
**Chair of Computer Networks**

# Master Thesis

Generic Frontend for Exploring Sensor and Information Services

*Author:*
M.Sc. Uliana Andriieshyna
Matrikel-Nr: 3828303

*Supervisors:*
Dr.-Ing. Josef Spillner
Prof. Dr. rer. nat. habil.
Dr. h. c. Alexander Schill

April 08, 2014

# Declaration of Authorship

I, Uliana Andriieshyna, declare that this thesis, titled "Generic Frontend for Exploring Sensor and Information Services", and the work presented in it have been done on my own without assistance. All information directly or indirectly taken from external sources is acknowledged and referenced in the Bibliography.

Dresden, 08.04.2014 _____

TECHNISCHE UNIVERSITÄT DRESDEN

**Fakultät Informatik**, Institut für Systemarchitektur, Lehrstuhl Rechnernetze

# AUFGABENSTELLUNG FÜR DIE MASTERARBEIT

| | | | |
|---|---|---|---|
| Name, Vorname: | Andriieshyna, Uliana | | |
| Studiengang: | DSE 2011 | Matrikelnummer: | ???? |
| Forschungsgebiet: | Service and Cloud Computing | Forschungsprojekt: | DaaMob |
| Betreuer: | Dr.-Ing. J. Spillner | Externe(r) Betreuer: | |
| Verantwortlicher Hochschullehrer: | | Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill | |
| Beginn am: | 01.11.2013 | Einzureichen am: | 10.04.2014 |
| Thema: | Generic Frontend for Exploring Sensor and Information Services | | |

## ZIELSTELLUNG

Research on smart cities, connected vehicles, complex logistics across organisations and similar trends suggests a need for channeled forwarding of information from heterogeneous sources to creative third-party services and applications. Currently, most of the research is concerned with the protocol and middleware levels, whereas the potential of a generic interactive access to sensor and information services needs to be explored. This involves their selection, mash-ups, and usage within a client-controlled interface.
In this master thesis, a first web-based prototype (portal) for such services is to be created. Along with it, a light-weight scenario service registry will be needed. Users should be able to explore not just services, but also the information provided by them, and eventually be led to advanced usage patterns such as the development of third-party applications to access the information data and real-time streams.

## SCHWERPUNKTE

- Concept for a generic information and sensor service portal
- Development of the portal and associated dependency tools
- Demonstration using a convincing scenario

---

*Unterschrift des verantwortlichen Hochschullehrers*
*Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill*

# Contents

# Chapter 1

# Introduction

The increasing numbers of sensor devices has increased the number of sensor-specific protocols, platforms and software. The definition "sensor" means not only a physical device that can be an actuator or measuring device, but also a context-dependent information that is available through the Web, e.g. feeds, tweets, news, weather forecast and other real-time data streaming. As a result, various approaches have been proposed to connect, maintain and monitor various types of data sources[1, 2, 3]. Those are specifically focused on a platform development, protocol definition and software architecture for the concrete user-oriented requirements. They are typically narrowly focused and do not define a common approach. Proposed approaches are mainly focused on following issues: security and privacy between sensor-specific protocol and platform, online aggregation of secure data in the cloud, monitoring of different types of data sources, social dimensions[4]. Not a single project from the list above covers aspects of providing a generic extensible user-friendly interface for real-time streaming data preview, which should be an essential part of working with sensor systems.

Therefore the area of research of this master thesis is dedicated to building a common approach of generic frontend with a Graphical User Interface for exploring data sources, that can be easily and universally connected to any platform or a system that require GUI. Continuously changing information can be retrieved as a real-time stream, independently from its data type. Together with an option of getting up-to-date values from a data source, a possibility to have access to its historical archives can be also provided. The following sections ground the motivation of the chosen research field, define the central research questions and goals of this master thesis, and describe the overall structure of the work.

## 1.1 Motivation

In the recent years with the technological progress in information systems, web technologies and in particular sensor data systems, have become an essential part in daily life of the modern society. More and more aspects of human life are shifted to the Web and mobile applications. This allows fast and easy way to get information from any point on Earth by using any web, mobile or traditional desktop clients. In the same time, a range of sensor specific platform and interfaces have increased. People have started to use them more often not only for business, education, manufacture, but also for private reasons (e.g.

"smart house", "smart device"). Currently, most of researches are concerned with the protocol and middleware levels, whereas the potential of a generic interactive access to sensor and information services via user-friendly GUI needs to be reseached. As was already mentioned, most of works in this area are focused mainly only on a backend side, thus an essential goal for this thesis is to provide simple and universal for integration web-based generic frontend with a focus on a dynamic GUI composition. Users and developers should be able to explore not only information provided by Web services, but also from real sensors around them. The system architecture of a concept should be highly distributed in order to connect already implemented projects with it.

Creating composite third-party services and applications from reusable components is an important technique in software engineering and data management. Although a large body of research and development cover integration at the data and platform levels, less work has been done to facilitate it at a generic level. This master thesis discusses the existing web frameworks and component technologies used on presentation level integration, illustrates their strengths and weaknesses, and presents some opportunities for future works. Real-time web applications are connection-oriented, providing web-based user interfaces that display information as soon as it was published. Examples include social news aggregators and monitoring tools that continously update themselves with data from an external source.

## 1.2 Application Area

In order to define concrete research questions, it is important to determine which kind of concept is going to be proposed and implemented, as well as to clarify the common terms which are used throughout the thesis:

- **Application.** This thesis is focused on design and implementation of a generic multi-tier web-based frontend that delivers dynamic real-time content to end-users via user-friendly graphical interface. Such type of universal frontend acts as a middleware between provider-specific data source and a user. Any backend system which supports a concept architecture can easily send data to the user, without a need to implement a narrowly oriented frontend.
- **Infrastructure.** Implies a distributed system powering an application stack and running real-time data streaming.
- **Tools and Protocols.** Assumes a set of frameworks and Internet of Things (IoT) protocols for real-time data streaming to explore and maintain services of the aforementioned application.

## 1.3 Research Questions and Goals

As mentioned in the previous section, many solutions for creating sensor-aware applications already exist. But such platforms are mainly focused on a single area of usage and they

are not commonly suitable to support the dynamic and adaptable composition of different types of data sources in one application. Moreover, they typically specify a data exchange protocol, rather then provide a GUI and discovery tools for real-time sensor data.

Within this thesis the following research questions should be answered in order to design, implement and evaluate a desired generic frontend:

- Which architecture should have a concept of a generic frontend?
- How should it be designed in order to provide ease of integration with any backend system?
- What types of data sources have to be retrieved and what is the most universal interface for collaboration between backend and frontend systems?
- How can be a generic GUI designed and implemented by providing dynamic content composition?
- Which protocol can perform real-time data streaming and access to historical archives of data streams independently from type of data in it and bind it with user personal preferences?
- Which software components might be applied to the concept in order to be applicable to the most of available data sources and platforms?

To sum up, this thesis is aimed at the development of a concept that provides users a possibility to integrate a personalized visual environment for exploring, managing and monitoring sensor data, regardless of technical configuration and implementation details of data sources. A concept of a generic frontend can become an essential approach to create project templates for stream-driven applications.

## 1.4 Structure

After the introduction in *Chapter 1* the thesis is structured in the following way:

*Chapter 2* defines the background of the master's thesis, describes the basics of used terminologies and the foundation platforms. Requirements to a concept of a generic frontend that need to be developed are also introduced in this chapter.

*Chapter 3* is devoted to the state of the art analysis. Related research and consumer-based works in the area of sensor-based data retrieving approaches, such as: portal and mashup systems, the browser based and non-browser based systems are investigated against the defined requirements.

*Chapter 4* focuses on the concept of the generic frontend for exploring sensor and information services, considering possible approaches, strategies, frameworks and necessary criteria, defined in *Chapter 3*.

*Chapter 5* provides the implemented functionality of the concept and evaluated by using convinsing scenario. An example of use case scenario is proposed, described and performed in order to cover the fulfillment of defined requirements by the developed solution. Important methods and extensions of an interface are described in details.

*Chapter 6* concludes the master thesis underlining and evaluating achieved goals and providing prospects for the possible future work.

# Chapter 2

# Foundation and Requirements Analysis

The main goal of the following Chapter to define requirements to a generic frontend for exploring sensor and information services. To achieve formalized goals in the Chapter 1, fundamental terms in the presented research area at the beginning of the Chapter are defined. In the Section 2.2 basic aspects concerning data source definition, types of information services in the web are described. Summary section underlines this Chapter by combining needs of frontend and data sources and defines requirements to a common concept, which is described in details in the Chapter 4.

## 2.1 Frontend's Requirements

In computer science, the frontend is responsible for collecting input in various forms from the user and processing it to confirm to a specification the backend can use. The frontend is an interface between the user and the backend[5] and the separation of software systems into front- and backend simplifies development and separates maintenance. Therefore needs to be distinguished what are the main requirements to a generic frontend, in order to be applicable to any backend system and type of a data source, i.e. :

- *Loose coupling:* each of systems components has, or makes use of, little or no knowledge of the definitions of other separated components. Where the main goal is to avoid dependencies between new components, modules and easy deployment of future enhancement.
- *Fine-grained structure:* split the system into a small parts (logic modules, universal interfaces for every type of data format, information resources in the Web), such that it can be distributed across Internet, and can be applied to any resource and used from any mobile device. It also includes ease of expansion and integration with backend system, independently from platform used on a backend and frameworks used on a frontend side.
- *Multi-user capabilities:* defining to users visibility rules according to their type and rights and load personilized bookmarks/preferences, targeting by using ID and pass-

word. Every account have to be uniquely binded with a data source, contracts or personalized preferences. After that, GUI can be targeted according to it.

- *Cross-platforming or multi-platform:* a possibility of application to be run on any type of device (e.g., smartphone, notebook, tablet) without special preparation or changes. This gives an possibility to be flexible.

- *Responsive design:* an ability of a GUI automatically adapt to any size of device screen, by provisioning high usability performance.

- *Usability:* web-based interface, which gives an easy to understand user-friendly interface of a different data types. User-oriented design provides fast and simple awareness of what frontend propose and how it can be explored by user.

### 2.1.1 Fine-grained structure

The fine-grained structure of a system provides a possibility to distribute tasks between responsible modules. It guarantees performing tasks in parallel, moreover, exchange, enhance and add new parts of a system, without influence to an any another module. Also it guarantees necessary performance in response time and makes a system to be highly scalable. If the granularity is too fine, the performance can suffer from the increased communication overhead. On the other side, if the granularity is too coarse, the performance can suffer from the load imbalance. Hereby, generic frontend have to be splitted to independent modules, responsible for a separated tasks, that in the same time gives a possibility to scale the system.

### 2.1.2 Loose Coupling

The goal of loose coupling is to reduce dependencies between system's parts. And this term mostly related to an internal software structure of a system, e.g. classes, methods, asynchronous requests. After a comprehensive comparison of tight- and loose coupling properties in the Table 2.1[1] becomes clear, that loose coupling plays an important role in building abstract and reusable frontend's modules.

[1] A High-level Comparison of Tight and Loose Coupling,`http://wiki.scn.sap.com/wiki/display/BBA/Loose+Coupling+Through+Web+Services`

| Target | **Tight Coupling** | **Loose Coupling** |
|---|---|---|
| Physical Connection | Point-to-Point | Via mediation (a) |
| Communication Style | Synchronous | Asynchronous |
| Data Model | Complex common types | Simple common types |
| Service Discovery and Binding | Static | Dynamic |
| Dependence on Platform-Specific Functionality | Strong/many | Weak/few |
| Interaction Pattern | Via complex object trees | Via data-centric, self-contained messages |
| Transactional Behavior | Controlled by a central transaction manager (e.g., two-phase commit) | Compensation (b) |
| Control of Process Logic | Central control | Distributed control |
| Deployment | At the same time (c) | Different point in time if desired |
| Versioning | Explicit/forced upgrades | Implicit upgrades |

Table 2.1: A High-level Comparison of Tight and Loose Coupling

Where:
a. Mediation implies that some mechanism must handle communication between the composite application and the backend system, filling the role of the service contract implementation layer.
b. Compensation means that for every modifying service, a dedicated compensational service must explicitly be developed for rollback purposes. If the modifying service is part of a service chain that has to be executed as one transaction and an error occurs, the compensational service helps to undo the first modifying operation and sets the system back to its initial state.
c. For tightly coupled applications, the parts must always be deployed at the same time. Loosely coupled applications do not have this requirement (though of course the parts could be deployed at the same time). Consider an interface change for a web service that performs a write changes. When a tightly coupled service is called, if the interface has changed, it simply will not work. A loosely coupled service operation will write the data to the service contract implementation layer and continue. Once the new write operation is in place, it can handle all buffered calls.

The degree of the loose coupling can be measured by noting the number of changes in data elements that could occur in the sending or receiving systems and determining if the computers would still continue communicating correctly[6, 7]. These changes include items such as:

- adding new data elements to messages/GUI elements, since real-time streaming are going to be implemented;
- changing the order of data elements;
- changing the names of data elements;
- changing the structures of data elements;
- omitting data elements.

Benefits of loose coupling include flexibility and agility. A loosely coupled approach offers unparalleled flexibility for adaptations of a changes in content which have to be retrieved by GUI. Another aspect to consider is the probability of layout changes during the lifetime of the application. Due to mergers and acquisitions and system consolidations, the layout underneath the application is constantly changing. Without loose coupling, user will be forced to adapt application again and again. In essence, loose coupling means reducing the number of assumptions to a bare minimum. The goal of loose coupling is to minimize dependencies between systems.

### 2.1.3 Multi-User Capabilities

In order to provide better user experience needed to be distinguish main user types of a system itself:

- Backend developer, which needs an simple and clear interface for integration with a backend and adaptable modules for its data structure. Thereby, any backend can be easily connect with a generic frontend. Comprehensive and detailed service descriptions is a first entry point for developers. Well or poorly described service measures the amount of efforts, made by developers.
- Developer of a mobile application, who has a task to find necessary sensors as fast as possible, explore metadata provided by data soutce, SLA if exist, explore the data provided by sensor and get to know of configuration of end-points and system architecture.
- Simple user of an application, that wants to monitor and control sensors and their statistics.

According to defined types of users, needed to be specified what are the main requirements from a user prospective to the researched frontend approach.

Backend developer, who starts to work with the frontend, needs to get comprehensive and detailed service descriptions. Therefore generic frontend have to support common standards in a field of interface, protocols, approaches to build web services. Very helpful for a developer will be a possibility to connect new modules directly from a user interface by completing a respective form.

Developer of a mobile application plays a role of a new user, that knows the purpose of a system and needs to find sensors in order to connect it with his/her own mobile application. Firstly, he/she needs to find necessary sensor, accept an SLA with sensor provider, if exist, get to know of what type of data retrived by choosen sensor and if it satisfies his/her requirements, download an SDK, documentations and configuration of end-points.

A simple user log in to the system, without any knowledge of a system, should be clarified what should be shown on a main screen and how to structure the data in such a way, that user get used to the dashboard very fast. A dashboard responsibility is to make clear of which steps have to be performed by user in order to get started receiving real-time data.

In a context of a generic frontend the term multi-user capabilitites means to bind personalized users' requirements with corresponding credentials, preferences and personal settings. To use client-side data binding and dependency injections to build dynamic views of data that change immediately on response to user actions.

### 2.1.4 Cross-Platforming

A concept of a generic frontend have to work not only on more than one desktop system platform (e.g., Unix, Microsoft Windows, Macintosh), but also on every mobile device independently from version or type of device. Possible solutions and approaches which can satisfy such requirements will be discovered in detailes in the Section 3.2. Nowadays, cross-platforming application become popular more and more and it opens a lot of additional properties to the system. Such properites are include:

*Separation of functionality*: Separation of functionality attempts to simply omit those subsets of functionality that are not capable from within certain client browsers or operating systems, while still delivering a fully-functional application to the user.

*Multiple code base*: Multiple code base applications present different versions of an application depending on the specific client in use. This strategy is arguably the most complicated and expensive way to fulfill cross-platform capability, since even different versions of the same client browser (within the same operating system) can differ dramatically between each other. This is further complicated by the support for "plugins" which may or may not be present for any given installation of a particular browser version.

*Third-party libraries*: Third-party libraries attempt to simplify cross-platform capability by "hiding" the complexities of client differentiation behind a single, unified API.

### 2.1.5 Responsive design

Responsive design [8, 9] is a web design approach aimed at crafting sites to provide an optimal viewing, experience-easy reading and navigation with a minimum of re-sizing, panning, and scrolling-across a wide range of devices (from mobile phones to desktop computer monitors). An application designed with responsive design[10, 11] adapts the layout to the viewing environment by using fluid, proportion-based grids, flexible images, and adaptive styles.

- The fluid grid concept calls for application elements sizing to be in relative units like percentages, rather than absolute units like pixels or points.

- Flexible images are also sized in relative units, so as to prevent them from displaying outside their containing element.

- Allowed usage of different styles rules based on characteristics of the device the application GUI is being displayed on, most commonly the width of device.

### 2.1.6 Usability

Usability is the ease of use and learnability of a human-made object. In human-application interaction and computer science, usability studies the elegance and clarity with which the interaction with an application is designed.

ISO defines usability as "The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use.". The word "usability" also refers to methods for improving ease-of-use during the design process. Usability consultant Jakob Nielsen has written: "Studies of user behavior on the Web find a low tolerance for difficult designs or slow sites. People don't want to wait. And they don't want to learn how to use a home page. There's no such thing as a training class or a manual for a Web site. People have to be able to grasp the functioning of the site immediately after scanning the home page—for a few seconds at most"[12]. Otherwise, most casual users simply leave an application and use another one. Computer science professor Ben Shneiderman have written about a framework of system acceptability, where usability is a part of "usefulness" and is composed of[13]:

- Learnability: How easy is it for users to accomplish basic tasks the first time they encounter the design?

- Efficiency: Once users have learned the design, how quickly can they perform tasks?

- Memorability: When users return to the design after a period of not using it, how easily can they re establish proficiency?

- Errors: How many errors do users make, how severe are these errors, and how easily can they recover from the errors?

- Satisfaction: How pleasant is it to use the design?

In scope of this master thesis will be covered such characteristics of usability as: learnability, efficiency and satisfaction. Rest of it together with possible enhancement will be proposed in Chapter 6 in the section Future Work.

## 2.2 Types of Data Sources

In order to distinguish what type of data sources can be retrieved, all data sources was differentiated as hardware and software sensors. Sensor in such a way not just a hardware, but everything that "sense" a data.

### 2.2.1 Hardware Sensors

A sensor is a converter that measures a physical quantity and converts it into a signal which can be read by an observer. A hardware sensor provides a real value from surrounding world, e.g. temperature, humidity, lighting level, electricity or water measurement etc. A backend after receiving this signal have to reconvert it to a signal that can be retrieved by frontend. The main task of a generic frontend to determine what are the type of hardware sensors, what kind of data it provides, when and how retrieve a real-time values inside the graphical container on a GUI, when the connection have to be established, how can be a sensor described and presented. A global standard data format have to be discoverd in order to describe real-time data received from sensor and information about sensor itself. Thereby any backend and frontend can easily integrate new sensors and rely on.

Together with software sensors the proposed data standard is justified and explained in the Section 4.3.1

### 2.2.2 Software Sensors

Nowadays more and more physical information move to a web by simplifying the way of sharing information. Already implemented thousands of public services such as: GPS coordinate, weather forecast with all arising from it data, e.g. ambient temperature, humidity, pressure, traffic feeds. Not only hardware sensors become available through the web, but through Web 2.0, blogs, tweets, RSS feeds, social networks makes possible to create human-behavioral sensors or automatic computer-made sensors like crawlers or analytics. People got used to it, and today has become an inseparable part of their lives. Software sensor is not only an information from the web, but everything that can be connected through the standard web interface by using internal software from provider.

An important research question is: how to manipulate data, made by different vendors, by using single dashboard? To get an answer to this question and define most common approach, was used categorization by vendors made in scope of research in the paper[14]. The facts about the data vendors were gathered by means of a web search. As every vendor or marketplace has a web site, this publicly available information was used to determine how to categorize each vendor.

| Dimension | Categories | Question to be answered |
|---|---|---|
| Type | Web Crawler, Customizable Crawler, Search Engine, Pure Data Vendor, Complex Data Vendor, Matching Vendor, Enrichment Tagging, Enrichment Sentiment, Enrichment Analysis, Data Market Place | What is the type of the core offering? |
| Time Frame | Static/Factual, Up To Date | Is the data static or real-time? |
| Domain | All, Finance/Economy, Bio Medicine, Social Media, Geo Data, Address Data | What is the data about? |
| Data Origin | Internet, Self-Generated, User, Community, Government, Authority | Where does the data come from?Who is the author? |
| Pricing Model | Free, Freemium, Pay-Per-Use, Flat Rate | Is the offer free, pay-per-use or usable with a flat rate? |
| Data Access | API, Download, Specialized Software, Web Interface | What technical means are offered to access the data? |
| Data Output | XML, CSV/XLS, JSON, RDF, Report | In what way is the data formatted for the user? |
| Target Audience | Business, Customer | Towards whom is the product geared? |
| Trustworthiness | Low, Medium, High | How trustworthy is the vendor? Can the original data source be tracked or verified? |
| Size of Vendor | Startup, Medium, Big, Global Player | How big is the vendor? |

Table 2.2: Categorization of Data Vendors in the Web

Such categorization clarify main important attributes in which customer can be interest in, namely: data access, data output, trustworthiness, data origin and time frame. Such typization appear as an appropriate attribute in a data standard, which was described in details in the Section 4.3.1

As shown in the Table 2.2 the flexibility and modularity of APIs have made these the most popular of all data access methods. More than 70% of all vendors offer an API. However, less than 30% of all vendors have an API as their only way to access data. Most vendors offer an API next to other methods. For example, web interfaces or file downloads are used to give previews of the dataset, to make it easier and more accessible for the customer to see what the actual data looks like. Preview of a real-time data and description of a data source, will be structurized and accessible through the Web API by using data standard description.

