Friedrich-Alexander-Universität Erlangen-Nürnberg

Technische Fakultät, Department Informatik

Philipp Eichhorn

BACHELOR THESIS

# A Notification Service for an Open Data Service

*Submitted on:*

16.10.2014

*Supervisor:*

Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander University Erlangen-Nürnberg

## Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 16.10.2014

## License

Erlangen, den 16.10.2014

# Abstract

The Open Data Service developed at the professorship allows clients to query and fetch numerous open data sources without having to write custom adapters first. While being able to access large quantities of data in itself is already an accomplishment, clients often wish to perform complex and potentially long running analysis on this data, which becomes difficult in cases where client software is running on mobile devices due to hardware limitations. The notification service as designed and implemented in this thesis outsources the computationally intensive task of actually running the analysis to an external service, by accepting so called “rules” from clients that describe a set of conditions that should be met before sending out a notification. To better demonstrate and evaluate the powers of this notification service the Android application *Pegel Alarm* was developed, that supports people living in Germany in times of potential flooding by enabling them to define alarms that trigger whenever water levels of rivers have risen above critical values.

# Table of Contents

# List of Figures

# 1 Introduction

## 1.1 Original Thesis Goals

Initially the goal of this thesis was to design, implement and evaluate an Android client library for an Open Data Service (ODS) under development at the professorship, as well as an Android client application that uses this library. The primary goal of this application is to support people living in Germany in times of severe flooding. The exact requirements and functionalities of the application were to be determined during the project by conducting interviews with experts in the water domain.

## 1.2 Changes to Thesis Goals

After a round of initial discussions with German water level providers (in particular PegelOnline and Wuppertal) and potential users of the Android application, a new functionality for the Android application emerged, the so called “water level alarms”. By defining such an alarm users could be warned whenever water levels of selected rivers have risen above a critical value, enabling them to prepare their homes for the incoming water. Unlike the original thesis goal, this alarm feature would require mayor modifications of the ODS, which lead to a shift in the thesis goal by primarily focusing on integrating a notification feature into the ODS, while still keeping the two subgoals of creating an Android library and application. The new title of the thesis is: A Notification Service for an Open Data Service.

# 2 Research Chapter

## 2.1 Introduction

Gathering and consolidating unstructured data from multiple inhomogeneous data sources is often a cumbersome and dull task, requiring developers to design, implement and test multiple adapters before even beginning to build the actual application that makes use of this data. The Open Data Service (ODS), which is a Java-based open source server software developed at the professorship, tries to eliminate some of the manual work by tapping into multiple open data sources, gathering available data, consolidating it by applying common data formats and then offering this modified data via a REST API to the public. Sample sources currently in use are PegelOnline[1], a water level portal operated by the *Wasser- und Schifffahrtsverwaltung des Bundes* (WSV) which offers up-to-date information about German rivers relevant for the shipping industry, or Open Street Map[2], a popular service for collecting and sharing various map data. By offering these heterogeneous sources under a single domain using a well defined query API and common data formats, client application developers can spend more time focusing on their actual application logic, without having to worry about how to fetch and interpret the data they need.

While the ODS is an improvement in accessing various data sources, there are many use cases where just having plain, clean and validated data is only the first step in a long processing chain until this data actually becomes useful for users. This is especially the case when looking at sensor data, such as the water levels provided by PegelOnline. Learning that a measuring station along a river is currently reading x cm above some absolute value is great, but only if the user also possesses the necessary domain knowledge to interpret this value x.

Knowing that there already exist a number of widely adopted tools[3] for analyzing large sets of data, this thesis is not going to attempt to build yet another one of these toolkits. Instead it focuses on solving a different not completely unrelated problem: detecting, analyzing and extracting specific events from large sets of data. These events can be as simple as sensor readings exceeding a certain threshold, such as temperatures reaching a critical value, or complex conclusions derived from multiple parameters, such as weather predictions which could depend on temperature, wind and pressure changes over a longer period of time. Surely the previously mentioned tools can be used to some extent to arrive at the same conclusion if periodically fetching the latest data from the ODS. However in the world of mobile computing, any application which depends on periodically downloading and analyzing large quantities of data is not likely going to be successful, as mobile devices such as smartphones and tablets usually suffer from unstable network connections and limited battery reserves.

This problem becomes worse the more data a source contains. Assuming that a client application wants to be informed whenever water levels of German rivers have risen above some

1 http://pegelonline.wsv.de
2 http://www.openstreetmap.de
3 For example http://www.cs.waikato.ac.nz/ml/weka or https://commons.apache.org/proper/commons-math

average value, that application would be forced to fetch all stations from PegelOnline, approximately 560 KB, on regular intervals. Seeing that PegelOnline is gradually moving towards updating water levels every minute, that would lead to a total of about 806 MB[1] of data to download every day. While there is an optimization potential here to reduce this number, for example by fetching stations that are far from a critical value less often, it does not solve the underlying architectural problem: analyzing large data sets should not be done on mobile client devices.

Instead the “heavy lifting” should happen on the server side where the ODS resides. This thesis focuses on accomplishing exactly that. By incorporating a notification service in the ODS, clients should be able to upload some form of rule describing a set of conditions, which when met trigger a notification from the ODS to the client informing him about the new situation. That way communication between client applications and the ODS is kept down to a bare minimum while still allowing clients to run arbitrary complex analysis.

Having extend the ODS on the server side with the notification service, the second part of this thesis focuses on the client side, by firstly designing an Android client library for interacting with the ODS, and secondly by implementing a sample Android client application to better evaluate how well all these components work together. Since there are already numerous well established Android REST libraries and frameworks available[2], the library in this thesis is aimed less at building and running queries to fetch content from the ODS, but rather on automatically keeping data on the ODS in sync with the mobile device, by handling common tasks such as managing timers or setting up the local database while trying to conserve as much battery as possible.

As a last step the Android client application demonstrates the usage of the ODS, the notification service and the client library. The application is targeted at people living in Germany that wish to be alarmed whenever water levels of nearby rivers have risen above critical values. While offering support in times of flooding, this feature also nicely illustrates how the notification service can be used to bring extra computation power to mobile devices, and through that achieve goals that were previously unreachable.

1 540 KB * 60 * 24 = 806.4 MB

2 For example the popular spring platform (http://spring.io/guides/gs/consuming-rest-android/) or Retrofit (https://square.github.io/retrofit/)

## 2.2 Research Approach

This section covers some of the primary requirements that were defined prior to design and implementation of the notification service, the client library and the actual Android application.

### 2.2.1 Notification Service Requirements

As an absolute minimum the notification service should include the following features, which are further described in the subsequent sections:

- The ability to define „complex rules“
- Allow clients to access the data which caused a rule to fire
- A public API for registering / unregistering rules
- The freedom to let clients choose how they would like to receive notifications

#### Complex rules

The term „complex rule“ is rather loosely formulated, partially because the notification service should allow clients as much freedom as possible in defining rules to foster potentially very different client applications. However, to narrow down the term at least a little, a number of additional requirements were defined, regarding the kind of rules users should be able to construct:

- Sliding windows, both length-based and time-based (e.g. all measurements in the last `x` minutes or the last `y` measurements)
- Aggregation across sliding windows (e.g. average temperature)
- Correlation between data items (e.g. `temperature1` followed by `temperature2`)
- Ability to access and combine data from different ODS data sources (similar to SQL joins)

The above list is a minimal set of features that try to outline how potential rules could look like. It has largely been derived from the requirements for the Android client application that are discussed in section 2.2.5. For a detailed analysis of these features and on how such a rule language could look like, please refer to Luckham (2007), who has nicely outlined some of the mayor design decision and requirements that went into building the event pattern language RAPIDE (chapter 8, pp. 145-174).

#### Access to computed data

This requirement is only relevant for mobile clients, which have a limited ability to receive push notifications from servers such as the ODS. For Android the Google Cloud Messaging[1] (GCM) platform allows mobile applications to receive small messages from outside without first having to query servers for news. While these push messages are allowed to carry a payload, their size is

1 https://developer.android.com/google/gcm/index.html

limited to 4096 bytes[1], making GCM unsuitable for transmitting any large amounts of data from the ODS to the client. The Google guidelines suggest Android developers to implement a „send-to-sync“ approach when using their push notification framework, meaning that these push notifications should only be regarded as a „tickle“ informing the application that new data is available on the server, causing the client to schedule a sync operation.

In many scenarios clients might be content to just receive this „tickle“ from the notification service, telling them that their rule has been triggered. If a rule however monitors and aggregates values over a long period of time, clients could wish to access the resulting aggregation without first having to query for all the raw data and performing the calculations themselves. The notification service should therefore give clients direct access to the computed data which caused a rule to fire.

#### A public API

The notification service requires a public API for handling registration requests as well as fetching the underlying data of triggered rules. To stay consistent with the current ODS design, the notification service also uses a REST API for all client interactions.

#### Notification mechanisms

While this thesis largely focuses on Android clients, the notification service should not be limited to working with Android only. Instead it should offer a framework for quickly extending the notification mechanism to include different clients as well, such as applications reachable via HTTP or devices running iOS.

### 2.2.2 Complex Event Processing

To understand what “Complex Event Processing” (CEP) is all about, it makes sense to start with the definition of a “regular” event. According to Etzion and Niblett (2010) an “event is an occurrence within a particular system or domain; it is something that has happened, or is contemplated as having happened in that domain”. When dealing with sensor data such as provided by PegelOnline, interpreting the role of events is easy by defining that each new measurement is equal to one new event in the water domain. In this case there is a clear mapping between things that have happened in the real world and objects in the ODS. In other domains however this linkage becomes less obvious. Taking OpenStreetMap as an example, the fact alone that a new item (e.g. street) has been added to a map does not necessarily mean that this object has just been built. It might also have been discovered only recently by users. Consequently the semantics of a new event can vary depending on what source is being used.

Analyzing regular events is in many cases not particularly exiting. As mentioned before, only knowing that a water level currently has the value `x` will most likely not be very interesting. Receiving notice that a water level has risen above a value `y` on the other hand sounds more like having a potential value for users, for example in an alarm application. However, to arrive at this conclusion two events need to be combined: first an event `A` which states the water level is currently

1 https://developer.android.com/google/gcm/adv.html

below `y`, followed by an event `B` saying that the water level is now above `y`. The resulting event, here “water level has risen above `y`” , requires the two simpler events `A` and `B` to be constructed, and is also called a complex event (Luckham, 2007, p. 7). CEP consequently refers to the analysis, detection and manipulation of such complex events, and is sometimes also known as event processing, event stream processing (ESP) or distributed stream computing (Schulte, 2014).

Since building a whole new event processing language as well as the required compiler components would go far beyond the scope of this thesis, a number of open source event processing libraries have been analyzed regarding their fitness to serve as a basis for the notification service. Three available tools are Drools Fusion[1] developed by JBoss Inc, Siddhi[2] owned by WSO2 and Esper[3] from EsperTech. While these three have mainly been chosen due to their open source nature, there are a number of alternative solutions available (for example see Schulte, 2014), mostly commercial products that often go far beyond being mere CEP frameworks. The following section does a brief comparison between the three and highlights the reasons for using Esper.

### Drools Fusion

While Drools Fusion offers various event processing capabilities, it is actually one component of a much larger business rules management system (BRMS) called Drools. As such the BRMS is more geared towards handling business logic on an enterprise level than offering the bare CEP functionalities required by the notification service. This mismatch becomes even more apparent when looking at the other components that are part of Drools, such as the web UI for managing rules (Drools Workbench), the rule engine (Drools Expert), the business project management suite (jBPM) or the business resource management tool (OptaPlanner) among many others. Even though Drools has been around since 2001, is still being actively developed and maintained and offers a wealth of excellent documentation (e.g. Bali, 2013), the framework would introduce a lot of unnecessary overhead if being used in the notification service, which ultimately lead to the decision to favor alternative CEP solutions.

### Siddhi

Siddhi is an open source CEP engine written in Java which was originally developed as part of a research project at the University of Moratuwa, Sri Lanka and is now being improved by WSO2 Inc. Siddhi tries to eliminate a number of limitations of similar CEP engines and has even proven to outperform rivals such as Esper in terms of performance (Suhothayan et al., 2011). While the bare CEP engine is freely available[4], documentation and examples are unfortunately difficult to find and in many cases outdated[5]. This fact and the lack of advertisement of successful products which are actually using Siddhi, suggest that the engine has not been widely adopted, which makes

1 http://drools.jboss.org/drools-fusion.html
2 http://wso2.com/products/complex-event-processor
3 http://esper.codehaus.org
4 https://github.com/wso2/siddhi
5 For example the Siddhi Google user group was updated last on 03.12.2012 and includes broken references to documentation material (https://groups.google.com/forum/#!topic/siddhi-dev/pL1QJIREMp8). A similar state can be seen when looking at a Siddhi developer blog, which was also last updated in 2012 and contains broken links to the source code (http://siddhi-cep.blogspot.de/).

Siddhi unsuitable as a basis for the notification service.

