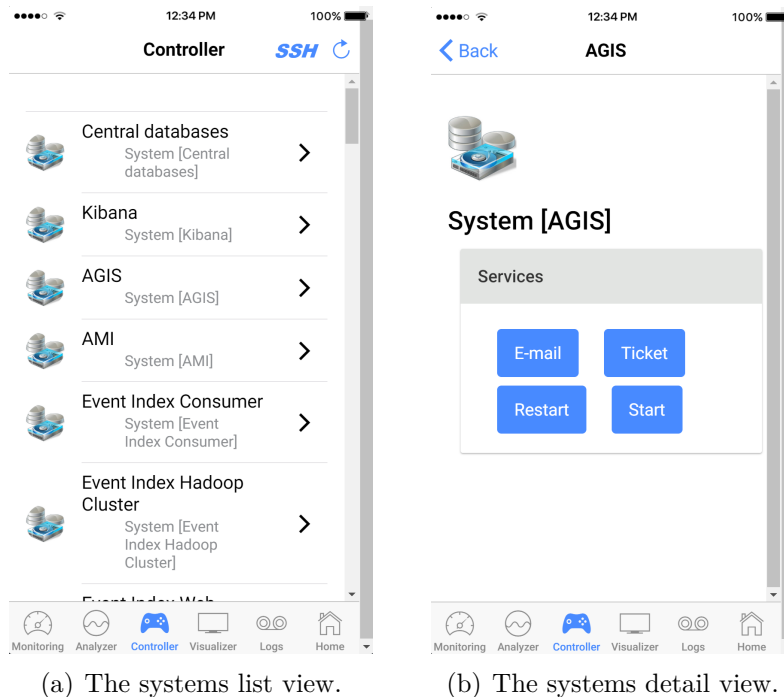


Figure 4.3.: The controller tab.

Further details on the ssh connection system and its authentication are available in section 4.5.2.

4.4.4. Visualizer Tab

This tab is not primarily intended to be used for monitoring/repairing purposes of the individual modules and systems, but for displaying overall information about the monitored instance. This could be diagrams of the ATLAS luminosity, plots of the successful and failed Grid jobs and so on. To do this, the visualizer tab displays the images from the links stored in `visualizers.json` selectable via a dropdown menu (shown in Figure 4.4(a)). The action bar for this tab hosts only a circular arrow to initiate a reload in `DataModel.ts` and subsequently reprint the images.

4.4.5. Logs Tab

Sometimes the displayed information in the overview pages of the different monitoring systems might be misleading or wrong because the error had an impact there too. In that case, it is a good idea to have a direct look into the log files produced by the various software packages in the instance. These logs are displayed in the logs tab as shown in Figure 4.4(b).

4. Smartphone application

The desired log is selectable via a dropdown menu on top of the view. In the action bar the logs tab contains a circular arrow to reload the list of the logs. It is important to note that, due to the fact that some of them are relatively large, the logs are loaded independently after they are selected, to not jam up the reload process of other tabs.

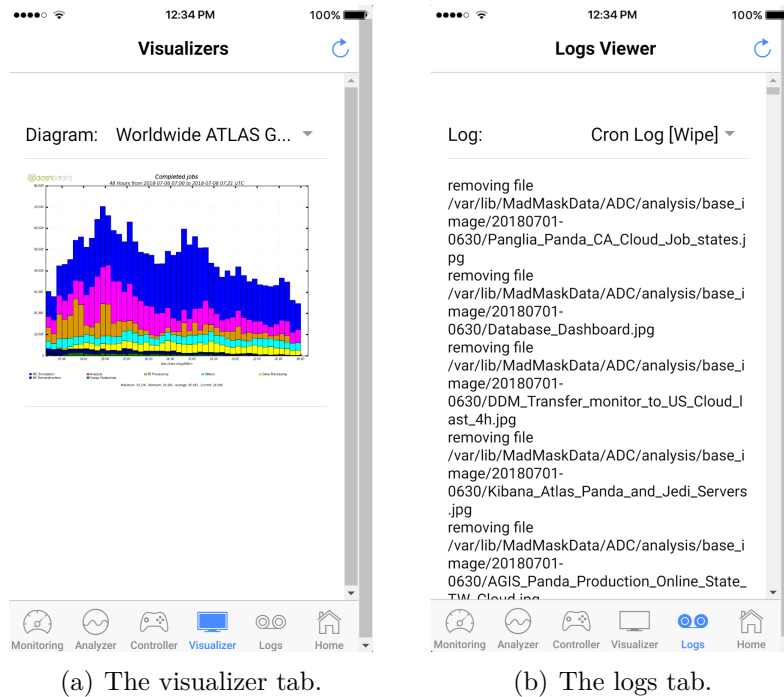


Figure 4.4.: The visualizer and logs tabs.

4.4.6. Home Tab

The rightmost tab in the tab-bar is called the home tab. The task of this tab is to provide the widget system to the user.