Besides aforementioned properties of a metadata format, the main issue for generic frontend is to dynamically retrieve information provided by data sources. It has to be done independently from the data type. Data can change its value during some period of time and as soon as it happens, user have to receive it immediately. A Frontend is responsible for collecting and aggregating data, keeping connection with it, notifying user and visualize it on a GUI. Does not matter how often the data will be changed because all data transfered as real-time streaming. The term streaming data, usually applied to such type of data as

video, audio, or map of values and usual approaches as HTTP GET/POST become unuseful in this case. Therefore, one of the most important requirement to the concept is to define an interface between backend and frontend which can support real-time data retrieval.

The data categorization on which concept of a generic frontend a mainly focused on are:

- feeds from the web, RSS, video and music streaming;
- traffic information sensor network from the city ("smart city");
- home automation sensors (“smart home”);
- continuously growing log files from researchers;
- crowdsourced sensors, news;
- industrial/embedded sensors, robots;
- virtual data sources which result from aggregation.

## 2.3 Summary

This chapter gives an overview of basic requirements to a generic frontend. Starting from main properties in the Section 2.1 namely, loose coupling, fine-grained structure, multi-user capabilities, cross-platforming, responsive design and usability. And continuing in the Section 2.2 with differentiation of data sources, typization and separation to a hardware and software sensors. It helps to define main channels of retrieved data in which user can be interested in and based on that build a system architecture. An essential part of any data retrieval is to determine a most appropriate interface between backend and frontend for a real-time data streaming. Based on requirements to the generic frontend next chapter presents state-of-the-art in area of researched field and gives an overview of already existent solutions.

# Chapter 3

# State of the Art

The following chapter covers an overview and analysis of existent solutions in related research areas of sensor-driven projects. At the beginning of the chapter in the Section 3.1 most familiar research works are studied. It covers public and private sector, where is a lot of work done in the field of Web.

In the section 3.2 prominent approaches of a frontend development are studied and evaluated against the requirements described in the Section 2.1 with a purpose to clarify their capabilities and properties.

## 3.1 Web of Sensors

This thesis is targeted at the creation of a generic user-friendly interaction system. Nowadays every project is focused on a specific area of realization and handling a narrow range of data types, such as urban environment[3], "smart homes", robots or geospatial data over the web[1] and much more as described below. A lot of work related to Cloud-based information resources and Internet of Things (IoT), where the main focus on information which is the result of human behaviour or usage of already implemented tools or platforms are already done.

*Smart city*[1] tool uses information technology and communication (ICT) to help local government to monitor what currently happening in the city (Fig. 3.1). Web-based application for monitoring city in a single dashboard to help summarize the current conditions of a city. The architecture system use network sensor consisting of sensor nodes that has function to capture city condition like temperature, air pollution, water pollution, traffic situation. Besides exist a possibility to add another information, e.g. socio-economic situation like public health service, economic indicator, energy supplies, etc. The results of work presents a prototype of the smart city dashboard of a Bandung City, one of big cities in Indonesia. It consists of network sensor, server, application and also communication protocol which is used for city monitoring. The summary information of the city displayed in a single view to help people watching and analyzing of what being happen at the real-time in the city.

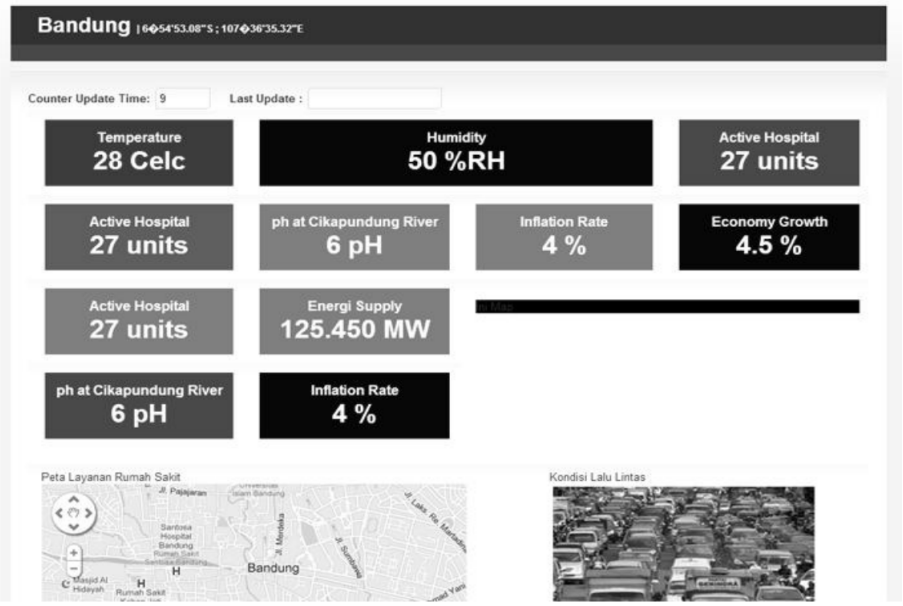


Figure 3.1: Sample of Smart City Dashboard Application

*Microsoft SensorMap[15, 16]*(Fig. 3.2): a system to monitor, search and visualize spatially and geographically related data such as driving directions, directory entries, weather and traffic conditions based on Microsoft Virtual Earth and overlay with housing prices, crime rates, bus locations, and other data on top of browsable maps. SensorMap is a web

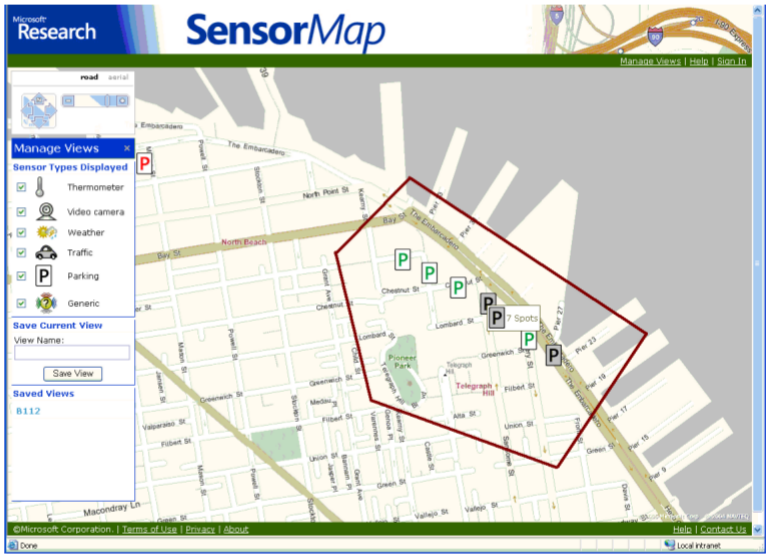


Figure 3.2: Sample of SensorMap

portal that allows owners of sensor networks to register their physical sensors and publish their data on SensorMap. They use GeoDB to store the sensor network information, housing sensor metadata including the publisher's name; the sensor's location, name, and type; the data type; and free text descriptions. DataHub to retrieve the new sensor data to enable real time services. The aggregator to summarize sensor data in a specific area to clients. And SensorMap GUI is based on Windows Live Local and therefore provides features such

as zooming, panning, street maps, satellite images, and 3D views. SensorMap supports three types of queries: geographic queries that drawing geometric shapes directly on the map (for example, within a region, near a route); type queries that sensor types specify within the viewport; and free text queries that keywords describing sensors specify. The SensorMap describes first constructive idea how to connect different types of sensors available in the Web through API and provides system architecture. But within this project authors do not consider questions as: retrieval of real sensors in a room, sensor control, live data streaming, adaptive GUI and integration with another platform and systems without specifically added libraries.

*LiveWeb Portal*[17] presents the architecture, design, and approach to build a sensorweb service portal, where sensorweb is a global observation system for varied sensor phenomena from the physical world and the cyber world measured in a real-time. This system has been used to represent and monitor a real-time physical sensor data and cyber activities from ubiquitous sources. The LiveWeb meets its goal of providing an efficient and robust sensor information oriented on a web service, enabled with real-time data representation, monitoring and notification. LiveWeb has the following properties: the system enables sensorweb service accessible from anywhere, makes sensor network data readable by anyone, provides methodology of building searching engine and sensor network sharing. Based on open standards in the world of web, they propose an approach of how to build a generic system with heterogeneous resources.

*Internet of Things*[2] presents a service platform based on the Extensible Messaging and Presence Protocol (XMPP) for the development and provision of services for Internet of Things (IoT) mainly focusing on the integration of things based on domains like smart cities, automotive or crisis management require service platforms involving real world objects, backend-systems and mobile devices. Was argued necessary usage of XMPP client as protocol for unified, real-time communication and introduce the major concepts of a platform. Based on two case studies demonstrated real-time capabilities of XMPP for remote robot control and service development in the e-mobility domain. And in the same time a proof of concept was made by authors in a project called ACDSense[1] that XMPP and its numerous extensions have gained momentum as a multi-purpose lightweight middleware for social computing, multi-device interaction, and communication with sensors (Fig. 3.3). The concept was demonstrated with a use case of three distributed and connected XMPP-driven pervasive systems: standard XMPP client, sensor App and Web browser sensors dashboard. The aggregation of data as a web application done by using multiple widgets with the help of an Inter-Widget Communication (IWC) approach based on portlets[18].

*Dynvoker Portal*[19] a generic human-driven ad-hoc usage approach, by including rapid service testing and dynamic inclusion of services as plugins into applications. Dynvoker is an engine, which consists of a relatively small application core which can be run as a servlet, a web service or a command-line application. Explore method-centric and resource-centric services alike, output forms in various formats or integrate GUI services to provide a richer user experience. The generic design of many parts of Dynvoker has yielded a lightweight architecture which is freely available to any interested person as an open source project.

[1] ACDSense project, `http://www.rwth-aachen.de/`

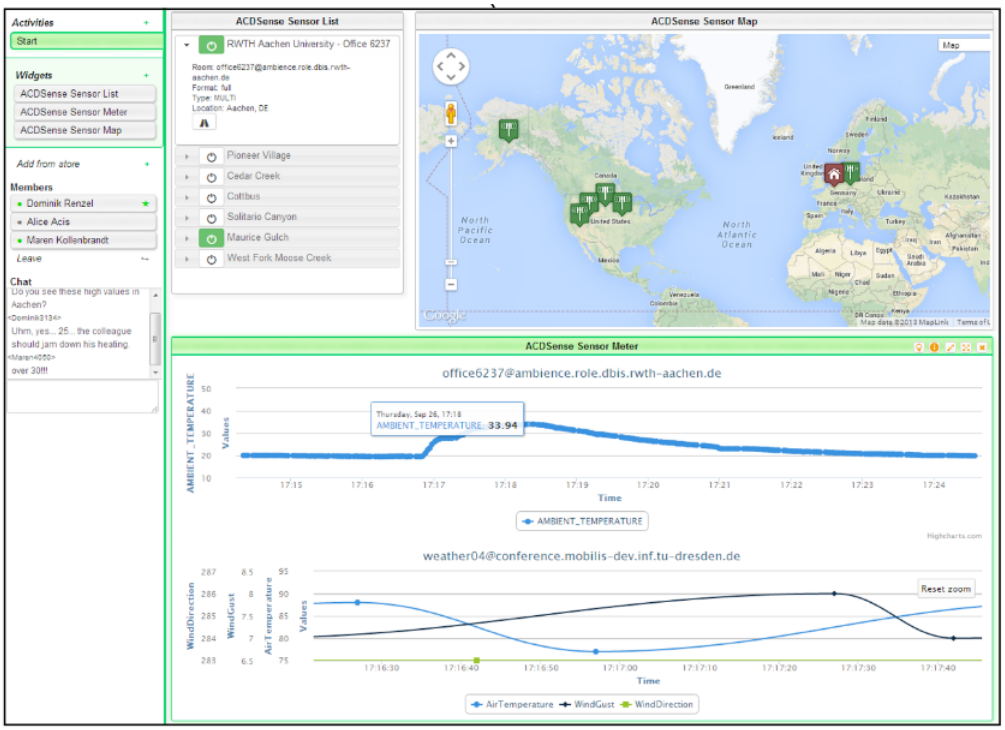


Figure 3.3: Sample of ACDSense

*Sensor Web Enablement* project[20] is focused on developing standards to enable the discovery of sensors and corresponding observations, exchange, and processing of sensor observations, as well as the tasking of sensors and sensor systems. Open Geospatial Consortium (OGC), Inc. members specifies interoperability interfaces and metadata encodings that enable real-time integration of heterogeneous sensors into the information infrastructure. Developers will use these specifications in creating applications, platforms, and products involving web-connected devices such as flood gauges, air pollution monitors, stress gauges on bridges, mobile heart monitors, Webcams, and robots as well as space and airborne earth imaging devices. In this publication by OGC was defined XML-based standards as: Sensor Model Language (Sensor ML), Sensor Observation Service (SOS), Web Notification Service (WNS) etc. As subproject calls SANY (Sensors Anywhere) focuses on interoperability of in-situ sensors and sensor networks. The goal for the SANY architecture is to provide a quick and cost-efficient way to reuse data and services from currently incompatible sensors and data sources in future environmental risk management applications. By developing a standard open architecture and a set of basic services for all kinds of sensors, sensor networks, and other sensor-like services, the SANY IP supports and enhances both GMES (Global Monitoring for Environment and Security, a major European space initiative) and GEOSS (Global Earth Observation System of Systems) in the area of in situ sensor integration. Though the SANY work enhances interoperability for monitoring sensor networks in general, the application focus is on air quality, bathing water quality, and urban tunnel excavation monitoring.

*VICCI Project* (Visual and Interactive Cyber-physical Systems Control and Integration)[21, 22] (Figure 3.4). The scope includes smart home environments and supporting people in the ambient assisted living, considers the software-technical side of so-called “Cyber-physical

systems" (CPS). This term includes complex, embedded systems, which connect the virtual and the physical world with each other (IoT) in different application scenarios. The main uses of CPS are in logistics, traffic optimization, in the use of robots in the industrial and domestic sectors, in modern energy networks (Smart grid), in the building and factory automation (Smart factory), as well as in the field of intelligent office installations (Smart Office). The aim of project VICCI is the creation of software engineering principles that are necessary for the development of complex cyber-physical systems. Firstly, CPS should be made understandable and accessible by means of a comprehensive control center. Secondly, platforms that enable the development and marketing of software for complex CPS through a pure control panel are to be developed. A domestic environment is considered a sample scenario in which a person with reduced mobility is supported by sensors, actuators and a service robot, which is currently seen as a complex cyber-physical system. No concrete frontend or any kind of user-friendly GUI have been not yet developed.

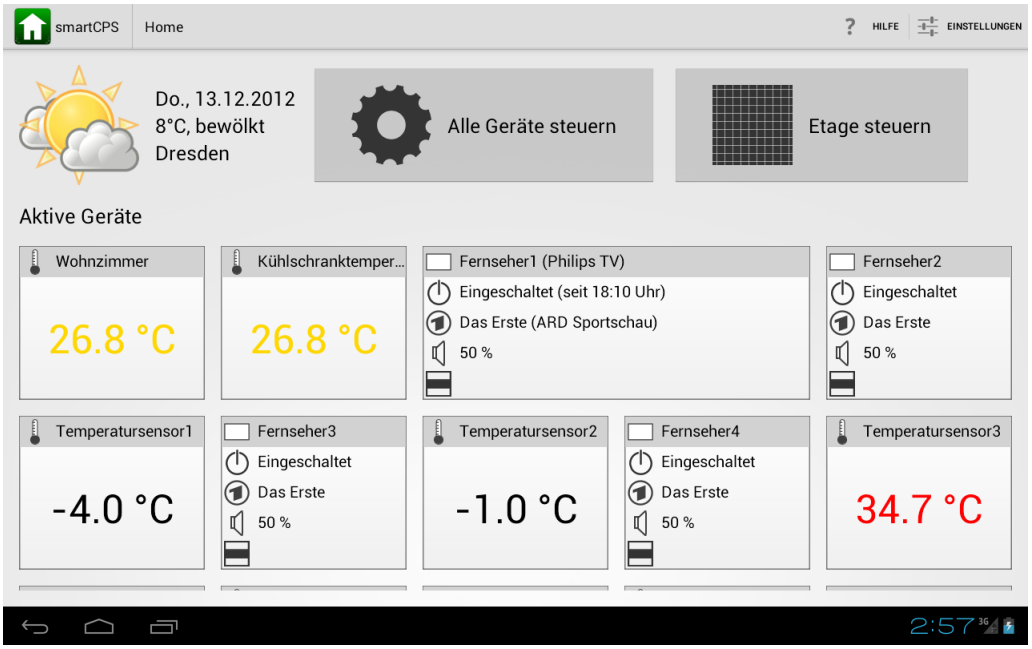


Figure 3.4: Mockup of VICCI project

A series of articles devoted to integrate sensor data into a Cloud. Special attention is given to a privacy-relevant or otherwise sensitive information that is stored in a Cloud.

*SensorCloud*[23], a cloud design for user-controlled storage and processing of sensor data proposed security architecture enforces end-to-end data access control by the data owner reaching from the sensor network to the Cloud storage and processing subsystems as well as strict isolation up to the service-level. In this paper authors presents transport security mechanisms for communication with the Cloud, applies object security mechanisms to outbound data items, and performs key management for authorized services.

*CloudRemix[24]* a Personal and Federated Cloud Management Cockpit, an interactive cockpit to manage personal clouds and their federations (Figure 3.5). Is a new techniques for users to perform asset discovery, exchange and management in Cloud area. The CloudRemix

prototype demonstrates its utility to manage personal clouds in both social and market-driven environments. The goal of CloudRemix is to be open, user-centric regarding the manageable assets, and flexible regarding their free or commercial exchange, with or without explicit contract negotiation. CloudRemix is an open-source web-based cockpit application with support for multiple users. Each user gets to see an aggregated list of both local and remote services of each of the asset types.

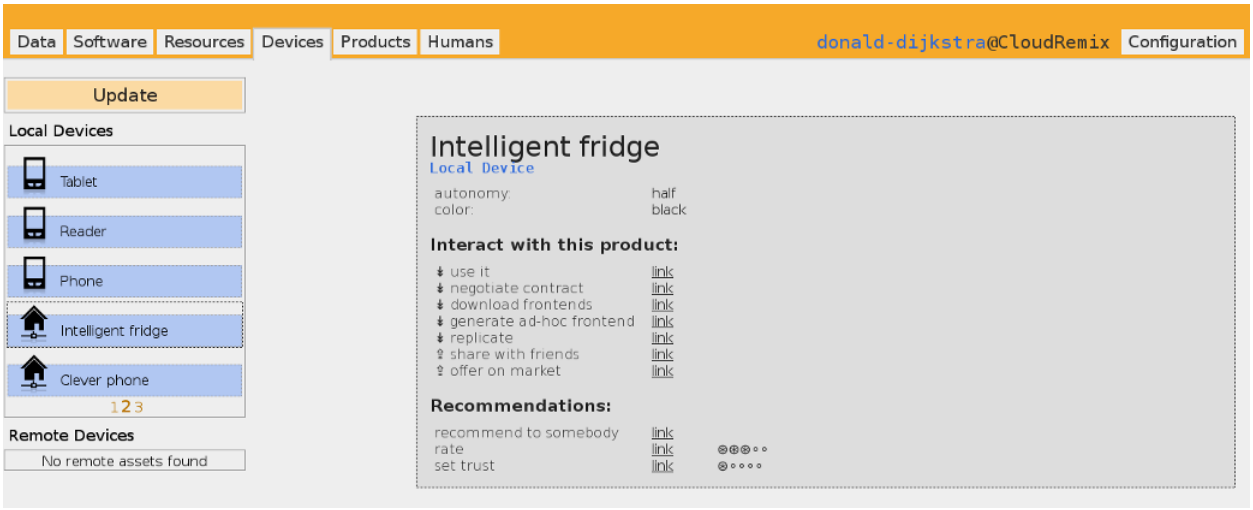


Figure 3.5: Sample of CloudRemix

*Xively Cloud Services*[2] (Figure 3.6) is the Public Cloud which was particularly built for Internet of Things, by giving developers standards-based services and tools, elastic scalability, and intuitive lifecycle management capabilities oriented on a business customer. With Xively, user can gain the agility and efficiency needed to meet market demands for compelling connected products and solutions. Xively's Platform as a Service (PaaS) provides the tools and services needed to create compelling products and solutions on the Internet of Things.

*Optique*[3] (Figure 3.7) is a web-based semantic access to Big Data for effective data analysis and value creation. Optique brings a data access by providing a semantic end-to-end connection between users and data sources; enabling users to formulate intuitive queries using familiar vocabularies and conceptualizations; seamlessly integrating data spread across multiple distributed data sources, including streaming sources[25, 26, 27]. According to publications, no research was done in a field of visualization and presentation. Implemented Frontend with adaptive and cross-browser GUI was developed specifically for this project by using up-to-date technologies (jQuery, HTML and CSS).

The Table 3.1 presents all researched projects according to next characteristics: status (active or inactive in nowadays), availability of frontend as full functional system with business logic and GUI, Mockup (planed GUI), type of project (public or private).