#### Esper

Esper is a CEP engine developed and maintained by EsperTech, a US based software company focusing on event processing solutions. While the company offers various commercial products, their base product, the Esper CEP engine, is open source and published under the GNU General Public License (GPL) v2. Unlike the Drools framework, Esper only contains the bare CEP engine, including an event processing language (EPL) for defining rules, without any UI components or planning features and, unlike Siddhi, offers a well documented API[1] containing numerous examples. This ultimately lead to the decision of adopting Esper as a basis for the notification service, even though this resulted in an additional requirement regarding the license compatibility between Esper and the ODS as discussed in the following section.

### 2.2.3 License Considerations

Despite the fact that Esper as a CEP engine met all of the requirements to serve as a basis for the notification service, directly integrating the engine into the ODS was not possible due to incompatible licenses between the two. While the Esper engine is being distributed under the GNU GPL v2, the ODS is published under the GNU Affero General Public License (AGPL) v3, which cannot be used in conjunction with one another (see section 3.1 for a detailed discussion on licenses). In addition the notification service should not limit our ability to change the license of the ODS in the future, for example in case the ODS were to be used in a closed source commercial product. Consequently, the Esper engine should be integrated in such a way, that it would not impose any additional restrictions (e.g. those defined by the GPL) on the ODS or its license.

### 2.2.4 Android Library Requirements

The library can be divided into two distinct parts: an ODS and a notification service client implementation. As mentioned in the introduction, the ODS library should be less focused on providing yet another REST client framework, but rather on the ability to keep remote ODS content in sync with a local database while maximizing battery life. The notification service library should be responsible for handling rule registration, managing those rules in a local database as well as implementing GCM related components to receive notifications and forward those to the actual application.

### 2.2.5 Android Application Requirements

In order to explore possible use cases for the Android client application, a series of five informal interviews was conducted with various experts of the water domain, ranging from water level portal operators to disaster management professionals, asking them what they would like to see in a flooding application. The results of these discussions were as follows:

- Consolidating different German water level portals into the ODS

1 http://esper.codehaus.org/esper-5.0.0/doc/reference/en-US/html/index.html

- Displaying all water levels along a river in a single graph
- Alarming users when water levels have risen above a certain point

Unfortunately the first idea, although appealing as it seemed to perfectly fit the work description of the ODS, had to be discarded early on due to the missing cooperation of too many portal providers. Out of the 14 distinct providers that are operating in Germany, 12 were contacted via mail and phone, requesting permission to include their data in the ODS. Eleven states replied but only two[1] agreed to have their data integrated (see also section 3.2) Therefore, the advanced river water level analysis tool (second feature) and the water level alarms (third feature) were left to implement in the Android application. Out of these two only the alarms feature is really relevant for this thesis and is hence analyzed in a bit more detail.

The rationale behind the alarm feature is that people living along rivers which are at risk of being flooded during rainy seasons, usually know how high the river can rise before water starts reaching their home. Consequently they would like to be warned whenever water levels have reached a critical value such that they can prepare for the imminent water. The Android application partially automates this warning procedure by allowing users to register water level rules on the notification service.

1 Mecklenburg-Vorpommern and Brandenburg.

## 2.3 Research Result

This section analyses the final architecture and functionalities of the notification service, the client library and the Android application. From now on the term CEP service (or CEPS in short) is used as a synonym for the notification service.

### 2.3.1 The Basic Architecture

Out of all the previously discussed requirements, the inability to directly include the Esper CEP engine in the ODS due to license conflicts (see 2.2.3) had the greatest impact on the final architecture of the system and is hence discussed first. In order to rule out any potential license issues, the ODS cannot use or depend on the Esper engine in any way. As a result the code which actually interacts with Esper related components had to be completely separate, ideally in such a way that the ODS could operate even without any CEP functionalities. In order to achieve this strong separation between the two, the notification service was built in the form of a server application which could be started and stopped independently, much like the ODS. Clients wishing to register CEP rules have to communicate with this new server, while using the "regular" ODS for fetching data. Figure 1 shows this setup.

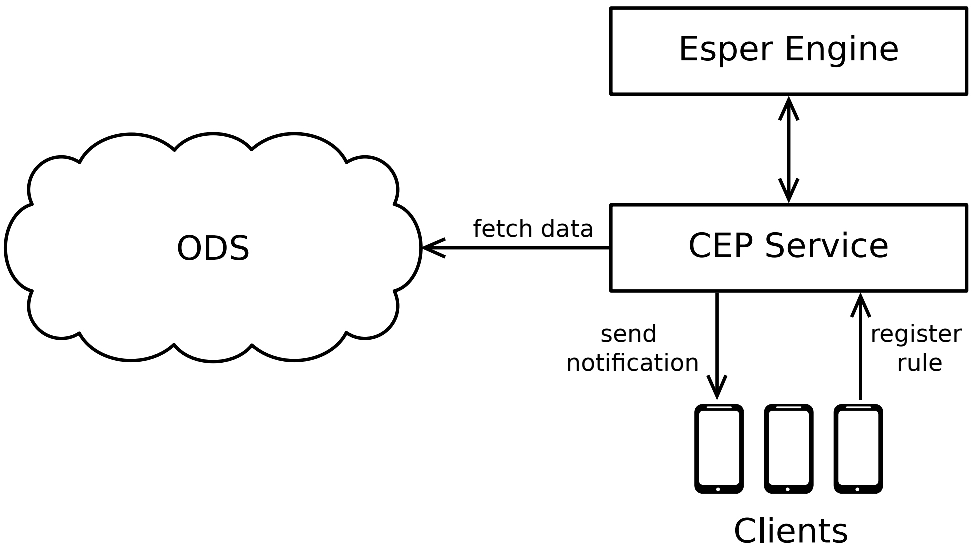


***Figure 1**: Interaction between the ODS, the CEPS and clients that wish to receive notifications. In this setup clients have to communicate with the ODS to receive bare data, and use the CEPS for managing notification rules.*

By using this two server architecture, the ODS is unaware of the CEPS and the Esper engine and can therefore be considered "free" of any GPL obligations. The CEPS on the other hand, as it directly interacts with the Esper engine, is subject to the GPL and must therefore use the same license[1]. While this architecture manages to preserve the license of the ODS, it adds additional

1 Whether this is truly a "must" is partially up for discussion, see 3.1 for details.

complexity to the overall project by requiring the two services to communicate over a network, as well as forcing clients to deal with two service providers instead of one.

### 2.3.2 A Simple ODS Notification Feature

Before starting to build the actual CEPS, the communication between the two services had to be designed. While using a pull-based approach to regularly fetch data from the ODS to forward to the Esper engine for processing is certainly a technically feasible approach, it suffers from all the common weaknesses of pull-based implementations, be it an increased load on both the ODS and CEPS, the chance of missing some new data or the unavoidable delay between new data being published and being fetched.

To overcome those limitations a simple push-based notification architecture was integrated into the ODS, which essentially implements a publish-subscribe system where the ODS is responsible for forwarding any new data to registered clients such as the CEPS. This notification architecture was hooked into the filter framework of the ODS. For every source on the ODS, a filter chain can be defined which takes care of fetching data, validating its content and storing the data in a database among other tasks. Figure 2 shows a simplified setup of such a chain for the PegelOnline source.

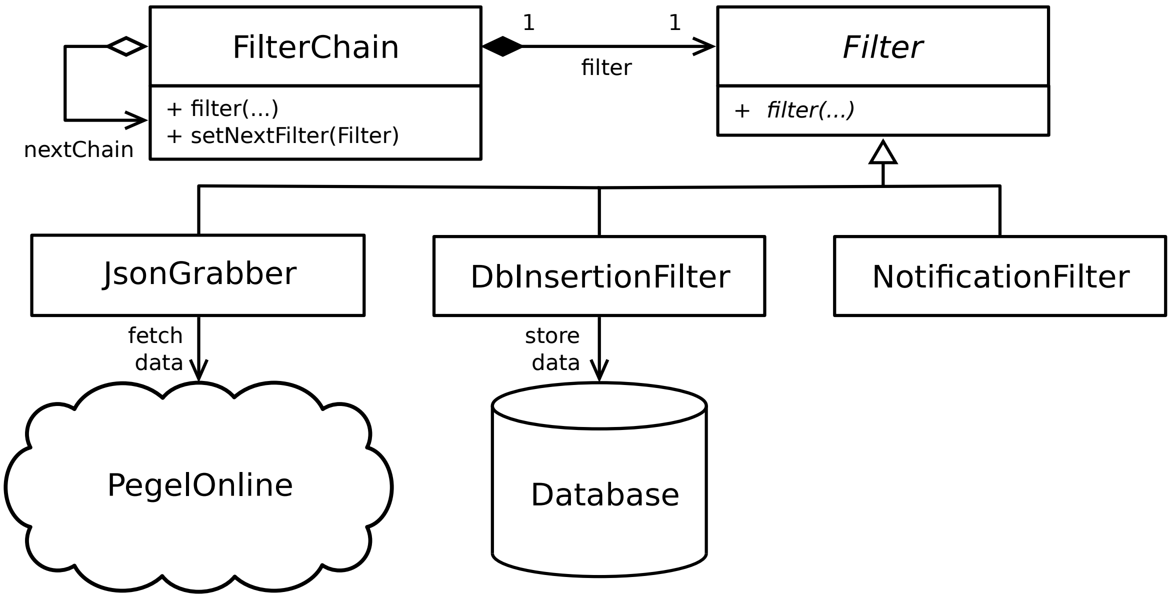


***Figure 2**: The ODS filter framework. While each data source can define its own filter chain, these chains usually contain some form of grabber for fetching data (here* `JsonGrabber`*), a* `DbInsertionFilter` *for storing data in a database and a* `NotificationFilter` *for alerting clients that new data is available. All these filter have to implement the* `Filter` *interface. The class* `FilterChain` *is responsible for the linkage between the individual filters and also serves as a starting point for the filter chain.*

The notification framework itself is comprised of a number of components, most notably the `NotificationManager`, which receives registration and deregistration requests from the REST API, stores the client information in a local CouchDB[1] instance, and forwards any new data to separate sender implementations that take care of actually notifying clients. Figure 3 shows a class diagram of the architecture of the notification framework and figure 4 a corresponding sequence diagram. The `NotificationFilter` is part of the filter chain and acts as an entry point for new data into notification framework.

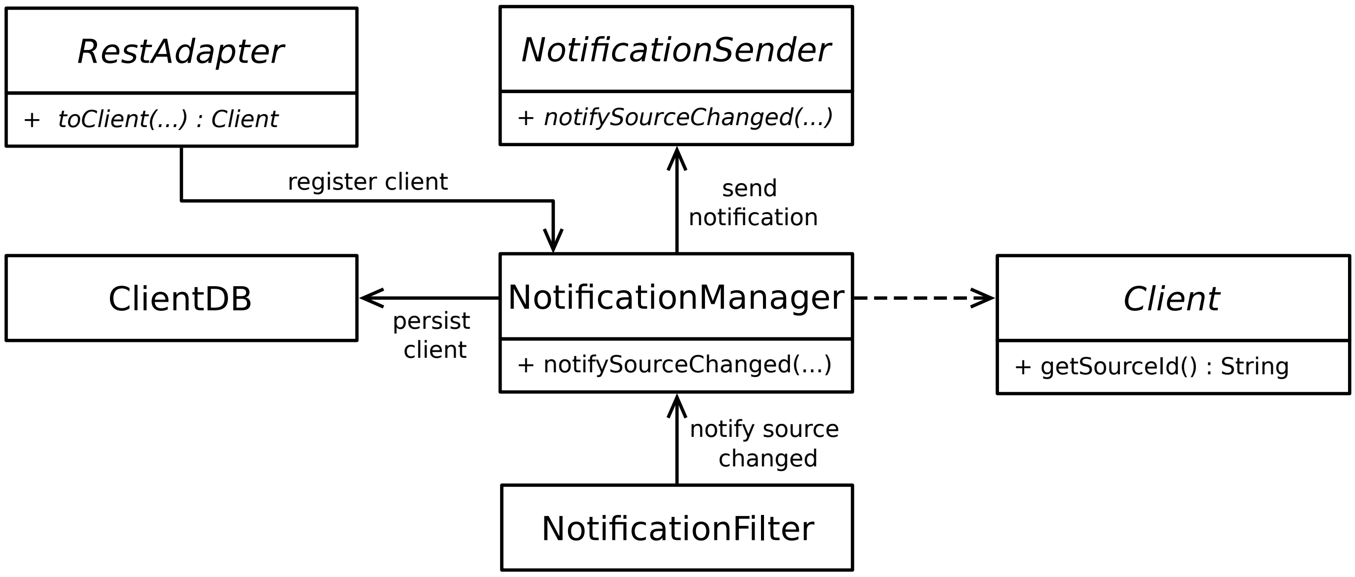


***Figure 3****: The ODS notification framework class diagram. Once the* `NotificationFilter` *receives new data from a filter chain, it forwards this data to the* `NotificationManager`*. If clients have previously registered themselves with the ODS via the* `RestAdapter`*, the* `NotificationManager` *retrieves this client data from the* `ClientDB` *and forwards the notification request to an instance of* `NotificationSender`*. As the ODS supports multiple notification mechanisms (GCM and HTTP callbacks), there is one* `NotificationSender`, `Client` *and* `RestAdapter` *implementation for each client type.*

Section 2.2.1 discusses a requirement for the CEPS to work with different client types, such as Android or iOS devices. This requirement was also applied to the ODS. In order to add a new client type, the following classes have to be extended:

- `Client`: contains all relevant information for contacting a single client application. In the case of GCM clients (`GcmClient`) they only relevant item is a GCM registration id which uniquely identifies a mobile device in the GCM framework[2].
- `NotificationSender`: the sender implementation which takes care of forwarding a notification to clients. The `GcmSender` for instance sends the id of the updated source along with the client registration id to the Google servers, which handle the actual delivery of the message to an Android device.
- `RestAdapter`: an adapter class used by the REST API for converting incoming registration

1 https://couchdb.apache.org/
2 https://developer.android.com/google/gcm/client.html

requests into client objects, for example into a `GcmClient` instance.

