

Bachelor's Thesis

Erstellen und Veröffentlichen einer neuen Smartphone Application für das HappyFace Meta-Monitoring Tool

Creating and Publishing a new Smart Phone Application for the HappyFace Meta-Monitoring-Tool

prepared by

Fabian Kukuck

from Henstedt-Ulzburg

at the II. Physikalischen Institut

Thesis number: II.Physik-UniGö-BSc-2014/09
Thesis period: 21st April 2014 until 28th July 2014
First referee: Prof. Dr. Arnulf Quadt
Second referee: Priv.Do. Dr. Jörn Große-Knetter

Zusammenfassung

Um die immensen Datenmengen, die bei den Experimenten des LHC anfallen, zu verarbeiten, ist ein Netzwerk aus Rechen- und Speicherressourcen notwendig: das Worldwide LHC Computing Grid. Das Überwachen der beteiligten Grid-Rechenzentren ist unabdingbar zur Identifikation von Fehlern und deren Behebung, um letztendlich eine stabile Infrastruktur zu betreiben. Jedoch werden für das Überwachen der diversen Bestandteile eines Grids verschiedenste Dienste mit eigenständigen Zugangspunkten und eigenen Datenstrukturen benutzt. Um die verschiedenen Monitoring-Datenquellen zu vereinen, wurde das Meta-Monitoring Tool HappyFace ins Leben gerufen. Im Laufe dieser Abschlussarbeit wurde eine eher unkonventionelle Ergänzung zum HappyFace Tool entwickelt: Eine neue Smartphone Application, die eine neue Art des Zugriffs auf Monitoring-Daten von HappyFace Instanzen ermöglicht.

Stichwörter: WLCG, Grid Computing, Meta-Monitoring, HappyFace, Smartphone, App

Abstract

To process the enormous amounts of data that experiments create at the LHC, a network of computing and storage resources is necessary: the Worldwide LHC Computing Grid. Monitoring the involved grid sites is inevitable for running a stable infrastructure by identifying and correcting failures. However, to monitor the various integrated constituents of the grid, different services with separate access points and own data architectures are used. To centralise and unify the different monitoring data sources, the modular meta-monitoring tool HappyFace was initiated. In the course of this thesis, a rather unconventional addition to the HappyFace suite has been developed: a new smart phone application, enabling a new way of accessing the monitoring data of HappyFace instances.

Keywords: WLCG, Grid Computing, Meta-Monitoring, HappyFace, Smart Phone, App

Contents

1. Introduction	1
1.1. WLCG	1
1.2. Meta-Monitoring	2
2. HappyFace Meta-Monitoring Tool	3
2.1. HappyFace	4
2.1.1. DB Web Service	6
3. Smart Phone Application	11
3.1. Application Structure on the Front End	11
3.1.1. Side Menu View	12
3.1.2. Categories View	13
3.1.3. Modules View	14
3.1.4. Detailed Single Module View	15
3.2. Preparation of the Application's Back End	16
3.2.1. Introductory Words	16
3.2.2. Cordova	17
3.2.3. AngularJS	17
3.2.4. Ionic	20
3.3. Application Structure on the Back End	21
3.3.1. Basis	21
3.3.2. Cordova Plugins	21
3.3.3. From the Beginning	23
3.3.4. Overview	27
3.3.5. A View from Close To	30
3.3.6. Retrieving Meta-Monitoring Data	34
3.3.7. Background Fetch	37
4. Conclusion and Outlook	39
4.1. Conclusion	39

Contents

4.2. Outlook	39
A. Code Extracts	41
A.1. Query Result from the Example in Section 2.1.1	41

1. Introduction

The Large Hadron Collider (LHC) [1] is a proton-proton collider located at CERN, built within the 27 km tunnel of the previous experiment LEP (Large Electron Positron collider). Currently LHC is the largest of its kind, with a luminosity of $10^{34} \text{ cm}^{-2} \text{ s}^{-1}$ and a designed centre-of-mass energy of 14 TeV – unattained by its predecessors. The LHC mainly houses the experiments ATLAS, CMS, LHCb and ALICE.

The high luminosity experiments ATLAS and CMS were primarily designed to discover the Higgs boson, H^0 , predicted by the Standard Model (SM) (respectively the multiple Higgs bosons of possible extensions of the SM), a particle which is indispensable for the SM to be consistent. SM predictions however yield a probability of 1 in 10^{13} or more for the event of a H^0 production in pp collisions.

With the LHC design luminosity of $10^{34} \text{ cm}^{-2} \text{ s}^{-1}$ and a reliable operation of the experiments, some 10^{16} minimum bias events are expected in a full year, with still only some thousands of Higgs bosons being produced [2] – this is like looking for a needle in a haystack.

After the preselection of events, the data output of all four experiments is around 700 MB/s or about 15×10^6 GB/year [3] which need to be stored and analysed; a mammoth task which demands computing powers of a new kind. This is what the Worldwide LHC Computing Grid (WLCG) was established for.

1.1. WLCG

The goal of WLCG is to provide “a data storage and analysis infrastructure for the entire high-energy physics community that . . . use the LHC” [4, p. 1] – over 5000 scientists in about 500 research institutes and universities worldwide. This infrastructure is not centralised and located near the experiments, but in order to find a reliable solution within the funding capacities of LHC, it is realised by a network of many clusters around the world, a *computing grid*.

As of today, the WLCG links over 140 computing centres from 35 countries, hierarchical structured in so called *Tiers* [5]:

1. Introduction

Tier 0 Located on the CERN area, this Tier (also called CERN Data Centre) primarily stores all *preselected* event data detected at the experiments. It is also used for first pass reconstruction of the raw data into useful information and is responsible for forwarding the data to Tier 1s.

Tier 1 Tier 1 centres store and archive an agreed share of raw and reconstructed data from Tier 0. They also provide computational resources for reprocessing the data along with storing its output. Tier 1s are connected with the CERN Data Centre with data transfer rates of 10 GB/s. At present, there are 13 Tier 1 sites spread across the globe [6].

Tier 2 Mainly consisting of universities and scientific research centres, Tier 2 sites provide computational resources and intermediate storage for end-user analysis and production plus reconstruction of Monte Carlo simulated events. Tier 2s are fed with data from Tier 1 centres and the generated data at Tier 2s is sent to the Tier 1 sites for permanent storage. At present, there are around 155 Tier 2 sites spread across the globe [6].

1.2. Meta-Monitoring

To secure a smooth workflow of a computing site, failures of its involved systems need to be identified and localised, preferably when they indicate themselves in an early state, having no serious impacts on the operation yet. Hence, *monitoring* of the grid parts provides information about almost all involved components (both hardware and software). However, checking the monitoring data of various involved systems is often a tedious work. To provide a remedy for that, the meta-monitoring tool HappyFace was designed [7]. It functions as a single point of contact to obtain monitoring information from different sources.

Currently, the meta-monitoring data of HappyFace monitored sites is available in the form of a website. The goal of this thesis is to bring this output to mobile devices, in the form of a smart phone application. This will further improve the handiness of HappyFace, as the data will be even more easily observable.

2. HappyFace Meta-Monitoring Tool

Maintaining a computing site in everyday life can be a cumbersome task. Not only its computing and storage hardware has to be observed, but also the integration of the local system into the grid as a whole, with all the incoming and outgoing traffic [8]. Additionally, the outputs of many monitoring applications are insufficiently structured and hard to be analysed by non-professionals [7]. Furthermore, it is uncomfortable to check all the different systems with their different ways of access and presentations of data. Nevertheless, monitoring the site is indispensable to meet the “strict obligations concerning the operational availability” [7, p. 1] imposed on members of the grid (e.g. the cause of occurring failures has to be identified within 8 hours after their appearance).

An improvement to this bring *meta-monitoring* tools, such as *HappyFace*: the idea is to collect monitoring data of different sources and make them available on a single access point.

Meta-monitoring tools gather information from various existing monitoring applications and unify the output. The unification is composed of the single access point on the one hand, and on the other hand a meta-monitoring tool could introduce a rating schema by which the states of all monitoring applications appear in a consistent manner. Overall, meta-monitoring tools should preferably embody certain attributes such as [7]:

- **compatible**: be functional with all common operating systems.
- **accumulative**: smartly summarise all relevant information in one output.
- **current**: display the latest monitoring data.
- **retraceable**: provide a *history functionality* to obtain outdated data.
- **accessible**: consist of a lightweight architecture, enabling high performance.
- **comfortable**: do not allocate results more than three navigational steps away.
- **evident**: present results in an easy to grasp, visual way.
- **customisable**: means to set up alert algorithms and tests.

2. HappyFace Meta-Monitoring Tool

Meaning	Value r
<i>ok</i>	$r \in [0.66, 1]$
<i>warning</i>	$r \in [0.33, 0.66)$
<i>critical</i>	$r \in [0, 0.33)$
execution error	$r = -1$
data acquisition failure	$r = -2$

Table 2.1.: Possible ratings for a module [10].

2.1. HappyFace

A meta-monitoring suite designed with all the above attributes in mind is the HappyFace Project [9]. It exactly fulfils the task of gathering status data of various monitoring applications of a site and displaying them in a lucid manner. It is a Python-written modular software framework consisting of the HappyFace core (HappyCore) and site-specific *modules*. The core and module development currently (version 3.0) is a joint project of the Karlsruhe Institute of Technology (KIT) and the Georg-August-University Göttingen, while modules are also being developed at the University of Aachen [10].

HappyFace is structured modular to provide maximum flexibility for arbitrary monitoring sources. HappyCore is responsible for initialising the gathering process of the active modules periodically (typically every 15 minutes). A gathering process accumulates data from the modules which are then stored in an SQLite database, managed by the core. The second part of the framework consists of the entirety of the modules: one module is needed for each monitoring application HappyFace is supposed to surveil. A module includes means to address the respective monitoring application it was designed for. Besides the instruction on how to extract information (and which) from the individual monitoring application, a module also contains an algorithm that determines how to process the aggregated data from a gathering process to form a homogeneous statement of the module's *status* in form of a *rating*. The rating schema of HappyFace assigns each set of monitoring information a value, calculated based on the algorithm defined in the module. Possible values are shown in Tab. 2.1.

With each gathering process, the HappyFace modules deliver the respective monitoring information along with the calculated rating – indicating the summarised state of the module – to the core. The latter then stores the information into the HappyFace database, for later visualisation and access of past module monitoring data.