[2]Xively ,https://xively.com/
[3]Optique ,http://www.optique-project.eu/

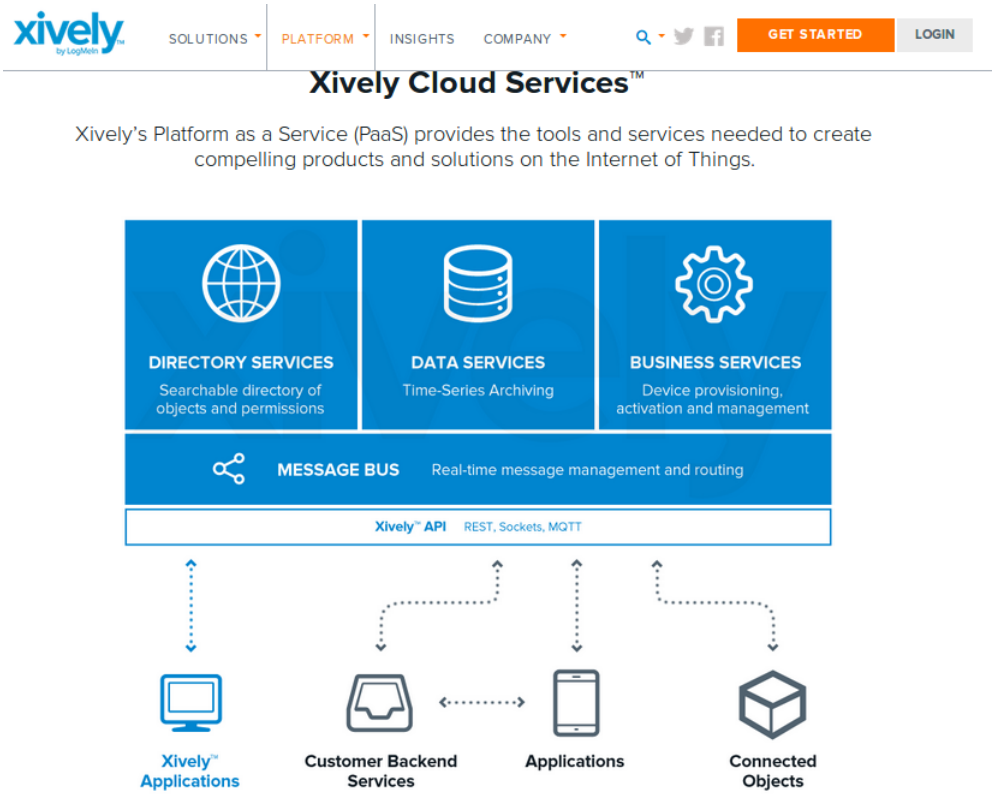


Figure 3.6: Sample page of Xively

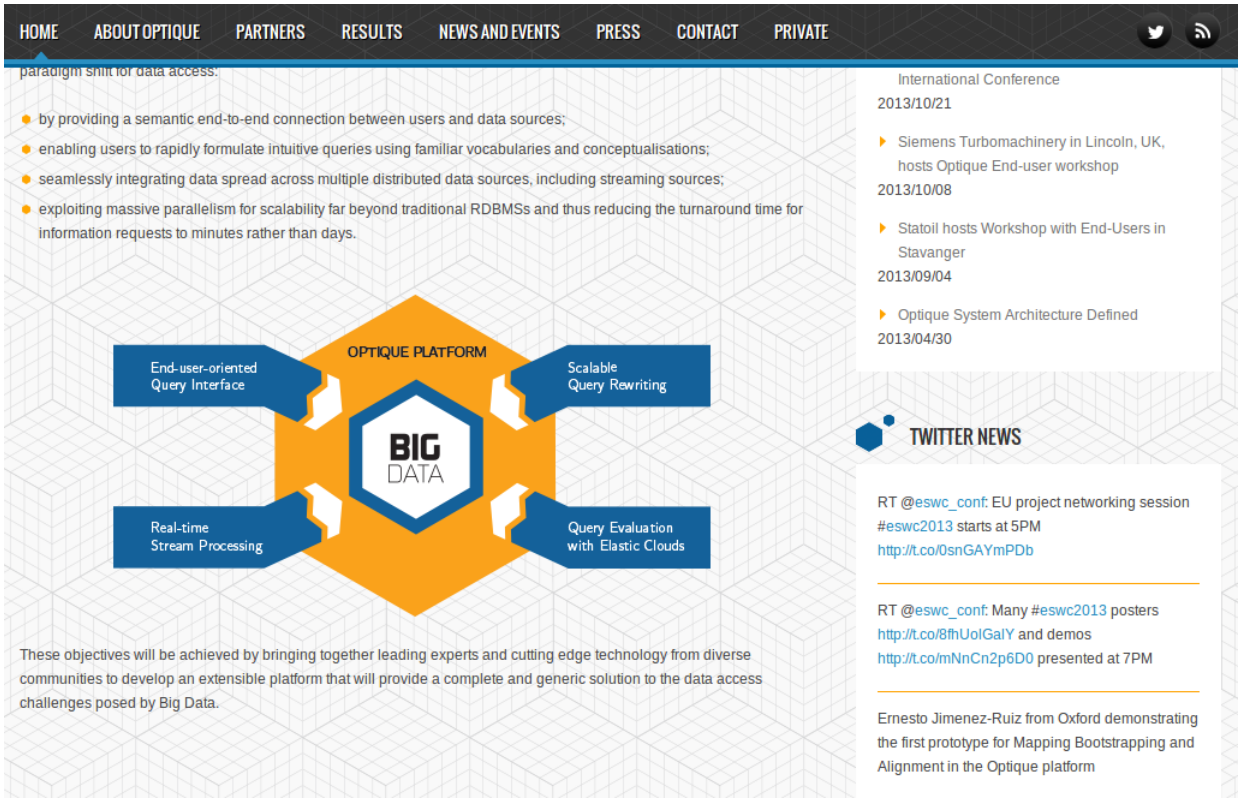


Figure 3.7: Sample page of Optique

| Target | Status | Frontend | Mockup | Type of Project |
|---|---|---|---|---|
| Smart City | active | yes | yes | public |
| Microsoft SensorMap | inactive | yes | - | public |
| LiveWeb Portal | inactive | no | no | public |
| Sensor Web Enablement | inactive | no | no | public |
| SensorCloud | inactive | no | no | public |
| CloudRemix | active | yes | - | public |
| VICCI Project | active | no | yes | public |
| Dynvoker Portal | inactive | no | no | public |
| Internet of Things | active | no | no | public |
| Xively | active | yes | - | private |
| Optique | active | yes | - | private |
| ACDSense | active | yes | yes | public |

Table 3.1: State of the Art Summary

Among all of presented research projects interactive frontend was not implemented. All presented screenshots of an dashboards developed only to satisfy concreate requirements from a backend. Most of them simply visualise data provided by backend, no manipulation can be done in such a way. Only few of them have interactive web-based interfase, such as ACDSense or CloudRemix, but unfortunately, due to concreate realization, no enhancement on frontend side (adding new modules, new visual parts) can be done independently from technology used on a backend side. Most of the research projects have no frontend at all, or only some mockup as a simple picture. Thus, concluding all researched projects above, become clear that needs of a generic frontend, that can be easily integrated with any type of data-driven platform is high. To do so, needed to be clarify what are the main methodologies to build frontend are existent nowadays, specify types of data that have to be retrieved and possible interface for aggregation of data and collaboration between all modules defined by system architecture.

## 3.2 Frontend Development Approaches

In computer science, a frontend is responsible for collecting input from user and processing it to a backend system and another direction - collecting data from backend, namely sensor data steam, and processing it to the user-friendly interface. Therefore, on the one side, generic frontend has to satisfy architecture requirements from backend, such as: fine-grained structure, cross-platforming, loose coupling; and on the other side, implement a dynamic user-friendly interface for a end-user with a client-side multi-user data binding. Thus, to satisfy aforementioned requirements it is necessary to compare all available web-oriented solutions.

To retrieve data from different resources in one web-based interface was specified next existent approaches:

- portal with portlets,

- mashup[4],
- browser based systems,
- non-browser based system.

### 3.2.1 Portal

The main concept is to present the user with a single web page that brings together or aggregates content from a number of other systems or servers.

Usually, each information source gets its dedicated area on the page for displaying information (a portlet); often, the user can configure which ones to display. The extent to which content is displayed in a "uniform way" may depend on the intended user and the intended purpose, as well as the diversity of the content. Very often design emphasis is on a certain "metaphor" for configuring and customizing the presentation of the content and the chosen implementation framework and/or code libraries[28, 29]. In portal technologies end-user can customize number of retrieved data sources, but for that he has to be aware what is it and how to integrate it in portal. User interface in portals have fixed layout, style and location on the web page. To make changes in it, end-user needs to have a deep knowledge of the system architecture and of whole portal entirely. The portal allows the administrator to define specific sets of applications, which are presented to the user in a single page context. The Portlets themselves are more than simple views of existing Web content. A Portlet is a complete application having multiple states and view modes, plus event and messaging capabilities.

Portlets run inside the Portlet container of a portal server, similar to a servlet running on an application server. The Portlet container provides a runtime environment in which portlets are instantiated, used, and finally destroyed. Portlets rely on the portal infrastructure to access user profile information, participate in window and action events, communicate with other portlets, access remote content, look up credentials, and store persistent data.

A portal may use a search engine API to permit users to search intranet content as opposed to extranet content by restricting which domains may be searched. Apart from this common search engines feature, web portals may offer other services such as e-mail, news, stock quotes, information from databases and even entertainment content. Portals provide a way for enterprises and organizations to provide a consistent look and feel with access control and procedures for multiple applications and databases, which otherwise would have been different web entities at various URLs. The features available may be restricted by whether access is by an authorized and authenticated user or an anonymous site visitor.

**Main features Integration** — Ability to integrate with current tools or the possibility of adding new tools.

**Security** — Enable user or group based security to secure documents and sites throughout the intranet portal.

[4] `http://www.programmableweb.com/applications`

**Customization** — Software that is flexible to allow for organization. Web Parts can be used to create custom modules which can make interaction easier with the site. Ability for users to customize tools and resources they use most often.

**Collaboration** — People are now able to collaborate their work with each other. Example would be multiple people working on one document.

**Communication Channels** — Allows corporations to promote corporate culture and present information in a more interactive way than before.

**User Friendly** — Application must be easy to use and understand due to a wide range of technical abilities.

**Remote Access** — Ability for users to access content away from the office/home.

**Targeted Content** — Business portal administrators can target content by business group area, e.g., HR, Marketing, Legal, Corporate Executives, etc.

Portal technology has proven powerful but complex. Mashups offer the other extreme - simplicity, but not as much power.

### 3.2.2 Mashup

Mashup is a web application, that uses content from more than one source to create a single new service displayed in a single graphical interface. The term implies fast integration, frequently using open application programming interfaces (API) and data sources to produce enriched results that were not necessarily the original reason for producing the raw source data. The term mashup originally comes from pop music, where people seamlessly combine music from one song with the vocal track from another-thereby mashing them together to create something new. The main characteristics of a mashup are combination, visualization, and aggregation. It is important to make existing data more useful, for personal and professional use. To be able to permanently access the data of other services, mashups are generally client applications or hosted online. But to customize retrived resources end-user have no option, as use only predefined type and numbers of applications, that was created by application or platform developer.

The mashup application is a composite Web 2.0 application that aggregates and integrates heterogeneous web resources offered in a form of available Web APIs and sources for creating a new service. Mashups differ from traditional component-based applications in providing more situational character of these applications[30]. In general there distinguish the following three types of mashups[31]. Customer mashups are aimed at the combination and reformation data from various public sources according to users' needs. Data mashups aggregate similar types of resources from diverse sources into a new single data representation. And business, or enterprise, mashups define composite applications that are focused on solving heterogeneous business problems by supporting collaborative activities[32]. The architecture of a mashup is divided into three layers:

- *Presentation / user interaction:* this is the user interface of mashups. The technologies used are HTML/XHTML, CSS, Javascript, Asynchronous Javascript and XML (Ajax).

- *Web Services:* the product's functionality can be accessed using API services. The technologies used are XMLHTTPRequest, XML-RPC, JSON-RPC, SOAP, REST.

- *Data:* handling the data like sending, storing and receiving. The technologies used are XML, JSON, KML.

Concerning architectural styles of mashup applications, server-side and client-side mashups are distinguished. In server-side mashups a content aggregation is realized on a server[33]. The opposite client-side mashups aggregate content on a client, typically, at a client's web browser[34]. Server-side mashup is similar to traditional Web applications using server-side dynamic content generation technologies. The data from multiple sources are aggregated at the server side and the final results are rendered at the client's browser[35].

**Portal vs Mashup**

Mashups differ from portals in the following respects:

| | **Portal** | **Mashup** |
|---|---|---|
| **Classification** | Older technology, extension to traditional Web server model using well-defined approach | Using newer, loosely defined "Web 2.0" techniques |
| **Philosophy/ Approach** | Approaches aggregation by splitting role of Web server into two phases: markup generation and aggregation of markup fragments | Uses APIs provided by different content sites to aggregate and reuse the content in another way |
| **Content Dependencies** | Aggregates presentation-oriented markup fragments (HTML, WML, VoiceXML, etc.) | Can operate on pure XML content and also on presentation-oriented content (e.g., HTML) |
| **Location Dependencies** | Content aggregation takes place on the server | Content aggregation can take place either on the server or on the client |
| **Aggregation Style** | "Salad bar" style: Aggregated content is presented "side-by-side" without overlaps | "Melting Pot" style – Individual content may be combined in any manner, resulting in arbitrarily structured hybrid content |
| **Event Model** | Read and update event models are defined through a specific portlet API | CRUD operations are based on REST architectural principles, but no formal API exists |
| **Relevant Standards** | Portlet behavior is governed by standards JSR 168, JSR 286 and WSRP, although portal page layout and portal functionality are undefined and vendor-specific | Base standards are XML interchanged as REST or Web Services. RSS and Atom are commonly used. More specific mashup standards such as EMML are emerging. |

Table 3.2: Portal vs Mashup Technologies

Mashups and portals are both content aggregation technologies. Portals are an older technology designed as an extension to traditional dynamic Web applications, in which the process of converting data content into marked-up Web pages is split into two phases: generation of markup "fragments" and aggregation of the fragments into pages. Each markup fragment is generated by a "portlet", and the portal combines them into a single Web page. Portlets may be hosted locally on the portal server or remotely on a separate server. Portal technology is about server-side, presentation-tier aggregation. It cannot be used to drive more robust forms of application integration such as two-phase commit.

The portal model has been around longer and has had greater investment and product research. Portal technology is therefore more standardized and mature. Over a time, increasing maturity and standardization of mashup technology will likely make it more popular than portal technology because it is more closely associated with Web 2.0 and lately Service-oriented Architectures (SOA). New versions of portal products are expected to eventually add mashup support while still supporting legacy portlet applications. Mashup technologies, in contrast, are not expected to provide support for portal standards. Another possibility is that the overall concept of portals is replaced by new technologies like Ajax widget libraries and even JavaFX. Many of the limitations of JSR-168 portlets seem quaint now that we are in the age of the interactive Web 2.0 or even Web 3.0.

### 3.2.3 Non-Browser Based Systems

Another synonym of a non-browser based system is a native application, that is targeted to create software specificaly for operation system used on a device. It is become popular when usual mobile phone started to have parts of computer functionality, e.g. smartphones, tablets. Of course enormous numbers of portable devices increased number of operation systems, requirements and restrictions. But native application always provide best user experience and possibility to use all resources of a mobile device. Android and Apple's iOS have the greatest market share, but there are others, including the Blackberry and Windows Phone operating systems. Developing native apps involves targeting one or more of these platforms, each of which has its own software development kit (SDK)[5]. A native mobile app is a smartphone application that is coded in a specific programming language, such as Objective C for iOS and Java for Android operating systems. Native mobile apps provide fast performance and a high degree of reliability. They also have access to a phone's various devices, such as its camera and address book. In addition, users can use some apps without an Internet connection. However, this type of app is expensive to develop because it is tied to one type of operating system, forcing the company that creates the app to make duplicate versions that work on other platforms.

Rather than being accessed via the Web, native apps are mainly deployed through app marketplaces that are also mostly targeted at particular platforms. These markets allow apps to be downloaded for free or commercially, with the app store taking a percentage cut of sales revenue. Native user interfaces provide an interaction level and quality that currently cannot

[5]Native App,http://www.techopedia.com/2/28134/development/web-development/native-app-or-mobile-web-app

be achieved through a Web app running in a browser. In addition, native app processing can employ mobile device hardware features, such as GPS and other localization facilities, accelerometers and touchscreens. As HTML5 develops, Web apps will also be able to exploit some or all of these features. But for now, these bells and whistles are mostly exclusive to native apps.

A native app also has the ability to use offline data storage. But, the advance of Web technologies, such as HTML5, will begin to close this gap because Web apps will be able to store data for offline use as mobile caching models continue to improve.

The number one disadvantage, for native apps is the amount of resources businesses require to invest in the development process. Each platform has its own framework, and to target more than one involves multiple programming languages - not to mention an understanding of the different application frameworks. In addition to the initial development project, maintenance of native apps is an ongoing concern, as the platforms they are designed to work with are constantly changing.

### 3.2.4 Browser Based Systems

An application that runs within the Web browser (such as Firefox, Internet Explorer or Chrome). The instructions, typically written in a combination of HTML and JavaScript, are embedded within the Web page that is downloaded from a Web site. The advantage of browser-based applications is that they can run in a Windows PC, Mac or Linux device, since all Web browsers are required to render HTML and execute JavaScript in the same manner, no matter their environment. In practice, there are minor differences in page rendering, which are generally tolerable.

One of the main benefit of browser-based applications is that there are no downloads necessary in order to make them run. And no need anymore to reinstall and update software manually. This means that, generally, even users behind firewalls can benefit from using these types of tools[6].

Web applications optimized for mobile use also offer significant benefits for certain projects. This is an area that is set to undergo enormous change over the next few years, particularly through technologies such as HTML5/CSS3 and jQuery Mobile, not to mention improvements in network connectivity. It is clear that, in terms of functionality, they will greatly impact the ability of Web apps[7] to compete with native apps.

The major advantage of using Web apps to deliver services is the simple fact that only one application needs to be developed. Of course, a successful Web app is tested and refined to cope with browser, operating system and hardware differences, but the bulk of application processing remains accessible from any mobile user environment. Mobile browsers are advancing at a fast pace, and the functionality gap between them and their desktop counterparts is gradually narrowing.

[6]Browser based system definition, http://www.pcmag.com/encyclopedia/term/61816/browser-based-application

[7]WEB APPLICATIONS, http://www.w3.org/2008/webapps/

Benefits of browser-based interface:

- Easier to manage: no installation required on user machines, upgrades need only be performed on server side and are immediately available to all users. Data backup can be performed on a single machine as data will not be spread out across multiple clients.
- Application can be accessed from any machine with a browser.
- Can easily support multiple platforms consistently.
- Memory and CPU requirements may be considerably less on the client side as intensive operations can be performed on the server.
- Increased security: data is stored on a single server instead of multiple client machines and access can be better controlled.
- Many other benefits of a centralized environment including logging, data entered from multiple sources can immediately be available from other clients, etc.
- It is often easier to debug and faster to develop web-based solutions.
- They require no upgrade procedure since all new features are implemented on the server and automatically delivered to the users;
- Web applications integrate easily into other server-side web procedures, such as e-mail and searching.
- They also provide cross-platform compatibility in most cases (i.e., Windows, Mac, Linux, etc.) because they operate within a web browser window.
- With the advent of HTML5, programmers can create richly interactive environments natively within browsers. Included in the list of new features are native audio, video and animations, as well as improved error handling.
- Modern web applications support greater interactivity and greatly improved usability through technologies such as AJAX that efficiently exchange data between the browser and the server.
- Web applications allow for easier introduction of new user devices (e.g. smartphones, tablets) because they have built-in browsers.

**Choosen Methodology**

Such a wide range of methodologies and approaches gives an opportunity to mix advantages of a mashup and non-browser based application in a best way. Thus, to satisfy one of the main requirement about dynamic user-friendly interface, adaptable to any kind of device, mashup architecture should be enhanced with a browser-based approach. Based on various design principles, that truly embody a new vision of possibility and practicality[36]. The mashup methodology has a clear and easy adoptable for the web three level structure, described above: presentation, web services and data.

## 3.3 Summary

The Chapter has introduced main approaches of building web-based applications for different types of data source. The Section 3.1 provides list of projects devoted to retrieve sensed data from the web and another resources such as: Cloud, Big Data, personal sensors (temperature, humidity etc.), traffic sensors. Was studied not only scientific research projects but also private business-oriented solutions: Xively and Optique. Based on the frontend type and implementation 12 projects were researched and compared. Concluding all researched projects above, become clear that needs of a generic frontend, that can be easily integrated with any type of data-driven platform is high.

The Section 3.2 consists main methodologies of a design and implementation of a generic frontend: portal with portles, mashup, native application and non-browser based system. As a result, mashup technology within a browser satisfies all necessary requirements to create generic frontend for exploring sensor and information services. In the Chapter 4 based on this decision a system architecture and described responsibility of an every module will be defined.

# Chapter 4

# Concept

The chapter describes a concept of a user-friendly generic frontend for exploring sensor data, going through a software architecture design and a content aggregation of a web-based user interface controlled and provisioned by end-user requests. The concept is developed based on the analysis of the current state-of-the-art, modern technologies and requirements formulated in the Chapter 2.

The Section 4.1 begins this chapter with a software design according to 3-tier architecture, which contains client, application and data tiers. Next sections presents detailed description of an every tier, with corresponding functional modules based on a fine-grained structure. Every part of a system is responsible for providing application functionality of a corresponding tier. Summary of this chapter underlines main responsibility and requirements for every part of the system infrastructure. It clarifies requirements to a prototype implemented in the Chapter 5.