The widget system is intended to display the most important information from the data in a compressed and user-defined way, so that the user gets an overview of the instance that fits precisely to his/her workflow. To do that, the idea of the home tab is to display small independent views, so-called widgets, which are interchangeable, to fit the workflow of the user. Two example widgets are visible in Figure 4.5(a). Each widget displays certain, well-defined, information to the user and is totally independent of the others to provide a maximum of stability. The widgets are programmed as independent `angular` components in their own `.ts` files with their own user interfaces in their own `.html` files. These files are then loaded using a dynamic loading system during the runtime of the application and are displayed on the home tab. To modify the widgets, the user has

to start the widget modification mode via the tools button in the action bar of the home tab (shown in Figure 4.5(b)). In this mode, every widget displays a light blue header bar which states the name of the widget and a cross button which closes it. To add a new widget, the user can press the floating add button (plus sign) in the bottom left corner, which will then search for any available widgets in one of the applications widget folders (`src/assets/widgets` and `src/pages/home/loader/static-widgets`) and display a list of available widgets. These widgets can be selected using check boxes and the OK button below the list will end the adding mode and display them. Which of the folders is checked for widgets depends on if the current browser version for progressive web applications (or the `cordova webkit` version in the case of the smartphone application) supports the JavaScript dynamic `import` statement. In this case the folder `src/assets/widgets` is used, otherwise the `src/pages/home/loader/static-widgets` folder is used. Note that in case the dynamic import is not supported, the widgets need to be added in `src/pages/home/loader/StaticLoader.ts` and the application needs to be recompiled for the widget to appear in the new widget list. Further information on how to write/compile/deploy a widget can be found in Appendix A. All widgets currently developed, and their behaviour, are described in Subsection 4.5.5.

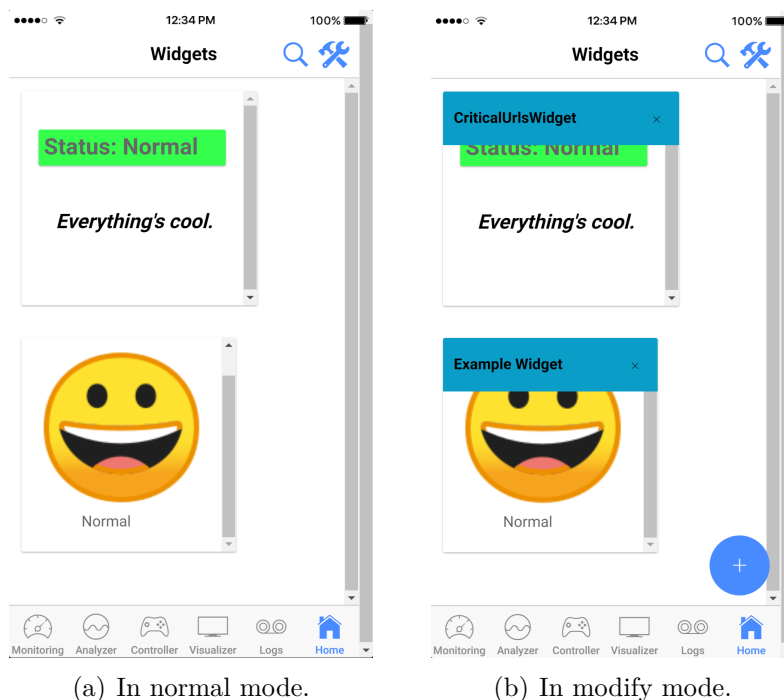


Figure 4.5.: The home tab.

4. Smartphone application

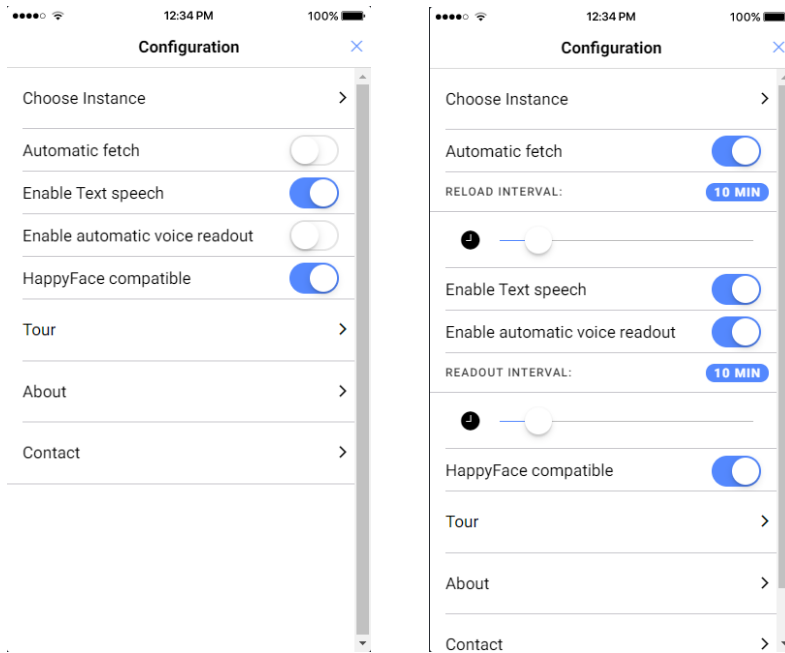
4.4.7. Settings page

By using the gear button on the monitoring tab the settings page can be opened. The settings page is displayed as a *modal*, meaning that if the screen is big enough it is displayed as a small quadratic overlay window, which can be closed with a Windows-like cross button in the top right corner. If the screen is not big enough, the modal will take up the whole screen, but will still be closable via the cross button.