In its current form the ODS supports handling GCM as well as HTTP clients. An HTTP client can be any service which implements a REST API as defined by the `HttpSender` for receiving ODS source data. The `HttpSender` then forwards any new data as part of the HTTP request body to that client. The CEPS is an example for such an HTTP client.

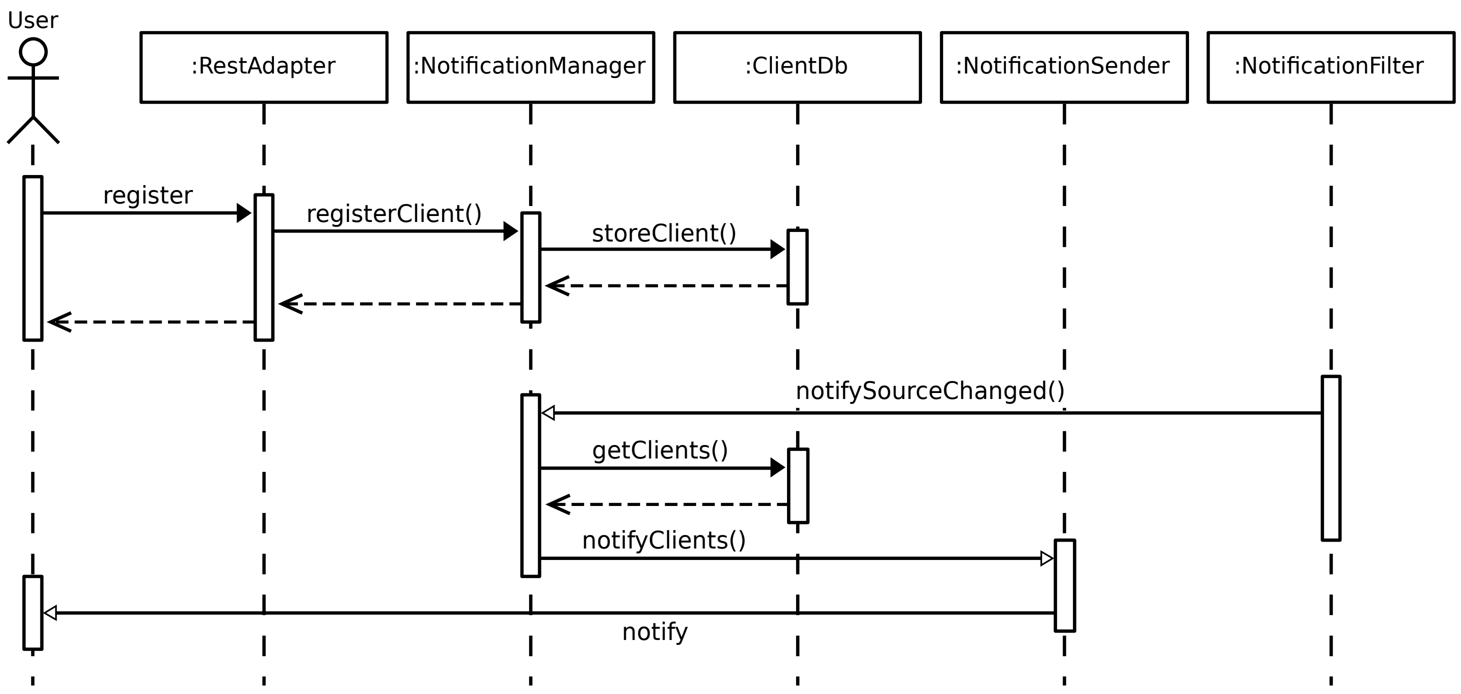


***Figure 4**: The ODS notification framework sequence diagram. The `NotificationManager` receives new client data from the `RestAdapter` and gets notified about new data from the `NotificationFilter`, which can be hooked into an ODS filter chain.*

### 2.3.3 The CEPS architecture

Before going over the CEPS architecture, some terms require clarification:

- *Client data*: in order to register a rule on the CEPS, a client must supply the actual CEP rule (an EPL statement in the case of Esper) as well as information required to contact a client machine (GCM id in the case of Android GCM). Together both pieces are referred to as the client data which is stored for each registration.
- *Event*: while the term event usually refers to data going into a CEP engine, such as individual sensor measurements, the CEPS also uses this term for data which is “leaving” the engine. E.g. when the EPL statement `select price from StockTick.win:time(10 sec)` is triggered, the Esper engine passes an array of the type `EventBean`[1], containing stock prices of the last 10 seconds, to any listeners.

1 http://esper.codehaus.org/esper-5.0.0/doc/api/com/espertech/esper/client/EventBean.html

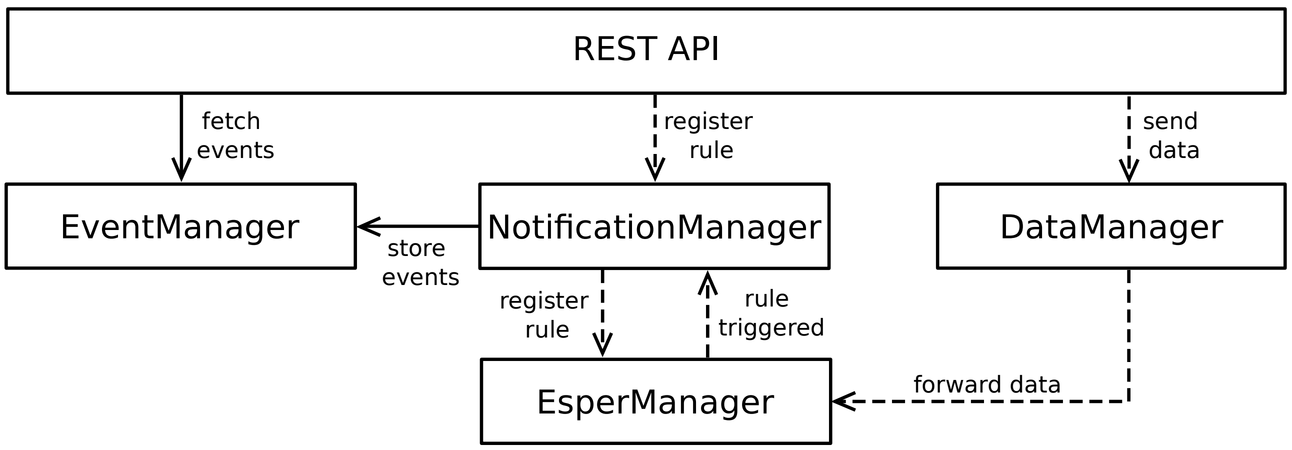


***Figure 5****: The CEPS architecture class diagram. A typical workflow starts with a client that wishes to be informed of certain conditions and that registers itself with the* `NotificationManager` *by using the REST API. This registration request is then converted into an Esper internal format (EPL statement) and forwarded to the* `EsperManager`*. Once a source has new data available the ODS sends this data to the* `DataManager`*, which is responsible for receiving, parsing and forwarding the data to the* `EsperManager`*. The* `EsperManager` *then, if the client's conditions are met, notifies the* `NotificationManager` *which actually sends out a notification to the client. In addition the* `EventManager` *will store the data which caused the notification in a local database for later retrieval by the client.*

When analyzing the architecture of the implemented CEPS, four primary classes can be identified that reside below a REST API layer (see figure 5 for a class diagram and figure 6 for a sequence diagram):

- `DataManager`: responsible for all communication with the ODS, including subscribing the CEPS as an HTTP client to sources offered by the ODS on first start and receiving, parsing and forwarding new data to the `EsperManager`.
- `EsperManager`: a thin wrapper around the Esper engine which is primarily designed for managing an instance of the engine, accepting new data and EPL statements and forwarding those to the engine. This also includes translating any data and schema information into the internal format used by the Esper engine. Lastly the `EsperManager` alerts listeners whenever a rules has fired, such as the `NotificationManager`.
- `NotificationManager`: its functionality is similar to that of the `NotificationManager` residing in the ODS, that is handling client registration and deregistration, storing clients in a local database and forwarding notifications received from the `EsperManager` to sender implementations. Additionally it passes any events to the `EventManager` in case a client is unable to receive event data directly via push notifications, such as Android GCM clients.
- `EventManager`: stores events in a local database for later retrieval, since getting this data can potentially require client applications to perform a lot of work.

- REST API: a collection of classes that are responsible for validating and parsing incoming requests and forwarding those to the underlying managers. The CEPS REST API also implements the REST API as defined by the `HttpSender` of the ODS such that the CEPS can act as a valid HTTP client for the ODS.

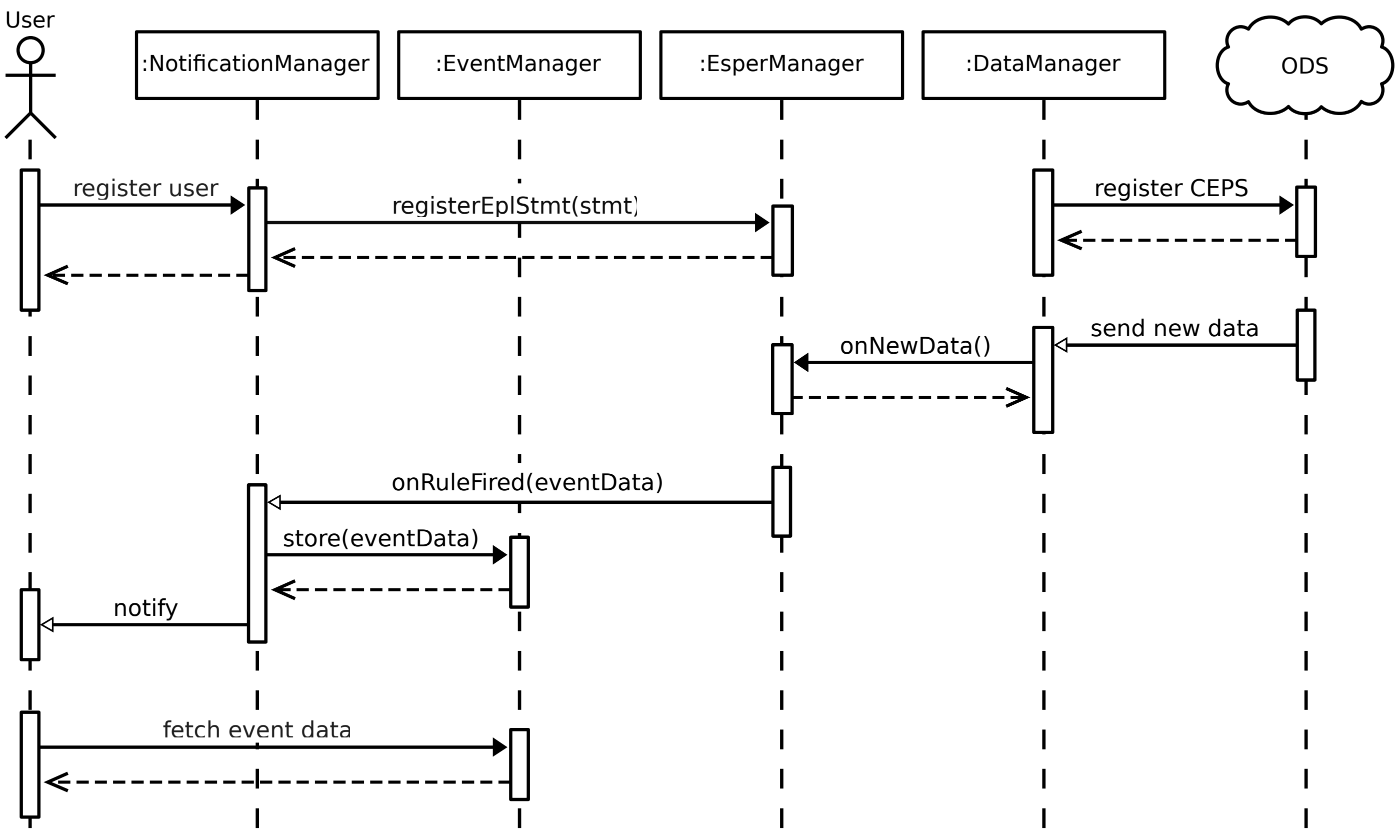


***Figure 6****: The CEPS architecture sequence diagram. In this workflow the user represents a client that is incapable of directly receiving event data via the notification that is sent from the* `NotificationManager` *(e.g. GCM clients). In that case the client has to the option to fetch this data manually from the* `EventManager`.