## 4.1 Concept in 3-tier Architecture Projection

Building a system architecture based on a fine-grained structure satisfies one of the important requirement defined in the Section 2.1. Such a structure of a generic frontend should be scalable and easily integrated with any kind of a backend, where every module is responsible for its personal independent task. Thus, deployment of new changes to any module have no influnce on another parts of an architecture. System becomes consistent and reliable. An important task is to determine the software design according to 3-tier architecture, where presentation, application processing, and data management functions are logically separated. The multi-tier architecture provides abstract structure of modules and gives a possibility to define in which concrete module of a system developer is interested in. Also it describes how different parts of frontend are connected with each other and which extensions and integration points for backend are available.
The Figure 4.1 shows the concept infrastructure:

- **Client Tier:** web-based GUI and client framework;

- **Application Tier:** application logic, interface of communication between tiers, back-end integration points;
- **Data Tier:** provides description of data sources based on defined data standard and real-time data streaming of all sensors registered in the system.

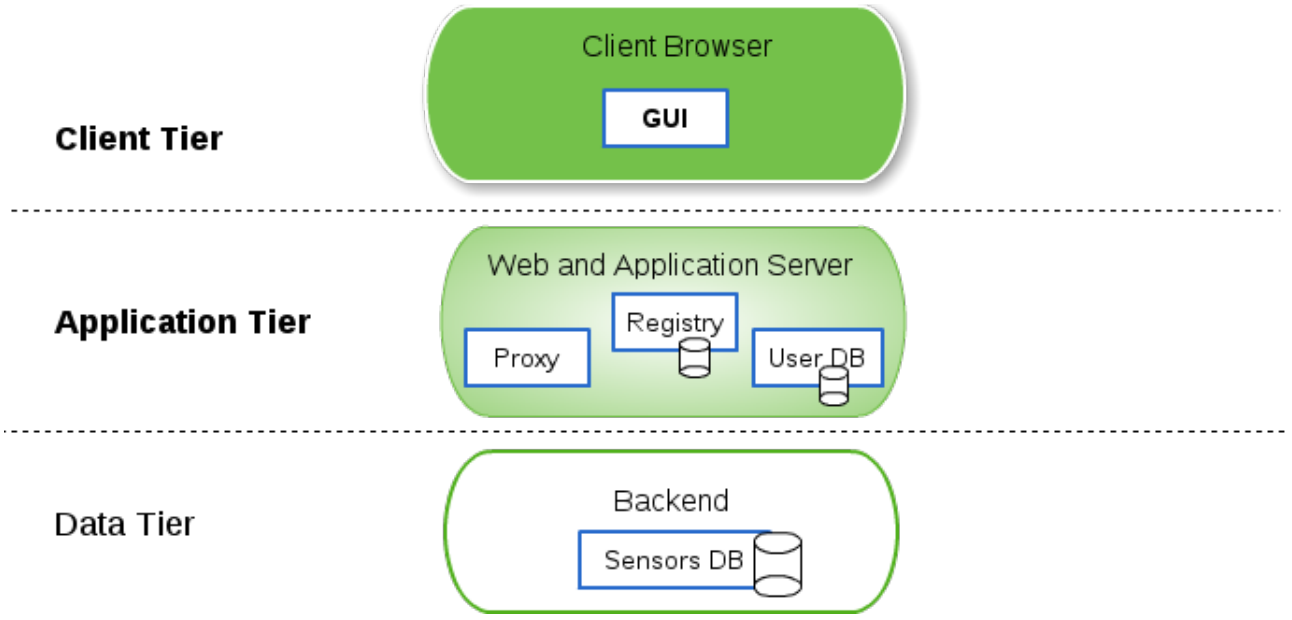


Figure 4.1: 3-tier Architecture

*Client Tier* hosts the presentation layer components. The main function of the interface is to translate tasks and results from application tier into a user-friendly GUI via a client framework. This tier was defined in order to satisfy next requirements to the concept: cross-platform application, usability properties and responsiveness of a concept, which is formulated in the Section 2.1.

*Application Tier* includes business logic and data access tiers. It controls an application's functionality by performing detailed processing, transformation of a one type data to another, defines an interface between the client tier and the data tier. Besides possessing application logic between two another tiers of infrastructure, this tier also contains integration point with a backend system. Loose coupling and multi-user binding is discovered and implemented in this tier.

*Data Tier* contains source of data that has to be retrieved by the application tier to a client tier, by request from a user. This tier keeps data neutral and independent from application server or business logic. Backend generates a description of a data source in a system-defined way and provides an access to the description and data itself through the standard intreface.

From a historical perspective the three-tier architecture concept emerged in the 1990s from observations of distributed systems[37] (e.g., web applications) where the client, application and data tiers ran on physically separate platforms. Nowadays, MVC and similar model-view-presenter (MVP) patterns are used for a separation of concerns. It discovers

design patterns that apply exclusively to the presentation layer of a large system. In simple scenarios MVC may represent the primary design of a system, reaching directly into the database. This way, to ensure highly adaptive GUI independently from a data and application tiers, MVC pattern comes into a picture. As a part of frontend logic it will be described in the next subsection.

The multi-tier architecture model may seem similar to the model-view-controller (MVC) concept. However, topologically they are different. A fundamental rule in a three tier architecture is the client tier never communicates directly with the data tier; in a three-tier model all communication must pass through the middle tier. Conceptually the three-tier architecture is linear. However, the MVC architecture is triangular: the view sends updates to the controller, the controller updates the model, and the view gets updated directly from the model.

The next section describes every functional module of a respective tier according to 3-tier architecture.

## 4.2 Client Tier

The client or presentation tier a layer which user can directly access from any type of portable device. In the Section 3.2 the necessity to implement web based portable application was proved, so that a user can access it by using browser. This tier consists user-friendly GUI which includes widgets structured according to the responsive layout and client framework. First of all, client tier gives an overview of a design layout (Fig.4.2), content provided by data source and managment panel (e.g. technical details of a system architecture such as: end-points configuration, API documentation or SDK downloads). Secondary, it contains a client-based library to bind client and application tier. This tier also responsible for adaptation of a GUI to any kind of mobile or desktop devices.

### 4.2.1 Web-based GUI Composition

The Figure 4.2 presents a simple content layout that has to be presented on a web-page in order to satisfy all possible user requirements. It contains:

- *Main navigation tabs*: a sensors list contains a list of all available sensors; subscriptions - show all data sources to which subscribed by a user; favorites tab saves favorite data sources among already subscribed; in the settings tab a user can manage own profile and also add new sensors by using corresponding form; and the admin references tab clarify steps needed to be established in order to develope own application.

- *Log in form* with user name and password. After user logged in, the system defines his/her rights and applies visibility rules according to assigned role. All users can explore description of every data retrieved by system, but only after subscription to a sensor it becomes possible to get real-time streaming data. Users that have admin

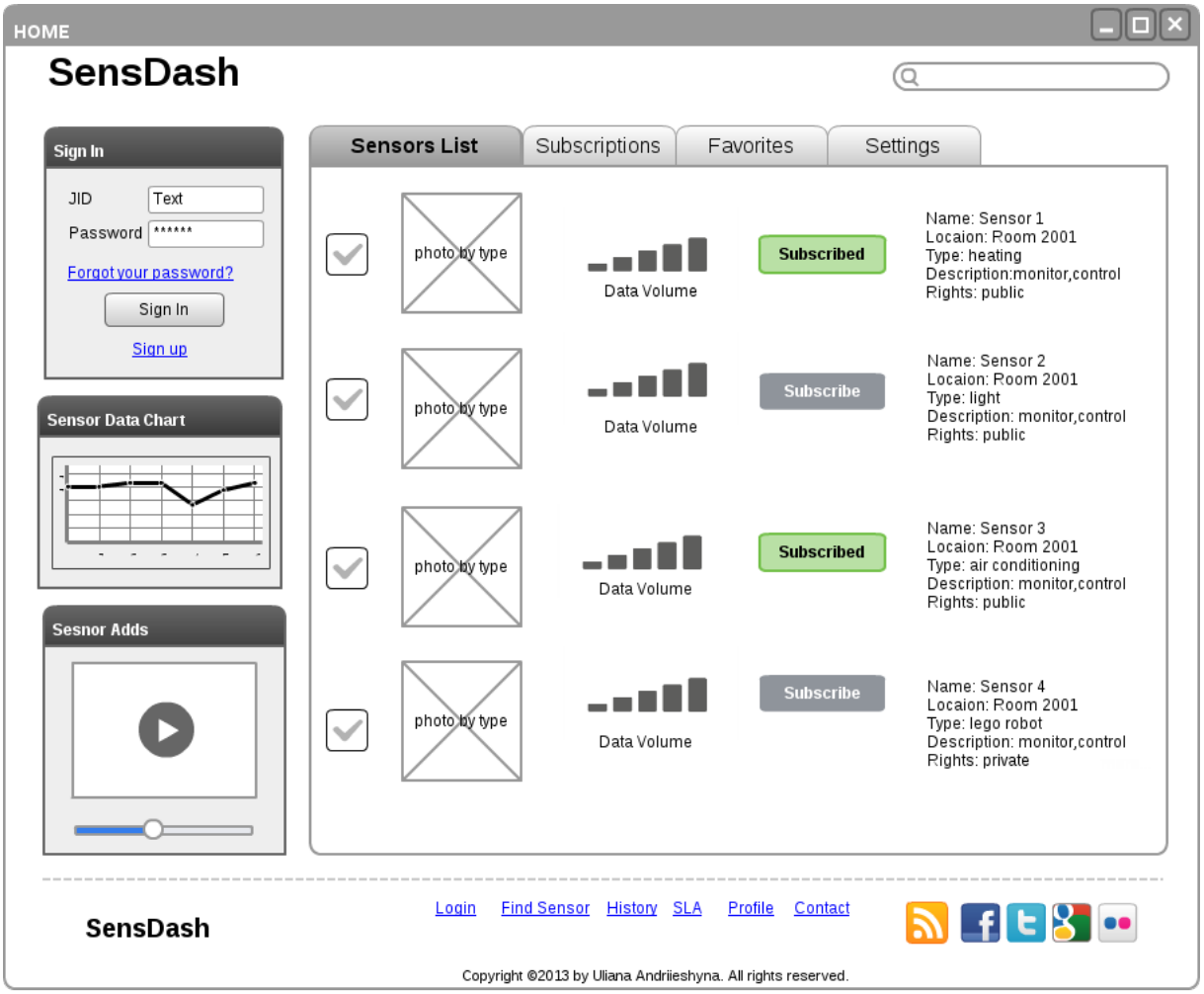


Figure 4.2: GUI Mockup

rights get an interface to manage sensors. Simple user without privileges gets an interface to receive statistic and information from sensors and to maintain his/her own account data.

- *Sensor icon* defines what the current type of sensor is, e.g. light, temperature, heating, robot lego battery status etc. It helps easily and quickly understand what is the main function of a sensor in the list.

- *Availability or unavailability* of a data source. User can subscribe only to services which are online. If some services become offline it will be automatically marked as inactive and after page reload will be deleted from the list of available sensors. As soon as new data will be sent, a user will immendiatyely see it on the subscriptions tab. If user has already subscribed to any sensor, this sensor automatically added to a list/tab of subscriptions made by user. Also a user can define hierarchy in which sensor information has to be displayed. It is done by using "favorite" label/tab. It helps user

to receive information from a sensor in a fast way.

- *Data Volume icon* shows the average data stream volume needed to retrieve sensor data (Kb/s). User can define his possibilities of getting such type of streaming data according to his Internet connection. Ideally, the dashboard should automatically adapt quality of streaming data based on connection throughput. Not only data volume depends on quality of a service itself, but also security level, reliability and performance. Such type of data description can be substituted by most relative icons such as: "lock" icon to define security level or appearance of a reliability label.

- *Description and preview.* The best way to give a user full information about data source is to provide a preview or examples of source data. It is not only description but also real-time drawing graphics, real example of video or audio, images etc.

- *Access and providers.* Based on provider of a data source, data can be private or public. For public type of data a user do not have to accept any SLA to subscribe to sensor. But for private data it is important to accept SLA between subscriber and provider before user will get any real data.

- *Search panel.* Need to filter and search between available sensors, which only based on information available for client tier. Without any queries to application or data tier.

The general use case is shown on the Figure 4.3. User can use any type of mobile device and his favorite browser to receive information from data sources (sensors) by using web-page as a dashboard. Once a user logs in to the dashboard, he/she can explore all available sensors. If he/she is already a user of the dashboard all his/her preferences will be loaded from a server automatically and appear in respective tabs.

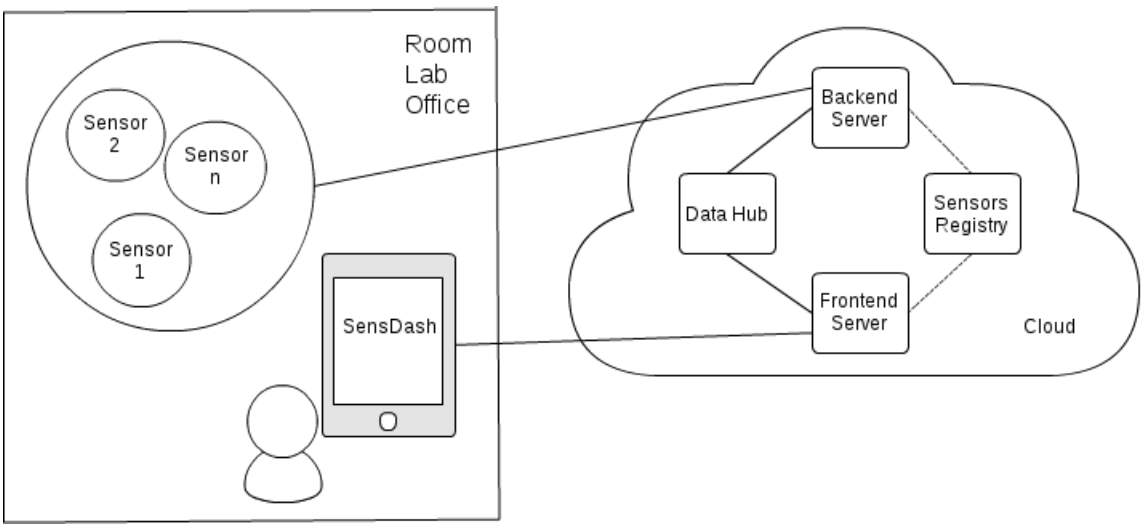


Figure 4.3: Use Case

### 4.2.2 JavaScript MVC

As it was mentioned on the begining of the section, the second responsibility of a client tier is to bind application and client tiers. It can be done by using a client-based framework which is based on the MVC pattern.

In the design shown on the Figure 4.4, a common Model-View-Controller pattern takes place. Model represents the application object that implements the application data and business logic. The View is responsible for formatting the application results and dynamic page construction. The Controller is responsible for receiving the client request, invoking the appropriate business logic, and based on the results, selecting the appropriate view to be presented to the user. The Model represents data and the business rules that govern access to and updates to this data. A View renders the contents of a Model. It accesses data through the Model and specifies how that data should be presented. It is the View's responsibility to maintain consistency in its presentation when the Model changes. This can be achieved by using a push Model, where the View registers itself with the Model for change notifications, or a pull Model, where the View is responsible for calling the Model when it needs to retrieve the most current data. A Controller translates interactions with the View into actions to be performed by the Model. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web application, they appear as GET request. The actions performed by the Model include activating business processes or changing the state of the Model. Based on the user interactions and the outcome of the Model actions, the Controller responds by selecting an appropriate View.

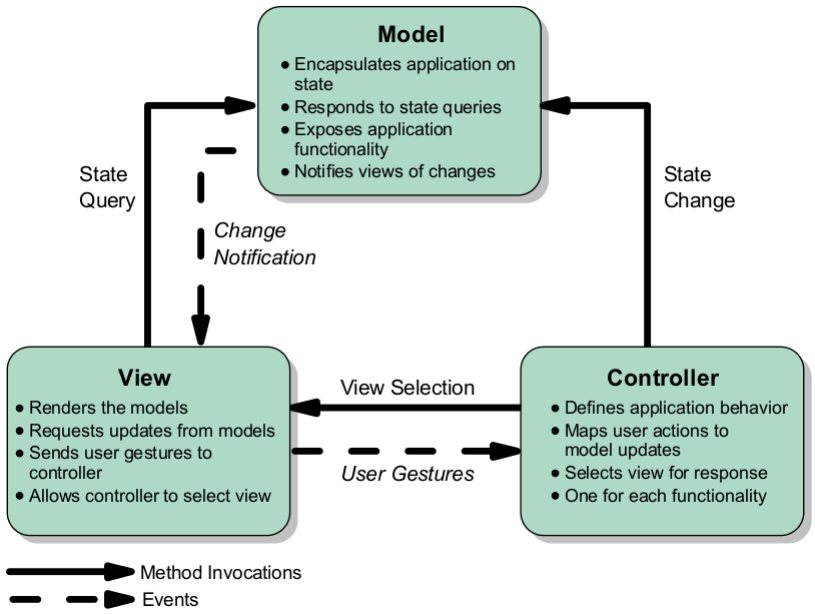


Figure 4.4: MVC Pattern

Not necessary to follow the MVC pattern strictly. The idea is to separate Model, View and application logic for the best separate of concerns.

## 4.3 Application Tier

This layer coordinates commands of processes, makes logical decisions and performs calculations. It also moves and processes data between the two surrounding layers.

Application tier contains all logical modules: Web server, Registry, Data Hub and Web-based Frontend. All these modules connect to each other as shown on the Figure 4.5.

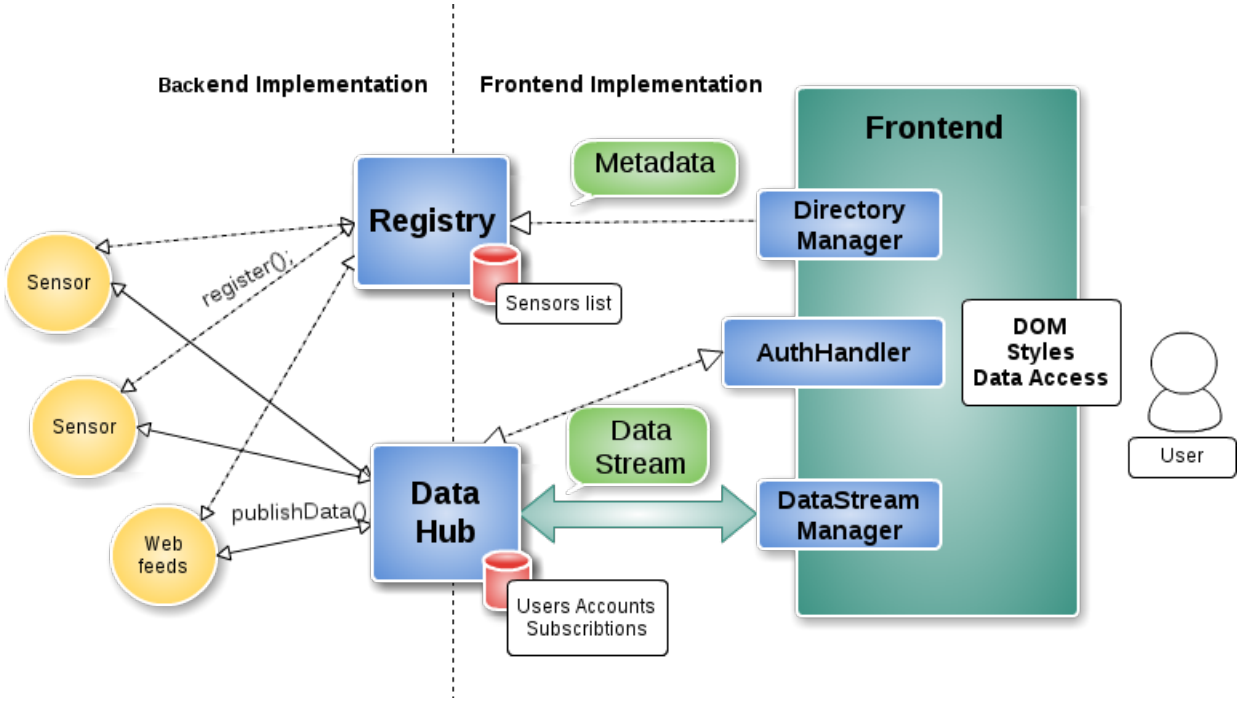


Figure 4.5: System Architecture

The System Architecture is splitted to a backend and frontend, that gives an overview of how these two parts are connected. Respectively "backend/frontend implementation" means that all modules of each side have to be implemented separately. Such module as Registry and Data Hub defined and standardized by frontend, in order to clarify interface of collaboration with backend. Implementation of Registry and Data Hub modules relate to the backend responsibility.

### 4.3.1 Registry

The Registry is a module responsible for storing an info about all registered sensors in order to provide descriptional overview to a user. The Frontend's GUI requests and aggregates information from the Registry about data sources registered in it and dynamically presents it to the user. The Registry contains next necessary attributes: unique id of a sensor, title, description, availability, private or public access description, data provider, service-level agreement (SLA) and its last update time, number of end-points available for one sensor and data format. JSON format fully satisfies described metadata format. The type of connection interface between the Registry and the frontend will be defined in the Section

4.3.3. Structured sensors' attributes within JSON format describes sensors metadata and gives an opportunity to transform it automatically to the graphical container of as a part of the GUI. Registries which provides defined standard can be easily added to the frontend in a runtime. The Registry specifies simple interface schema in JSON format to specify all available attributes and properties. It is a lightweight format, native to browser and much simpler to parse than XML. It separates metadata of a sensor from a real-time data stream.