Aside from that, each module generates an HTML¹ fragment which contains how a typical dataset of the module should be visualised on the HappyFace output [10]. It also includes how the database should be later queried to grasp the desired information from the database, the fragment should be filled with. The fragments are later pieced together by HappyCore, shaping the visual representation of the HappyFace web output. The structure of HappyFace is summarised on Fig. 2.1.

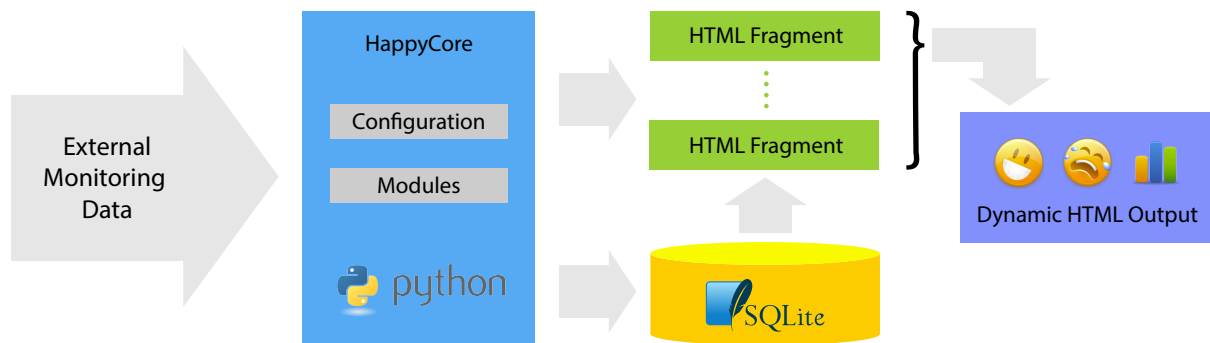






Figure 2.1.: HappyFace Structure, based on diagram from [7].

In the customisation of HappyCore, modules can be subsumed under so called *categories*. Defined categories appear in the output of HappyFace with their name and also a rating, which is calculated from the ratings of the individual modules contained.

The actual output of HappyFace has two different forms: an HTML one and an XML one (Fig. 2.2), created by HappyCore. It is common to deploy them on a server to make them accessible through the web. Every time a user requests an output, the HappyFace tool queries its database for the newest available data and fills the HTML template with the return, generating a visual representation of it. Since the database contains data from every gathering process, a history functionality is relatively easy to implement: the desired date just has to be specified in the database query.

HTML Output Most notably this output of HappyFace consists of the category bar and the navigation bar. A category is represented by its name and an arrow. The arrow represents the rating of the respective category:  for *happy*,  for *warning*,  for *critical* and  for *error*. When the HTML output is called from a user, it displays by default the information of the latest gathering process. If it is wished to see earlier data, the navigation provides the adequate feature.

A click on a category reveals it contained modules. On the left, a small bar gives a glance

¹HappyFace version 2 actually used PHP templates, this was abandoned however.

2. HappyFace Meta-Monitoring Tool

at the individual states each module is currently in. The main body is the part of the output where the HTML fragments are integrated. It offers the detailed insight into each category's monitoring data.

XML output The second type of output HappyFace offers is the XML format. This output form is somewhat reduced compared to the HTML one. It is currently limited to a rather superficial overview of the categories and their contained modules. To all categories and modules it provides the names and ratings and also a time stamp. In addition, the modules here come with a link to their detailed view in the HTML output.

2.1.1. DB Web Service

A module which plays a key role throughout this thesis is the *database web service* (DB web service or simply web service). It is a rather unconventional module, since it is not assigned to any monitoring application. Its purpose is to provide means for data extraction from the SQLite database of HappyFace via the HTML output.

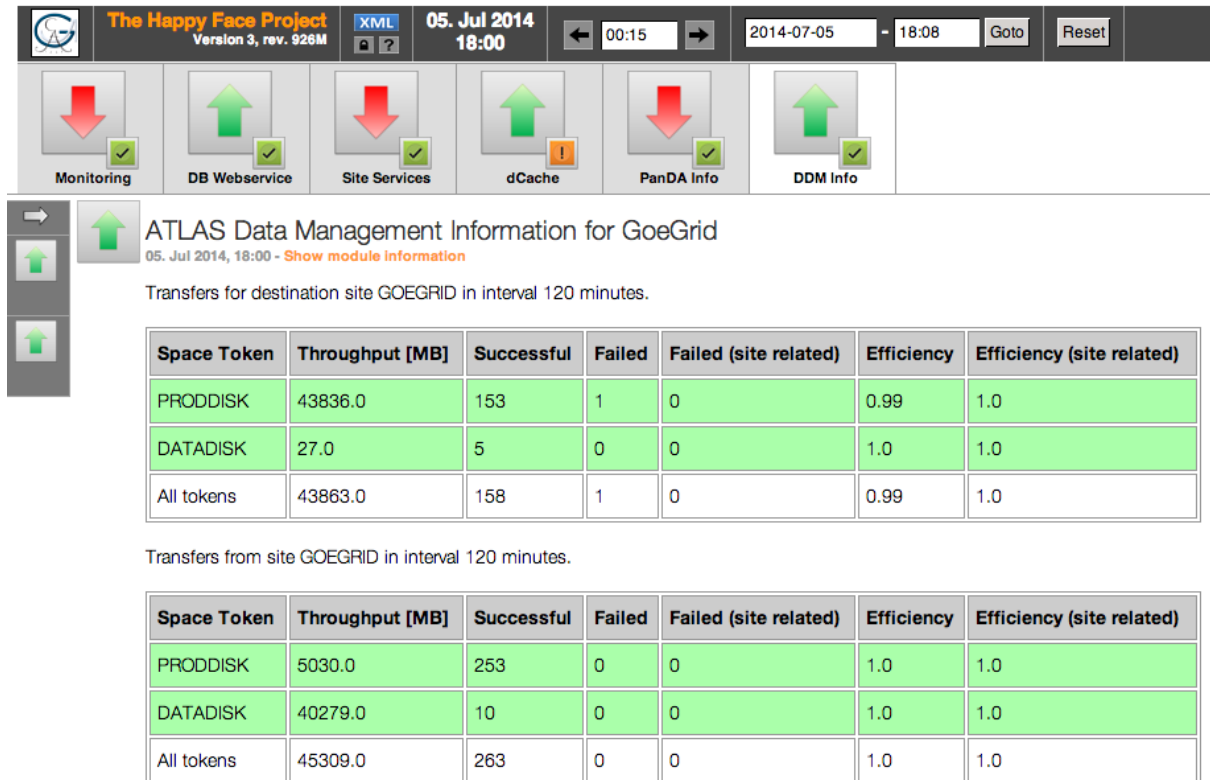
DB web service supports two procedures:

- Extract data from the database via a GUI.
- Extract data from the database by querying specific URLs.

Via GUI If a user of HappyFace wishes to have a fast and lucid overview of the database of a HappyFace instance, or just wants to extract specific data from it, the GUI provides great convenience to do so.

It is accessible from the HTML output in form of a module (or might be wrapped in its own category). The HTML template provides an input for the desired time period, the user wants to extract data from. A click on the button *GET IT!* reveals the database structure in the form of selectable checkboxes – every module has its own entry and only one module can be queried at a time. When the desired module is checked, a second column with the respective module tables and subtables appears – here, too, only a single entry can be selected at a time. In the last column, which will appear when an entry has been selected, the exact content which will be included in the data return can be specified. A click on the button *QUERY!* will query the database and then reveals a link, where the requested monitoring information can be downloaded in JSON² format.

²JavaScript Object Notation



(a) HTML

```

<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=2014-07-05&time=18:00</link>
  <module>
    <name>ddm</name>
    <title>ATLAS Data Management Information for GoeGrid</title>
    <type>rated</type>
    <status>1.0</status>
    <time>2014-07-05 18:00</time>
    <link>http://happyface-goegrid.gwdg.de/category/data_management?date=2014-07-05&time=18:00#ddm</link>
  </module>
  <module>
    <name>ddm__deletion</name>
    <title>Dataset Deletion Efficiency for GoeGrid</title>
    <type>rated</type>
    <status>1.0</status>
    <time>2014-07-05 18:00</time>
    <link>
      http://happyface-goegrid.gwdg.de/category/data_management?date=2014-07-05&time=18:00#ddm__deletion
    </link>
  </module>
</category>

```

(b) XML

Figure 2.2.: Exemplary outputs of the *GoeGrid* HappyFace instance, an ATLAS Tier 2 site of the WLCG, based in Göttingen, Germany.

Via URLs Internally, the logic behind the GUI of the web service performs the desired request by querying the database via a RESTful service implemented with a Python-

2. HappyFace Meta-Monitoring Tool

written CGI script [10]. This query can also be done manually by directly addressing the script, which expects requests in form of a compound URL:

1. The URL begins with the path to the script, called `db_backend.py` :
`http://path/to/webservice/db_backend.py` .
2. Next, the desired (sub)tables and their columns are to be specified, only one (sub)table can be queried at a time:
`http://path/to/webservice/db_backend.py?data=SELECT` .
For each wanted entry, the snippet `+<(sub)table name>.<(sub)table column>`, has to be added. And from the last added one, the `,` has to be removed.
3. After the column specification, some static instructions follow, plus the information in which (sub)table to look for the previous set columns:
`+FROM+hf_runs+INNER+JOIN+<(sub)table name>`
`+ON+hf_runs.id==+<(sub)table name>.id` .
4. As a last portion, the desired time interval where the data should come from has to be specified:
`+WHERE+'<from year>-<from month>-<from day>`
`+<from hour>:<from minute>:<from second>.000000'+<+hf_runs.time`
`+AND+'<till year>-<till month>-<till day>`
`+<till hour>:<till minute>:<till second>.000000'+>+hf_runs.time%3B` .

For example, a valid query URL might be

```
http://happyface-goegrid.gwdg.de/webservice/db_backend.py
?data=SELECT+hf_runs.id,+hf_runs.time,+mod_ddm.instance,+mod_ddm.status
+FROM+hf_runs+INNER+JOIN+mod_ddm+ON+hf_runs.id==+mod_ddm.id
+WHERE+'2014-01-01+13:37:00.000000'+<+hf_runs.time
+AND+'2014-01-01+14:37:00.000000'+>+hf_runs.time%3B3.
```

Calling a URL of this kind executes the script with the specified parameters; the result is stored on the server in form of a JSON file, which can be accessed under `http://path/to/webservice/query_result.json` . Notice that, as soon as a new query is transacted, the old `query_result.json` will be overwritten, so multiple users can not use the web service at the same time – unexpected results might occur if they do so.