The contents of the settings page are shown in Figure 4.6(a). It is important to note that this is not always the active layout. If the source code is executed as a progressive web application inside a browser, the first page of the settings looks as shown in Figure 4.6(c) and the “normal” settings are shown under “Advanced settings”. The reason for this layout difference is that, since browsers support tabs that open different webpages at once, users of the progressive web application may want to open HappyFace monitoring applications for different instances in different tabs. To support this behaviour, the settings page in a progressive web application contains, on its first page, the list of available subinstances as links which open directly in another browser tab and the application settings are moved to “Advanced settings”. Additionally the web settings page also contains a direct link to the HappyFace web instance.

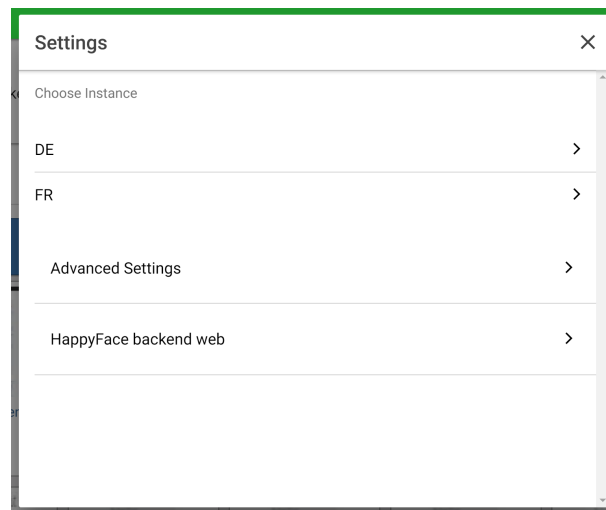
As shown in Figure 4.6(a), the settings page contains a list with various entries:

- **Choose instance** The first entry is a link which opens the instance chooser. This view is described in great detail in Subsection 4.5.1.
- **Automatic fetch** The automatic fetch switch controls the automatic reload process in `DataModel.ts`. Once activated, the application will reload all data after a discrete time interval. The time interval is defined by the user in a slider menu just below the automatic fetch switch (shown in Figure 4.6(b)) which becomes visible after the switch is set to active. This behaviour is useful when the user wants the app to monitor the instance while he/she performs another task.
- **Enable Text speech** The enable text speech switch activates or deactivates the voice readout initiated by a tap on the status card on the monitoring tab or the analyzer tab, or by the timed automatic readout. Selecting to deactivate the voice readout might be preferable for situations, such as during meetings or presentations, where a unintended tap on the status card should not lead to a disruptive voice readout.
- **Enable automatic voice readout** If the voice readout in the previous setting is enabled, it can be initiated automatically by this setting. Once activated a slider



(a) The settings page on a mobile phone.

(b) The settings page on a mobile phone with opened menus.



(c) The settings page in a browser.

Figure 4.6.: The settings page

menu below the switch is visible to set the time interval in which the automatic reload is executed.

- **HappyFace compatible** The HappyFace compatible switch selects the `http` server, which is used to download the necessary JSON files. If HappyFace compatible is disabled, the `ionic` server, which also provides the progressive web applications, is used for distributing the JSON file while, if HappyFace compatible is

4. Smartphone application

enabled, an `apache http` server residing on the same machine is used. The reason for this redundancy is to have a backup system in case one server breaks and to have, during development, a second system to check for development failures.

- **Tour** The application tour is a fullscreen overlay which is displayed once the user starts the application for the first time. Presented as a slide show, the tour explains the different features of the app and the configurations the user can take and where to find them. This basic explanation should help the user to orient himself/herself and to get to know the application. After the user finishes the tour, it will be deactivated so that the user will not be disturbed with it every time he/she opens the app. If something is still unclear or the tour is skipped, the tour link in the settings page will reopen it.
- **About** For legal reasons, every smartphone application and every webpage needs to have a declaration on who is responsible for its content. In this application, this is done in the legal notice reachable with the about link in the settings page. This legal notice contains all information about the developers, maintainers and legally responsible people, as well as the data protection statement.
- **Contact** Even though the application has an impressum containing contact information about responsible people for legal purposes, anyone who has technical problems or questions about the application should use the contact link in the settings page. It will open the mail client with the email address of the group responsible for maintaining the application, since they might not be the ones legally responsible.

4.5. Special Functions

Apart from the regular views described in Section 4.4, the application contains some views and functionality which only appear under certain conditions. These will be described in this section.

4.5.1. The instance chooser

A key feature of the smartphone application, in comparison to the webpage of HappyFace, is the ability to easily switch between instances to monitor from inside the application without having to remember the correct URLs. To realise this is the responsibility of the instance chooser, which is available under the choose instance link in the settings page.

The instance chooser, as seen in Figure 4.7(a), consists of two large sections, each containing a list of instances to be chosen. The bottom section in Figure 4.7(a), with the headline “Cloud”, contains all the instances available in the current level. A tap on one of them will set the selected instance as the current one for the whole application and initiate a complete reload in `DataModel.ts`, now from the new instance. The contents of this list are thereby not hard-coded inside the application, but rather downloaded when the instance chooser starts from a remote server (the concrete URL is available in Appendix A). If the user does not tap on the name of the list entry but on the small arrow on the right side, the subinstances inside this instance are downloaded from the server and this list will display them. This accommodates to the “tree-like” structure the instances are supposed to have, as shown in Section 4.1. These subinstances could have subinstances themselves, marked again by a small arrow on the right side of the list entry which can be tapped to get another level down. To get back a level, and display the instances one level above, an arrow to the other side is displayed next to the headline of the list if the user is not on the top level (shown in Figure 4.7(b)).