Before publishing data to the Registry, a data publisher, which is a part of a backend, should specify all required fields based on a description of a sensor. An important attribute of a sensor is its "id", which in order to avoid inconsistency between different Registries must be unique for every data source. Id can be an alphanumeric scring of arbitary length.

To get an access to a real-time data streams a user have to be subscribed to it. The process of subscription to a public and private data sources differ. Sensors with a public access do not require explicit SLA accepting, while private data sources obligate a user to accept provider SLA before getting any real-time data. Once a user accepts the corresponding SLA real-time data becomes available as long as the SLA is up-to-date. When the provider of a data source makes changes in the SLA, a user has to be immediately notified and to avoid SLA disagreements automatically unsubscribed from a sensor. The implementation details are described in the Section 5.3.2.

End-points for a sensor handle real-time data streaming and have to be sctuctured in a heuristic order, in order to switch end-points when another one fails. The number of sensor end-points for data streaming is not limited.

### 4.3.2 Data Hub

Since the the Registry responsible for collecting metadata of sensors, the Data Hub is responsible for mapping interface of particular sensor data stream format into a format supported by the frontend and delivered through the common universal protocol. It means that the Data Hub has to satisfy next requirements:

- be aware of a metadata provided by the Registry to the frontend;
- bind metadata from the Registry with real-time data streaming from a sensor;
- get and parse sensor streaming data and reconvert it to the type supported by universal protocol;
- implement universal protocol to provide exchange message with a server in order to retrieve streaming data from a sensor;
- store history from sensors and user personal preferences.

A GUI depends a lot on user personal preferences. It may contain sensor subscriptions list, favorite data sources, user profile and list of available sensors. All these data has to be stored in the Data Hub and loaded after authentication process was sucessfully passed.

### 4.3.3 Web-based Frontend

#### Web server

The primary function of a web server is to deliver web content to clients. The communication between client and server takes place using the Hypertext Transfer Protocol (HTTP).

In proposed concept Web server is responsible for robust and efficient serving of static files (*.html, *.css, *.js etc.). The goal is to exclude dependencies on concrete backend platforms or frameworks and to provide generic frontend as an easily plugable component. Such a common and simplified design makes it possible to extend and scale every part of a distributed system independently. Specific operation logic like authentication of user, registration of sensors and users are delegated to external components such as Registry, Data Hub and Authentication Hadler (AuthHandler), these external components can be interchanged without dependency to the system itself.

#### AuthHandler

Authentication Handler (AuthHandler) is responsible for logging in and optionally registering a user in the the system. After a user gets necessary ID and confirms his/her personality using password and name, system automatically applies visibility rules. After verification and confirmation of credentials, stored on Data Hub, it becomes possible to bind user ID with personal preferences. These preferences include: user subscriptions, favorites, social sharing information and session data.

#### Interfaces

On the Figure 4.5 exist 3 communication channels:

- Registry to Directory Manager (one-way connection);
- Data Hub to/from DataStream Handler (asynchronous duplex connection);
- Data Hub to/from AuthHandler (synchronous duplex connection).

**Registry to Directory Manager Interface**

The Registry contains pairs of attributes – values, which describe sensors. These type of data can be structured by using JSON format and retrieved by Directory Manager. The Directory Manager can send HTTP GET request to the Registry and get a list of availale sensors with their metadata. Once JSON file is parsed by frontend, values of metadata are extracted and aggregated, resulting with appropriate system and user interface updates. HTTP GET requests and JSON responses are a part of RESTful API approach. It will be called Web API in following text.

If system needs to use more then one Registry, requests will be sent to all of them. It is important to minimize total waiting time at this point, preferring parallel Registry

querying. After getting all JSON lists of sensors, frontend will reparse all received data foming a conbined sensor list and immediately representing it on a web page.

**Data Hub – Directory Manager Interface/AuthHandler**
Communication between the Data Hub and another 2 modules: DataStream Handler and AuthHandler has to be supported through a single universal interface. It has to satisfy next requirements:

- be based on open formats;
- handle HTTP requests to work with browser;
- support multiple data streaming channels in a single HTTP connection;
- support different types of data in one channel, message differentiation;
- keep connection alive, and reconnect in case of failures;
- simplicity of enhancement and customization;
- popuratity within developers.

The Sensors, Data Hub and Frontend communication is shown on the Figure 4.6. Such architecture decouples sensor specific interface and interface between frontend and backend.

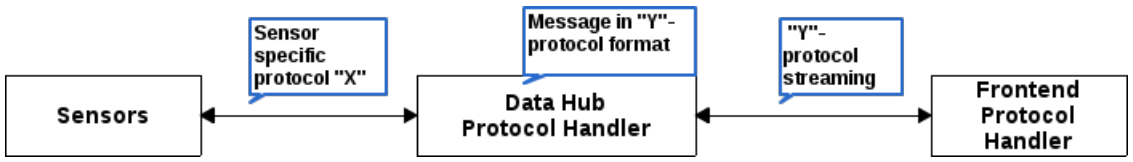


Figure 4.6: Protocol flow

In order to satisfy all aforementioned requirements, two major protocols have been found: *XMPP*[38]:a protocol for extensible messaging with a special case of the device-to-server pattern, since people are connected to the servers and *MQTT*[1]: a protocol for collecting device data and sending it to servers.

**MQTT**
the Message Queue Telemetry Transport, targets device data collection (Fig. 4.7[2]). As its name states, its main purpose is telemetry, or remote monitoring. Its goal is to collect data from many devices and transport that data to the IT infrastructure. It targets large networks of small devices that need to be monitored or controlled from the cloud. MQTT makes attempt to enable device-to-device transfer, nor to "fan out" the data to many recipients. Since it has a clear, compelling single application, MQTT is offering few control options. In this context, "real time" for MQTT is typically measured in seconds. All the devices connect to a data concentrator server. So the protocol works on top of TCP, which provides a simple,

[1]MQ Telemetry Transport, `http://mqtt.org/`
[2]What is MQTT?, `https://www.ibm.com/developerworks/`

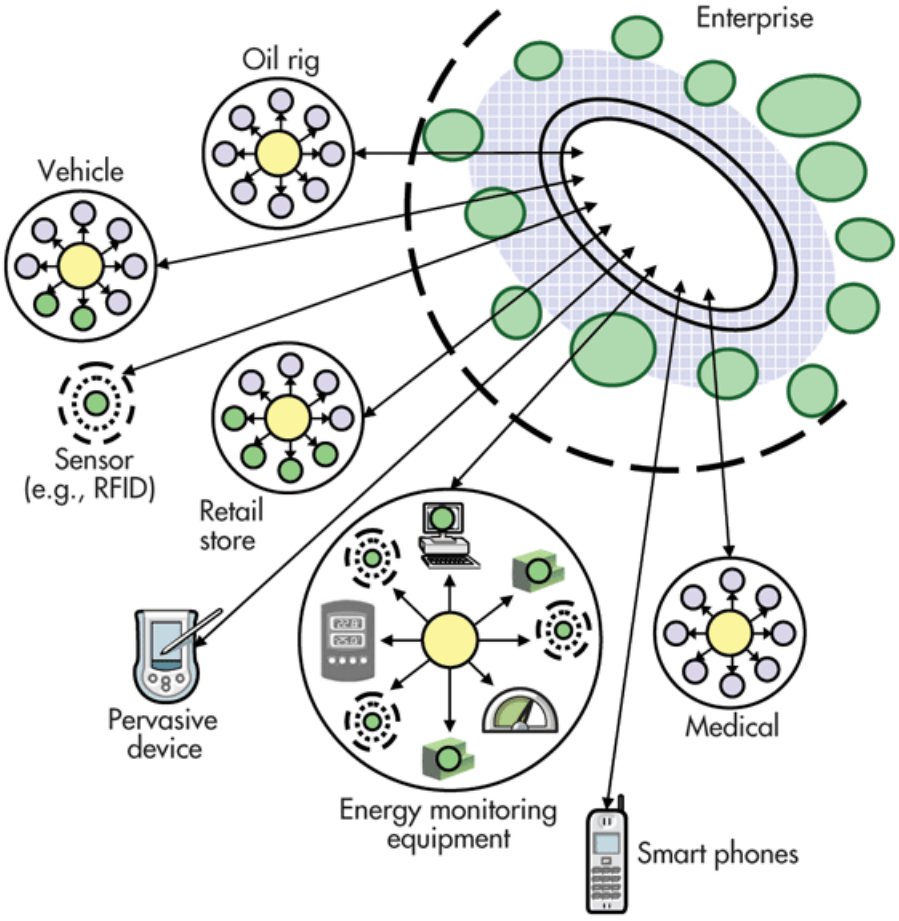


Figure 4.7: Message Queue Telemetry Transport

reliable stream. Since the IT infrastructure uses the data, the entire system is designed to transport data into enterprise technologies.

MQTT enables applications like monitoring a huge oil pipeline for leaks or vandalism. Those thousands of sensors must be concentrated into a single location for analysis. When the system finds a problem, it can take action to correct that problem. Other applications for MQTT include power usage monitoring, lighting control, and even intelligent gardening. They share a need for collecting data from many sources and making it available to the IT infrastructure.

**XMPP**

XMPP was originally developed for instant messaging (IM) to connect people via text messages (Fig. 4.8[3]). XMPP stands for Extensible Messaging and Presence Protocol. The targeted use is people-to-people communication.

XMPP uses the XML format messages, making person-to-person communications natural. Like MQTT, it runs over TCP, or over HTTP on top of TCP. Its key strength is a name@domain.com addressing scheme that helps connect the needles in the huge Internet haystack. XMPP powers a wide range of applications including instant messaging, multi-user

[3]IoT, `http://electronicdesign.com/embedded/understanding-protocols-behind-internet-things`

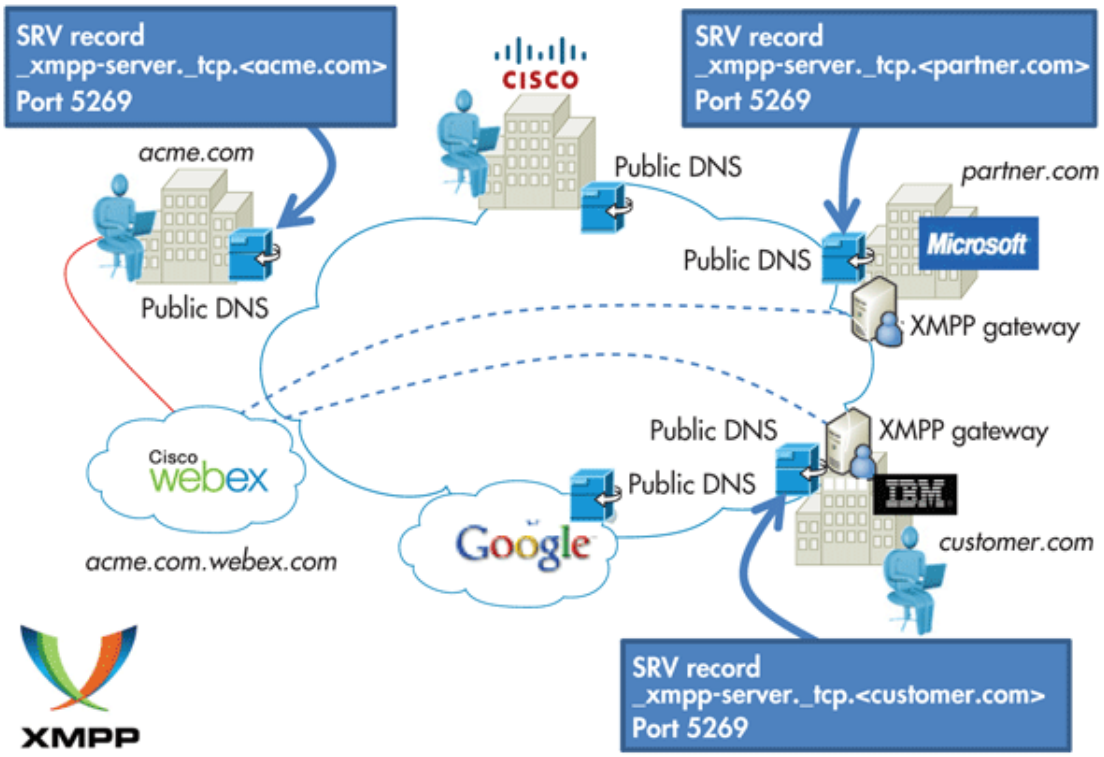


Figure 4.8: The Extensible Messaging and Presence Protocol (XMPP)

chat, voice and video conferencing, collaborative spaces, real-time gaming, data synchronization, and even search. Although XMPP started its life as an open, standardized alternative to proprietary instant messaging systems like ICQ and AOL Instant Messenger, it has matured into an extremely robust protocol for all kinds of messaging purposes.

Most implementations work directly with TCP connections, but *BOSH* extension (bidirectional streams over Synchronous HTTP) lets servers push XMPP data through HTTP long polling sessions, enabling "real time" XMPP experience in web-application. Like HTTP, XMPP is a client-server protocol, but it differs from HTTP by allowing either side to send data to the other asynchronously. XMPP connections are long lived, and data is pushed instead of pulled. XMPP has nearly 200 extensions, providing a broad and useful range of tools on which sophisticated applications can be build.

After a short research XMPP pecularities clearly show that this protocol can fully satisfy all requirements and be used in generic frontend. So the interface *Data Hub to/from Directory Manager Interface/AuthHandler* will be powered by using XMPP. Thus, it should be discovered in details.

XMPP, like all protocols, defines a format for moving data between two or more communicating entities. In XMPP's case, the entities are normally a client and a server, although it also allows for peer-to-peer communication between two servers or two clients. Many XMPP servers exist on the Internet, accessible to all, and form a federated network of interconnected systems. Data exchanged over XMPP is in XML, giving the communication a rich, extensible structure. One major feature XMPP gets by using XML is XML's insensibility. This extensibility is put to great use in the more than 200 protocol extensions registered with

the XMPP Standards Foundation and has provided developers with a rich set of tools. XML is known primarily as a document format, but in XMPP, XML data is organized as a pair of streams, one stream for each direction of communication. Each XML stream consists of an opening element, followed by XMPP stanzas and other top-level elements, and then a closing element. Each XMPP stanza is a first-level child element of the stream with all its descendant elements and attributes. At the end of an XMPP connection, the two streams form a pair of valid XML documents. The Extensible Messaging and Presence Protocol is the IETF's formalization of the base XML streaming protocols for instant messaging and presence developed within the Jabber community[39].

**Pushing Data**

HTTP clients can only request data from a server. Unless the server is responding to a client request, it can not send data to the client. XMPP connections, on the other hand, are bidirectional. Either party can send data to the other at any time, as long as the connection is open. This ability to push data expands the possibilities for web applications and protocol design. Instead of inefficient polling for updates, applications can instead receive notifications when new information is available.

**Pleasing Firewalls**

Some web applications support the use of HTTP callbacks, where the web server makes requests to another HTTP server in order to send data. This would be a handy feature to push data if it were not for firewalls, network address translation (NAT), and other realities of the Internet. In practice it is very hard to enable arbitrary connections to clients from the outside world. XMPP connections are firewall and NAT friendly because the client initiates the connection on which server-to-client communication takes place. Once a connection is established, the server can push all the data it needs to the client, just as it can in the response to an HTTP request.

**Improving Security**

XMPP is built on top of Transport Layer Security (TLS) and Simple Authentication and Security Layer (SASL) technologies, which provide robust encryption and security for XMPP connections. Though HTTP uses Secure Sockets Layer (SSL), the HTTP authentication mechanisms did not see much implementation or use by developers. Instead, the Web is full of sites that have implemented their own authentication schemes, often badly.

**Statefulness**

HTTP is a stateless protocol; XMPP is stateful. Stateless protocols are easier to scale because each server does not need to know the entire state in order to serve a request. This drawback of XMPP is less onerous in practice because most non-trivial web applications make extensive use of cookies, backend databases, and many other forms of stored state. Many of the same tools used to scale HTTP-based applications can also be used to scale XMPP-based ones, although the number and diversity of such tools is more limited, due to XMPP's younger age and lesser popularity.

Main XMPP properties are:

- *Decentralization.* The architecture of the XMPP network is similar to email; anyone can run their own XMPP server and there is no central master server.
- *Open standards.* The Internet Engineering Task Force has formalized XMPP as an approved instant messaging and presence technology under the name of XMPP (the latest specifications are RFC 6120 and RFC 6121). No royalties are required to implement support of these specifications and their development is not tied to a single vendor.
- *History.* XMPP technologies have been in use since 1999. Multiple implementations of the XMPP standards exist for clients, servers, components, and code libraries.
- *Security.* XMPP servers can be isolated from the public XMPP network (e.g., on a company intranet), and strong security (via SASL and TLS) has been built into the core XMPP specifications.
- *Flexibility.* Custom functionality can be built on top of XMPP; to maintain interoperability, common extensions are managed by the XMPP Standards Foundation. XMPP applications beyond IM include group chat, content syndication, collaboration tools, file sharing, gaming, remote systems control and monitoring of geolocation, cloud computing, VoIP and Identity services.

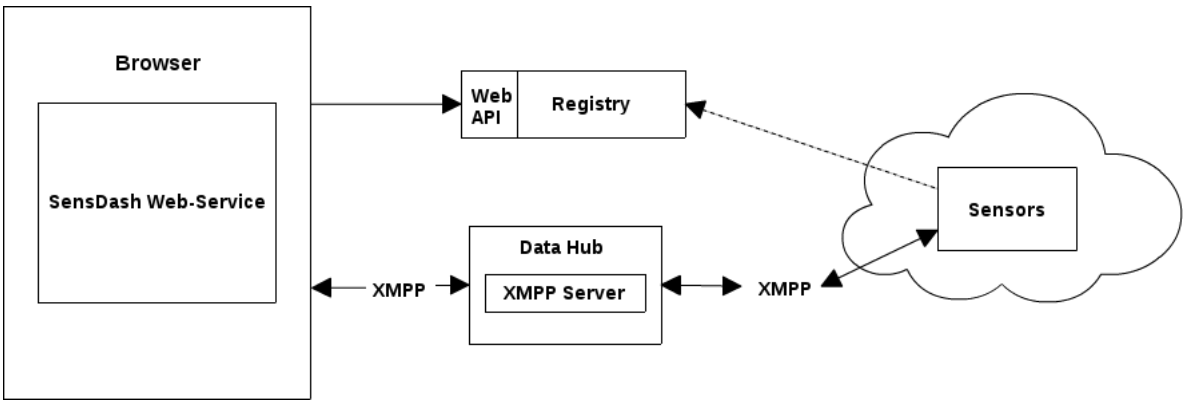


Figure 4.9: Interface

The XMPP network uses a client-server architecture (clients do not talk directly to one another). However, it is decentralized by design, there is no central authoritative server. Anyone may run their own XMPP server on their own domain. Every user on the network has a unique Jabber ID (usually abbreviated as JID). To avoid requiring a central server to maintain a list of IDs, the JID is structured like an email address with a username and a domain name (or IP address) for the server where that user resides, separated by an at sign (@), such as username@example.com.

### 4.3.4 Backend Entry Points

Data Hub and Registry are two modules that have been standardized by frontend and have to be fully implemented on a backend side. Since both of these parts support common standards

such as Web API, AJAX, JSON or XML file format, it makes possible to implement every functional module on a backend side without dependency on OS type, framework or language of implementation. Separation between metadata and streaming data increase scalaility of a system, such that any number of Registries and Data Hubs can be deployed in runtime. Defined Web API and XMPP-based interfaces between frontend's and backend's sides based on open standards. In case of XMPP-based interface, all communication are flows through a distributed XMPP network. Thus, Data Hub needs to have its own XMPP server or sends a requests through any alternative server.

## 4.4 Data Tier

As was mentioned in the Section 4.1 data tier contains data sources that have to be retrieved via application tier to a client tier. Data tier consists of hardware and software sensors, which provide information to a user. The data format which can be retrieved by frontend via XMPP connection has no limitation, but in scope of this master thesis such type of information was defined as text and hashmap of values. Text format is used as an example of information provided by software sensor. In the same time hardware sensor periodically sends measured map o values. These types of data changes with some time-frequency and automatically retrieved by the client tier as a real-time data stream.

An important aspect in streaming data that some of data can be cashed on a server side, thus become possible to retrieve data after it was produced. But sometimes relevant data has to be live, thus there is no other options except of live streaming, where the connection configuration, aliveness and quality become a key aspect. All these properties are already covered by XMPP. Considering the absence of any concrete backend system, cashing of a data can be done based on XMPP server configuration.

**Sensor Functional Characteristics**
An essential part of a concept is to guarantee reliable and secure data transportation. Thus, every data source can acquire additional properties based on a system architecture:

- reliability
- perfomance
- security

All these three characteristics rely on a quantity of available Data Hubs which include XMPP modules and handle data streaming. Since Data Hub has to provide data from sensor to frontend by using XMPP connection, it has to support XMPP data channel configurations and may play a role of end-point for frontend. If sensor has 2 end-points it should be mentioned in Registry together with a type of data transfer protocol (covered in the Section 5.2).

A simple sensor with a low level of reliability has only a single end-point (Figure 4.10 a). A reliable sensor has two, where first one is primary, and next serves as backup/failover

(Figure 4.10 c). A highly reliable sensor has three or more end-points, which guarantee data delivery in case of n-1 endpoints failing. A high-performance sensor has two or more

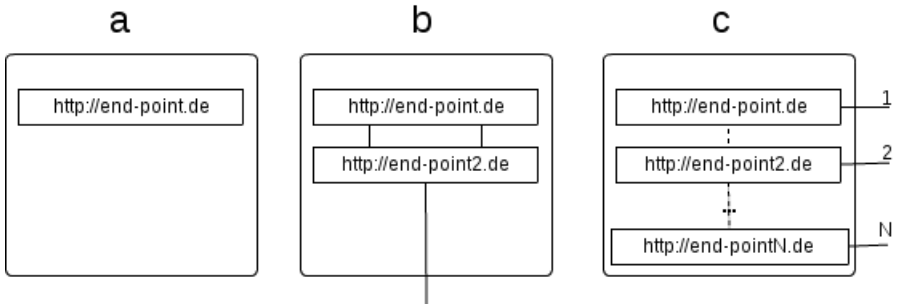


Figure 4.10: Sensor Functional Characteristics

end-points (Figure 4.10 c) working in parralell. In contrast to reliability end-points, high performance does not perform failovers, using all endpoints in the same time to achieve better bandwith and minimize latency.