³The JSON result of this query can be seen in section A.1

The JSON file contains all the columns specified in the query URL, with data from all HappyFace gathering processes happened in the asked time period.

DB web service is not yet a default module shipped with the current version of HappyFace (version 3.0). However, in the future it may be and the application developed throughout this thesis already supports it.

3. Smart Phone Application

The HTML output of HappyFace is conceived for classical browsers. Currently, the output has no dedicated mobile version which would adapt to the conditions that mobile devices are subjected to (first of all the small screen size). However, only adjusting the HTML output for better readability on mobile devices would waste great potential. Nowadays, it is common that people have smart phones as their constant companions. In addition to providing a smart phone suitable visual output, one hence could make use of the smart phone's capabilities and create functionality like alerting the user when certain events in the monitoring data occur.

In the course of this thesis, a new smart phone application (app) for the HappyFace meta-monitoring tool has been created. It functions as an addition to any HappyFace monitored site and while the use of it is not required, it brings further comfort to monitoring with HappyFace. The app was designed to be usable with any HappyFace monitored site and primarily extends the web output of HappyFace onto mobile devices. It also features a form of alerting the users when changes in category states occur. The app is available on iOS and Android devices.

3.1. Application Structure on the Front End

In this section the user interface (UI) of the HappyFace smart phone app will be explained. The UI is structured in so called *views*. The term *view* describes a section of the app – full-screen displayed to the user – which has a dedicated purpose. The HappyFace app has four major views:

1. the side menu,
2. the categories view (main view),
3. the modules view,
4. and the detailed single module view.

3. Smart Phone Application

3.1.1. Side Menu View

The side menu of the app is what makes it generic – usable with arbitrary HappyFace instances¹. It provides an interface to select which instance should be monitored or to add new instances to the stock. It also gives entry to some app internal settings.

The current stock of HappyFace instances is listed in the body of the side menu. The app can only monitor *a single instance* at a time, the so called *active instance*. It can be set by tapping on the desired instance from the list. The active instance appears highlighted in grey. An instance can be removed from the stock by pressing its *Delete* button, visible after swiping left on an entry.

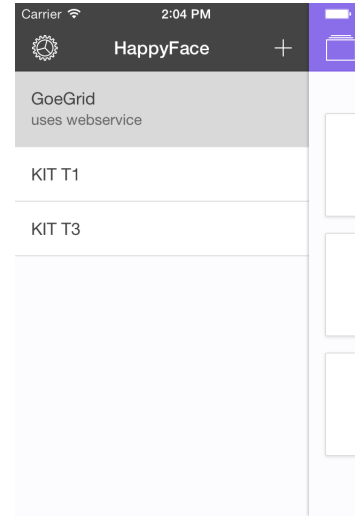


Figure 3.1.
Side menu view.

Adding an Instance To add new instances, a form can be called by tapping the + button on the top right. From here (Fig. 3.2a) the user can either add a custom instance, or choose an instance to add from a preset list (Fig. 3.2b). While the latter is rather self-explanatory, adding a custom instance demands further information.

The HappyFace app hooks into the XML output of a HappyFace instance and all information eventually displayed on the category and module view was first retrieved from there. When adding a new instance it is therefore necessary to specify the exact URL to the XML output of the desired instance. An example for this is

`http://happyface-goegrid.gwdg.de/category?action=getxml.`

Aside from the URL of the instance, input fields for the instance’s *name* and *web service URL* are provided. Both fields are optional and can be left blank while adding an instance. However, for obvious reasons, it is highly recommended to name a new instance. If the web service input field is left empty, the app will not make use of a potential DB web service. If the user wants to use web service, a proper URL has to be provided, an example would be `http://happyface-goegrid.gwdg.de/webservice/`. If this is the case, and a URL has been provided when tapping the *Add This Custom Instance* button, the new instance is marked as *uses web service*.

¹*Instance* shall describe an entity of the HappyFace suite running on a grid site.

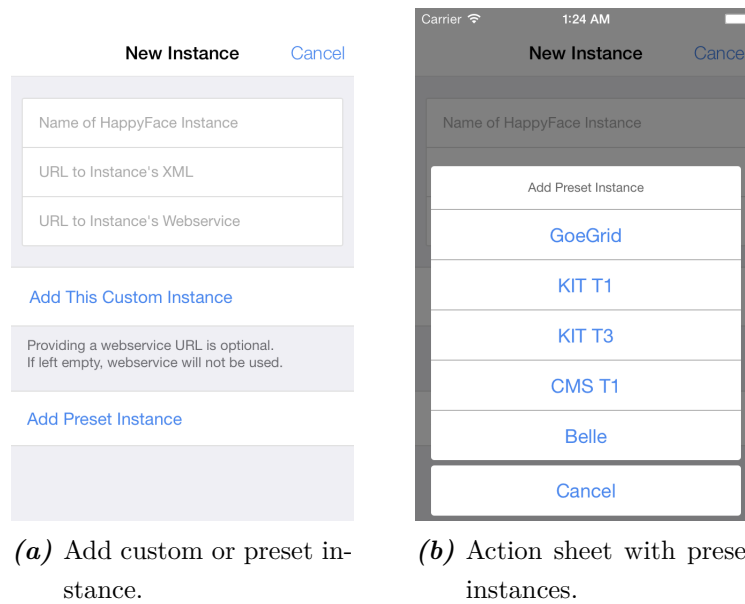




Figure 3.2.: Add new instance form accessible through the side menu (+ button).

Settings The internal settings page of the app might be called by tapping the  button on the top left. Apart from an *About* page, the settings page provides the option to reset the instance stock to default. Upon tapping and confirming the *Reset Instances* button, all instances in the stock are deleted and replaced by default ones.

The *Extended Notifications* toggle is intelligible in the context of the *background fetching* feature of the app: If your platform is iOS 7 or higher, the HappyFace app will automatically refresh the data of its active instance, even if the app is not in the foreground (for more information see section 3.3.7).

If a *change* in a category's status occurred after this process, the app will notify the user. However, if *Extended Notifications* are toggled , the app will alert *every time* it has performed a fetch, even if there were no changes in states.

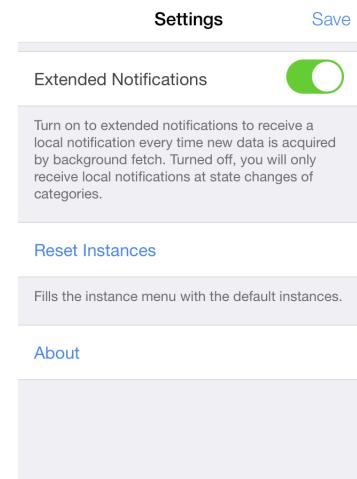








Figure 3.3. Settings page accessible through the side menu ( button).


3.1.2. Categories View

The categories view (Fig. 3.4a), or main view, is the view which shows the categories of your active instance. It is that view, which is shown after the app has finished launching. From the side menu (section 3.1.1), the view is accessible by tapping on a listed instance.

3. Smart Phone Application

Each category of the active instance is represented by its title and its status, indicated by an icon:  for *happy*,  for *warning*,  for *critical* and  for *error*.

The button  on the top left opens up the side menu, which can also be opened by dragging the categories view to the right.

The title of the categories view is the *name*, which the active instance was given. The button  on the top right refreshes the data for the active instance from the server. For this, an Internet connection is required. A time stamp (*data from ...*) indicates the date of the displayed data. The date originates directly from the loaded XML from the HappyFace server – it is not the date of the last successful refresh.

What is however dependent on the last date of refreshing, is the criterion for the data being *outdated*. The app will automatically blend in a grey overlay (Fig. 3.4c) on the view, when more than 20 minutes have passed since the last successful refresh – the data is then considered outdated. This, of course, is instance-bound, meaning a refresh on instance A will not revert an outdated state of instance B.

A tap on an instance leads to the *module view*.

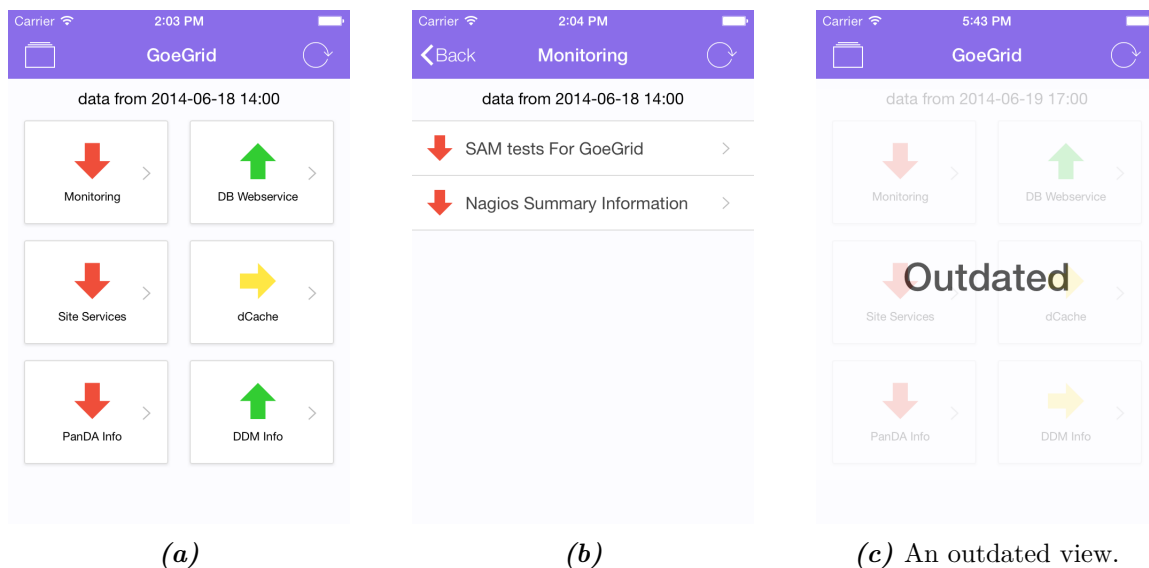




Figure 3.4.: Categories (main) and modules view.