As a significant downside of the above design, the `NotificationManager` directly receives EPL statements from the REST API and forwards those to the `EsperManager`. While this allows maximum flexibility on the client side to construct arbitrary complex rules, it comes at a cost. Not only will client application code contain elements of the Esper event processing language (EPL), bringing up again the license discussion about whether client applications have to be released under the GNU General Public License, but it also introduces an inherent security risk to CEPS as a whole. The EPL is powerful enough to allow dynamic deletion, addition and mutation of events directly in the Esper engine, all from within an EPL statement[1], making it possible for clients to influence what events other clients can or cannot see. Lastly exposing the underlying CEP engine to the public makes swapping out the engine in future releases nearly impossible, without breaking backwards compatibility.

1 http://esper.codehaus.org/esper-4.2.0/doc/reference/en/html/epl_clauses.html

As a solution an additional layer of abstraction was introduced, the `EplAdapter`, which converts a set of parameters passed in from clients into a valid EPL statement. While this requires the configuration of specific adapters for each domain (such as the `PegelOnlineAdapter` for water level alarms), it frees the CEPS REST API of any Esper or CEP references. Figure 7 shows how this adapter was integrated into the REST API, here represented by the `RegisterRestlet` which extends `org.restlet.Restlet`, the chosen REST framework[1] for the CEPS. By injecting instances of the `EplAdapter` into the `RegisterRestlet`, the ability to add new client types was kept independent from the EPL mappings configured in the CEPS.

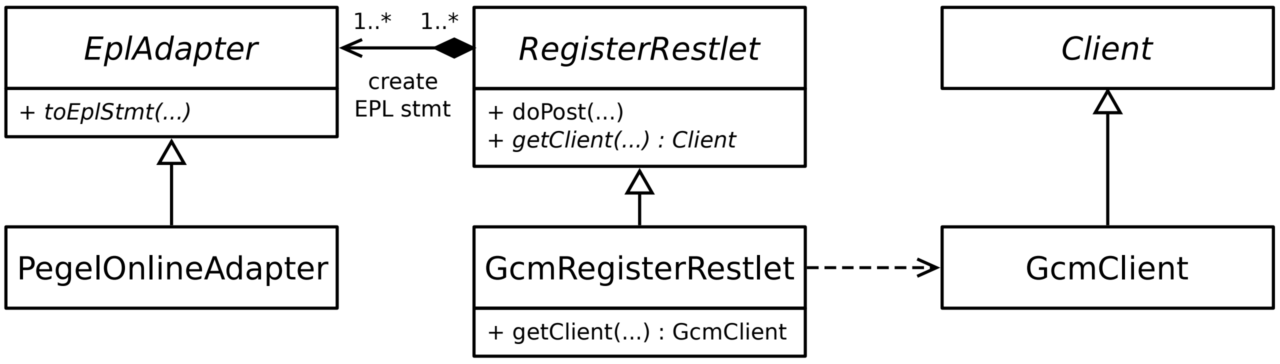


***Figure 7**: CEPS `EplAdapter` integration. For each domain specific rule one subclass of `EplAdapter` has to be implemented which translates parameters passed in from a client to an EPL statement (e.g. `PegelOnlineAdapter` in the case of the* Pegel Alarm *application). Each adapter can then be combined with an instance of `RegisterRestlet`, which determines what notification mechanism should be used (e.g. GCM or HTTP callbacks). Keeping the `EplAdapter` logic separate from the notification logic allows CEPS developers to reuse the same `RegisterRestlet` for different domain specific rules and vice versa.*

### 2.3.4 CEPS Garbage Collection

The last topic to be discussed on the CEPS side is the topic of garbage collection. Both the `EventManager` and `NotificationManager` require a database for storing event and client data respectively. And in both cases there is no guarantee that this data is ever disposed of when no longer needed, which makes explicit garbage collection necessary.

While the `EventManager` already allows clients to explicitly remove events they no longer need, there is always the possibility of ill behaving clients. The implemented garbage collection for events is fairly straight forward. It runs at regular intervals and discards any events which are older than a certain time interval `x`, assuming that `x` is large enough for clients to fetch any data that they require.

Removing inactive clients on the other hand is a little trickier, in the sense that an accidental removal of an active client has to be avoided at all costs, ruling out a simple age-based approach. Additionally there is no general algorithm for checking whether a client is still active or not, as the mechanisms for communicating with a `GcmClient` are highly different from those of an

1 http://restlet.com/

`HttpClient`. At the same time it is possible that Android applications are uninstalled, application data is removed (leaving rules registered on the server but not on the client) or clients simply cease to exist altogether (e.g. an `HttpClient` goes offline and does not come back). In an attempt to account for all the above, the implemented garbage collector for clients consists of one instance of `GargabeCollector` for each client type, much like the `NotificationSender`, leaving it up to the individual collector instances to determine the best way of checking whether a client is still active or not. For instance the `GcmGarbageCollecor` periodically sends a heart beat message to clients, relying on the GCM framework to notify the collector whether the client application is still installed on the target device.

## 2.3.5 The Android Client Library

Based on the fact that the ODS and CEPS are two separate applications, the Android client library was designed to reflect this setup by containing two independent components, one for interacting with the ODS and one for the CEPS.

### ODS library

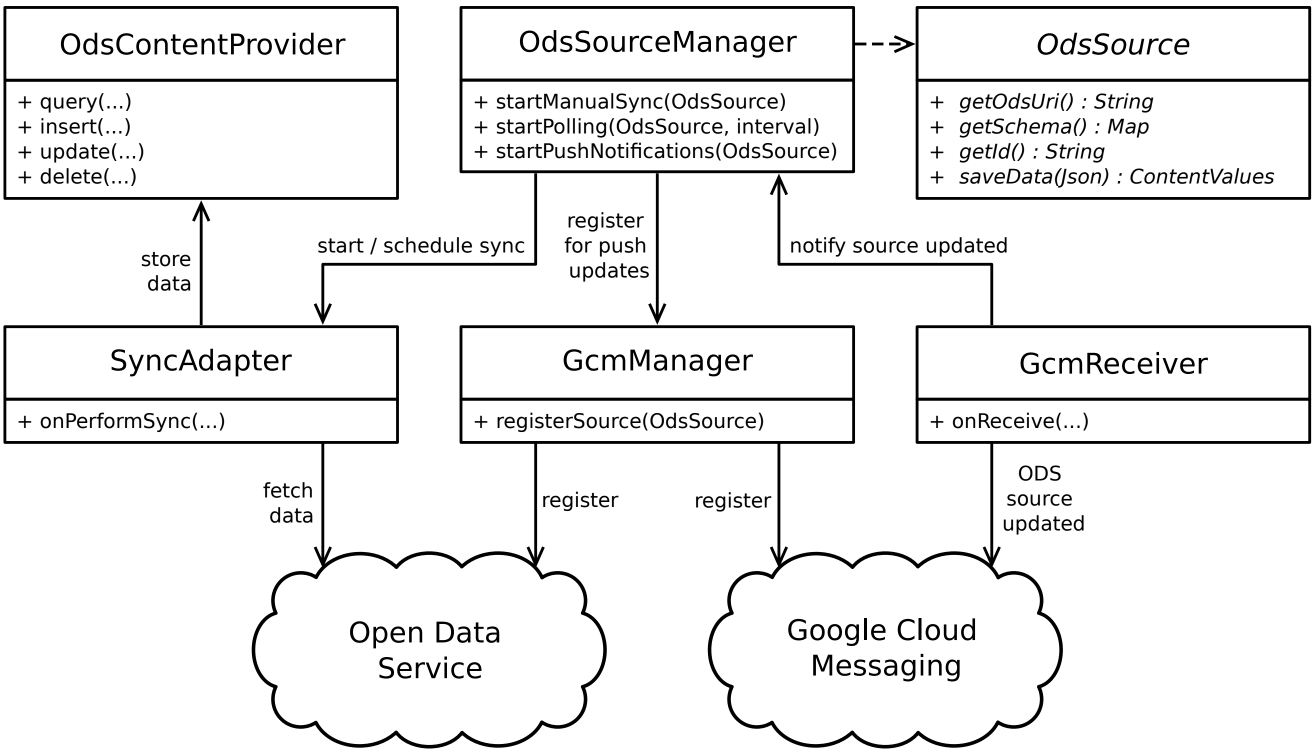


***Figure 8****: The ODS Android client library architecture. A typical workflow starts with a client application initiating a sync request via the* `OdsSourceManager` *by passing in an instance of* `OdsSource`*. This triggers the* `SyncAdapter` *which takes care of actually fetching and parsing data from the ODS and finally stores this data in a local database using the* `OdsContentProvider`*. Client applications can then fetch this new data through the content provider. In cases where clients would like to keep local content in sync with the ODS using push notifications, the* `GcmManager` *will register the application for push notification with the ODS and the GCM servers.*

Designing a REST client on Android in itself is not a difficult task. There are multiple options, ranging from manually running network requests in the background using a combination of `android.os.AsyncTask`[1] and `java.net.HttpURLConnection`[2] or by integrating one of the many networking libraries available such as Android Volley[3]. While offering a lot of fine control over which request gets executed when or, in the case of Volley, being able to run requests in parallel without having to come up with a custom implementation for this (Kirkpatrick, 2013), none of those solutions are tailored to the task that is most common for REST clients: keeping local data synchronized with data available on a server. The architecture chosen for this REST client follows a design laid out by Dobjanschi (2010). This solution leverages all the advantages of the Android Sync Adapter framework[4], allowing the operating system to bundle network requests across multiple applications and execute those whenever it sees fit to maximize battery life. As pointed out by Qian et al (2012) and contrary to intuition, repeatedly sending small chunks of data from a mobile devices has greater negative impact on the battery than sending the same amount of data in one request[5]. In that sense the Sync Adapter framework is an ideal solution for collaborating with other application developers on conserving as much battery as possible.

Figure 8 shows the components that form the foundation for the ODS Android client library:

- `OdsSource`: client applications using this library need to extend this class for every ODS source they wish to monitor. Besides providing basic information such as where a source resides on the ODS (`getOdsUrl`) or its schema (`getSchema`), base classes are also responsible for implementing the logic which determines what parts of an ODS data item should be stored in the local database (`saveData`). Leaving this decision up to the client application leads to less storage used on the Android device, since an application might only require a small number of parameters present in a data source. The water level alarm application for example only needs current water levels but not wind and temperature values.

- `OdsSourceManager`: after having provided a custom `OdsSource` implementation, client applications can use this manager to keep their source in sync with the local database. This can happen either in the form of a manual one time fetch (`startManualSync`), polling for changes on regular intervals (`startPolling`) or by automatically receiving updates from the ODS whenever new data is available (`startPushNotifications`). When to use which update mechanism largely depends on how often new data becomes available on the server and how up to date a client application wishes to be. Using push notifications to frequently trigger updates can actually require more battery than it does good, as the Google cloud services become an unnecessary intermediate system in the case of static or almost static update intervals (Burgstahler, Lampe, Richerzhagen, Steinmetz, 2013). Additionally client

1 https://developer.android.com/reference/android/os/AsyncTask.html
2 https://developer.android.com/reference/java/net/HttpURLConnection.html
3 https://developer.android.com/training/volley/index.html
4 https://developer.android.com/training/sync-adapters/index.html
5 Qian et al (2012) discovered one version of the Facebook application, where “periodic transfers account[ed] for only 1.7% of the overall traffic volume but contribute[d] to 30% of the total handset radio energy consumption”.

applications usually don't require such short update intervals over a long period of time. For example the PegelOnline source is publishing new data on average about every minute, which if constantly fetched, will quickly drain the battery in the process of doing so.

- `SyncAdapter`: responsible for doing the actual synchronization with the ODS and part of the Android sync adapter framework.
- `OdsContentProvider`: provides client applications access to the local ODS database which contains the data of sources that were registered with the `OdsSourceManager`. Only the most recent data is stored, meaning that running the `SyncAdapter` a second time will update old values.
- `GcmManager` and `GcmReceiver`: while the manager is responsible for handling GCM related details, the receiver is called by the GCM framework whenever a push notification has been received from the ODS. This message is then forwarded to the `OdsSourceManager` which triggers a sync ("send-to-sync" approach).

Figure 9 summarizes how the ODS library can be utilized by client applications.