A secure sensor should be maintained by at least two end-points, each transporting an unrecoverable part of a data (Figure 4.10 b). This way, even by compromising some of endpoints, a full message could not be stolen.

All functional characteristics of sensors should be automatically retrieved by frontend from predefined attributes located in Registries. Users then would be able to estimate conditions and quality of data streams before subscribing to them. Frontend should have a logic which calculates a number sensor end-points, analyses their basic characteristics and graphically represents it using icons and labels, presenting most important technical meta-data in a user-friendly way. In case of end-point problems, frontend should automatically perform a failover to the next available endpoint, if it exists. Developers that want want to use generic frontend as a proxy for their application, should care about correct ordering of end-points, and pay attention to connection process in order to provide high-secure, reliable and high-performance data retrieval.

## 4.5 Summary

In this chapter, according to a 3-tier architecture, the first web-based concept for sensor streaming services has been created. Fine-grained structure provides clear separation of concerns between different modules of the concept. The client tier consists of GUI content and client framework; the application tier provides an application logic to interconnect backend and client tier; and finally, the data tier describes format of a data in order to easily connect it with the application tier and visually represent it by using the client tier. Every data souce was assigned such functional characteristics as reliability, perfomance and security level of information streams. Every characteristic relies on a number of end-points responsible for sensor.

As a result a fine-grained structure of the concept was built on top of next modules:

Registry responsibilities:

- stores an metadata about available sensors registered in the network;
- provides HTTP Web API in JSON format.

Data Hub responsibilities:

- be aware of metadata provided by the Registry;
- bind metadata from the Registry with real-time data streaming from a sensor;
- get and parse sensor streaming data and reconvert sensor-specific protocol to XMPP;
- implement XMPP services and guarantee message exchange between the frontend and sensors;
- store history of data sources and personal user preferences.

Web-server responsibilities: handles delivery of static web content.

Frontend responsibilities:

- interconnect all modules by using appropriate interfaces: Web API for Registry and XMPP interface for Data Hub and AuthHandler;
- build a responsive and adaptive GUI;
- implement a scalable and efficient system structure (adding new Registries/sensors through a web form, changing personal preferences, and other typical frontend functions should productive, user-friendly, and easily extensible).

# Chapter 5

# Implementation and Evaluation

The chapter contains practical part of the work, describing implementation of modules defined in the Chapter 4 within a convincing scenario. The prototype implements major aspects proposed in the concept, including following tiers:

- **Client Tier** includes adaptive GUI with related libraries, modules for dynamic content updates and session management;
- **Application Tier** contains XMPP server with protocol extensions, ensuring appropriate interface for communication between tiers;
- **Data Tier** contains descriptional metadata of sensors and is responsible for streaming text, images or arbitary data provided by heterogeneous sources.

To make precise evaluation, system will use real data from temperature sensor which is provided by ACDSense project, together with TU Dresden, BTU Cottbus-Senftenberg and RWTH Aachen University. It is located in the room INF3084, Faculty of Computer Science, Chair of Computer Science.

## 5.1 Implementation requirements

### 5.1.1 Programming language and libraries

In order to maximize application compatibility, standard web-stack tools have been selected for this work: Javascript for frontend logic, HTML[1] for layout markup, CSS[2] for block styling.

To select additional software tools responsible for css selectors, function chaining, event handlers, AJAX etc, a comprehensive comparison of libraries was made, including most popu-

[1] HTML specification, `http://www.w3.org/wiki/HTML/Specifications`
[2] CSS specification, `http://www.w3.org/Style/CSS/specs.en.html`

lar web toolkits like jQuery[3], Dojo[4], Prototype[5], Yahoo User Interface(YUI) and ExtJS[6], as shown in the Table 5.1.

| Target | jQuery | Dojo | Prototype | YUI | ExtJS |
|---|---|---|---|---|---|
| License | MIT | BSD & AFL | MIT | BSD | GPL and Commercial |
| Size | 32 kB | 41 kB | 46–278 kB | 31 kB | 84–502 kB |
| Dependencies | JavaScript | JavaScript + HTML | JavaScript | Javascript + HTML + CSS | JavaScript |
| Layout Grid | yes | yes | yes | - | yes |
| DOM wrapped | yes | yes | yes | no | yes |
| Data retrieval formats | XML, HTML | XML, HTML, CSV, ATOM | - | yes | XML |
| Server push data retrieval | yes | yes | - | via plugin | yes |
| Touch events | with plugin | yes | yes | - | yes |

Table 5.1: Comparison of JavaScript frameworks

Also the most important part is browser support which is presented in the Table 5.2 . jQuery[7], Dojo[8], Prototype[9], YUI[10], ExtJS[11].

| Target | jQuery | Dojo | Prototype | YUI | ExtJS |
|---|---|---|---|---|---|
| Chrome | 1+ | 3 | 1+ | - | 10+ |
| Opera | 9+ | 10.50+ | 9.25+ | 10.0+ | 11+ |
| Safari | 3+ | 4 | 2.0.4+ | 4.0 | 4+ |
| Mozilla Firefox | 2+ | 3+ | 1.5+ | 3+ | 3.6+ |
| Internet Explorer | 6+ | 6+ | 6+ | 6+ | 6+ |

Table 5.2: Browser Support

Considering aforesaid evaluation, jQuery library has been selected as a main Javascript dependency. The goal of such software components selection is to make resulting code more short, readable, and easier to support for other developers.

[3]jQuery Javascript library, http://jquery.com/
[4]Dojo documentation, http://dojotoolkit.org/features/
[5]Prototype documentation, http://prototypejs.org/
[6]ExtJS documentation,http://docs.sencha.com/extjs/4.2.2/
[7]jQuery browser support, http://jquery.com/browser-support/
[8]Dojo browser support,http://livedocs.dojotoolkit.org/releasenotes/1.4
[9]Prototype browser support, http://prototypejs.org/doc/latest/Prototype/Browser/index.html
[10]YUI browser support, http://yuilibrary.com/yui/environments/
[11]ExtJS browser support, http://www.sencha.com/products/extjs/

### 5.1.2 Frontent Frameworks

#### Integrating CSS toolkit

Twitter Bootstrap was selected as one of the most popular and widely used css frameworks nowadays, offering basic style and usability components for web pages, such as responsive CSS grid, adaptive class mixins, various widgets, etc.

It consists of four main modules:

1. Scaffolding – global styles, responsive 12-column grids and layouts. Has some expressive features like tablets and mobile grids which maintain the grid column structure instead of collapsing the grid columns into individual rows when the viewport is below 768 or 480 pixels wide.

2. Base CSS – this includes fundamental HTML elements like tables, forms, buttons, and images, styled and enhanced with extensible classes.

3. Components – collection of reusable components like dropdowns, button groups, navigation controls (tabs, pills, lists, breadcrumbs, pagination), thumbnails, progress bars, media objects, and more.

4. JavaScript – jQuery plugins which bring the above components to life, and adding transitions, modals, tool tips, popovers, scrollspy, carousel, typeahead, affix navigation, and more.

It was decided to use first three modules for GUI development, maximally reducing Javascript dependencies. All animations, appearence, and dynamic adaptivity was done by using special tags, anchors and classes.

Examples of used modules are shown on a screenshots in the section 5.3.1. It contains buttons, navigation tabs bar, log in form, search field, 4/3/2-columns grid layout, modals, tooltips and carousel for previews.

#### Integrating JavaScript MVC

As mentioned in the section 4.3.3, it is important to implement the system in a loosely-coupled way. Visualization, user management, content retrieving and data aggregation modules have to be separated and accessable through strictly defined interfaces.

In order to structure the code and enforce module decoupling, AngularJS[12] framework was selected. It assists running single-page application with a goal to augment it with model–view–controller (MVC) capability, making development, testing and support simpler.

Angular.js parces HTML that contains additional custom tag attributes; it then obeys the directives in those custom attributes, and binds input or output parts of the page to a model represented by standard JavaScript variables. The values of those JavaScript variables

[12] AngularJS, `http://angularjs.org/`

can be manually set, or retrieved from static or dynamic JSON resources[40]. AngularJS is a toolset for building the framework most suited to application development. It is extensible and works well with other libraries such as jQuery. Every feature can be modified or replaced to suit unique development workflow and feature needs.

The framework adapts and extends traditional HTML to better serve dynamic content through two-way data-binding (Figure 5.1) that allows automatic synchronization of models and views. As a result, AngularJS deemphasizes DOM manipulation and improves testability.

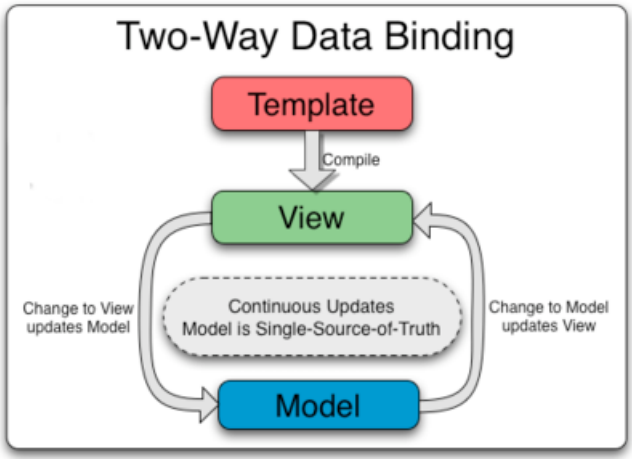


Figure 5.1: Two-way data binding[13]

*Design goals*

Decouple DOM manipulation from application logic. This improves the testability of the code. Decouple the client side of an application from the server side. This allows development work to progress in parallel, and allows for reuse of both sides. Angular follows the MVC pattern of software engineering and encourages loose coupling between presentation, data, and logic components. Using dependency injection, Angular brings traditional server-side services, such as view-dependent controllers, to client-side web applications. Consequently, much of the overheads on the backend is reduced, leading to much lighter web applications.

*Two-way data binding*

AngularJS two-way data binding is a most notable feature and reduces the amount of code written by relieving the server backend from templating responsibilities. Instead, templates are rendered in plain HTML according to data contained in a scope defined in the model. The $scope service in Angular detects changes to the model section and modifies HTML expressions in the view via a controller. Likewise, any alterations to the view are reflected in the model. This circumvents the need to actively manipulate the DOM and encourages bootstrapping and rapid prototyping of web applications.

The way Angular templates works is different, as illustrated on the Figure 5.2. They are different because first the template (which is the uncompiled HTML along with any additional markup or directives) is compiled on the browser, and second, the compilation step produces a live view. Any changes to the view are immediately reflected in the model, and any changes in the model are propagated to the view. This makes the model always the

single source for the application state, simplifying the programming model for the developer. View is therefore an instant projection of a model.

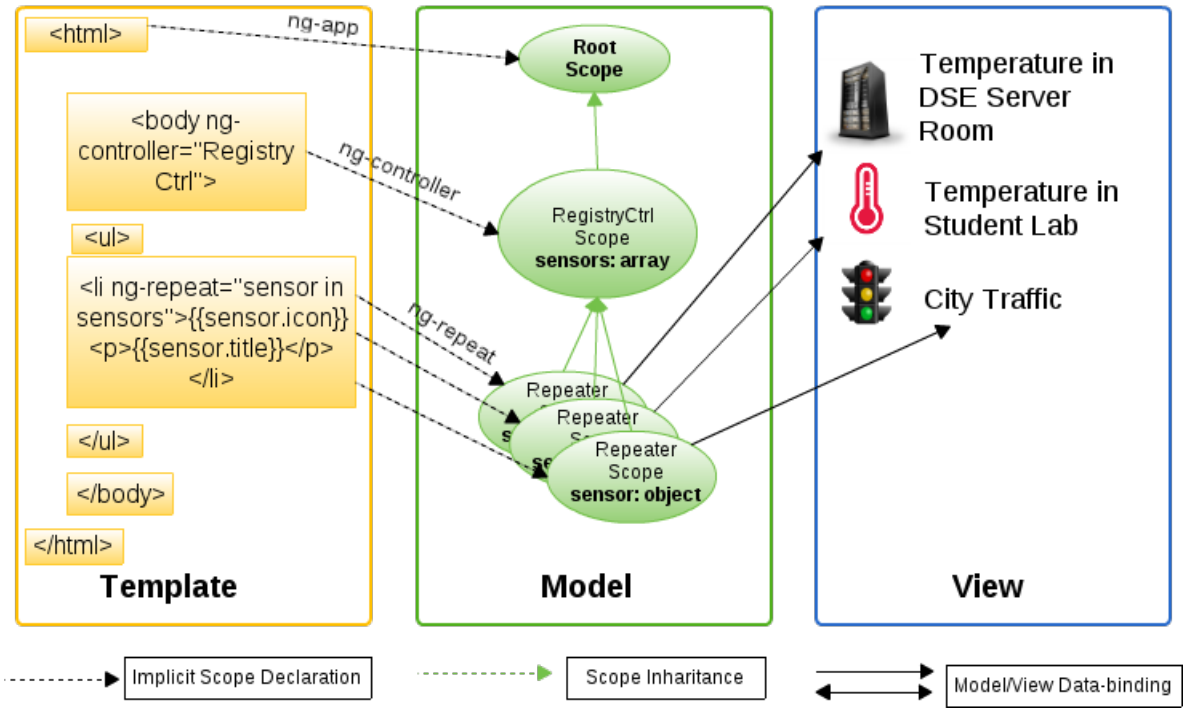


Figure 5.2: Template Model View

Because the view is just a projection of the model, the controller is completely separated from the view and unaware of it. As shown on the Figure above, the resulting view can be applied to every data available on a backend. Model handler automaticaly generates a view for every new sensor. No need to change code or add new id and dependent handlers, variables, channels.

The template declaire the structure of what have to be shown on user view. Model repeats to generate the same view of every sensor in the list, which is coming from a controller as parameter sensor with possible attributes sensor.icon and sensor.titel, till the list of registered sensors will be finished.

The code of such flow is shown on the Listing 5.1.

```
1  <div id="sensor_list">
2      <div class="grid-sizer"></div>
3      <div class="masonry-brick sensor-wrapper" id="{{sensor.id}}" ng-repeat="
           sensor in sensors | filter:{title: query}">
4          <div class="sensor" ng-controller="RegistryCtrl" ng-click="open()">
5              <div class="icon">
6                  <img class="img-responsive" ng-src="{{sensor.icon}}">
7                  <h4>{{sensor.title}}</h4>
8                  <span class="label label-success" ng-show="user.
                       check_subscribe(sensor.id)">Subscribed</span>
9              </div>
10             <div ng-show="sensor.picture">
11                 <img class="img-responsive" ng-src="{{sensor.picture}}">
```

```
12              </div>
13              <span class="description">{{sensor.description}}</span>
14          </div>
15      </div>
16  </div>
```

Listing 5.1: Template registry.html

Model is explicitly integrated to the HTML by using directives: ng-repeat, ng-src, ng-click and sensor prototype attributes sensor.*, as shown on the listing 5.1. Next, listing 5.2 shows the basic implementation of "RegistryCtrl" controller.

```
1  var sensdash_controllers = angular.module("sensdash.controllers", []);
2
3  sensdash_controllers.controller("RegistryCtrl", ["$scope", "Registry", "User",
4      function ($scope, Registry, User) {
5          Registry.load().then(function(sensors){
6              $scope.sensors = sensors;
7          });
8          $scope.user = User;
9      }]);
```

Listing 5.2: Registry Controller

### 5.1.3 XMPP support

#### Message broadcasting protocol

In scope of this Master Thesis some limited conferencing functionality is required to broadcast messages from sensors to users. This functionality is partially covered in XMPP extensions. Two main extensions: MUC and PubSub will be described next, followed by decision of protocol support.

**XEP-0045: Multi-User Chat**
Multi-User Chat (MUC), is a standard XMPP conference protocol, supporting features like invitations, message presence, room moderation and administration, and specialized room types[14].

Each room is identified as a "room JID" <room@service> (e.g. <sensor@conference.tu-dresden.de>), where "room" is the name of the room and "service" is the hostname at which the multi-user chat service is running. Each occupant in a room is identified by "occupant JID" <room@service/nick>, where "nick" is the room nickname of the occupant as specified on entering the room or subsequently changed during the occupant's visit. A user enters a room (i.e. becomes an occupant) by sending directed presence to <room@service/nick>. An occupant can change the room nickname and availability status within the room by sending presence information to <room@service/newnick>. Messages sent within multi-user chat rooms are of a special type"groupchat" and are addressed to the room itself (room@service),

[14]XEP0045, http://xmpp.org/extensions/xep-0045.html

then reflected to all occupants. An occupant exits a room by sending presence of type "unavailable" to its current <room@service/nick>.

The MUC conference has next structure(Figure 5.3):

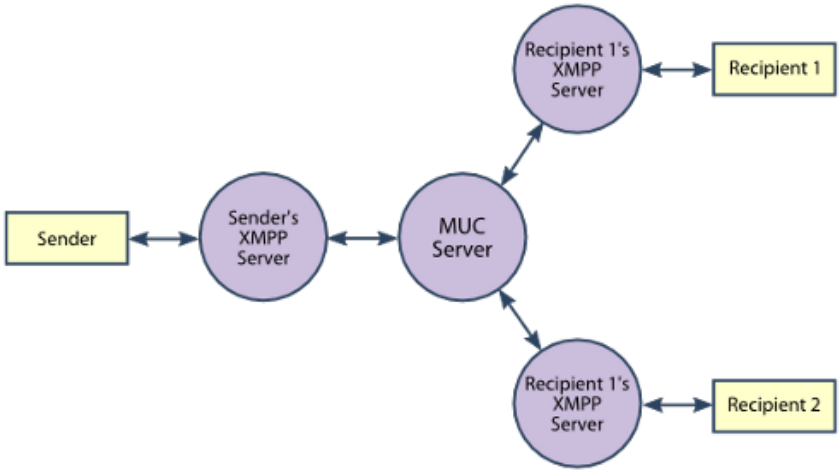


Figure 5.3: MUC System structure

Group chat is provided as a service, usually having own domain. Each room on the group chat service gets its own address, which looks like a JID. Rooms can have access controls, moderators, administrators, and automatic logging and archival of the group communications.

*Entering and Leaving a Room*
Joining and leaving a room is done using <presence> stanzas. Users can join a group chat room by sending available presence to the room. Similarly, to leave, unavailable presence is sent to the room.

If a user wants to join the group chat room with the Temperature sensor, they will both need to send directed presence to their desired identity in the room temperature@chat.sensor.lit. Their stanzas are shown in the Listing 5.3:

```
<presence to="temperature@chat.sensor.lit/sensor"
          from="sensor@example.lit/sensor">
    <x xmlns="http://jabber.org/protocol/muc"/>
</presence>
```

Listing 5.3: Stanzas Format for MUC

Once they have joined the room, the group chat service will broadcast all the other participants' presence statuses to them. After all the other participants' presence stanzas are sent, the server concludes the presence broadcast by sending the arriving participant's presence to everyone, including the new arrival. Thus, when a new participant sees their own presence broadcast back to them, they know they have fully joined the room.

The room sends the affiliations and roles of each participant along with their presence. Sensor's own presence broadcast also includes a status code of 110, which signals that this presence refers to the sensor itself. Just as with presence updates from sensor's roster, sensor

will also receive presence updates from the room as people leave and new people join on the listing 5.4.

```
<presence to="sensor@example.lit/sensor"
          from="temperature@chat.sensor.lit/sensor">
  <x xmlns="http://jabber.org/protocol/muc">
      <item affiliation="member" role="participant"/>
      <status code="110"/>
  </x>
</presence>
```

Listing 5.4: Server Presence Notification

*Creating Rooms*

Creating rooms is accomplished in the same manner as joining a room. Assuming the service allows the user to create new rooms, sending directed presence to the desired room JID of the new room will cause the room to be created and the user to be set as the room's owner. On the Listing 5.5, sensor creates a new room for the News feed.

```
<presence to="chatter@chat.news.lit/sensor"
          from="sensor@news.lit/drawing_room">
    <x xmlns="http://jabber.org/protocol/muc"/>
</presence>
```

Listing 5.5: MUC Room Creation

The chat.news.lit service responds with the presence broadcast for the room's new and only occupant. Sensor has the owner affiliation and the moderator role. These attributes give the sensor special permissions within the room. More comprehensive information about roles, affiliations, erros and etc can be found in documentation [38].

**XEP-0060: Publish-Subscribe**

Publish-subscribe is another conference XMPP extension[15] that provides a framework for a wide variety of applications, including news feeds, content syndication, extended presence, geolocation, trading systems, workflow systems, and any other application that requires event notifications (image 5.4.

There is a channel of communication, subscribers who are interested in data sent on that channel, and publishers who can send data across the channel. The first thing an application must do for a presenter is to create a channel to publish information. In XMPP pubsub these channels are called *nodes*. The protocol enables XMPP entities to create nodes (services) at a pubsub service and publish information at those nodes; an event notification (with or without payload) is then broadcasted to all entities that have subscribed to the node. Pubsub therefore adheres to the classic Observer design pattern.