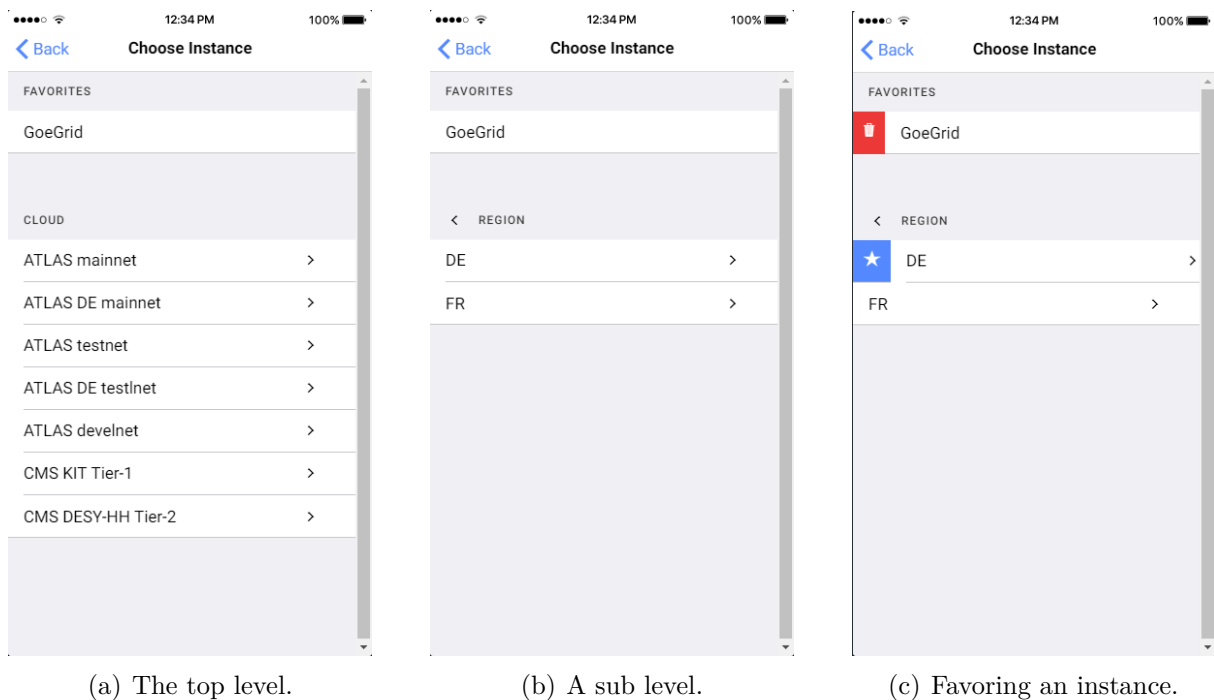


Figure 4.7.: The instance chooser.

The top section, with the headline “Favorites” as shown in Figure 4.7(a), is intended to help users, who mostly want to switch fast between certain instances. Since these instances could be hidden deeply in the tree structure, changing between these instances could become a tedious task. Therefore, every instance in the bottom list can be favoured

4. Smartphone application

by swiping it to the right. This displays a button with a white star on blue background (visible in Figure 4.7(c)) and a tap on this button will add the instance to the favourites, displayed in the favourites list in the top section. The favourites are saved between startups, so a restart of the app will keep the favourites. To remove one of the favourites, the red trashcan button can be revealed by swiping it to the right. This button will remove the instance from the favourites list. Favourites can be selected just as instances in the bottom list, but will not display their subinstances with an arrow button, since this behaviour would spoil the intended usage of the favourites list.

4.5.2. The ssh terminal

As already stated in Section 4.4.3, it is important that the user has the ability to react once an error occurs. To do that, the user most likely needs to have access to the terminal of the monitored instance to apply possible repairs (change configurations, shift data or reboot in extreme cases). For this purpose, the controller tab provides a set of buttons for each system, which can execute functions in the system.

However, occasionally the repair is more complex than just the execution of a predefined function, or needs a command which is not in the list of functions available in the controller page of the system. For these cases, the `ssh` terminal was implemented. It is based on the `xterm.js` terminal emulation system [20] and displays a Unix-like terminal (visible in Figure 4.8(a)) to connect via ssh to the server of the instance to perform repairs. To help administrators who are already experienced in Linux the terminal also supports colour-coding, but unfortunately it does not yet support bash-history, reverse-i-search or pseudo-terminals (used for example by multiplexers like *tmux*).

The `ssh` terminal is initialised using the **SSH** button in the action bar of the controller tab, which loads and activates the terminal emulation. To start the connection, the user needs to write the command `ssh` and press enter. Note that this is not the original `ssh` command from OpenSSL [21] but simply the name of the service the user intends to start, therefore it does not take any arguments. Instead the hostname, port, username and password should be entered using the authentication modal (Figure 4.8(b)) which opens automatically when `ssh` is entered. This modal will ask for different connection settings:

- **host and port** The host and port of the server of the monitored instance. The host can be a URL or directly an IP-address while the port has to be a number between 1 and 65535. Both of these pieces of information need to be inserted manually, since they might be too sensitive to be stored in the publicly readable JSON files.
- **username** The username to connect via ssh to the server. It, as well as the host and

port, will be stored in the applications memory when the switch "save configuration" is activated.