3.1.3. Modules View

The module view (Fig. 3.4b) reveals the modules which are contained in a category. The modules are depicted within a list. Each contained module is represented by its title and its status. The same icon indications as for categories (see section 3.1.2) are used. The data of the module view is reloaded alongside the refreshing process of the categories

view. Nevertheless, the same refresh button  as for the categories view can be found on the module view: it refreshes both the state of the categories and of the modules in all categories. The data here will be *outdated* just as the data on categories view. Also, the same time stamp as in the categories view is shown in this view.

Tapping on a listed module will lead to the *detailed single module view* of that particular module. The Back button directs to the categories view.

3.1.4. Detailed Single Module View

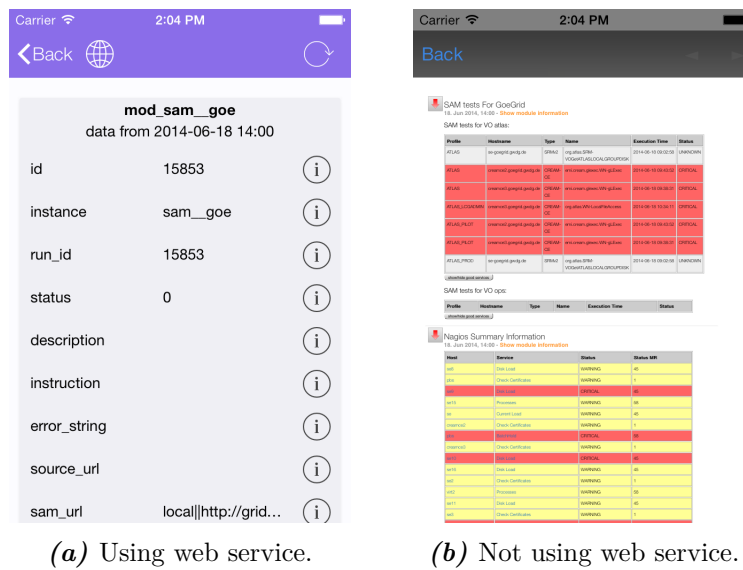


Figure 3.5.: Detailed Single Module View.

This view gives information about the detailed status of a single module. It comes with two faces, and the usage of web service determines which one shows. If the active instance uses web service (in the sense of section 3.1.1) a tap on a listed module in the modules view will lead to the view depicted in Fig. 3.5a. If web service is not used, it will lead to the view depicted in Fig. 3.5b.

Using Web Service (Fig. 3.5a) This part of the detailed view makes use of the active instance’s DB web service. It connects to the web service server and loads all tables and subtables which are linked to the active module. After a successful loading process, the retrieved data is displayed – *all* key/value pairs the respective tables contain, are listed. A table or subtable is surrounded by a grey box and its title is displayed in bold. The time stamp from each table is not related to the time stamps of the categories or modules

3. Smart Phone Application

view, but also stems from the retrieved web service data.

The key/value pair of each row can also be displayed as a whole by tapping on the ⓘ button next to the entry. This is especially useful when any text is cut off (with dots . . .) due to limited space.

The refresh button ↻ on this view refreshes the categories and the modules view, but also queries the DB web service for new data. Also, this view can be outdated in the same sense as categories and modules view (if the last refresh was more than 20 minutes ago). The ⬅Back button directs to the modules view. Additionally, the 🌐 button opens up the detailed module view *without web service*.

Not Using Web Service (Fig. 3.5b) On this view, the HTML output of the selected module is displayed, as it would appear on the instance's web page. It supports *pinch-to-zoom* for better readability of the content. This view is essentially an Internet browser showing a slightly altered version of the respective module's web page. In principle, all functionality provided on the original website can be made use of. Thus, provided the proper hyperlinks on this web page, the complete Web could be accessed. To prevent this, all hyperlinks are disabled.

3.2. Preparation of the Application's Back End

In order to expound how the app works *inside*, several preconditions have to be elucidated.

3.2.1. Introductory Words

The smart phone market is not dominated by a single operating system, but rather by two – *Android* and *iOS* (in 2013 either Android or iOS was shipped on 94% of all sold smart phones [11]). The desire for having an application which is available for both major smart phone operating systems Android and iOS, was one of the top priorities during its development. Android and iOS necessitate code written in different programming languages, namely Java and Objective-C/Swift. Without any further effort, it is not possible to write code once and run it on both operating systems. Because coding two different native apps with two different code-bases in different languages is especially time consuming, several *cross-development frameworks* were (and are) developed by third-parties. All of these frameworks have in common that they enable the developer to write an app only once, and then run it on several operating systems. This enhances productivity massively.

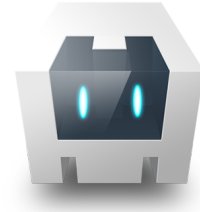
The decision was made that the HappyFace app should be based on the cross-development

framework Cordova. Also, the front-end framework for building the UI was chosen to be Ionic, which is based on the framework AngularJS.

3.2.2. Cordova

Cordova [12] is a framework which provides the ability to write cross-platform apps in HTML, CSS and JavaScript (JS). Cordova is open-source² and is a top level project within the Apache Software Foundation [13].

Mobile applications made with Cordova are hybrid ones. Meaning they are neither native apps, because the UI is made with web technologies, nor web-based apps, because they are packaged and deployed on the regular app markets and have access to the APIs of the host-device. Cordova functions as a *wrapper* for essentially a web page written in HTML, CSS and JS [14]. But the output product is, in all cases, a binary application (.apk for Android, .ipa for iOS). The wrapper also enables the access to the native device API and thus makes features like vibration, file storage, geolocation or camera access possible [15].



Apache Cordova.

3.2.3. AngularJS

AngularJS (commonly abbr. Angular) [16] was made to address the flaw that HTML was not designed for dynamic web pages, but rather for static ones: once loaded, the page remains in its state until the browser reloads the site. This is not what a user would expect from an app, as content should be updated in a dynamic sense.

Angular is an open-source³ client-side JavaScript framework maintained by Google. It uses Model-View-Controller (MVC) architecture, undertakes DOM⁴ manipulation, improves testability and encourages a modular application structure [17]. This makes Angular highly attractive for the development of web applications.

A design goal of the framework is to separate DOM manipulation from the logic of your application. To achieve this, several new components come with Angular. The relevant ones will be introduced in the following [17].

Controllers The application's business logic behind its views is wrapped in Angular *controllers*. A controller can be seen as the direct implementation of anything on an

²Cordova is Apache 2.0 licensed.

³AngularJS is MIT licensed.

⁴Document Object Model

3. Smart Phone Application

app's view which requires further processing. Prominent controller tasks are

- logic executed on user interaction with the UI or
- providing variables of the view.

Factories The application's business logic which is *independent of its views* is wrapped in Angular *factories*. Anything which might be intended to be used on more than two views is likely to be implemented within a factory. Prominent factory tasks are

- means to communicate with external servers or
- storage and manipulation of variables across controllers.

Directives Angular *directives* extend HTML by providing custom attributes and elements. All DOM manipulation (UI changes) is realised with directives. Prominent directive tasks are

- showing/hiding DOMs,
- repeating specific DOMs (such as table rows) or
- registering taps on DOMs (user interaction).

Models The data which is shown to the user in the application's views are Angular *models*. These models can primarily be modified by controllers. However, Angular uses *two-way data binding*. This implies that models which are altered through *user input*, are automatically changed in the controller as well – and changes which arise from the controller immediately alter the user input. Prominent examples are

- all non static data in the view or
- text inputs and toggles of the UI.

Modules Angular modules are containers for parts of an app, most notably they can contain controllers, factories or directives. Modules are the aspect of Angular which makes Angular-written apps *modular*.

A controller which is packed in `module ctrl` might use a factory, which could be packed in `module fac`. One can then make use of the Angular feature *Dependency Injection*, which wires up the different modules and makes sure `module ctrl` can access the functions of `module fac`. Therefore, a defined way must be used to declare Angular modules. For example, the definition of a controller wrapped in a module is done by


```
angular.module(/*module name*/, [/*dependencies on other modules*/)
  .controller(/*controller name*/, [/*factory dependencies*/,
    function (/*factory dependencies*/) {/*controller content*/}
  );
```

An example for a controller `CtrlTest` depending on a factory `FacTest`, wrapped up in a module `ModTest` is

```
angular.module('ModTest', [])
  .controller('CtrlTest', ['FacTest', function (FacTest) {
    //controller content
  }]);
```

Factories and directives are declared analogously.

Promises Angular provides advanced mechanisms which make asynchronously performed routines easy to implement, the *promises*.

A problem that developers face when making web applications is that certain parts of the app require content which first has to be loaded from external sources. In general, it is not clear how long such a loading process might take and whether it even succeeds. However, solving this issue by simply blocking any user interaction with the UI is often considered bad practise. A more convenient way of solving this brings the promise/deferred implementation of AngularJS.

Instead of actual return values, functions are able to return promises. These promises come with the concept of `then` which takes a code block following the promise, but waits with executing it as long as the promise remains unresolved. As soon as the promise gets resolved, `then` executes an appropriate code block, dependent on the outcome of the promise: whether it *succeeded* or *failed*.

Any code which is not in the scope of `then` but also comes after the promise, will be executed right away, regardless of the promise outcome or how long it might take. That is why the whole program flow does not get blocked by an action that might take some time (like an HTTP request).

To give an example, suppose the function `takingSomeTime()` returns a promise. Then

```
takingSomeTime()
  .then(
    function(){
      //success handler here
    },
    function(){
      //error handler here
    }
  );
```

3. Smart Phone Application

results in an execution of the function `takingSomeTime()` and a wait for the `then` block until `takingSomeTime()` resolves (\rightarrow success handler) or rejects (\rightarrow error handler) its promise.

3.2.4. Ionic

Ionic [18] is an open-source⁵ framework, designed to develop the front-end of mobile hybrid apps using the languages HTML, CSS and JS. It relies on Cordova as a wrapper to deploy the created web page on mobile devices. While there are quite a lot of web page frameworks to help creating UIs, Ionic is quite unique – apps feel very ‘native’ and not like a web page, because Ionic was exclusively conceived to create UIs for *mobile apps* and not for web pages or web apps.

Ionic itself is based on AngularJS and consists of a solid set of CSS and JS components like headers, buttons, lists, grids, check boxes, sliders, navigation bars or side menus. All of them are easy to use and designated to be part of dynamic views in mobile applications.

Directives Ionic’s features are first and foremost realised by Angular directives – Ionic provides a large palette of them [19]. Examples are⁶

- `ionSideMenus` to implement side menus in views,
- `ionHeaderBar` to display a bar at the top of a view,
- `ionContent` to define the content area of a view and
- `ionList` and `ionItem` to display lists and its content.