**Creating a Node**

A pubsub node is created by sending an IQ-set stanza to the pubsub service, as shown in the listing 5.6, where User1 creates node "sensor_data" within "pubsub.sensor1.lit" XMPP host server.

[15] XEP-0060: Publish-Subscribe, http://xmpp.org/extensions/xep-0060.html

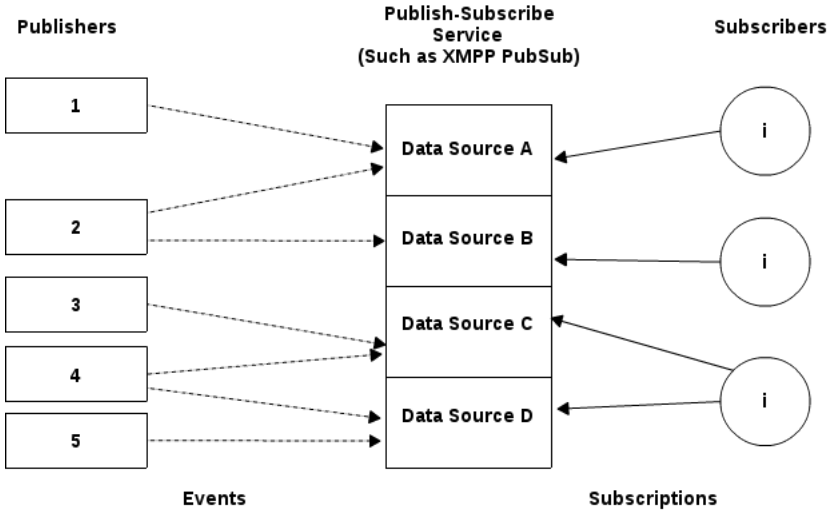


Figure 5.4: General Event Subscription

```
<iq to="pubsub.sensor1.lit"
    from="user_1@sensor1.lit/sensor_registry"
    type="set"
    id="create1">
  <pubsub xmlns="http://jabber.org/protocol/pubsub">
      <create node="sensor_data"/>
  </pubsub>
</iq>
```

Listing 5.6: PubSub Node Creation

Most actions on pubsub nodes will look very similar to this one, the difference between MUC and PubSub stanzas is the <pubsub> element. Pubsub nodes and their configuration are necessary and useful, but they don't do much by themselves. The real value of pubsub nodes is in the events that are published to them and broadcast to subscribers. Anything can be included in a pubsub event. The pubsub service doesn't know or care what is inside the event; it simply broadcasts this data to a node's subscribers.

*Retrieving Item*

User2 just subscribed to user1's sensor_data node, and has missed his earlier event broadcasts. User1 configured his node to persist items and anyone can query his node for the most recently published items. In listing 5.7, user2 requests the last five items by sending an IQ-get stanza to the node with the <items>:

```
<iq from="user2@longbourn.lit/outside"
    to="pubsub.sensor1.lit"
    type="get"
    id="items1">
  <pubsub xmlns="http://jabber.org/protocol/pubsub">
    <items node="sensor_data" max_items="5"/>
  </pubsub>
</iq>
```

Listing 5.7: PubSub: requesting last 5 items from history

The <items> element contains a node attribute just like the other actions. User2 has also set the max_items attribute to 5 because he is only interested in the recent history. If he had omitted max_items, the server would interpret it as a request to send all the historical data it has been configured to keep. If he had set max_items to 500, which is much larger than the configured maximum for the node, the server would have sent as many as were available.

*PubSub Data Flow*

The Figure 5.5 shows the process flow of how and when a client can subscribe/retrieve data from a publisher of service through the XMPP PubSub approach. The first thing the DataSource 1 and 2 must do for a would-be presenter is to create a channel for them to publish data - nodes. Once a node is created and configured, Publishers can start send data. Once these events are published, pubsub takes over and makes sure that they get delivered to the subscribed users. In case Publishers want to get a list of their subscribers, it can be retrieved from the pubsub node so that it can present this data to them. If Publisher will become offline its <presence> will be changed automatically to "unavailable" and next time, becoming online and sending new data, all subscribers will receive this information immediately.

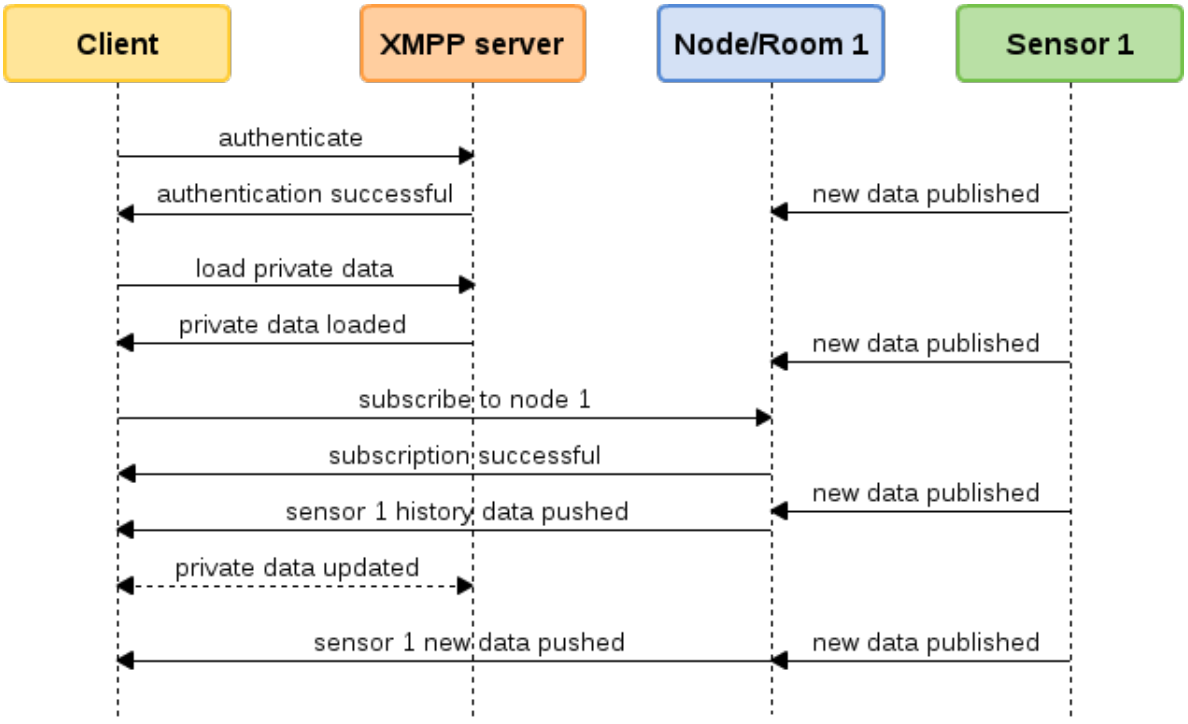


Figure 5.5: Service PubSub

**Comparison and Decision**

After reviewing possible ways to power message broadcasting mechanism, it is required to select the most appropriate one for implementation and support. A quick recap of muc/pubsub advantages and disadvantages is shown in table 5.3.

| | **Publish-Subscribe** | **Multi-User Chat** |
|---|---|---|
| **Advantages** | • Pubsub extension is generic, assuming nothing about the subscribers.<br>• Pubsub nodes and subscriptions are arranged in a tree-based hierarchy.<br>• Events can be published as notifications or as full payloads, and the subscriber can choose which is most appropriate.<br>• Retrieval of the publishing history is built in and fine grained.<br>• The subscriber has more fine grained control over the delivery destination. | • MUC is optimized for chat-related use cases and builds on huge experience of previous chat systems, e.g. IRC.<br>• Presence handling is built in to MUC at a low level.<br>• Common moderation, administration and privelege features are supported.<br>• MUC has many implementations, both of clients and of servers.<br>• MUC allows for multiple levels of anonymity to be used as well as private communication. |
| **Disadvantages** | • By being generic, Pubsub is not optimized for specialized cases.<br>• Support for Pubsub features varies in quality and depth, e.g. no tools for node creation and configuration.<br>• No special handling of presence built in.<br>• Tooling for pubsub node creation and configuration is lacking.<br>• No built-in mechanism for subscribers to interact or find each other. | • It is possible to have bots as room occupants, but the experience is designed for human consumption.<br>• There is no way to organize chat rooms except as a flat hierarchy.<br>• There is no way to share configurations or participation across collections of rooms.<br>• Unlike pubsub, MUC implementations have a lot of edge cases in order to be user friendly and robust. |

Table 5.3: Pubsub and MUC comparison

Although both approaches are very similar, each one has own set of strength and weaknesses. MUC implementation would ensure support of many existing systems, while having Pubsub would be more future-oriented. Therefore, it was decided to fully cover MUC extension, but also include a basic Pubsub integration, so that both systems can work in parralel if needed.

### XMPP connection with JavaScript

In the web-application case, a connection to XMPP server can be only established through HTTP layer. This can be achieved by usnig Bidirectional-streams Over Synchronous HTTP (BOSH). Essentially, BOSH helps an HTTP client to establish a new XMPP session, then transports messages back and forth over HTTP wrapped in a special <body> element. It also provides some security features to make sure that XMPP sessions cannot be easily hijacked. The DataStream Manager communicates with the XMPP server as a normal client. In this way, an HTTP application can control a real XMPP session. Because of the efficiency and low latency afforded by the long polling technique, the end result competes with native connections.

Web applications are cross-platform, easily deployable, and come with a large user base already familiar with them. Web technologies rely on HTML, and it is usually the case that tools for manipulating HTML are also compatible with XML, making a good basis for work with XMPP stanzas. In order to implement web-based client-side application supporting XMPP streams Strophe.js[16] library was used.

**Strophe.js** is a library that will be used for invoking the XMPP protocol from a web-browser. While most of XMPP libraries are focused on chat-based applications, Strophe.js can to also power real-time games, notification systems and search engines, etc. It is production-ready since 2009, and therefore well tested, documented, and easy to extend. It uses BOSH, a standard binding of XMPP to HTTP using long polling.

## 5.2 Interface Implementation

General architecture, as desined in the concept chaper, is shown on image 5.6

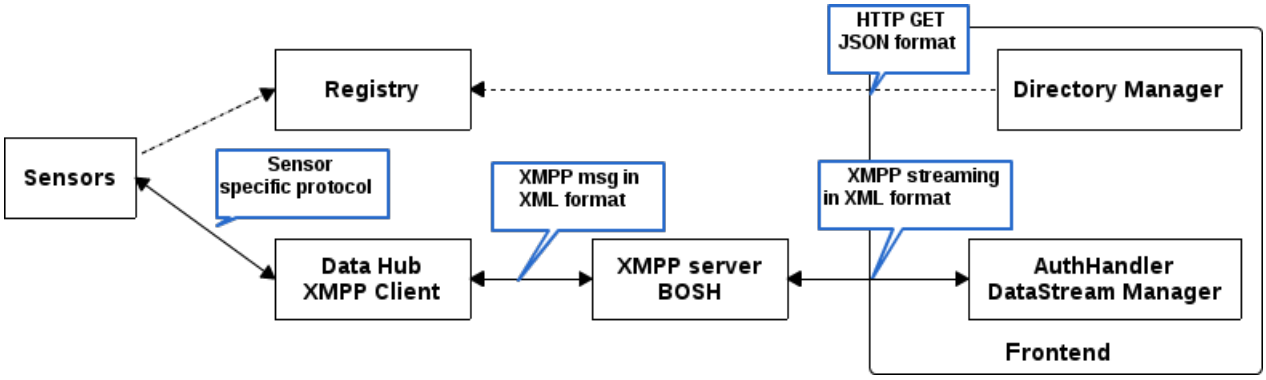


Figure 5.6: System Interfaces

According to it, frontend application has to implement two main interfaces:

1. Registry – Directory Manager

[16]Strophe.js an XMPP library for JavaScript, MIT licensed, http://strophe.im/strophejs/

2. XMPP/BOSH Server – DataStream Manager and AuthHandler

Implementation of aforesaid interfaces and related modules will be described next.

### 5.2.1 Directory Manager

Directory Management module is responsible for querying registries and forming a list of available sensors along with metadata, such as availability and access information, SLA with expiration details, list of access endpoints etc.

As stated in the concept, in order to provide this data, Registries implement a simple Web-API, responding to GET requests with JSON list of available sensors. Since a user might have an unlimited number of registries, it is important to reduce his waiting time, so that a relation between amount of Web-API end-points and total loading time would not grow linearly.

A straightforward way to address the this issue is to query user Registries in parallel, making maximal loading time qeual to loading time of the slowest Registy. To accomplish this, the asynchronous Javascript nature will be used, combined with extension that implements defer/promise pattern: “q-object”, provided by Angular.js.

The purpose of this deferred object is to expose the associated Promise instance as well as APIs that can be used for signaling the successful or unsuccessful completion and the status of the task. The q-object advantage over traditional callback approach is that the promise API allows for composition, in particular serial or parallel joining of promises.

The implementation of Directory Manager service, responsible for parallel querying of registries is shown on listing 5.8. It is using http service requests combined into a single q-object, returned as a promice.

```
1  var sensdash_services = angular.module('sensdash.services', ['ngResource']);
2
3  sensdash_services.factory("Registry", ["$http", "$q", "User", function (
       $http, $q, User) {
4      var registry = {
5          load: function () {
6              var requests = [];
7              var all_registries = User.registries.concat(Config.REGISTRIES);
8              for (var i = 0; i < all_registries.length; i++) {
9                  requests.push($http.get(all_registries[i]));
10             }
11             var q = $q.all(requests);
12             var flat_list = q.then(function (result) {
13                 var list = [];
14                 for (var i = 0; i < result.length; i++) {
15                     list = list.concat(result[i].data);
16                 }
17                 return list;
18             });
19             return flat_list;
20         }
```

```
21          }
22          return registry;
23      }])
```

Listing 5.8: Parallel requests to Registies combinied in q-object

This way, controllers that require combined registry data will receive a promise object, resolved upon completion of all parallel HTTP requests.

### 5.2.2 DataStream Manager

A DataStream Manager is responsible for maintaineing an XMPP connection and provides access to it via HTTP long polling technique. This connection is going to be used for sending and receiving XMPP messages (stanzas) over XMPP stream. Stanzas will be regarded as a basic information block sent from Sensor to User, or from User to Data Hub.

Before any stanzas are sent, an XMPP stream is necessary. Before an XMPP stream can exist, a connection must be made to an XMPP server. Typically clients and servers utilize the domain name system (DNS) to resolve a server's domain name into an address they can connect to.

In the presented web application XMPP connections will be managed through the Connection object, implemented by Strophe library. DataStream Manager includes a pool of BOSH connection managers that require HTTP-bind URL to establish connections. Major XMPP servers come with support for BOSH built in, and they typically expose HTTP-bind service at http://example.com:5280/http-bind or http://example.com:5280/xmpp-httpbind.

Although getting XMPP into a browser certainly involves extra development effort, this technique has some advantages over direct XMPP connections:

- Interactions with the connection manager are request by request, which allows the client to move from network to network. The managed connection stays available even if the end user's IP address changes several times.
- Because one failing request does not terminate the managed connection, managed sessions are extremely robust and tolerant of temporary network failure.
- Because connection managers cache and resend data for a request, no data will be lost when connection is interrupted.
- HTTP is firewall friendly, and because most connection managers run on standard HTTP ports, managed connections still work even in limited network environments that don't allow anything but HTTP.

Creation of the new Strophe.Connection object within "xmpp" service is shown on listing 5.9. Once a connection object was created, calls connect() and disconnect() can be used to start and end communication with the server:

```
var BOSH_SERVICE = Config.BOSH_SERVER;
var xmpp = {
    connection: {connected: false},
connect: function (jid, pwd, callback) {
        xmpp.connection = new Strophe.Connection(BOSH_SERVICE);
        xmpp.connection.connect(jid, pwd, callback);
    }
    ...
}
```

Listing 5.9: Sample code of connecting/disconnectiong to XMPP BOSH

The first two parameters to connect() are the JID and password to use to authenticate the session, and the last parameter is the callback function. The callback function will be called with a single parameter that is set to one of the statuses (CONNECTED, DISCONNECTED, AUTHFAIL, CONNFAIL etc.). A fragment of function that checks connection status is shown on listing 5.10:

```
var update_connection = function (status) {
    ...
    if (status == Strophe.Status.CONNECTED) {
        console.log('XMPP connection established.');
        $scope.xmpp.connection.send($pres().tree());
        $scope.process = '';
        $scope.in_progress = false;
        User.reload();
    }
    else if (status == Strophe.Status.AUTHFAIL) {
        $scope.process = 'Authentication failed';
        $scope.xmpp.connection.flush();
        $scope.xmpp.connection.disconnect();
    }
    else if (status == Strophe.Status.DISCONNECTED) {
        $scope.in_progress = false;
        User.init();
        $scope.xmpp.connection.connected = false;
        $scope.xmpp.connection.disconnected = true;
    }
}
```

Listing 5.10: Fragment of update-connection callback

Every time the connection changes its status, this callback function is executed. It checks selected statuses and triggers appropriate actions.

### Session Mechanism

XMPP is a TCP-based protocol, like HTTP, and communication happens over an established, mostly reliable socket between two endpoints. The BOSH extension to XMPP provides a bridge between this bidirectional, stateful protocol and HTTP, which is unidirectional and stateless. Because a web browser cannot directly connect to an XMPP server, a

BOSH connection manager responds to requests from a browser using HTTP and uses them to manage an XMPP connection on behalf of the user (Figure 5.7). XMPP's basic model of communication is Client -> Server -> Server -> Client, and in support of this it defines a Client to Server protocol and a Server to Server protocol.

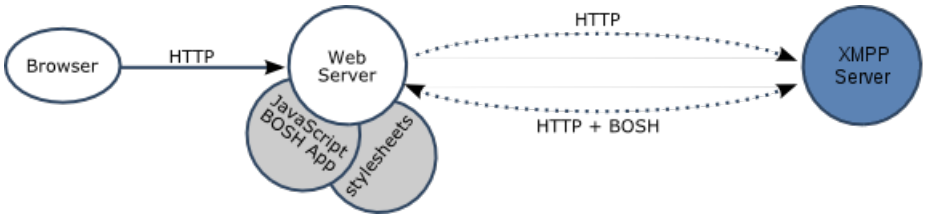


Figure 5.7: XMPP BOSH

Aside from the socket needed for XMPP communication, each managed connection has two other pieces of data associated with it: the session identifier(SID) and the request identifier(RID). SID stands for Session Identifier. This uniquely identifies the managed XMPP connection, and it is often a long, opaque alphanumeric string. Even though it is enough to identify the session, it is not very useful on its own. The RID identifies a particular HTTP request associated with a BOSH-managed connection. Before a connection is established, the client sends a random RID to the DataStream Manager along with its first request. Each subsequent request increments the RID by one. The SID and the RID together provide enough information to interact with the underlying XMPP connection. Because the RID is generated randomly from a very large range of numbers, it is virtually impossible to guess the RID. Also, the DataStream Manager will reject RIDs that fall outside of a narrow window around the current request. In this way, the BOSH-managed connection is tolerant of small errors like out of order delivery but robust to attacks like hijacking the connection. Because these two identifiers are enough to both address and make use of a managed XMPP session, if an application knows the SID and the RID, it can take over or attach to the underlying session.

The attach() function demonstrats sending SID and RID through a BOSH connection in the Listing 5.11):

```
1  var connection = new Strophe.Connection(BOSH_URL);
2      connection.attach(jid, sid, rid, callback);
```

Listing 5.11: BOSH Callback

BOSH sessions can be encrypted, and often the underlying XMPP sessions are encrypted as well. Because XMPP makes use of SASL, the authentication mechanisms are strong.

### Message Handler

In order to support both MUC and PubSub message broadcasting, `XMPP` service in frontend has a set of abstract methods like `subscribe`, `unsubscribe`, `check node` and `handle_message` that work with abstract entities omitting extension protocol details. It also has final methods

`handle_incoming_muc` and `handle_incoming_pubsub`, implementing message consuming logic specific to each protocol.

Listing 5.12 shows simplified implementation of MUC-handler.

```
handle_incoming_muc: function (message) {
        var sensor = xmpp.find_sensor(message);
        if (!(sensor)) {
            return true;
        }
        var text = Strophe.getText(message.getElementsByTagName("body")
            [0]);
        if (sensor.type == 'text') {
          if (typeof text == "string") {
              Text.updateTextBlock(text, sensor["id"]);
           } else {
              throw_error("Message is not a Text");
           }
        } else if (sensor.type == 'chart') {
            try {
                var data_array = JSON.parse(text);
                }
            } catch (e) {
                throw_error("message is not a valid JSON", text);
                return true;
            }
            if (Array.isArray(data_array)) {
                Graph.update(data_array, sensor.id);
            } else {
              throw_error("Message is not a JSON array");
           }
        }
        return true;
    },
```

Listing 5.12: Simplified MUC handler

### 5.2.3 User Private Storage

Since web-based application has no access to the local storage of a device and frontend should not directly rely on a database used on the Data Hub, it was decided to use one of the XMPP extensions. Private XML Storage[17] is such an extension, allowing a client to store any arbitrary XML on the XMPP server by sending an `<iq/>` stanza of type "set" to the server with a `<query/>` child scoped by the "jabber:iq:private" namespace. The purpose of using it is to make a storage for user personal preferences like subscriptions, favorites, accepted SLAs etc.