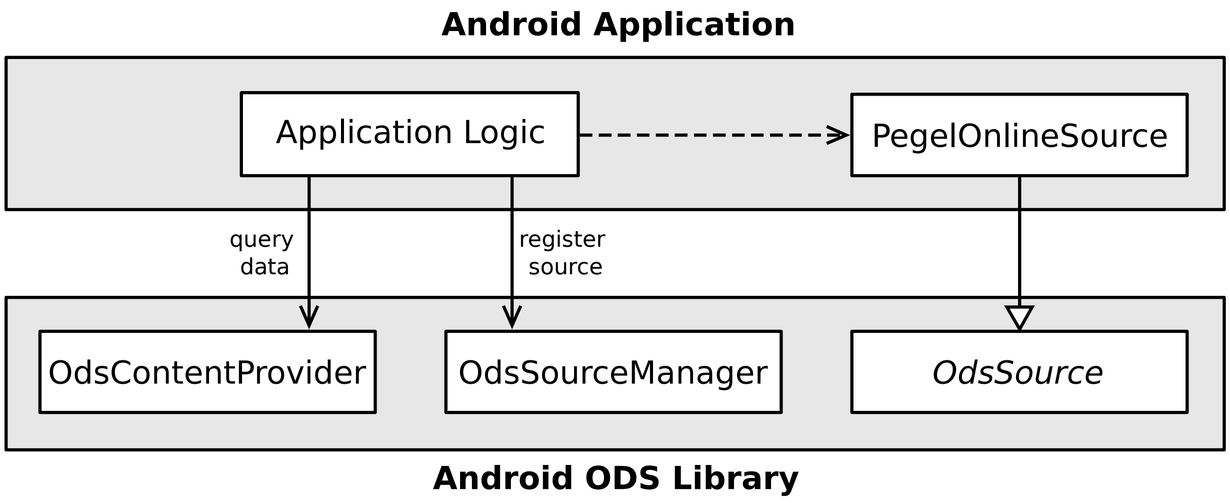


***Figure 9****: The ODS library usage. In order to work with the library the Android application needs to subclass* `OdsSource`*, e.g. like* `PegelOnlineSource` *which is being used by the* Pegel Alarm *application. The application can then use this source in combination with the* `OdsSourceManager` *to automate synchronization with the ODS, and the* `OdsContentProvider` *to query for local data which has been fetched from the ODS.*

### CEPS library

The part of the client Android library that is responsible for communicating with the CEPS consists of tree main classes that are discussed in the following paragraphs (see figure 10):

- `Rule`: a collection of parameters that, when forwarded to an `EplAdapter` on the CEPS, form the final EPL statement. For example in order to start an water level alarm on the CEPS, parameters included in a `Rule` are the name of a measurement station and a water level.

- `BaseEventReceiver`: the only class which has to be extended by client applications when using this library. Whenever a rule has been triggered on the CEPS, this receiver is invoked with the corresponding rule and an event id. The event id can be used by the client application to fetch the event data that was generated while firing this rule on the CEPS.
- `RuleManager`: handles the registration and deregistration of rules on the CEPS. As these actions are potentially long running operations, they, like all network operations, cannot be executed on the UI thread but are passed on to a worker thread instead. As such there needs to be some mechanism for the client application to determine the current status of a running operation (`getRegistrationStatus`), which it can use to display to users but also to schedule a retry in case of errors. The manager persists any registered rules in a local database along with their current status, which can either be `registered`, `unregistered`, `registration_pending`, `deregistration_pending`, `registration_error` or `deregistration_error`.

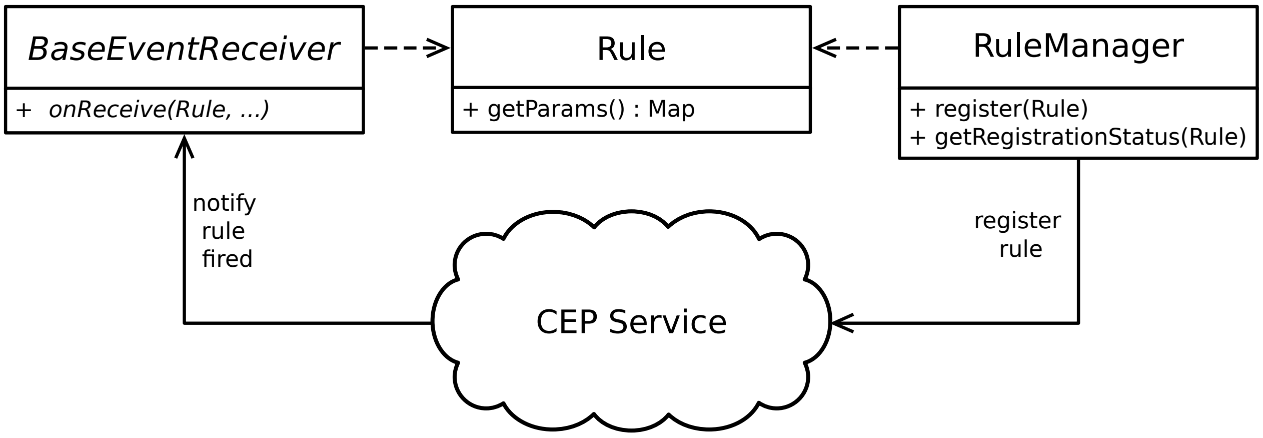


***Figure 10****: The CEPS library architecture. In order to register a rule on the CEPS, clients need to instantiate a* `Rule` *which contains parameters that will be used by an* `EplAdapter` *on the CEPS (e.g. river name and water level for the* Pegel Alarm *application). This rule is then passed to the* `RuleManager` *which does the actual communication with the CEPS. In order to receive notifications, client applications need to subclass and register and instance of* `BaseEventReceiver`*, which will be triggered by the library whenever a rule on the CEPS has fired.*

### 2.3.6 The Android Sample Application

The implemented sample Android application which makes use of all the above components is called *Pegel Alarm* (Water Level Alarm) and contains, roughly speaking, two features: the ability to schedule water level alarms that trigger whenever a river has risen above a certain threshold, and a collection of tools for viewing and analyzing current and past water levels across Germany. This section will cover the integration with the ODS and CEPS. See section 3.4 for details on the user interface.

To build the alarm feature, a custom `EplAdapter` was added to the CEPS (`PegelOnlineAdapter`) which takes the name of a measurement station, a river name and water level as parameters and returns an EPL statement that triggers every time two consecutive measurements were observed, where the first one is below the water level value, and the second one above (see also section 3.3). The only job left for the *Pegel Alarm* application is to gather those parameters from the user, put them into to a `Rule` object and pass it to the `RuleManager` for processing.

Viewing and analyzing water level data works by tapping into the PegelOnline source which has been defined on the ODS. On first startup, *Pegel Alarm* triggers a one time manual sync with the `OdsSourceManager`, such that users can immediately start browsing the individual measurement stations. Additionally a monitor functionality was build on top of the ODS library, which, if activated, schedules a periodic sync with the `OdsSourceManager` and copies the ODS database after each sync, allowing users explore water level trends over time. Fetching data about individual stations, for example when a user is viewing details about a single station, is not done through the library, but rather by issuing a regular REST request, as it is unnecessary to synchronize all data contained within a source in this case.

## 2.4 Limitations and Future Work

### 2.4.1 Limitations

From a technical perspective, the fact that the CEPS lacks the concept of users puts an additional burden on client application developers. As there is currently no support for fetching or updating all rules belonging to one user, persisting this information is delegated to client applications, in this case to the CEPS Android library. In many case, like the *Pegel Alarm* application, this limitation does not actually reduce the capabilities of the CEPS or what clients can do with it, but makes them more difficult to implement. However, building multiple client applications with run on different platforms (e.g. Android, iOS, web), but all want to share the same data belonging to one user (e.g. one user owns two phones), is nearly impossible without specifically handling synchronization between these devices on the client side.

Secondly the CEPS in its current design can only be extended to support additional CEP rules by creating a subclass of `EplAdapter`, which requires a Java developer with an event processing background to make this change. Chandy and Schulte however identified that "business users must be able to tailor event-processing applications to meet their changing needs" (2010, p. 58), something that the CEPS clearly does not support. While the framework built around the `EplAdapter` is already a first step towards an easily configurable system, it is limited in a sense that end users can only change rule parameters but not create completely new rules.

### 2.4.2 Future Work

Additional work on the CEPS may address the challenge of creating a generic interface for accepting CEP rules, which would eliminate the need to apply changes to the server for each new client application.

On the client side the *Pegel Alarm* application could be extended to support more complex alarms by letting users choose from a wider range of parameters to monitor (e.g. wind strength, discharge of a river or outside temperature) or by combining multiple alarms using simple boolean algebra. Additionally a web client could be developed to target a wider audience.

## 2.5 Related Work

Gusev, Gushev, Ristov and Velkoski (2014) proposed a system which offers similar functionalities as the CEPS but targets users wishing to receive updates about website they frequently visit. Their Alert Notification as a Service (ANS) allows users to specify keywords or patterns that should be matched on a particular site and how they would like to be notified about these events, be it email, Skype, Twitter or via other social media channels. While the ODS and CEPS are targeting developers rather than end-consumers, the main difference to the ANS is the integration of a full fledged CEP engine into the CEPS, allowing the definition of arbitrary complex rules.

MetaXA (Metadata-processing XMPP-based architecture) on the other hand includes such a CEP component on the service side (Astrova, Kleiner, Koschel and Nitz, 2013), which is for example capable of detecting whether a mobile device has been stolen based on sensor data provided by the mobile device (e.g. camera, GPS, etc.). The overall architecture is similar to that of the CEPS, as both include an external service and a publish-subscribe pattern, with the exception that MetaXA relies on clients using an extended version of XMPP, where as the ODS and CEPS are capable of communicating with clients using different protocols (e.g. GCM, HTTP callbacks, …).

On the client side numerous applications have been published in the Google Play Store that offer similar functionalities as *Pegel Alarm*. Most notably they are Pegel-Online[1], RiverApp[2], Pegelstände Passau[3] (Water Levels Passau), Hochwasser 2013[4] (Flooding 2013), KremsAlarm[5] and Hochwasser App Lahn[6] (Flooding App Lahn). The first four can be broadly classified as human interfaces to the data from PegelOnline or other water portals. They all offer some mechanism for selecting rivers and measuring stations, inspecting water levels and other parameters and in some cases even viewing hydrographs that can be directly downloaded from PegelOnline. None however support any form of water alarms. KremsAlarm on the other hand allows users to define an alarm for one specific station (Kremsmünster, Austria which is located next to the river Krems) by fetching the latest water data at regular intervals. The Hochwasser App Lahm does something similar by informing the user whenever an official flood warning has been issued for the state of Hesse. As such, *Pegel Alarm* with its ability to define alarms for arbitrary rivers and stations paired with a battery efficient framework for supporting those rules, seems to be unique in the Android market.

1 https://play.google.com/store/apps/details?id=org.cirrus.mobi.pegel
2 https://play.google.com/store/apps/details?id=de.android.riverapp
3 https://play.google.com/store/apps/details?id=de.uni_passau.fim.eislab.pub.pegelstandderdreifluesse
4 https://play.google.com/store/apps/details?id=de.se.hochwasser
5 https://play.google.com/store/apps/details?id=net.sauft.gasi.kremsalarm
6 https://play.google.com/store/apps/details?id=de.hwlz.app

## 2.6 Conclusion

In this thesis we designed, implemented and evaluated a notification service which allows clients to be informed about arbitrary complex events, that are processed and evaluated by an external service rather than directly on the client side. The notification service uses the Open Data Service under development at the professorship for tapping into multiple open data sources. Additionally an Android client library for communicating with these two services has been designed, that serves as a battery efficient framework for keeping remote data on the ODS in sync with a local device and as a client implementation of the notification service. Finally the Android application *Pegel Alarm* was built to evaluate these two services and how well they interact with the Android client library. The application allows users in Germany to define thresholds for water levels, which when crossed, trigger a notification, warning the user that the current water level is now above that threshold.

# 3 Elaboration of Research Chapter

This chapter serves as an extension which covers some of the design decision and work results more depth. Section 3.1 picks up on the license discussions of 2.2.3 by taking a closer look at the GNU GPL and applying it to the ODS. Section 3.2 provides an overview of which water level providers were contacted regarding having their data integrated with the ODS and their respective replies. Sections 3.3 and 3.4 focuses on the sample Android application (see 2.3.6), by firstly showing how client requests are translated into valid EPL statements inside the notification service, and secondly by briefly presenting the android application user interface.

## 3.1 Integrating a GNU GPL Component into a non GPL Project

This section tries to shed some light onto the question, whether the ODS and Esper can be combined in one software project without violating any of the respective license terms. EsperTech has published its core product, the CEP engine, under two licenses[1]: the GNU GPL v2[2] and a commercial license which can be purchased directly from EsperTech. The ODS is being distributed under the GNU AGPL v3[3], which goes beyond the viral copy-left effects of the GPL by also requiring developers to publish enhancements they have made to the ODS in cases where it is solely operated over the web. The following discussion does not only focus on combining GPL with AGPL software, but rather analyses how or whether arbitrary, including proprietary, licenses can be integrated with a GPL component by using Esper as an example.