Routing In Ionic, views can be defined and linked-up comfortably. This includes automatic tracking of the route the user takes, making features like a *back-button* easy to implement.

Delegates Further functionality of Ionic is provided by its Angular factories, called *delegates*. They can be invoked from controllers to access certain functions, for example to open the side menu of the app programmatically, without user interaction.

⁵Ionic is MIT licensed.

⁶Following the Angular typographic conventions, directives are written with camel-casing, e.g. `ionSideMenus`, even though the direct implementation in HTML is realised with `ion-side-menus`.

3.3. Application Structure on the Back End

In section 3.1, the app was presented as it would have been to a user; the app's features have been mentioned. This section will take another approach to the app: present it in its internal structure, proximal to its source code.

3.3.1. Basis

Cordova apps follow a strict directory structure. The key-folders are

- `platforms/`, where the specifics of each operating system the app shall support, are controlled; and the Cordova framework code is stored in;
- `plugins/`, where the implementations of *plugins* are stored (see section 3.3.2 for more on plugins);
- `www/`, where the cross-platform app itself, written in HTML, CSS and JS is stored.

The key-*file* which contains app meta-data like its *name* or its *version* is `config.xml` – it is found in the source code's root directory.

3.3.2. Cordova Plugins

Cordova's goal is to make apps work natively on different mobile operating systems, while being written in a single code base. The framework also strives to enable the use of the host device's native APIs. Cordova plugins give the implementation for that. They are pieces of source code, which provide an interface to access native functions (written in Java, Swift, ...) from JS code – they act like a bridge between JS and native code. Each plugin needs an implementation for each platform the app is supposed to run on.

The HappyFace app makes use of the following plugins:

Device Plugin The Device plugin [20] defines a `device` object, globally accessible in JS code. `device` has properties like `platform` or `version`, which for example return the strings `"iOS"` and `"3.2"` if the host device runs iOS 3.2. The Device plugin has an implementation for both Android and iOS and is maintained by the Apache Software Foundation.


3. Smart Phone Application

InAppBrowser Plugin The InAppBrowser plugin [21] provides an isolated web browser within the app. It listens to the JS `window.open()` function. In the HappyFace app this plugin is used to realise the detailed single module view without web service. The InApp-Browser plugin has an implementation for both Android and iOS and is maintained by the Apache Software Foundation.

Network Information Plugin The Network Information plugin [22] provides means to obtain the status of the host device's network connection. For example the network states *wifi*, *cellular* or *no connection* can be returned. In the HappyFace app this plugin is used to decide if an automatic refresh can be performed. The Network Information plugin has an implementation for both Android and iOS and is maintained by the Apache Software Foundation.

Status Bar Plugin With the Status Bar plugin [23] one gains access to manipulating the style of the host device's status bar on top of the screen – for instance hiding and showing it. In the HappyFace app this plugin is used to have the status bar harmonise with the UI correctly. The Status Bar plugin has an implementation for both Android and iOS and is maintained by the Apache Software Foundation.

Local Notification Plugin The Local Notification plugin [24] implements functions to schedule native local notifications on the host device. Local notifications are alerts of an app which are displayed to the user even if the app is not running in the foreground. In the HappyFace app this plugin is used to alert the user when a background fetch has been performed. The Local Notification plugin has an implementation for both Android and iOS and is maintained by third parties⁷. However, only the iOS implementation is currently used by the HappyFace app.

Network Activity Plugin The Network Activity plugin [25] gives the ability to activate and deactivate the network activity indicator  in the status bar. In the HappyFace app the indicator is shown when a refreshing process is currently ongoing. This plugin only has an implementation for iOS and is maintained by third parties⁸.

Background Fetch Plugin The Background Fetch plugin [26] provides means to access the background fetching feature of iOS 7. It is possible to define a routine which is executed when the iOS assigns background fetching time to the app (see section 3.3.7).

⁷By the user *katzer*.

⁸By the user *steve228uk*.

In the HappyFace app this feature is used to perform refreshes of the XML output and web service data from the HappyFace server of the active instance, even if the app is not running in the foreground. A working implementation of this plugin currently only exists for iOS and it is maintained by third parties⁹.

3.3.3. From the Beginning

The mechanics of the HappyFace app begin in the `www/` directory. The HTML templates for the app's UI are found in the folder `templates/`, the code of JS acting in the background is found in `js/`, while the Ionic framework is contained in `lib/ionic/`. Custom CSS which act on top of Ionic are located in `css/` and used images are stored in `img/`. Every Cordova app starts from the `index.html`, it is the entry point that the Cordova framework looks for.

`index.html`

```
<!DOCTYPE html>
<html ng-app="hf">
<head>
  <meta charset="utf-8">
  <title>HappyFace</title>
  <meta name="viewport" content="initial-scale=1, maximum-scale=1,
    user-scalable=no, width=device-width">

  <!-- no telephone number highlighting -->
  <meta name="format-detection" content="telephone=no" />

  <script>...dependencies on other files...</script>

</head>
<body ng-controller="CtrlMain">
  <ion-nav-view animation="slide-left-right"></ion-nav-view>
</body>
</html>
```

The app is initiated like an ordinary HTML5 website. The `<head>` tag most notably contains all the `<script>` dependencies, where the code of the app is found. Notice that the frameworks Ionic and *jQuery*¹⁰ are loaded as well, along with an XML to JSON parsing library¹¹. The `<head>` tag also contains the `viewport` - `<meta>` tag, which determines that there shall be not zooming in the app.

⁹By the user *christocracy*.

¹⁰A very popular open-source JS library [27], MIT licensed.

¹¹XML to JSON Plugin [28], MIT licensed.

3. Smart Phone Application

The remaining code is not ordinary HTML; it is HTML enhanced with Angular¹². The `<html>` is provided with an `ngApp`¹³ directive – this initiates the auto-bootstrap process of Angular, it loads the Angular module specified in its parameter, namely `hf`.

The `<body>` is provided with an `ngController` directive. This assigns the controller `CtrlMain` to everything in the scope of `<body>`; Angular takes only the controller's name and Dependency Injection finds out *where to actually seek for it* by itself.

The `ionNavBar` directive constitutes an entry point for the *views* declared later on.

js/app.js

```
angular.module('hf', ['ionic', 'hf.BackgroundRefresh', 'hf.CtrlCategories',
  'hf.CtrlDetails', 'hf.CtrlMain', 'hf.CtrlModalNewInstance',
  'hf.CtrlModalImpressum', 'hf.CtrlModalSettings', 'hf.CtrlModules',
  'hf.CtrlSideMenu', 'hf.FacCategories', 'hf.FacInstances', 'hf.FacPopup',
  'hf.FacRequest', 'hf.FacUIVariables', 'hf.FacWebpage', 'hf.FacWebservice'])
.config(['$stateProvider', '$urlRouterProvider',
  function ($stateProvider, $urlRouterProvider) {
    $stateProvider
      .state('sidemenu', {
        url: "/sidemenu",
        abstract: true,
        templateUrl: "templates/side-menu.html",
        controller: 'CtrlSideMenu'
      })
      .state('sidemenu.categories', {
        url: "/categories",
        views: {
          'view-content': {
            templateUrl: "templates/categories.html",
            controller: 'CtrlCategories'
          }
        }
      })
      .state('sidemenu.modules', {
        url: "/modules",
        views: {
          'view-content': {
            templateUrl: "templates/modules.html",
            controller: 'CtrlModules'
          }
        }
      })
      .state('sidemenu.details', {
        url: "/details",
        views: {
          'view-content': {
            templateUrl: "templates/details.html",
            controller: 'CtrlDetails'
          }
        }
      })
  })
```

¹²AngularJS is included with the `ionic.bundle.js`.

¹³Angular uses the namespace `ng` for its components.

```

        }
    }
    });
    $urlRouterProvider.otherwise("/sidemenu/categories");
}
});

```

This file contains the module `hf`, which was a parameter of the `ngApp` in `index.html`. This module can be seen as some sort of *mother-module* since it establishes connection to all other used modules in its dependencies¹⁴. Furthermore, this module contains configuration instructions (`config`) which are run when the app is auto-bootstrapping.

The `config` block gets injected the `$stateProvider`, a part of Ionic where the views of the app can be defined – just the step which happens next: four views (`sidemenu`, `sidemenu.categories`, `sidemenu.modules`, `sidemenu.details`) are declared, while the dot-notation indicates that the former is a *parent* of the others. Each view is defined with a template URL (containing HTML for the UI) and a controller (containing JS); the latter manages the logic of the HTML template. Controller and template harmonise together, and on their own they are rather pointless.

The `$urlRouterProvider.otherwise` ensures that if the app is in a condition with no defined state, the app automatically translates to the `sidemenu.categories` view – particularly when the app has just launched and no active state is set yet.

With the views being defined, the next step happens in `index.html`. Because of the configuration of the `$urlRouterProvider`, Ionic¹⁵ automatically translates to the `sidemenu.categories` view. Since this view is a child, the parent is concerned first. This results in the template of the `sidemenu` view being inserted into the `ionNavView` directive of `index.html` and `CtrlSideMenu` activated to control the view. The template of `sidemenu`, `side-menu.html`, also contains an `ionNavView`; this is where the template of `sidemenu.categories` is then inserted – because it is a child of `sidemenu`.

Fig. 3.6 provides a depiction of the structure of the app as a whole; it has not yet been fully explicated, but for the relations among the views that were introduced in this section, it offers a clear illustration.

Since the bootstrapping process of the app has now been described, it is convenient to continue with an overview of the app's core elements.

¹⁴The code of those modules are found in all the files included with the `<script>` tags in `index.html`.

¹⁵More precise: the *ui-router* [29] part of the `ionic.bundle.js`.