- **password** The password for the given username to log in into the server. Since this information is highly sensitive it is neither stored in the applications memory nor is it visible while typing. Even though the underlying ssh-connection system would support it, limitations on the `cordova` file plugin make it impossible to use keyfiles instead of passwords to connect (at least for now).
- **gateway** A tap on the gateway button opens two new fields asking for the gateway host and the gateway port. These two are set by default and only need changes once the gateway server is moved to a different location.

This gateway server is a transmitter server, which is necessary because an application written in HTML and TypeScript/JavaScript does not have direct access to the TCP Sockets which would be necessary to establish a `ssh` connection to a remote server. Instead, the application uses a gateway server, to which it connects by using a WebSocket (a TCP-Socket-like two-way connection using the http protocol) and which then itself establishes a ssh connection to the desired target using a TCP Socket. To protect the sensitive data transmitted over this WebSocket connection, the application uses the ECDH key exchange protocol to produce a unique session key and uses it to encrypt the sent data using a 256 bit AES algorithm in its cipher block chaining mode. The gateway server decrypts the data and sends it to the target using the `ssh` connection provided by the Node.js module `ssh2` [22] [23] [24].

Once the `ssh` connection is established, the user input from the terminal emulation is transmitted via the gateway server to the target and the responses are displayed in the terminal. The command `exit` ends the connection and the cross button on the top right closes the terminal emulation.

4.5.3. The HappyFace classical view

As already stated in Subsection 4.4.2, the new HappyFace smartphone application breaks with the usual design of the HappyFace web instance and the old HappyFace smartphone application, which can be an obstacle for any user who is familiar with the old HappyFace mobile application. To help such users to use the new smartphone application and provide an easy entry for them, the new HappyFace smartphone application contains a rewrite of the classical user interface in the analyzer tab under the function name "HappyFace classical rating".

4. Smartphone application

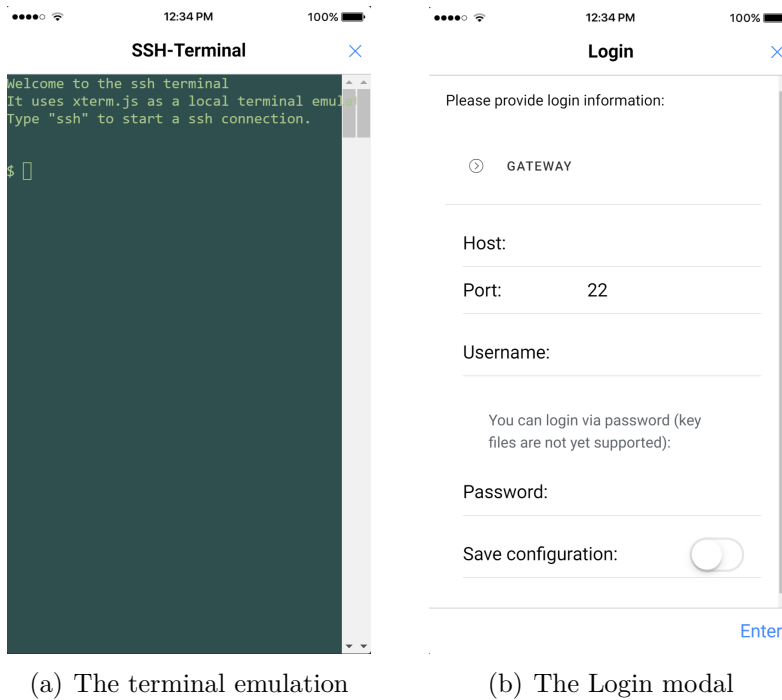


Figure 4.8.: The ssh terminal.

The “HappyFace classical rating” is furthermore based on its own `DataModel`, the `ClassicalDataModel`, available under `src/pages/analyzer/hf-classical/ClassicalDataModel.ts`. This `DataModel` downloads the XML file from the current instance saved in `DataModel.ts`, which can be selected via the instance chooser in the settings page, but is in any other meaning independent of the usual `DataModel` of the application. Therefore the “HappyFace classical rating” also serves as a backup system if the `MadAnalyzer` and `MadBrowser` modules in `HappyFace` fail and no JSON files are produced. Like the old `HappyFace` smartphone application, it contains an overview on its first page of all categories of the `HappyFace` instance (visible in Figure 4.9(a)). In the categories page, every category the `HappyFace` instance contains is displayed in a card with its name and an icon, which indicates the overall status of the category. The status is saved as a number in the XML file and is translated into a status using Table 4.1.

status number	status	icon
1.0	normal	🟢
0.5	warning	🟡
0.0	critical	🔴
else	error	❌

Table 4.1.: The status codes and their meanings.