As this discussion essentially evolves around the question when a project that uses a GPL component is covered by the viral license effects and when not, it is worthing taking a closer look at what exactly causes this viral effect. Jaeger and Metzger (2011) point out that the copy-left effect stems from the following paragraph of the GPL (section 2b):

> *„You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License."*

In short, any program which can be considered "derived" of some GPL element has to be published under the GPL as well. Unfortunately the GPL does itself not define what exactly constitutes a derived program, nor is there any general widely accepted definition for the term (Jaeger, Metzger, 2011).

One way of interpreting "derived", is by looking at the technical details of how two programs were combined to form a new product. The GPL FAQ published by the Free Software Foundation does just that, by stating that whenever a GPL component is statically or dynamically linked, the result must be subject to the GPL. On the other hand, "pipes, sockets and command-line arguments

1 http://www.espertech.com/download/public/EsperTech%20licensing.pdf
2 https://www.gnu.org/licenses/gpl-2.0.html
3 https://www.gnu.org/licenses/agpl-3.0.html

are communication mechanisms normally used between two separate programs"[1]. What sounds like a clear cut definition at first, is suffering from 2 mayor flaws:

- Firstly, the term "normally" leaves room for interpretation.
- Secondly, the FAQ, unlike the license itself, are not a legally binding document but merely the interpretation of the Free Software Foundation that has no affect on the agreement (the GPL) made between a developer and a user (Katz, 2007, p. 14).

In general defining the term "derived program" by only relying on the type of interfaces used without considering the content transmitted is bound to fail, as any GPL component could be easily converted into a non GPL version by simply adding a network layer of some sorts, which is made possible due to the open source nature of these projects (Jaeger, 2008).

Another way of going about interpreting the copy-left clause, is by looking at the copyright laws of the prospective countries. Rosen (2001) uses this approach and arrives at the conclusion, that "if the licensee doesn't modify the original GPL-licensed program, but merely runs it, he is not creating a derivative work", which is partially based on the fact that the GPL only uses the phrase "contains or is derived from", but not "combination". Hence merely combining components does not create a derivative work according to this explanation. However Rosen finishes her article with a word of caution, warning readers that the interpretation of the copyright law varies from country to country and that it always takes an knowledgeable attorney to review each case.

Lastly one could ask how other large software projects have dealt with this issue. After all, the GPL v2 has been released in June 1991 and numerous popular projects have been, and still are, using this license, such as the Linux Kernel[2]. One such study carried out by German and Hassan (2009) has identified a number of common patterns used in the software industry to circumvent this problem. The results divide the patterns into those which can be applied by a licensor or licensee to make a particular linkage possible. On the licensee side those patterns are most notably either explicitly asking for an exception or permission, requiring end users to build the derived program themselves by providing them with a patch, finding an alternate component with a different license or making sure that the new program is considered a collective and not a derived program. When comparing those options with the Esper ODS integration problem, only the last pattern is applicable as asking for permission were bound to fail should the ODS ever be published under a commercial license, and finding an alternative CEP engine would be besides the point of this discussion.

Having analyzed different interpretations of the GPL, the following discusses their consequences for Esper and the ODS. While the notification service does not require the CEP engine to be modified and therefore including the Esper library in the ODS should be fine from a GPL point of view according to Rosen, there will always be a remaining risk, especially since this integration was likely not intended by the developers at EsperTech. The solution we found realizes the notification service as a separate server program which receives data from the ODS via its regular REST API. This way the ODS no longer depends on the notification service and therefore

1 https://www.gnu.org/licenses/gpl-faq.html#MereAggregation
2 https://www.kernel.org/

also not on Esper, but rather the other way around. Additionally this setup leads to a true separation of concerns, allowing the components to scale according to their needs without having to affect the respectively other service.

Unfortunately such a solution is specific to the case of the ODS and notification service and can hardly be considered general advice on whether and how to integrate GPL components in a non GPL landscape.

## 3.2 Consolidating Water Level Portals in the ODS

Historically offering hydrological data to the public, be it flooding or quality related, has been the responsibility of the individual German states[1] (§7 and §79, Wasserhaushaltsgesetz / water economy regulations, 2013), with the exception of those rivers used by the shipping industry, which are managed on a national level[2]. Due to the lack of a central organization, a total of 14 distinct online portals[3] have emerged that are all using different approaches for publishing data.

| State | Permission to use data with the ODS | Reasons |
|---|---|---|
| Baden-Württemberg | No | State should be sole source of information |
| Bavaria | No | None given |
| Brandenburg | Yes | - |
| Hesse | No | State should be sole source of information |
| Lower Saxony | No | None given |
| Mecklenburg-Vorpommern | Yes | - |
| North Rhine-Westphalia | No | Only in exchange for a fee |
| Rhineland-Palatinate | No | State should be sole source of information |
| Saarland | No | None given |
| Sachen-Anhalt | No | State should be sole source of information |
| Schleswig-Holstein | - | No reply |
| Thuringia | No | State should be sole source of information |

***Table 1**: State permissions to integrate water level portals with the ODS.*

Seeing that the ODS was designed for dealing with heterogeneous data sources, consolidating all German water portals into one seemed like a perfect task for the ODS. Ultimately however this enterprise was abandoned due to the missing permission of too many portal providers. Out of 14 states that are publishing data online, 12 were contacted regarding their terms of use and whether they would approve of having the ODS fetch data from their sites. Eleven states replied, out of which only two agreed and all other nine refused to have their data redistributed through the ODS. Table 1 shows the outcome of these inquiries. States not contacted were Saxony and Hamburg.

1 More specifically the responsibilities are organized around so called water basin units, of which there are 10 in Germany. They are independent of state and national borders, requiring states to cooperate whenever they are within such an area that covers multiple states (Wasserhaushaltsgesetz / water economy regulations, 2013).

2 http://www.wsv.de/Wir_ueber_uns/index.html

3 A collection of those portals can be found at http://hochwasserzentralen.de/. Bremen links to the official site of PegelOnline and Berlin to the portal of Brandenburg, leaving 14 distinct water level providers.

## 3.3 Water Level Alarm EPL Statement

The `EplAdapter` used for creating water level alarms (`PegelOnlineAdapter`) takes three parameters called `STATION`, `RIVER` and `LEVEL` and converts those to an EPL instruction as shown below:

```
select station1, station2 from pattern
    [
        every station1=`de-pegelonline`
            (
                longname = 'STATION'
                and BodyOfWater.longname = 'RIVER'
                and timeseries.firstof(i => i.shortname = 'W') is not null
            )
        ->
        station2=`de-pegelonline`
            (
                longname = 'STATION'
                and BodyOfWater.longname = 'RIVER'
                and timeseries.firstof(i => i.shortname = 'W') is not null
            )
    ]
    where
        station1.timeseries.firstof(i => i.shortname = 'W'
                  and i.currentMeasurement.value <= LEVEL) is not null
        and
        station2.timeseries.firstof(i => i.shortname = 'W'
                  and i.currentMeasurement.value > LEVEL) is not null
```

The following paragraphs analyze the above rule by starting with the basic format for EPL statements in Esper:

```
select <selection> from <event source> where <filter>
```

The event source in this case is a so called “pattern”, which in Esper is mapped to any two consecutive events (here stations including their recorded measurements for one particular timestamp) that match a list of conditions specified in the parentheses that follow. The filter after the `where` keyword can be used to further narrow down which events should be chosen. In the case of water alarms only stations are kept which

- contain a water measurement (`timeseries[i].shortname = 'W'`), as PegelOnline stations can monitor multiple parameters at the same time, such as wind, temperature and water levels.
- have a water level above or below the user defined value, depending on whether looking at `station1` or `station2` (`timeseries[i].currentMeasurement.value >= LEVEL`).

Only when all of these conditions are met will `station1` and `station2` contain two measurements which indicate that a water level has risen above a certain value. Those two stations are then returned from the Esper engine and made available to clients in the `EventManager`.

## 3.4 *Pegel Alarm* User Interface

The *Pegel Alarm* user interface is structured around three main components: news, alarms and water levels. When first starting the application the user is presented with the main menu in the form of a navigation drawer[1] which shows those three sections (see figure 11). The following paragraphs give a brief explanation of each.

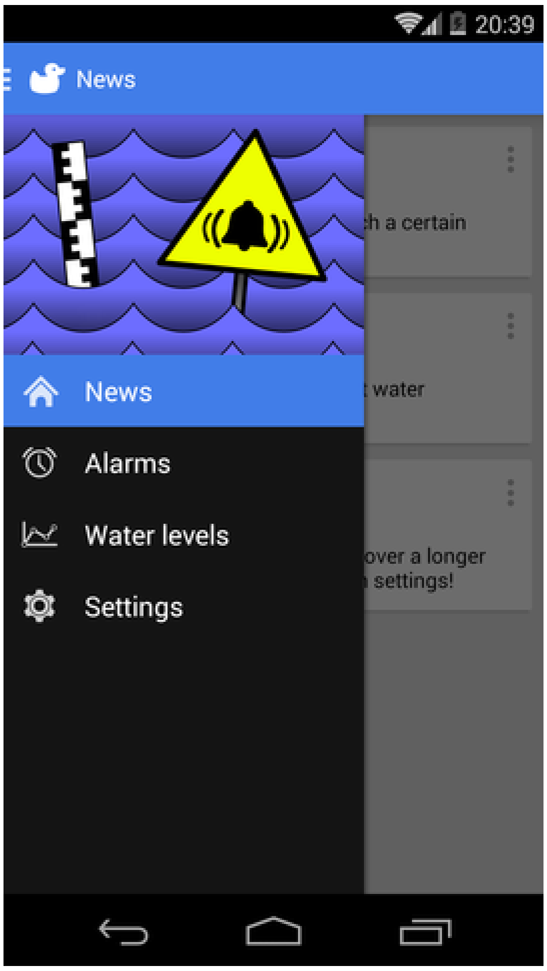


***Figure 11***: *The main menu showing the three primary sections (excluding settings): news, alarms and water levels.*

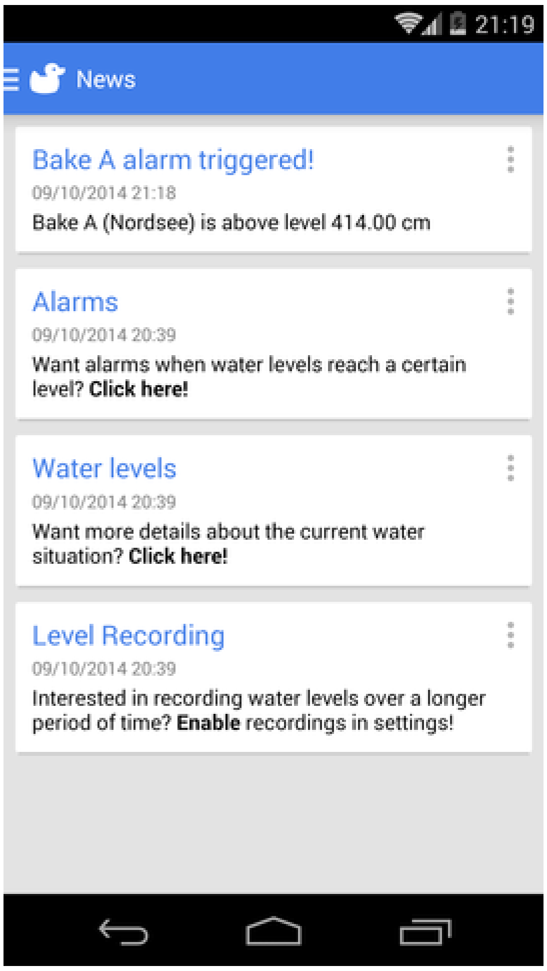
### News

This section is selected by default when opening the application and gives a quick overview of past important events, hence the name "News". Each item is displayed in the form of a swipeable card, similar to the Google Now application[2]. *Pegel Alarm* comes with a set of three news items already present which act as a quick tutorial for new users by helping them navigate to other sections when clicked. Each time an alarm has been triggered, an additional news item will be added to this list, telling the user about which alarm has fired and why. To better inform users about important events, each news item can be configured to cause an Android notification when being

1 https://developer.android.com/design/patterns/navigation-drawer.html
2 https://www.google.com/landing/now/

created, eliminating the need to regularly check back to this site to see what has happened.

Future work on this application might included additional sources to display in the news section, such as official warnings released by government offices (e.g. provided by the German central flooding organization[1]), unusual or critical weather conditions or information about collaboration efforts in times of flooding.


21:19

News

Bake A alarm triggered!
09/10/2014 21:18
Bake A (Nordsee) is above level 414.00 cm

Alarms
09/10/2014 20:39
Want alarms when water levels reach a certain level? **Click here!**

Water levels
09/10/2014 20:39
Want more details about the current water situation? **Click here!**

Level Recording
09/10/2014 20:39
Interested in recording water levels over a longer period of time? **Enable** recordings in settings!