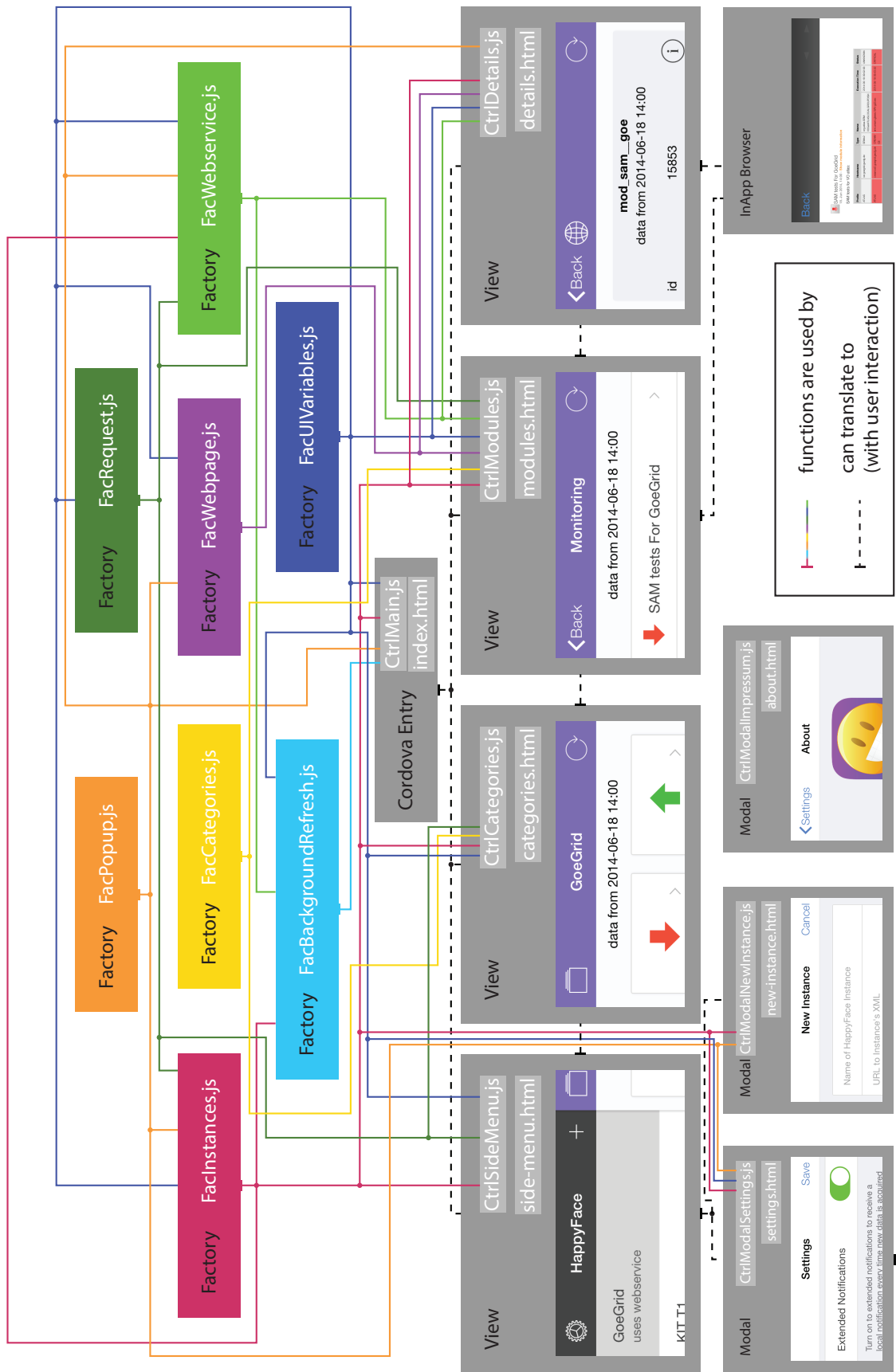


Figure 3.6.: Application code structure.

3.3.4. Overview

As already mentioned in section 3.3.3, each of the four views consist of a template and a controller. The controllers provide the logic directly necessary for the functionality of the template. However, they are not responsible for more demanding tasks; for these tasks they avail themselves of the factories contained in `js/`. The exact dependency tree is depicted in Fig. 3.6. What the individual factories are good for, will be touched by the following.

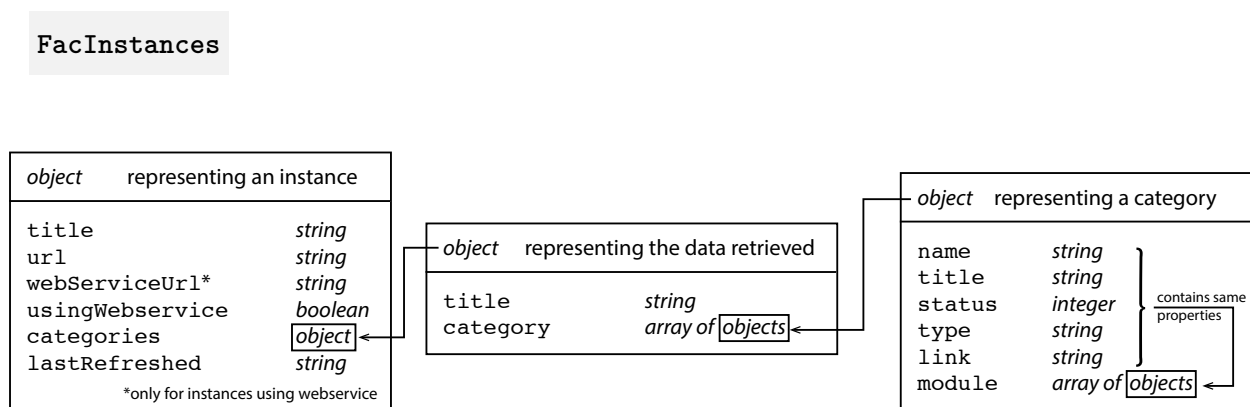


Figure 3.7.: An item of the array `instances`, schematically.

This factory is one of the core elements of the app. It manages the instances of HappyFace monitored computer grids which have been entered into the app. Each instance is an object of the kind depicted left in Fig. 3.7. The instances that are currently in the app's stock are stacked within the array `instances`. The purpose of having this factory is to provide an internal API to access the instances and perform convenient methods on them – from every controller. `FacInstances` provides the following functions, accessible from *outside*:




- `all()`: returns the internally maintained array `instances`.
- `getLastActiveInstance()`: returns the object representing the `activeInstance`.
- `setLastActiveInstance(index)`: sets the `activeInstance` to be `instances[index]`.
- `createInstance(instance, doSelect)`: appends the object `instance` to the array `instances` and sets it the `activeInstance` if `doSelect`. `instance` will already have content loaded via the web (property `instance.categories`).
- `refresh()`: updates the active instance's `categories` from the web. Also gives `lastRefreshed` the current date.

3. Smart Phone Application

- `conditionalRefresh()`: does the same as `refresh()`, but only when the host device has Internet connection and the active instance has not been refreshed in the last 2 minutes.
- `setInstancesToDefault()`: empties `instances` and fills it with the default instances (loading their content from the web).
- `removeInstance(instance)`: removes `instance` from `instances`.
- `lightweightRefresh()`: does the same as `refresh()`, but without any notification for the user on the UI (no loading indicators, etc.).

FacUIVariables

Another core factory is this one; it is meant to afford a single access point for variables which determine what the UI should show in certain situations. `FacUIVariables` provides the following functions, accessible from *outside*:

- setter and getter¹⁶ for `notFirstTimeLaunch`, a boolean which is `false` if and only if the app has been launched on the host device for the first time.
- getter for `isGreyedOut`, a boolean which is `true` if and only if the UI is currently greyed out, i.e. the data is *outdated*.
- `setTimeoutIsGreyedOut(dateString)`: sets `isGreyedOut` to `true` if the provided date (via `dateString`) and the current date differ more than 20 minutes. If they do not, `isGreyedOut` will be set to `true` as soon as the *remaining time* to a difference of 20 minutes has past.
- setter and getter for `sideMenuEnabled`, a boolean which is `true` if and only if the side menu is reachable.
- setter and getter for `isLoading`, a boolean which is set to `true` in order to have the refresh button  on the UI spinning.
- `beginLoading(withIonicLoading)`: Shows the network activity indicator  in the status bar (only for iOS) and overlays the UI with a loading screen if `withIonicLoading` is `true`.
- `stopLoading(withIonicLoading)`: Hides the network activity indicator  in the status bar (only for iOS) if there are no more processes using the network. It also hides the loading screen overlay if `withIonicLoading` is `true`.

¹⁶Functions to modify the value of the respective variable (set), and to retrieve its value (get).

- `getExtendedNotifications()` : returns `extendedNotifications` , a boolean which is `true` if the Extended Notification option is turned on.
- `toggleExtendedNotifications()` : sets `extendedNotifications` to `!extendedNotifications` .
- `isPlatformIOS()` : returns `true` if and only if the host device runs iOS.

FacRequest

This factory is a utility and is responsible for establishing HTTP connections to the HappyFace servers and get data from them. It features two functions accessible from *outside*:

- `Query(instanceUrl)` : returns an object including the retrieved data from `instanceUrl` .
- `cancel()` : aborts any ongoing HTTP request initiated by the factory.

FacWebservice

The actions to query the web service for an arbitrary module and process the data are taken by this factory. Here are the following functions accessible from *outside*:

- setter and getter for `activeModule` , an object which contains all information about a module provided by the instances XML.
- `setTableContent()` : begins the factory internal function chain which at success fills the object `ionicTableData` with updated content provided by the web service.
- `getTableContent()` : returns the object `ionicTableData` .

FacWebpage

This factory is a utility and is responsible for opening the HTML output of modules in an InAppBrowser. It makes the following function accessible from the *outside*:

- `openModuleInAppBrowser(mod)` : opens the InAppBrowser at the URL of `mod` . The function also changes some CSS on the displayed web page in order to remove the category bar which can be found on the top of the page normally.

FacCategories

Functioning as an access point for the current active category, this small factory provides

- a setter and a getter for the index of the active category.

3. Smart Phone Application

FacBackgroundRefresh

What the app ought to execute during a background fetch on iOS is set within this factory. The background fetching process knows what actions to perform once the following function has been called from the *outside*:

- `initializeBackgroundFetch()`.

FacPopup

This factory provides means to display pop-ups overlaying the UI. Therefore it makes available the functions

- `showPopupByName(name, instanceTitle)`: shows the pop-up specified by `name`. With `instanceTitle` it is possible to transmit a *string*, e.g. to display it within the pop-up.
- `showRowInfoPopup(subTableName, key, value)`: shows a pop-up with title `subTableName` and the information `'key: value'`.

3.3.5. A View from Close To

Since the factories which supply the controllers of the views have been introduced in section 3.3.4, this subsection will further explain how template and controller form a unity in Angular apps. Therefore the side menu view, introduced in section 3.1.1, will serve as an example.