Usually the error status is coded in the XML file with the status number -1.0 , but the application displays the error icon for every status code which is not in the table, therefore a wrong or missing status code for a category is also noted with the error icon. A tap on one of the categories opens its module page. This page, visible in Figure 4.9(b),

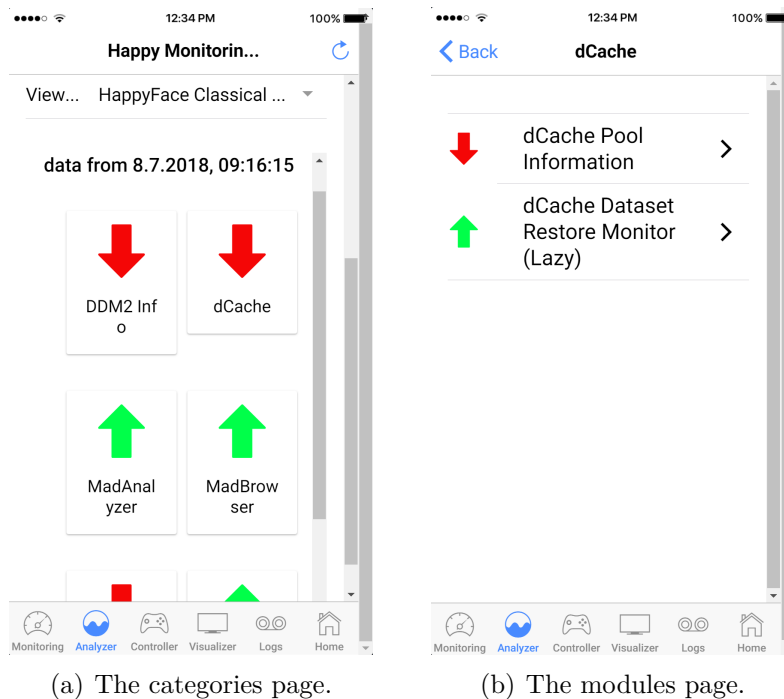


Figure 4.9.: The classical view.

contains all modules of the category in a list, each with its name and an icon showing its status. The status is again saved as a number in the XML file and again Table 4.1 is used to set the icon. Once the time elapsed since the download of the XML file reaches 20 minutes, the categories and the modules page will display an overlay stating that the data is outdated. This overlay prevents further interaction and requires the user to reload the data using the reload button in the action bar of the analyzer tab.

If the user taps a module, the webpage of this module will be opened either in a new tab (if the app is a progressive web application displayed in a browser) or using the `inAppBrowser` cordova plugin [25] (in the mobile phone application) which produces an overlay with the webpage displayed. This new implementation no longer displays the parsed table as the old application did, because the necessary service `db_backend` is largely disabled.

4. Smartphone application

4.5.4. The connection error modal

Due to non-optimal mobile network connection, server connection issues or bugs in the application it might not be possible for the `DataModel` to download the necessary JSON files as mentioned in Section 4.2. In that case, the user gets notified of the issue by the connection error modal, as shown in the example in Figure 4.10. The connection error modal tries to give the skilled user the ability to identify and repair the error, if possible, by providing the current host and ports to which the application tried to connect to, to download the JSON files. Mobile port in that case means the port of the `ionic` server and web port is the port of the `apache` server. Switching between those ports is possible by activating/deactivating the HappyFace compatible switch in the settings menu. The host and ports can be changed in the connection error modal before pressing the “Retry” button to try to download the files again. In case further information on the errors is needed, below the ports the files that resulted in errors are listed. These errors are also displayed using their `http` error code. This code is extracted from the `status` property of the `XMLHttpRequest`, therefore an error code of 0 means the application was unable to connect to the server for some reason, which could not be determined further. If all corrections were applied and the error has (hopefully) been resolved, the user can choose below the status code if he/she wants to save the new configuration for the future (it will be overwritten the next time he/she switches the instances in the instances chooser) and can then retry to download the JSON files with the “Retry” button in the bottom right. Alternatively, the user can decide not to reload the files (maybe he/she wants to switch instances anyway or does not know how to solve the issue) and close the connection error modal with the cross button in the top right.

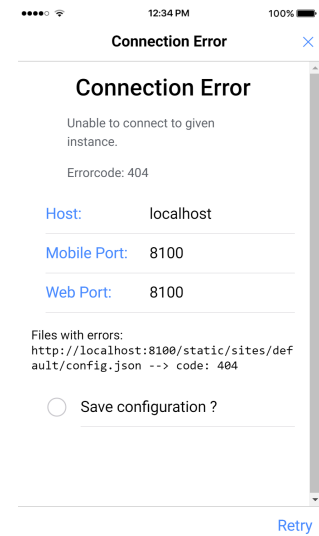


Figure 4.10.: The connection error modal.

4.5.5. Widgets

As stated in Subsection 4.4.6, the home tab provides a widget system to the user, which enables him/her to modify the user interface based on his/her own workflow and thereby get a personalised graphical user interface which speeds up the monitoring work. For this purpose widgets are required, which provide parts of the user interface together with the underlying logic. While this thesis was written, two widgets were implemented and are

shipped with the application (as well as four search widgets which are necessary for the search function from Subsection 4.5.6) and are described there in detail). These widgets are:

- **Example Widget** Intended as a proof-of-concept, this widget displays the current status icon in an enlarged form. The example widget is visible in Figure 4.11(a).
- **Critical urls widget** The critical urls widget is designed to speed up the process of identifying the modules which have failed. Instead of searching for them in the monitoring tab or the search function, the critical urls widget will display all failed modules as well as the current status level. The critical urls widget is visible in Figure 4.11(b).

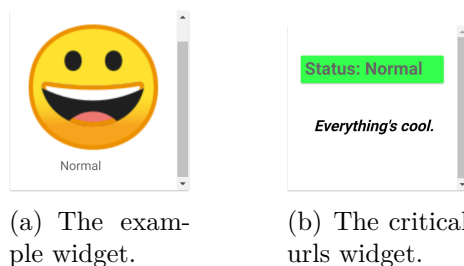


Figure 4.11.: The two current widgets.