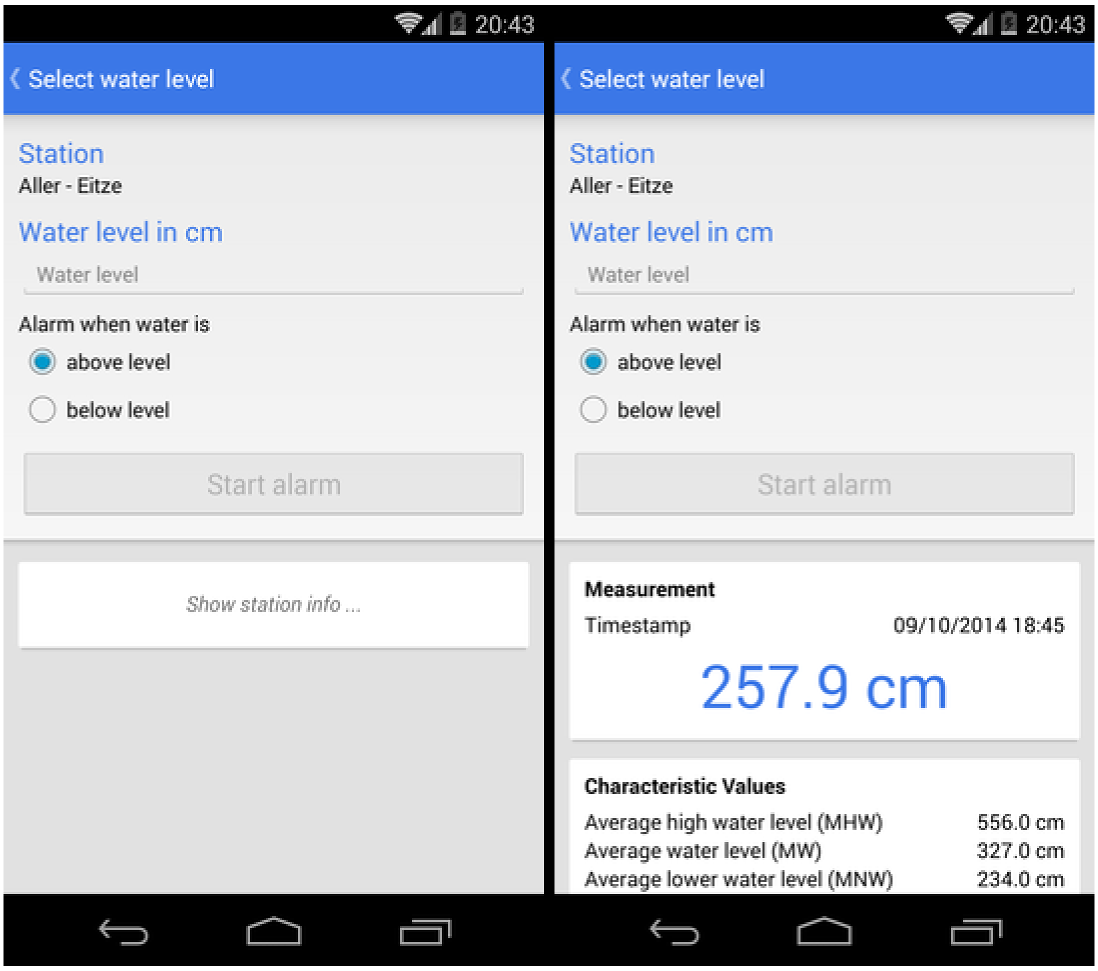
***Figure 12****: The news section. The three bottom cards show a quick tutorial for new users. The top card indicates that an alarm has been triggered.*

## Alarms

Here users can get an overview of all started alarms. New ones can be added by first selecting a river and station, either from a list or from a map (see figure 17), before entering the water level threshold. An additional option allows users to choose, whether the alarm should be triggered whenever the water level has risen above or below the provided threshold. In cases where users are unsure about what water level to enter or would like to review past values, the alarm creation screen features a button at the bottom which will display all current data about the selected station right on the same screen. This spares users from having to navigate back and forth between different parts of the application (see figure 13). Future work might include a graphical selection tool (e.g. by dragging the threshold up and down along a measuring scale), which would make picking a water

1 For example http://hochwasserzentralen.de/

level quicker by skipping the process of manually entering a number.



**Measurement**

Timestamp 09/10/2014 18:45

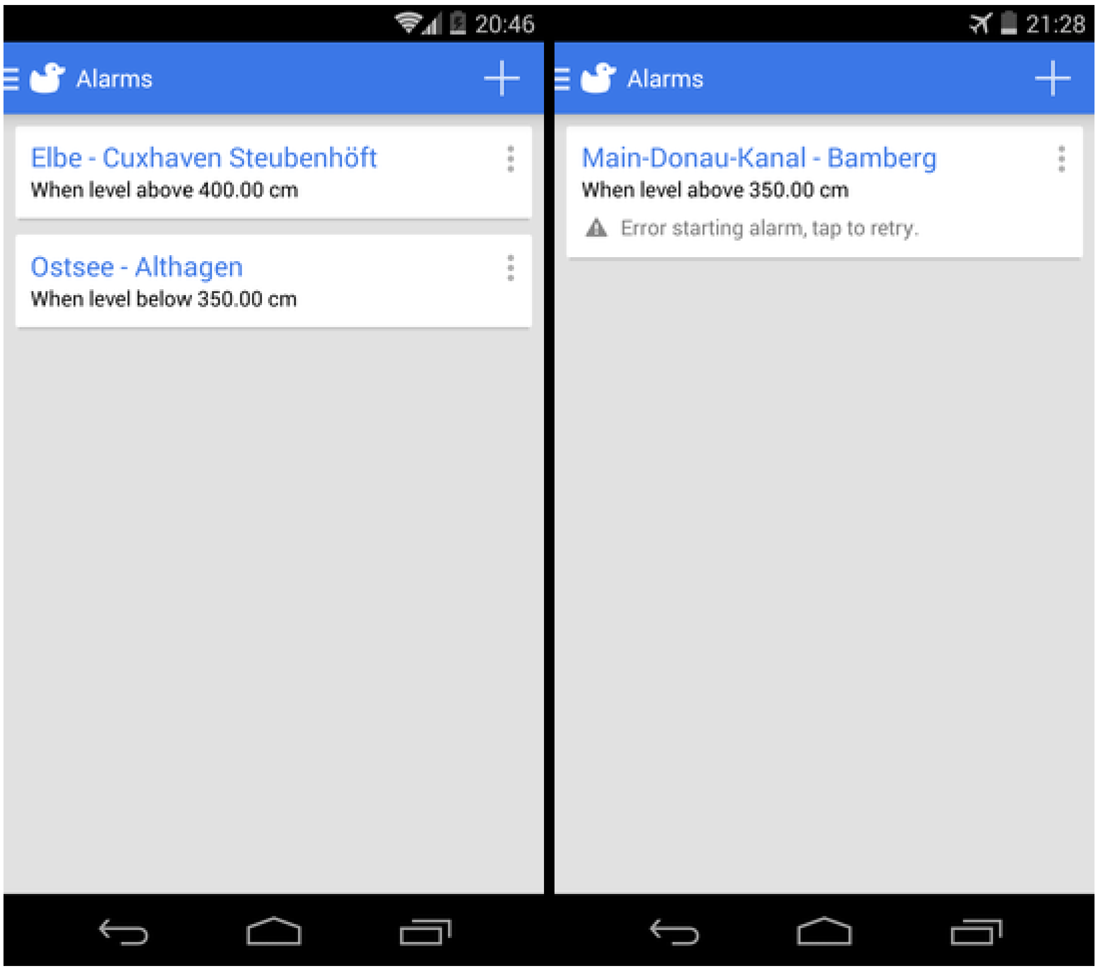
257.9 cm

**Characteristic Values**

| | |
|---|---|
| Average high water level (MHW) | 556.0 cm |
| Average water level (MW) | 327.0 cm |
| Average lower water level (MNW) | 234.0 cm |

***Figure 13***: *The screen for starting a new water level alarm. To avoid confusion, detailed data about the selected station is hidden by default.*

Since starting a new alarm requires the application to communicate with the CEPS server, the processes of starting an alarm can fail, for example due to a missing network connection. When this happens, the alarms overview screen will display a small error message, prompting the user to retry the registration (see figure 14).



Elbe - Cuxhaven Steubenhöft
When level above 400.00 cm

Ostsee - Althagen
When level below 350.00 cm

Main-Donau-Kanal - Bamberg
When level above 350.00 cm
Error starting alarm, tap to retry.

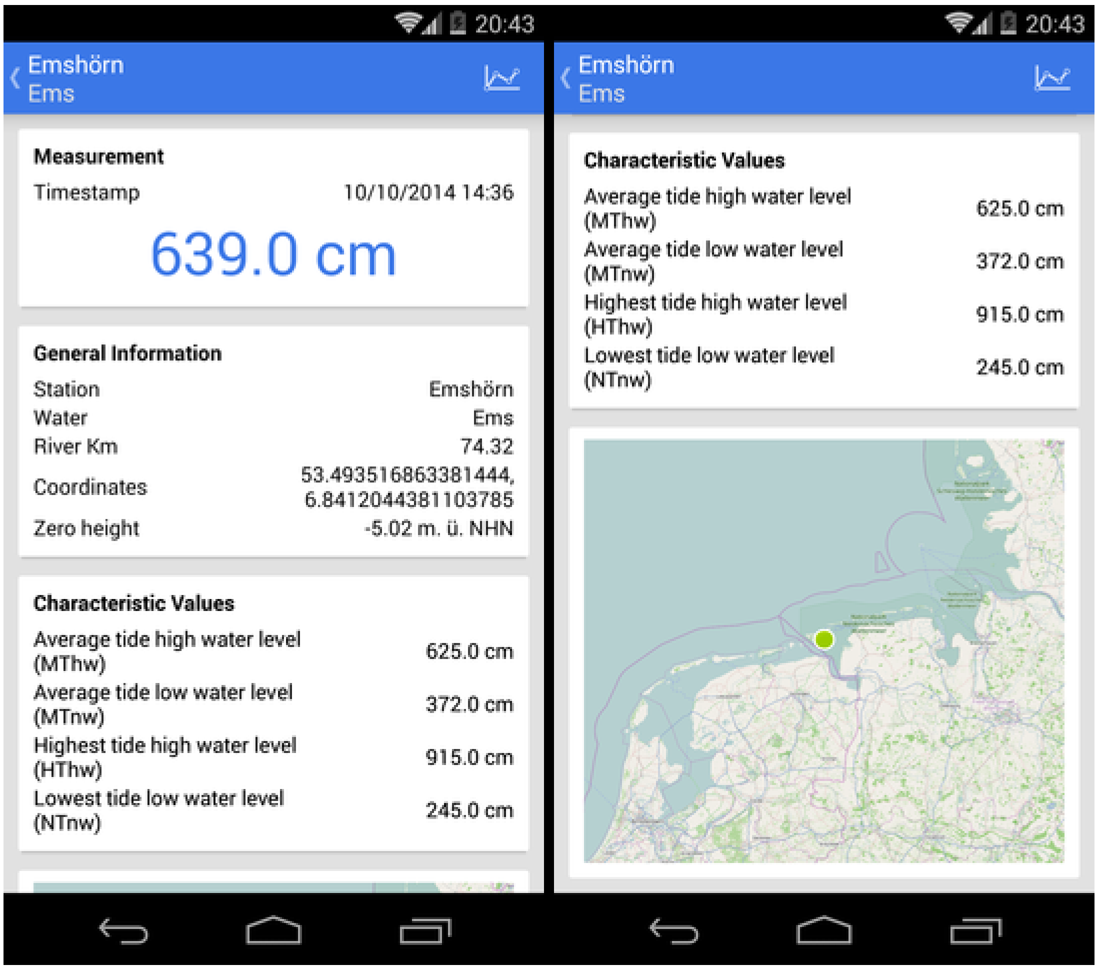
***Figure 14****: The alarms overview screen. The alarm on the right side experienced an error during registration (no network connection due to plane mode), but will retry when clicking it.*

### Water levels

This part of the application allows users to explore and analyze current and past water levels. They can choose between viewing information about a single station, or alternatively evaluating all water levels along an entire river. When navigating to a single station, the data displayed includes the latest water level at that station, various meta data about the station itself such as location or river km and characteristic values, which usually include average high and low water levels measured over several years (see figure 15). In order to maximize consistency within the application, the alarm creation screen reuses this layout for displaying information about a station.

The ability to evaluate all water levels along an entire river is targeted at users that wish to make some form of prediction about the future development of a flood. The amount of water relative to the current location along the river is a good indication, whether the tide wave has already passed or not. In oder to support this decision making progress, *Pegel Alarm* offers a river graph (see figure 16), showing the river km along the x axis and the water level along the y axis, as well as two tools for analyzing this graph.

The first tool addresses the problem, that while the graph might be visually appealing, it is difficult to draw any conclusions from it as every station defines their own zero height[1]. Consequently directly comparing water levels from different stations is hardly sensible without looking at their respective zero value. *Pegel Alarm* solves this issue by offering a so called "normalized" mode, which treats the distance between the average lowest and average highest water level of each station as 100% and displays the current water level relative to this distance. As a result users can now easily compare individual stations with one another and make predictions about how much water is coming from up higher along a river.