As mentioned in section 3.3.3, the `$stateProvider` glues together the template and the controller of a view. This way it is made sure that template and controller *know of each other*. The way controllers communicate with the overlying HTML template is `$scope`. `$scope` is what the glue between controller and template is made of, it is an object that refers to the application model. The `$scope` of a controller is the whole HTML template it is glued to. It can be given JS *properties* which then work as variables in the template.

`sidemenu`: `templates/side-menu.html` and `js/CtrlSideMenu.js`

In the template `side-menu.html` the Ionic implementation of a draggable side menu is used:

```
<ion-side-menus class="lavender-bg">
  <!-- Center content -->
  <ion-side-menu-content drag-content="true">
    ...
```

```

</ion-side-menu-content>

<!-- Left menu -->
<ion-side-menu side="left" is-enabled="sideMenuEnabled()"
  class="lavender-bg">
  ...
</ion-side-menu>
</ion-side-menus>

```

The `class` attributes are the standard HTML ones and modify the visual appearance of the tags. The side menu is only able to be opened if `$scope.sideMenuEnabled()` is `true`; this is the first time the template includes functionality it finds in the underlying controller: `CtrlSideMenu`. Here, the directive `is-enabled` is linked to the function

```

$scope.sideMenuEnabled = function () {
  return FacUIVariables.getSideMenuEnabled();
};

```

This is an implementation of the aforementioned properties of the `$scope` object. Declaring it this way makes `$scope.sideMenuEnabled()` directly dependent on `FacUIVariables.getSideMenuEnabled()`: Angular assumes the duty to change `$scope.sideMenuEnabled()` accordingly to any change of `FacUIVariables.getSideMenuEnabled()` at any time – effectively this results in an extension of the factory variable `sideMenuEnabled` to the template of the view.

The `ionSideMenuContent` is filled with the content which will be shown when the side menu is closed:

```

<ion-nav-bar class="bar-royal nav-title-slide-ios7">
  <ion-nav-back-button
    class="button icon-left ion-ios7-arrow-back button-clear"
    ng-click="goBack()">
    Back
  </ion-nav-back-button>
</ion-nav-bar>
<ion-nav-view name="view-content"></ion-nav-view>

```

The `ionNavBar` directive manages the user's view history and the `ionNavBackButton` always provides the possibility to get to the previous view. Notice the `ngClick` directive on the back button: Whenever the back button is tapped by the user, the function `goBack()` is called, defined in `CtrlSideMenu`:

```

$scope.goBack = function () {
  $ionicNavBarDelegate.back();
  $timeout(function () { //wait for the next $digest cycle
    if ($state.is('sidemenu.categories')) {
      FacUIVariables.setSideMenuEnabled(true);
    }
  })
}

```

3. Smart Phone Application

```
};
```

The implementation of `goBack()` mainly executes the Ionic service to transition back in navigation history. If the state changes to `sidemenu.categories`, the side menu becomes enabled again (resolving the disablement when transitioning to `sidemenu.modules`, implemented elsewhere).

Moreover, `ionNavView` gives the entry point for any child view of `sidemenu` – it will be *overwritten* with the template of the currently active child.

The actual content-wise implementation of the side menu begins in the scope of `ionSideMenu`:

```
<ion-header-bar class="bar-dark">
  <button class="button button-icon ion-ios7-cog-outline"
    ng-click="showModal('settings')">
  </button>
  <h1 class="title">HappyFace</h1>
  <button class="button button-icon ion-ios7-plus-empty"
    ng-click="showModal('instance')">
  </button>
</ion-header-bar>
```

With the appropriate controller function `$scope.showModal = function (string){...};`. After these declarations of buttons in a header bar, the list of the instances stock is introduced:

```
<ion-content scroll="true">
  <ion-list>
    <ion-item ng-repeat="instance in instances()"
      ng-click="selectInstance(instance, $index)"
      ng-class="{ 'active': activeInstance() == instance}"
      item="instance">
      {{instance.title}}
      <p class="instance-uses-webservice"
        ng-if="instance.usingWebservice">uses webservice</p>
      <ion-option-button class="button button-assertive"
        ng-click="removeInstance(instance)">
        Delete
      </ion-option-button>
    </ion-item>
  </ion-list>
</ion-content>
```

It is embedded in an `ionContent`, allowing the user to scroll by drag. The list itself is constituted of the `ionList` and `ionItem` directives – utilised as one expects. However, it is not known how many instances the user will have, ergo how many `ionItem` there have to be. This is where the directive `ngRepeat` with the parameter `instance in instances()` comes into play. It repeats any HTML it is equipped to, similar to a `for`-loop in JS. In this case, the markup is repeated for every item in

```
$scope.instances = function () {
    return FacInstances.all();
};
```

Additionally, the current item of the loop can be accessed within the scope of `ngRepeat` by the variable `instance` – and its index in `instances()` can be accessed with `$index`. The `ngClick` directive makes an `ionItem` tappable, executing the following function:

```
$scope.selectInstance = function (instance, index) {
    FacRequest.cancel();
    FacInstances.setLastActiveInstance(index);
    $ionicSideMenuDelegate.toggleLeft();
    $ionicScrollDelegate.scrollTop(false);
    $ionicScrollDelegate.resize();
    FacUIVariables.setTimeoutIsGreyedOut(
        FacInstances.getLastActiveInstance().lastRefreshed
    );
    FacInstances.conditionalRefresh();
};
```

Besides making use of the factory functions introduced in section 3.3.4, Ionic delegates are called. In the order of appearance they close the side menu, scroll the `ionContent` to the top without an animation, and resize the scroll container (because the newly selected instance might have more or less categories than the previous one).

The `ngClass` adds the CSS class `active` to an item when that particular instance is also the current `activeInstance()`, defined in the controller:

```
$scope.activeInstance = function () {
    return FacInstances.getLastActiveInstance();
};
```

Another new component inside the `ionItem` is the `{{}}`, introduced by Angular. This construct *evaluates*¹⁷ any JS contained, while having access to the underlying controller. Because of `ngRepeat` the `instance` is a place holder for an element of `instances()` (see Fig. 3.7). Hence its properties can be addressed with the dot notation `instance.title` and this results in the respective instance titles.

The `ionOptionButton` directive makes each `ionItem` swipe-able, to reveal an additional button. The list items and the option buttons are equipped with `ngClick`s, wired up to the respective controller function:

```
$scope.removeInstance = function (instance) {
    FacInstances.removeInstance(instance);
    if ((instance == $scope.activeInstance()
        || ($scope.instances().length == 0)) {
        FacInstances.setLastActiveInstance(-1);
    }
};
```

¹⁷ `{{1+1}}` would evaluate to `2` for example.

3. Smart Phone Application

```
        // -1 is code for deleting the active instance
    }
};
```

Also, the information if an instance is using web service is included in an item only if it actually does use the web service, ergo if `instance.usingWebservice` is `true`. This is realised with `ngIf`, which only renders its tag when the provided expression is `true`.

3.3.6. Retrieving Meta-Monitoring Data

The HappyFace app acquires its data from two different sources. First and foremost the app hooks into the XML outputs of HappyFace instances. All data displayed on the categories and modules view is sourced from there.

On the other side, the data on the detailed module view is retrieved from the DB web service module of the instance, provided that a DB web service has been associated with the instance. If this is not the case, the detailed module view is just displaying the ordinary HTML output of the respective module – with no data retrieval in such a sense that the data could be further processed (see section 3.1.4).

Acquiring and Processing the XML Output

The routines which are involved in obtaining and processing the XML output of HappyFace instances will be covered by this paragraph. Those routines are executed immediately after a HappyFace instance has been added to the stock or when the app or user performs a refresh.

The mechanisms described in the following all take place in the `FacInstances` and `FacRequest` and are performed by the `FacInstances` internal functions `createInstance`, `refresh` and `lightweightRefresh`.

The main process of acquiring the XML output begins with the query of the respective server. `FacInstances` therefore calls the `Query` function of the `FacRequest` factory. This function makes use of the `$http` service that comes with Angular [17]. It is essentially a wrapper for the built-in JS way of performing AJAX operations, revolving around the `XMLHttpRequest`. But additionally the `$http` makes use of the AngularJS promises system introduced in section 3.2.3.

Hence `FacRequest.Query` returns a promise whose outcome influences which code block is executed by the following `then`. In case of a promise rejection, i.e. a data retrieval failure of the `$http` request, appropriate notifications are given to the user. If the promise gets

resolved, i.e. data was retrieved from the XML output, the success handler is called. It proceeds with converting the retrieved XML into JSON, by using the library mentioned in section 3.3.3.

After assigning the proper variables with the newly retrieved content, the `lastRefreshed` date of the respective instance gets updated. This information is necessary to be able to assess data as outdated.

Acquiring and Processing DB Web Service Data

There are several steps involved in the procedure of querying the web service and processing the data retrieved. They proceed in a function chain within the `FacWebservice`, which is triggered by the user who opens the detailed single module view for a specific module.

The chain begins with `beginSetTableContentChain()`. This function is responsible for loading and processing the mapping of module names between XML and web service. This mapping exists because the module names delivered with the XML output differ from the ones the web service query URL demands – but the app does only know of the module names from the XML. The mapping also discloses the specific column names of each (sub)table, necessary for a correct query URL.

`beginSetTableContentChain()` takes the name of the active module – the one the user tapped to get into detailed view – from the XML and seeks its corresponding web service name with help of the mapping (the mapping is located on the server-side, within the DB web service module). If successive, the next function in the chain is triggered:

`prepareQueryUrls()` composes the URLs used to query the web service in a subsequent step. The function follows the composing scheme described in section 2.1.1. For a single module several query URLs might be necessary, since a module can be mapped to more than one (sub)table in the database and a query URL can only address one (sub)table at a time.

Since the web service module, in its current implementation, cannot handle queries for a *point* in time but only for intervals, the time interval for the produced query URLs is chosen to be 45 minutes into the past. Certainly, the web service might answer with several datasets contained in the JSON, but a later step will filter the returned data to make it contain only the newest entry.

As of now, the query URLs are prepared to request *all* columns for each (sub)table the mapping delivered in the previous step.

3. Smart Phone Application

After preparation of the URLs, the querying is initiated:

The current implementation of the web service module makes it necessary to execute all requests to it in a successive manner. That is because the result of the latest query gets stored server-side in form of `query_result.json`, but this file gets overwritten as soon as a new query is processed by the DB web service. Thus the way to approach this by the app is to query, download the result, and then proceed with querying the next (sub)table.