The widgets do not have direct access to the data stored in `DataModel`. Therefore, all widgets extend the `BaseWidget` class which can provide this access and some additional functions, for example to display an enlarged image. Furthermore, by extending it, the new widget needs to provide important information for the loading system, such as the name to be displayed in the header overlay, the preferred size (which will be rendered down if the preferred size is bigger than the screen) and its HTML template. A complete guide on how to write/compile/deploy a widget can be found in the appendix under A.

4.5.6. The search function

If specific information about concrete modules is needed, it might be tedious work to find all associated plots in the monitoring and analyzer tab. To simplify that, and to display the information in one central place, the home tab provides a search function.

The search bar of the search function becomes visible when the user taps on the magnifying glass button in the action bar of the home tab. If it is visible, a second tap on the button will make the search bar disappear again. In the search bar the user can enter

4. Smartphone application

the name of the module he/she would like to have information about, together with a statement indicating the kind of information he/she desires (further on called “action”). It is important that the action is placed in front of the name of the module for the search system to understand it. Additional words in the search bar are filtered out if they are contained in a stop word list, which contains the most common words of the English language, thereby eliminating all unnecessary words ([26]), so that the search function can also react to human-like questions in addition to simple commands, for example: “What is the status of Kibana Frontier?”. This is a valid command for the search bar as it contains the action (“status”) and the name of the module (“Kibana Frontier”). All other words are ignored as irrelevant. The case of the letters in the command is also ignored, so that the module name “Kibana Frontier” is equal to the module names “kibana frontier” and “KIBANA FRONTIER”. To make the search even more intuitive to use, each action name also has different synonyms to which the search function will react in the same way. Currently the search function knows four different actions:

- **status** This action will result in the current status of the named module, so it will display its latest available browser screenshot.
Its synonyms are: **state**, **situation** and **condition**
- **history** History is designed to show the output of the module over time, so the history function will display the browser screenshots from all times which are available in summary.json.
Its synonyms are: **chronic**, **past**, **annals** and **record**
- **dependency** The dependency action will show the dependency plot of the named module. Thereby it shows the modules on which the named module depends.
Its synonyms are: **dependence** and **depend**
- **analysis** This action will display the plot of the named modules stability over time.
Its synonyms are: **graph**, **time**, **plot** and **stability**

Since the search function is part of the home tab, the results of searches are displayed in the form of widgets. Currently each action has its own widget to display its results. This concept has the advantage that multiple searches can be executed one after another but the results from all of them are still visible. Also the results are bundled in one widget of a certain size, meaning that actions with very large results (for example the history action, which displays all available old status images at once) do not block the whole screen. The results of searches, therefore, need to be deleted in the widget modification mode of the home tab, which is available with the tool box button in the action bar.

5. Outlook and Conclusion

5.1. Outlook

During the time of development of this application, some ideas emerged for which there was not enough time to implement. A future extension of this application could therefore implement them to increase the functionality and usability of the application. These ideas include a movable and modifiable dependency network in the analyzer tab, which could provide a better readability of the currently often small and narrow lying network components. This idea was tried during the development phase with a network generator called `vis.js` [27] but was abandoned due to the highly computationally intensive network building process in `vis.js`. A future work on the application could potentially find a better working library for networks or optimise the building process in `vis.js`.

Furthermore the capabilities of progressive web applications have not been fully used. While `ionic` supports them, this application uses its deployment as a progressive web application solely to display the same interface on the browser. If the current trend from native to web based smartphone applications continues, the application can focus more on this deployment method by modifying the interface and functionality to fit its needs. Finally the application currently does not have a notification system. This system could automatically check the status in the background and notify the phone user if any changes occur. However, such a system of background tasks is not ideally implemented in `ionic` and would need two custom `cordova` plugins specifically optimized for this task each for iOS and Android.

5.2. Conclusion

The HappyFace smartphone application was developed to extend the capabilities of monitoring grid clusters and to simplify its workflow. It allows users to check the status either of the complete cluster or a single module from wherever they are, requiring only a smartphone with a mobile network or WLAN connection. It also provides the ability to perform simple repairs, thereby minimising the instances where a user requires access

5. Outlook and Conclusion

to a notebook or desktop computer.

The development of the application had its focus on the WLCG ATLAS Tier-2 centre GoeGrid but is easily extendible for other WLCG centres and Tiers. After finishing the implementation of the new application, it has been uploaded to the Apple App Store for iOS and the Google Play Store for Android. From there it is freely available to all interested users. The source code of the HappyFace smartphone application and its back end modules for HappyFace have also been submitted to GitHub, a free repository for open-source code. This will enable developers to extend its functionality in the future in order to accommodate the ever growing needs of the WLCG.