20:43

Emshörn
Ems

Measurement

| | |
|---|---|
| Timestamp | 10/10/2014 14:36 |

639.0 cm

General Information

| | |
|---|---|
| Station | Emshörn |
| Water | Ems |
| River Km | 74.32 |
| Coordinates | 53.493516863381444, 6.8412044381103785 |
| Zero height | -5.02 m. ü. NHN |

Characteristic Values

| | |
|---|---|
| Average tide high water level (MThw) | 625.0 cm |
| Average tide low water level (MTnw) | 372.0 cm |
| Highest tide high water level (HThw) | 915.0 cm |
| Lowest tide low water level (NTnw) | 245.0 cm |

20:43

Emshörn
Ems

Characteristic Values

| | |
|---|---|
| Average tide high water level (MThw) | 625.0 cm |
| Average tide low water level (MTnw) | 372.0 cm |
| Highest tide high water level (HThw) | 915.0 cm |
| Lowest tide low water level (NTnw) | 245.0 cm |

***Figure 15**: Information about a single station, here Emshörn which is part of the Ems river.*

The second feature deals with recording water levels to analyze trends over a longer period of time. As the ODS current only supports fetching the most current data, *Pegel Alarm* allows users to start a local recording mechanism, which will periodically fetch data from the ODS and store it in a local database. As such a feature can easily drain the battery if not configured carefully, the default recording settings will trigger a sync with the ODS every 30 minutes and only if the device has a Wi-Fi connection. By default the recording feature is turned off completely and has to be started by user manually. Once multiple recordings have been made, a user can selected a timestamp to display from within the river graph, and through that explore water levels over time in an interactive manner.

1 Water levels are always measured relative to this zero height. Unfortunately, due to historic reasons, each station defines their own arbitrary zero height that is measured in meters above sea level.

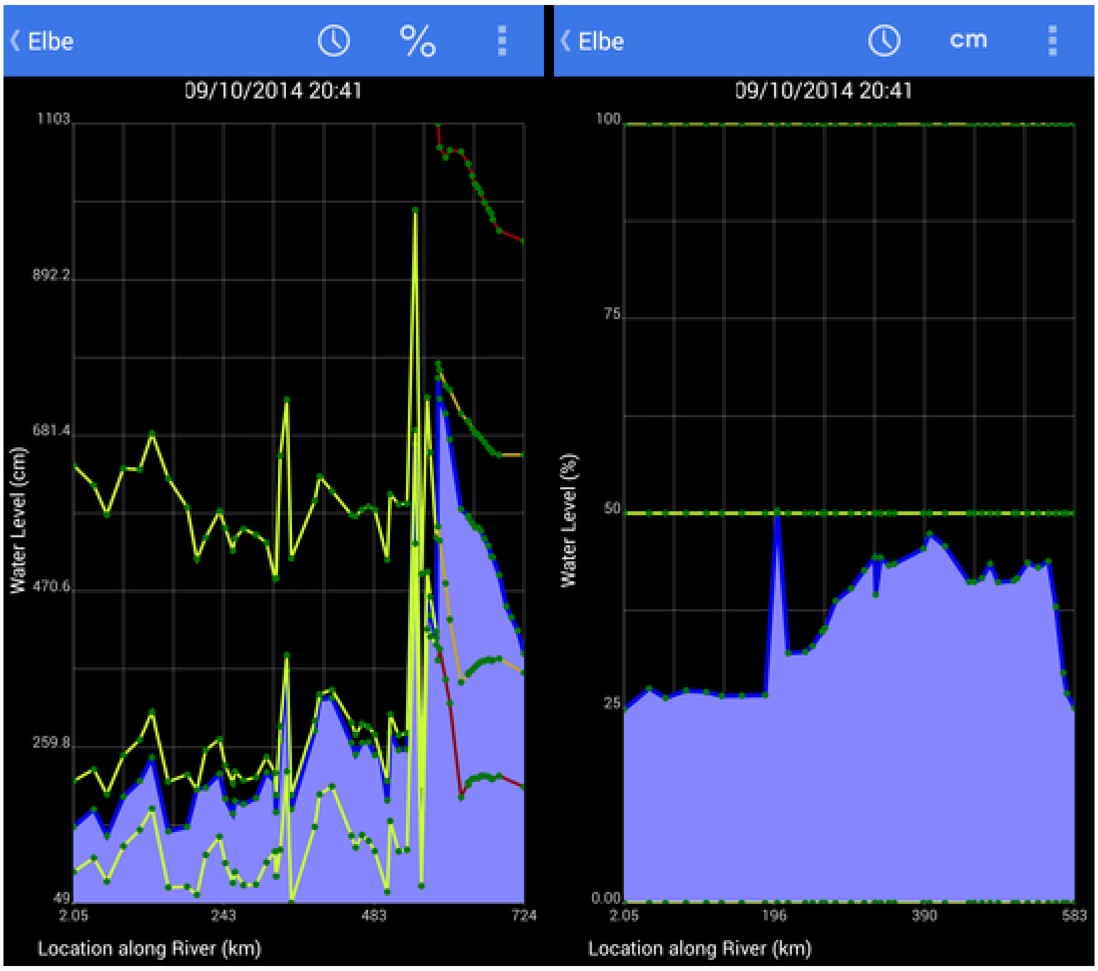
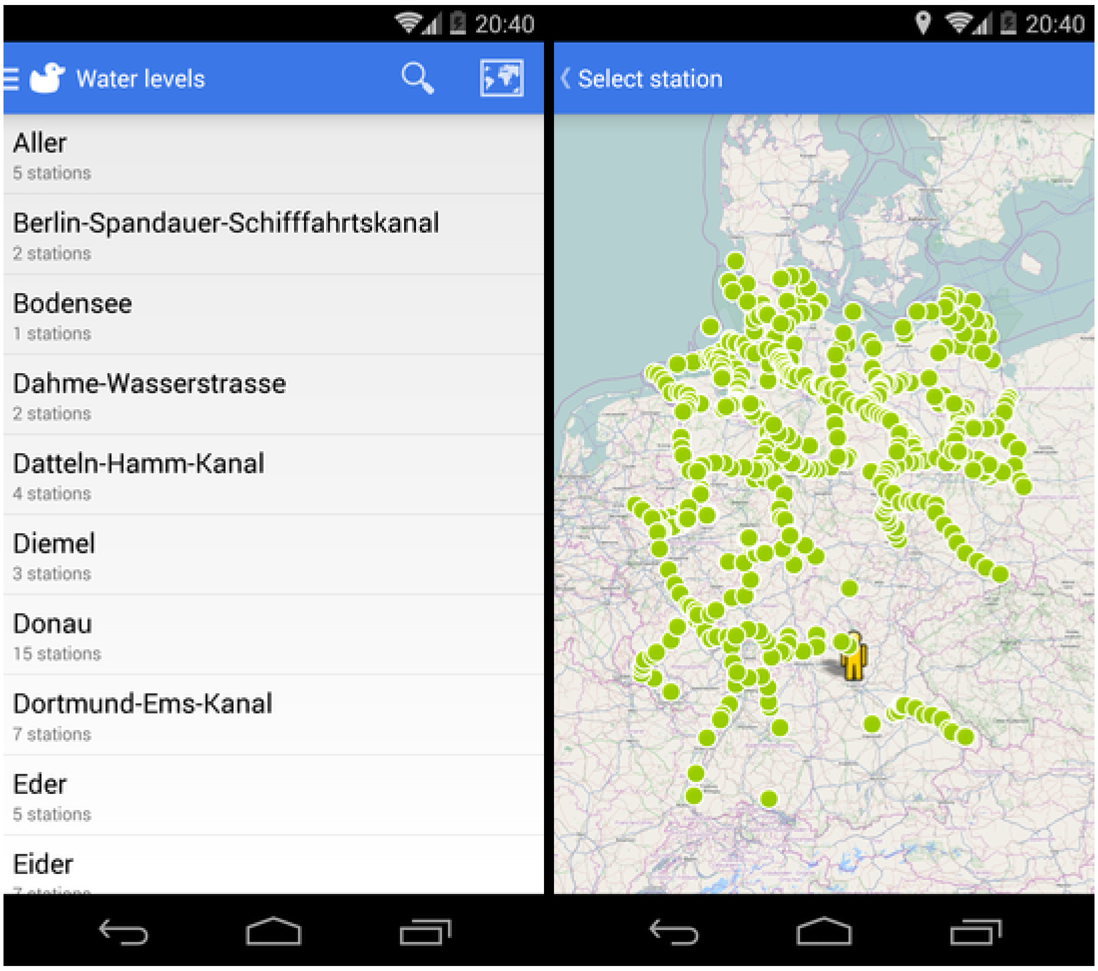


***Figure 16***: *The river graph screen. Water levels (y-axis) of all stations are shown along one river axis (x-axis), here Elbe. The right image shows a normalized version of the same river.*



Aller
5 stations

Berlin-Spandauer-Schifffahrtskanal
2 stations

Bodensee
1 stations

Dahme-Wasserstrasse
2 stations

Datteln-Hamm-Kanal
4 stations

Diemel
3 stations

Donau
15 stations

Dortmund-Ems-Kanal
7 stations

Eder
5 stations

Eider

***Figure 17***: *The river and station selection screen. Picking a station can either happen via a list or a map.*

## References

Astrova, I., Kleiner, C., Koschel, A., Nitz, S. (2013). Applying Event-Driven Architecture to Mobile Computing. *2013 International Symposium on Signal Processing and Information Technology*, 58-63, doi:10.1109/ISSPIT.2013.6781854

Bali, M. (2013). *Drools JBoss Rules 5.X Developer's Guide.* Packt Publishing.

Burgstahler, D., Lampe, U., Richerzhagen, N., Steinmetz, R. (2013). Push vs. Pull: An Energy Perspective. *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, 190-193, doi:10.1109/SOCA.2013.17

Chandy, M., Schulte, R. (2010). *Event Processing: Designing IT Systems for Agile Companies*. McGraw-Hill.

Dobjanschi, V. [Google Developers]. (2010, May 27). Google I/O 2010 - Android REST client applications [Video file]. Retrieved from https://www.youtube.com/watch?v=xHXn3Kg2IQE

Etzion, O., Niblett, P. (2010). *Event Processing in Action.* Manning.

German, D., Hassan, A. (2009). License Integration Patterns: Addressing License Mismatches in Component-Based Development. *2009 31st IEEE International Conference on Software Engineering*, 188–198. doi:10.1109/ICSE.2009.5070520

Gusev, M., Gushev, P., Ristov, S., Velkoski, G. (2014). Alert Notification as a Service. *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics*, 319-324, doi:10.1109/MIPRO.2014.6859584

Jaeger, T. (2008). *Kommerzielle Applikationen für Open Source Software und deutsches Urheberrecht* [Commercial applications for open source software and the German copyright law]. Retrieved from http://www.ifross.org/ifross_html/HoffmannLeible_Beitrag%20Jaeger.pdf

Jaeger, T., Metzger, A. (2011). *Open-Source-Software: rechtliche Rahmenbedingungen der Freien Software* [legal framework for free software]. Beck.

Katz, A. (2007). GPL – the Linking Debate. M*agazine of the Society for Computers and Law, Volume 18, Issue 3, August / September*, 12-16. Retrieved from http://www.moorcrofts.com/documents/GPL%20-%20the%20Linking%20Debate.pdf

Kirkpatrick, F. [Google Developers]. (2013, May 16). Google I/O 2013 - Volley: Easy, Fast Networking for Android [Video File]. Retrieved from https://www.youtube.com/watch?v=yhv8l9F44qo

Luckham, D. (2007). *The Power of Events*. Addison Wesley Pub Co Inc.

Qian, F., Wang, Z., Gao, Y., Huang, J., Gerber, A., Mao, Z., Sen, S., Spatscheck, O. (2012). Periodic transfers in mobile applications: network-wide origin, impact, and optimization. 2012 Proceedings of the 21st international conference on World Wide Web, 51-60, doi:10.1145/2187836.2187844

Rosen, L. (2001). The Unreasonable Fear of Infection . Retrieved from http://www.rosenlaw.com/html/GPL.pdf

Schulte, R. (2014, August 25). An Overview of Event Processing Software. Retrieved from http://www.complexevents.com/2014/08/25/an-overview-of-event-processing-software/

Suhothayan , S., Narangoda , I. L., Chaturanga , S., Gajasinghe , K., Perera S., Nanayakkara , V. (2011). *Siddhi: A Second Look at Complex Event Processing Architectures.* Retrieved from https://people.apache.org/~hemapani/research/papers/siddi-gce2011.pdf

Wasserhaushaltsgesetz vom 31. Juli 2009 (BGBl. I S. 2585), das zuletzt durch Artikel 4 Absatz 76 des Gesetzes vom 7. August 2013 (BGBl. I S. 3154) geändert worden ist [Water economy regulations from 31. July 2009, last changed on 7. August 2013]