`requestWithQueryUrls()` acts according to this pattern – realised with the promise system of Angular. After all query URLs have been processed and their results have been stored, the next item of the function chain is called:

`reformatTables()` rearranges the returned JSON to make further processing easier. A typical `query_result.json` might have been

```
{  "content": [
    ["2014-07-12 18:30:02.000000", 12345],
    ["2014-07-12 18:45:02.000000", 12346]
  ],
  "table_name": "mod_example", "table_columns": "time,mod_example.id"
}
```

The `reformatTables()` would rearrange that object to the following array:

```
[
  {"time": "2014-07-12 18:30:02.000000", "mod_example.id": 12345},
  {"time": "2014-07-12 18:45:02.000000", "mod_example.id": 12346}
]
```

The next function of the chain, `shrinkSubModuleTablesToOnlyNewestDate()`, discards all datasets but the newest. The above array would thus be shrunk to

```
[
  {"time": "2014-07-12 18:45:02.000000", "mod_example.id": 12346}
]
```

The last link of the chain is the function `prepareIonicTableData()`. In this final step the data is reorganised to ensure it to be suitable to be shown to users. The data is written into `ionicTableData`, an array that is directly shown on the detailed single module view. The current version of the app shows *all* columns the web service returns, without any preselection.

3.3.7. Background Fetch

Currently only on iOS, the app comes with a background refreshing capability. It automatically fetches new data from the HappyFace XML output and alerts the user when a change in a category's status occurred. This feature does work when the device is locked or the user has opened another app. However, the app has to remain open (visible in the multitasking switcher of iOS). A closed app will *not* perform background fetching.

The background refresh capability is based on the Background Fetch feature of iOS 7 [30], accessed with a suitable plugin (see section 3.3.2). That is why the interval of fetching however is not directly influenceable by the app – it is regimented by iOS itself. This results in an irregular periodicity of the content being updated. iOS takes into account the internal usage statistics of the app (time of day, usage frequency), to determine when the app gets assigned time to perform a background refresh.

4. Conclusion and Outlook

4.1. Conclusion

The HappyFace smart phone app has been finished in its first version, 0.0.1. It has not yet entered the *Apple App Store*, but has been submitted and currently finds itself in the review process. After an approval it will be downloadable from the regular App Store to any device running iOS 7.1 or higher.

The Android version of it was not submitted to *Google Play*, the Android pendant to the App Store, because the distribution of an app via alternative ways is significantly easier for Android than for iOS devices. In case of Android, the binary application in the `.apk` format will be available on the official HappyFace website [31], hosted by the KIT.

The app is ready to be used for every computing site running HappyFace. To the user, it brings several obvious benefits: To obtain the monitoring data that HappyFace aggregates it is no longer necessary to effort a classical computer. This results in further flexibility for the user, as it is now possible to check data at moments that previously did not allow for a quick glance at the monitoring status.

Also, with the feature of an automatic refreshing process in the background and the app reporting any changes in category states, it is not mandatory to check the status manually to stay informed.

4.2. Outlook

As of now the background fetching feature is only available for iOS devices, as Android manages background activity of running apps totally different than iOS. Future versions of the HappyFace app could provide support there.

Another aspect concerning the background refreshing on iOS is that this feature currently is based on the background fetching capability introduced with iOS 7. As explained in section 3.3.7, it is not possible to strictly control the fetching times with this capability. In case of a category-state-change this might result in a somewhat delayed alert from the

4. Conclusion and Outlook

app. In order to have background refreshes synchronised with the internal HappyFace gathering process, the background refreshing routines could be revamped; letting the new version base on *Push Notifications* [32, ch. 19]. A HappyFace server would then have to communicate with the Apple Push Notification Service.

At the present time the background refreshing routine only alerts the user when a category state change has happened. Since the app makes use of the DB web service module, which allows the data to be further processed, the reasons for an alarm could be chosen freely – and even the individual user could. One might implement a *customise* feature: allowing users to choose the elements (categories, modules, web service columns) they want the alert to be reactive to. Numerical web service columns might trigger an alarm when their value reaches critical limits, which in turn could be set by the user.

A. Code Extracts

A.1. Query Result from the Example in Section 2.1.1

```
[
  {
    "content": [
      [
        17100,
        "2014-07-01 13:45:02.000000",
        "ddm",
        1.0
      ],
      [
        17101,
        "2014-07-01 14:00:02.000000",
        "ddm",
        1.0
      ],
      [
        17102,
        "2014-07-01 14:15:02.000000",
        "ddm",
        1.0
      ],
      [
        17103,
        "2014-07-01 14:30:02.000000",
        "ddm",
        1.0
      ]
    ],
    "table_name": "mod_ddm",
    "table_columns": "hf_runs.id,hf_runs.time,mod_ddm.instance,mod_ddm.status"
  }
]
```


Bibliography

- [1] Lyndon Evans. “The Large Hadron Collider”. In: *New Journal of Physics* 9.9 (2007), p. 335.
- [2] Oliver Sim Brüning et al. *LHC Design Report*. Geneva: CERN, 2004. Chap. 19.
- [3] Christiane Lefevre. *LHC the guide*. Jan. 2008. URL: <http://cds.cern.ch/record/1092437/files/CERN-Brochure-2008-001-Eng.pdf> (visited on 07/22/2014).
- [4] Christoph Eck et al. *LHC computing Grid: Technical Design Report. Version 1.06 (20 Jun 2005)*. Technical Design Report LCG. Geneva: CERN, 2005.
- [5] Ian Bird. “Computing for the Large Hadron Collider”. In: *Annual Review of Nuclear and Particle Science* 61 (Nov. 2011), pp. 99–118.
- [6] *The Grid: A system of tiers*. URL: <http://home.web.cern.ch/about/computing/grid-system-tiers> (visited on 07/22/2014).
- [7] Viktor Mauch et al. “The HappyFace Project”. In: *J. Phys.: Conf. Ser.* 331.082011 (2011).
- [8] Friederike Nowak. “Search for Supersymmetry with Jets, Missing Transverse Momentum, and a Single Tau at CMS”. FB Physik, Univ. Hamburg, June 2012.
- [9] Volker Brüge et al. “Site specific monitoring of multiple information systems - the HappyFace Project”. In: *J. Phys.: Conf. Ser.* 219.062057 (2010).
- [10] Christian Georg Wehrberger. “HappyFace Meta-Monitoring for ATLAS in the Worldwide LHC Computing Grid”. MSc Thesis, II.Physik-UniGö-MSc-2013/07. II. Institute of Physics, Georg-August-University Göttingen, 2013.
- [11] Gartner Inc. *Worldwide Smartphone Sales to End Users by Operating System in 2013*. Feb. 13, 2014. URL: <http://www.gartner.com/newsroom/id/2665715> (visited on 07/22/2014).
- [12] *Cordova*. URL: <http://cordova.apache.org/> (visited on 07/22/2014).
- [13] *Cordova in the Apache Software Foundation*. URL: <http://projects.apache.org/projects/cordova.html> (visited on 07/22/2014).

Bibliography

- [14] Andrew Trice. *PhoneGap Explained Visually*. May 2, 2012. URL: <http://phonegap.com/2012/05/02/phonegap-explained-visually/> (visited on 07/22/2014).
- [15] *PhoneGap Supported Features*. URL: <http://phonegap.com/about/feature/> (visited on 07/22/2014).
- [16] *AngularJS*. URL: <http://angularjs.org/> (visited on 07/22/2014).
- [17] Brad Green and Shyam Seshadri. *AngularJS*. O'Reilly Media, Inc., 2013.
- [18] *Ionic framework*. URL: <http://ionicframework.com/> (visited on 07/22/2014).
- [19] *Ionic API*. URL: <http://ionicframework.com/docs/api/> (visited on 07/22/2014).
- [20] *Cordova Device Plugin*. Version 0.2.8. URL: <http://plugins.cordova.io/#/package/org.apache.cordova.device> (visited on 07/22/2014).
- [21] *Cordova InAppBrowser Plugin*. Version 0.3.3. URL: <http://plugins.cordova.io/#/package/org.apache.cordova.inappbrowser> (visited on 07/22/2014).
- [22] *Cordova Network Information Plugin*. Version 0.2.8. URL: <http://plugins.cordova.io/#/package/org.apache.cordova.network-information> (visited on 07/22/2014).
- [23] *Cordova StatusBar Plugin*. Version 0.1.3. URL: <http://plugins.cordova.io/#/package/org.apache.cordova.statusbar> (visited on 07/22/2014).
- [24] *Cordova Local-Notification Plugin*. Version 0.8.0dev. URL: <https://github.com/katzer/cordova-plugin-local-notifications> (visited on 07/22/2014).
- [25] *PhoneGap NetworkActivity Plugin*. Version 1.0.0. URL: <https://github.com/steve228uk/network-activity> (visited on 07/22/2014).
- [26] *Cordova BackgroundFetch Plugin*. Version 2.0.0. URL: <https://github.com/christocracy/cordova-plugin-background-fetch> (visited on 07/22/2014).
- [27] *jQuery*. URL: <http://jquery.com/> (visited on 07/22/2014).
- [28] Fynetworks.com. *jQuery XML to JSON Plugin*. Version 1.3. July 8, 2013. (Visited on 07/22/2014).
- [29] *Angular ui-router*. URL: <https://github.com/angular-ui/ui-router/> (visited on 07/22/2014).
- [30] Vandad Nahavandipoor. *iOS 7 Programming Cookbook*. O'Reilly Media, Inc., 2014. Chap. 16.3.
- [31] *HappyFace Meta monitoring framework*. URL: <https://ekptrac.physik.uni-karlsruhe.de/trac/HappyFace/> (visited on 07/22/2014).

- [32] Wei-Meng Lee. *Beginning iOS 5 Application Development*. John Wiley & Sons, Inc., 2012.

Acknowledgements

I want to thank Prof. Dr. Arnulf Quadt for giving me the opportunity to work on this interesting topic, as well as to extend my horizon beyond the tasks physicists are usually confronted with. Moreover, I am thanking Priv.Doz. Dr. Jörn Große-Knetter for dedicating his time for my thesis as second referee.

Furthermore I express my gratitude to Dr. Gen Kawamura, Erekle Magradze, Haykuhi Musheghyan, Dr. Jordi Nadal and Prof. Dr. Arnulf Quadt for affiliating me that kindly into the working team and for their support throughout the whole thesis.

Erklärung

nach §13(8) 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 22. September 2014

(Fabian Kukuck)