A. List of links

- The complete source code of the HappyFace smartphone application is publicly available in a GitHub code repository:

<https://github.com/HappyFaceGoettingen/HappyFace-MadMask/tree/master/HappyFaceMobileDevelopment>

- When the instance chooser starts, it downloads a specific file which serves as an entry point and provides the top level entries in the instances list. From then on, the list of subinstances of a specific instance is saved in the `meta-meta.json` file on the server of that instance, but the top level file does not have a server backend. Therefore, and for the best availability, it is also saved in the GitHub repository of the application:

<https://raw.githubusercontent.com/HappyFaceGoettingen/HappyFace-MadMask/master/HappyFaceMobile/sites/top.json>

- To customize the workflow, the home tab provides a widget system. Using this system, it is possible to develop a custom widget and deploy it to the application. A helpful guide on how to develop/compile/deploy a widget is available in the GitHub repository:

<https://github.com/HappyFaceGoettingen/HappyFace-MadMask/blob/master/HappyFaceMobileDevelopment/docs/widget-guide/GUIDE.md>

Bibliography

- [1] G. Apollinari, et al., *High Luminosity Large Hadron Collider HL-LHC*, CERN Yellow Report (5), 1 (2015), 1705.08830
- [2] C. Lefevre, *LHC: the guide (English version)*, Technical report (2009)
- [3] The ATLAS Collaboration, *The ATLAS Experiment at the CERN Large Hadron Collider* (2008)
- [4] CERN: Education, Communications and Outreach Group, *The Grid: A system of tiers* (2012), URL <https://cds.cern.ch/record/1997396>
- [5] C. G. Wehrberger, *HappyFace Meta-Monitoring for ATLAS in the Worldwide LHC Computing Grid*, Master's thesis (2013), II.Physik-UniGö-MSc-2013/07
- [6] C. Borrego, et al., *Aggregated monitoring and automatic site exclusion of the ATLAS computing activities: the ATLAS Site Status Board*, Technical report (2011)
- [7] Atlas Collaboration, *Rucio—The next generation of large scale distributed system for ATLAS Data Management*, J. Phys. Conf. Ser. **513**(4), 042021 (2014)
- [8] T. Maeno, et al., *Evolution of the ATLAS PanDA workload management system for exascale computational science*, J. Phys. Conf. Ser.
- [9] E. Imamagic, D. Dobrenic, *Grid infrastructure monitoring system based on nagios*, in *Proceedings of the 2007 workshop on Grid monitoring*, pages 23–28, ACM (2007)
- [10] M. L. Massie, B. N. Chun, D. E. Culler, *The ganglia distributed monitoring system: design, implementation, and experience*, Parallel Computing **30**(7), 817 (2004)
- [11] V. Buge, et al., *Site specific monitoring of multiple information systems: The HappyFace project*, J. Phys. Conf. Ser. **219**, 062057 (2010)
- [12] F. Kukuck, *Creating and Publishing a new Smart Phone Application for the Happy-Face Meta-Monitoring-Tool*, Bachelor's thesis (2014), II.Physik-UniGö-BSc-2014/09

Bibliography

- [13] T. Berners-Lee, *Uniform Resource Locators (URL) A Syntax for the Expression of Access Information of Objects on the Network* (1994), URL <https://www.w3.org/Addressing/URL/url-spec.txt>
- [14] V. Mauch, et al., *The HappyFace project*, J. Phys. Conf. Ser. **331**, 082011 (2011)
- [15] *The angular documentation*, accessed: 05.07.2018 12:45, URL <https://angular.io/docs>
- [16] *The ionic documentation*, accessed: 05.07.2018 12:47, URL <https://ionicframework.com/docs/>
- [17] *The cordova documentation*, accessed: 05.07.2018 12:48, URL <https://cordova.apache.org/docs/en/latest/>
- [18] J. Deacon, *Model-view-controller (mvc) architecture* (2009)
- [19] E. Magradze, *Monitoring and Optimization of ATLAS Tier 2 Center GoeGrid*, Ph.D. thesis, Georg-August University School of Science (2015), DISS 2017 B 9618
- [20] *Xterm.js*, accessed: 05.07.2018 12:50, URL <https://xtermjs.org/>
- [21] *OpenSSL*, accessed: 05.07.2018 12:52, URL <https://www.openssl.org/>
- [22] *SSH2*, accessed: 05.07.2018 12:54, URL <https://www.npmjs.com/package/ssh2>
- [23] E. Barker, L. Chen, A. Roginsky, A. Vassilev, R. Davis, *Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography*, NIST Special Publication 800-56A Rev. 3 (2018)
- [24] National Institute of Standards and Technology, *Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publication (FIPS) 197 (2001)
- [25] *inAppBrowser cordova plugin*, accessed: 05.07.2018 12:58, URL <https://www.npmjs.com/package/cordova-plugin-inappbrowser>
- [26] *English stopword list*, accessed: 05.07.2018 13:01, URL <https://www.ranks.nl/stopwords>
- [27] *Vis.js*, accessed: 05.07.2018 12:59, URL <http://visjs.org/>

Danksagung

Firstly, I would like to express my gratitude to Prof. Dr. Arnulf Quadt for giving me the chance to write this thesis and for the warm welcome he offered me in his research group. Also I would like to thank him and Priv.-Doz. Dr. Jörn Große-Knetter for being referees for this thesis.

I would also like to express my gratitude to Dr. Gen Kawamura for his indispensable support during the work on my thesis.

Lastly, I am very grateful to my family for their immeasurable motivation and support during my work.

Erklärung

nach §13(9) der Prüfungsordnung für den Bachelor-Studiengang Physik und den Master-Studiengang Physik an der Universität Göttingen:

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe.

Darüberhinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, im Rahmen einer nichtbestandenen Prüfung an dieser oder einer anderen Hochschule eingereicht wurde.

Göttingen, den 29. Oktober 2018

(Timon Vogt)