The `<query/>` element may contain any arbitrary XML fragment as long as the root element of that fragment is scoped by its own namespace. The data can then be retrieved by sending an `<iq/>` stanza of type "get" with a `<query/>` child scoped by the `jabber:iq:private`

[17] XEP0049 specification,`http://xmpp.org/extensions/xep-0049.html`

namespace, which in turn contains a child element scoped by the namespace used for storage of that fragment. Using this method, Jabber entities can store private data on the server and retrieve it whenever necessary. The data stored might be anything, as long as it is a valid XML. One typical usage for this namespace is the server-side storage of client-specific preferences. All available methods are displayed in table 5.8.

| | |
|---|---|
| get | Sent with a blank query to retrieve the private data from the server. |
| set | Sent with the private XML data contained inside of a query. |
| result | Returns the private data from the server. |
| error | There was an error processing the request. The exact error can be found in the child error element. |

Figure 5.8: Description of Acceptable Methods

### Elements

The root element of Private XML Storage namespace is query. At least one child element with a proper namespace must be included; otherwise the server must respond with a "Not Acceptable" error. A client must not query for more than one namespace in a single IQ get request. However, an IQ set or result may contain multiple elements qualified by the same namespace. Examples of saving and loading data are shown on the Listing 5.13 and 5.14.

```
CLIENT:
<iq type="set" id="1001">
  <query xmlns="jabber:iq:private">
    <exodus xmlns="exodus:prefs">
      <defaultnick>Alice</defaultnick>
    </exodus>
  </query>
</iq>

SERVER:
<iq type="result" from="alice@likepro.co/"
    to="alice@likepro.co/" id="1001"/>
```

Listing 5.13: Client Stores Private Data

```
CLIENT:
<iq type="get" id="1001">
  <query xmlns="jabber:iq:private">
    <exodus xmlns="exodus:prefs"/>
  </query>
</iq>

SERVER:
<iq type="result" from="alice@likepro.co/"
    to="alice@likepro.co/" id="1001">
  <query xmlns="jabber:iq:private">
    <exodus xmlns="exodus:prefs">
      <defaultnick>Alice</defaultnick>
```

```
14          </exodus>
15        </query>
16      </iq>
```

Listing 5.14: Client Retrieves Private Data

The message of format described above can be issued by using two main functions: save (for saving data on the XMPP server) and load (to retrieve saved data from the XMPP server), as shown on the listing 5.15. Both methods accept an arbitary string as a property key. The method init() initiates creation of registries, subscriptions, favorites and profile storages for a user with empty profile. An example of subscriptons map and favorites array are presented in the Appendix B. All the subsequent manipulations made by a user will be automatically saved and the GUI will be updated automatically by invoking the method reload().

```
1       sensdash_services.factory("User", ["XMPP", "$rootScope", function (xmpp,
            $rootScope) {
2         var user = {
3             init: function () {
4                 user.registries = [];
5                 user.favorites = [];
6                 user.subscriptions = {};
7                 user.profile = {};
8             },
9             save: function (property) {
10                xmpp.connection.private.set(property, property + ":ns", user[
                      property], function (data) {
11                        console.log(property + " saved: ", data);
12                    },
13                    console.log);
14            },
15            load: function (property) {
16                xmpp.connection.private.get(property, property + ":ns",
                      function (data) {
17                        user[property] = data != undefined ? data : [];
18                        $rootScope.$apply();
19                    },
20                    console.log);
21            },
22            reload: function () {
23                user.load("registries");
24                user.load("profile");
25                user.load("subscriptions");
26                user.load("favorites");
27            }
```

Listing 5.15: Snippet of Save/Load preferences to a private namespace

## 5.3 Evaluation

Evaluation is done as a proof of concept by demonstrating a use case scenario of an XMPP-driven web dashboard, which retrieves data from a hardware and software sensors. As an example of software sensor a bot was chosen, periodically sending IT news in text format. As a hardware sensor a university temperature sensor was used. Both of them are accessable from respective XMPP chat rooms. A prototype was given a name "SensDash" (Sensor Dashboard).

Temperature sensor was provided as a testing environment in scope of ACDSense project, together with TU Dresden, BTU Cottbus - Senftenberg and RWTH Aachen University. It locates in the room INF3086, Faculty of Computer Science, Chair of Computer Science. It represents a class of low-cost, high-performance sensors, which is implemented using a commercially available Raspberry Pi single-board computer with an affiliated USB thermometer and automatic WLAN and XMPP connections to a sensor MUC room established at a boot time. Everything concerning sensors was installed on Mobilis server. According to the system architecture on the Figure 5.9, everything concerning sensors-side real-time streaming is assumed to be implemented on a DataHub, which includes pre-installed XMPP server supporting BOSH and MUC extensions. The temperature sensor itself is a Data Source 1, and the Data Source 2 is respectively a software sensor presented on the Figure 5.9.

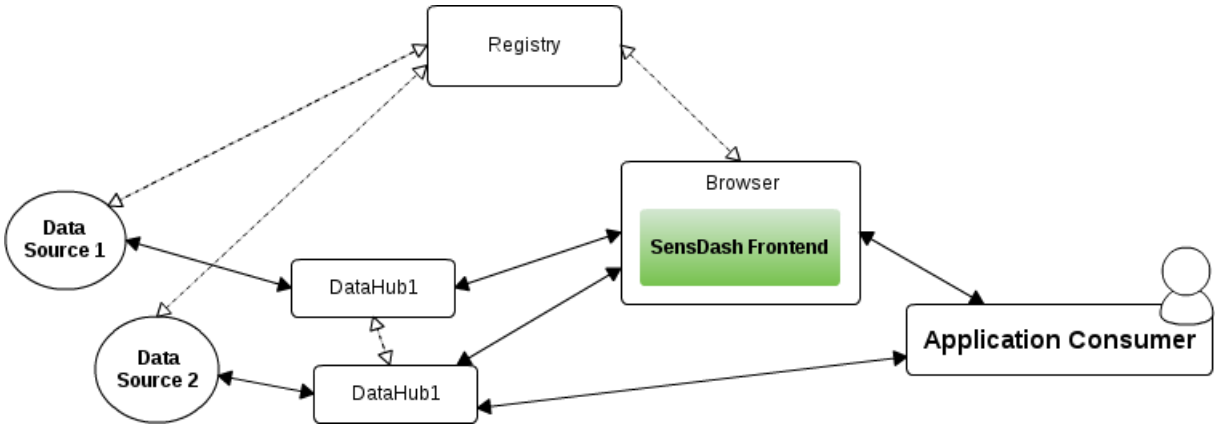


Figure 5.9: System Architecture Scheme

The temperature sensor publishes data to a MUC room, which can be accessed by jid `xmpp://inf3084@conference.mobilis-dev.inf.tu-dresden.de` and has the description shown on listing 5.16.

```
{
    "sensormuc": {
        "type": "AMBIENT_TEMPERATURE",
        "format": "short",
        "location": {
            "countryCode": "DE",
            "cityName": "Cottbus",
            "latitude": 51.076834,
```

```
            "longitude": 13.772586
        }
    }
}
```

Listing 5.16: JSON Description of Temperature Sensor

First, every sensor has to be put in the Registry with a unique id and fill-in the attributes defined in JSON Registry standard interface. Since not this Master Thesis nor ACDSense project had a task of building Registry software, metadata of temperature and news sensors was added manually as plaintext JSON stubs. The full example of metadata in the Registry can be found in the Appendix A, and the listing 5.17 presents an excerpt of its main charactristics.

```
[
    {
            "id": "30",
            "title": "Ambient Temperature INF3084",
            "availability": true,
            "description": "This sensor provides temperature updates with 5-
               seconds frequency and represents a class of low-cost high-
               performance sensors. It is implemented using an available
               Raspberry Pi single-board computer with an affiliated USB
               thermometer and automatic WLAN and XMPP connections to a
               sensor MUC room established at boot time.",
            "sla": "Sensor resolution is 0.5 C, measurements taken every 5
               seconds. Uptime 95% from 6:00 till 22:00.",
            "sla_last_update":"1395005996",
            "access": "private",
            "provider_name": "Provider TU Dresden",
            "location": "Cottbus",
            "type": "chart",
            "end_points": [
                {
                    "type": "muc",
                    "name": "xmpp://inf3084@conference.mobilis-dev.inf.tu-
                       dresden.de",
                    "pwd": null
                },
                {
                    "type": "muc",
                    "name": "xmpp://inf3086@conference.mobilis-dev.inf.tu-
                       dresden.de",
                    "pwd": null
                }

            ]
    },
    {
        "id": "20",
        "title": "IT News Feed",
        "availability": true,
        "last_update": "2014-04-05T11:14:34.000+02:00",
```

```
32          "description": "Provides up to date news in IT and Telecommunication
                area together with information about new gadgets.",
33          "sla": "",
34          "sla_last_update": "",
35          "access": "public",
36          "provider_name": "Provider TU Dresden",
37          "location": "Dresden",
38          "type": "text",
39          "end_points": [
40              {
41                  "type": "muc",
42                  "priority": "main",
43                  "name": "xmpp://testraum@conference.mobilis-dev.inf.tu-dresden
                        .de",
44                  "pwd": null
45              }
46          ]
47      }
48  ]
```

Listing 5.17: JSON Description of Sensors

Use case scenario is demonstrated from 3rd party user prospective – a mobile application developer Max. First of all he needs to define to which sensor to subscribe and how to retrieve real-time streaming from it.

### 5.3.1 Use Case Scenario

**Step 1:** In order to find necessary sensor, description and data format provided by it, Max has to log in into the SensDash by using peronal JID(max_sensdash@likepro.co) and password, received from an administrator of a SensDash(Fig. 5.10) or auto-created by XMPP registration software.

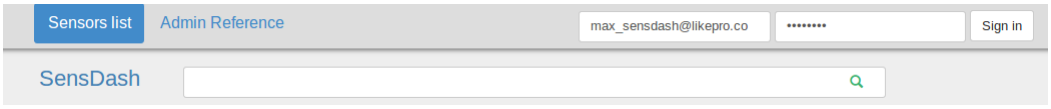


Figure 5.10: Log in to the SensDash

**Step 2:** After successfully completing the authentification process, Max can start browsing, searching and filtering sensors, as shown on the Figure 5.11.

By using a search field, he can enter keywords that define purpose or type of a sensor, and a list of sensors will be sorted immediately based on the search input. To get more detailed information about sensor user clicks on the sensor box. In a new modal window (Figure 5.12) user can get a detailed description about sensor, e.g. : SLA, location, provider, preview, development details such as end-points quantity, administrator name and more sophisticated dasription data if required. This information gives an overview of what type of data is provided by data source and which SLA should be accepted if the user wants to subscribe to it.

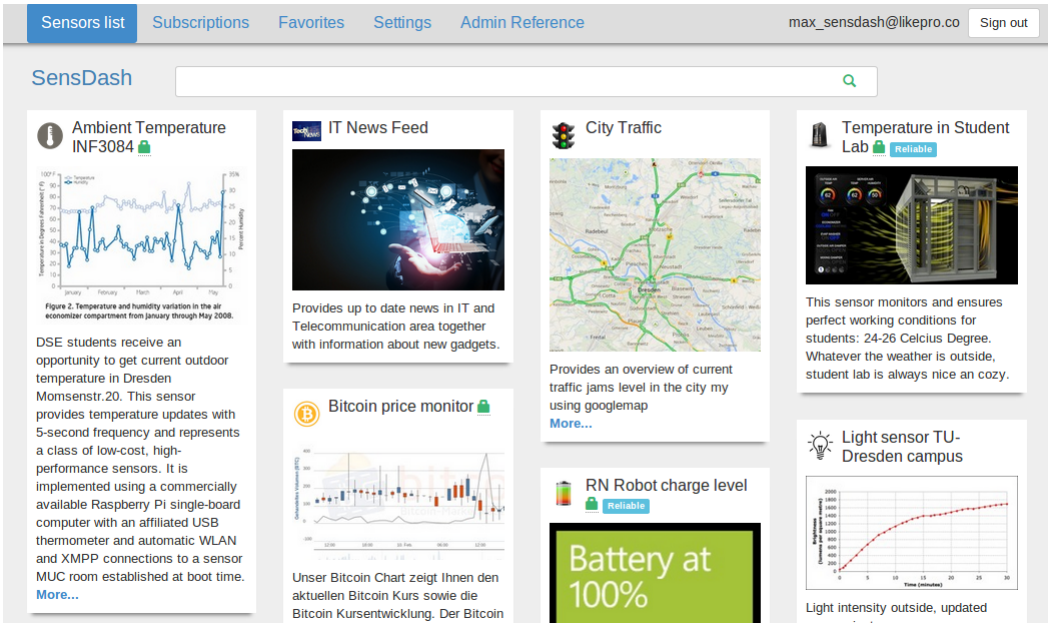


Figure 5.11: List of available sensors

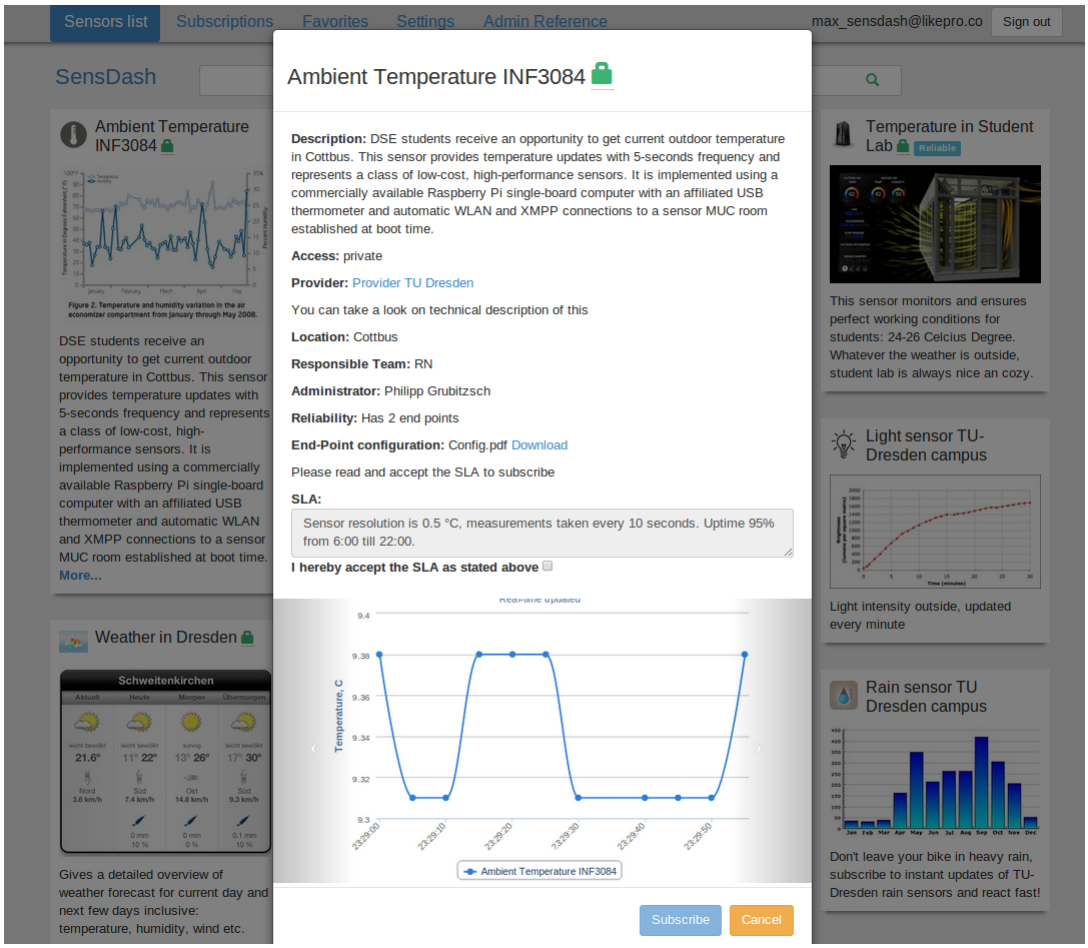


Figure 5.12: Personal Modal of a Sensor

In case of ambient temperature in room INF3084, Max can explore a predefined preview, which contains graphs of data, its frequency, security level and relialability. In case of

software sensor it can be sample of a news feed, provided by sensor.

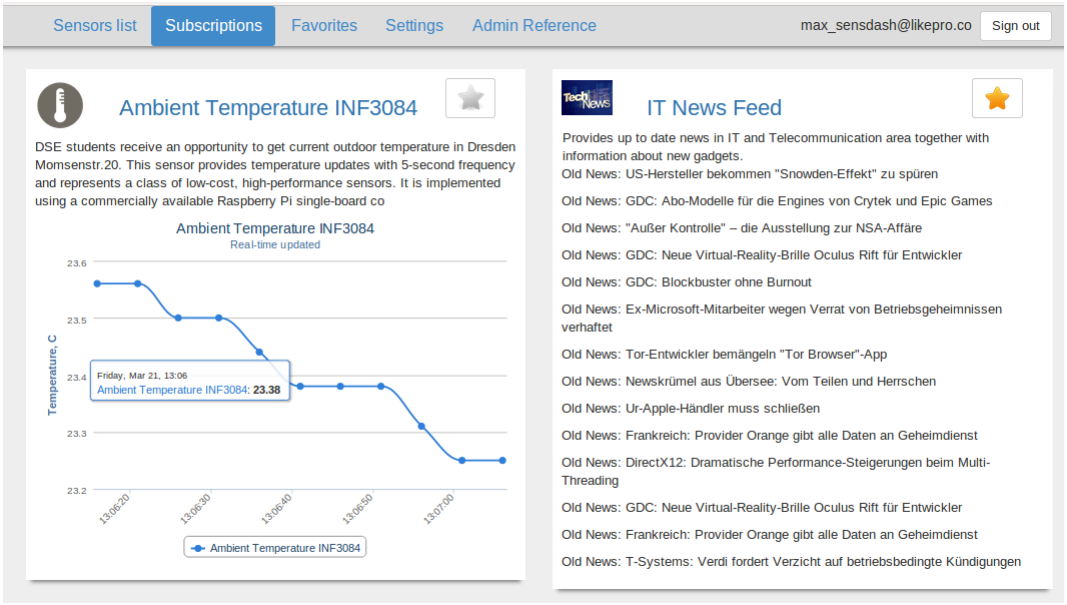


Figure 5.13: Subscriptions list

**Step 3:** Since the temperature sensor is a private sensor, Max has to accept SLA before getting a real-time data from it. In contrast with temperature sensor, a software sensor – IT news feed has a public access, thus, no need to accept any SLA in order to see real-time data provided by this sensor. The common approach, which is applied to every sensor independently from its access type (public or private) is to subscribe to it if desired conditions are met.

This way, all sensors that Max has subscribed to will appear in the next tab called "Subscriptions". First, desired amount of history data will be loaded. Then, as soon as new data becomes available SensDash retrieves it (Figure 5.13). By using Subscriptions Tab and icon "star" located in a right upper corner, user can add most relevant subscriptions to Favorites. After clicking on the favorites ison, sensors information will appear in the Favorites Tab.

**Step 4:** To get information about personal account user should navigate to Settings tab. He can find personal profile settings there, as shown on the Figure 5.14.

**Step 5:** As a developer, Max might be interested in technical details of a sensor data retieval process, e.g. API references, end-point configuration, sensor data format and system architecture in order to connect with backend directly or to add a new Registry URL. So Max should navigate to the Admin References Tab (Fig. 5.15), where he can find all available implementation detailes of a SensDash, e.g. used XMPP services and extensions, interface workflow etc.

### 5.3.2 SensDash Implementation

Next, SensDash prototype implementation details will be revealed, along with description of main functional modules and XMPP data stream integration.

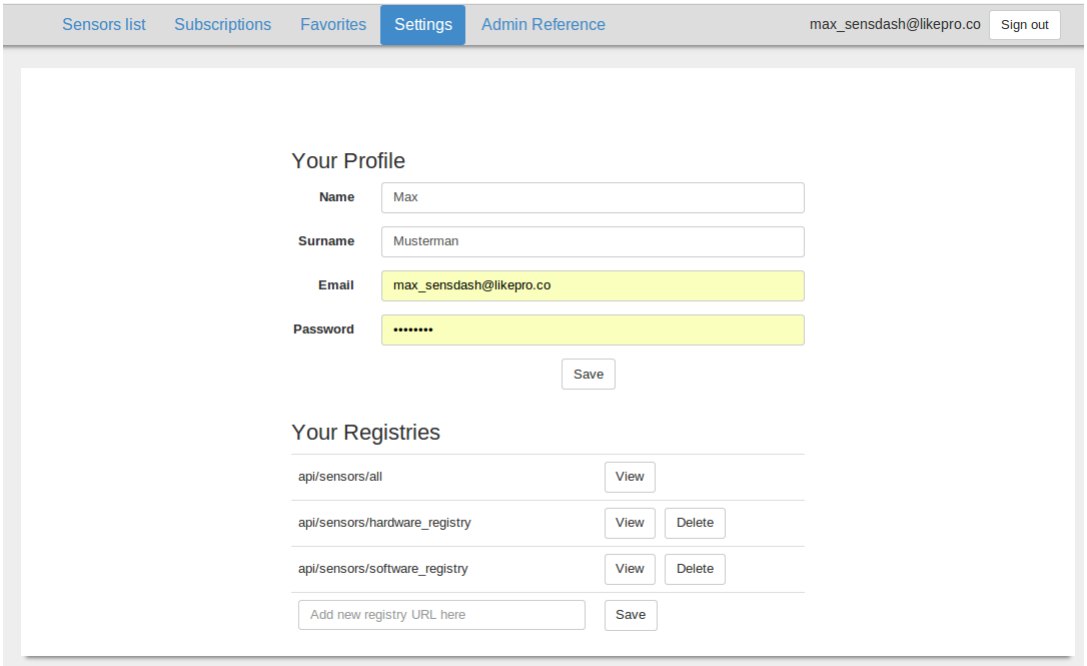


Figure 5.14: Settings Tab

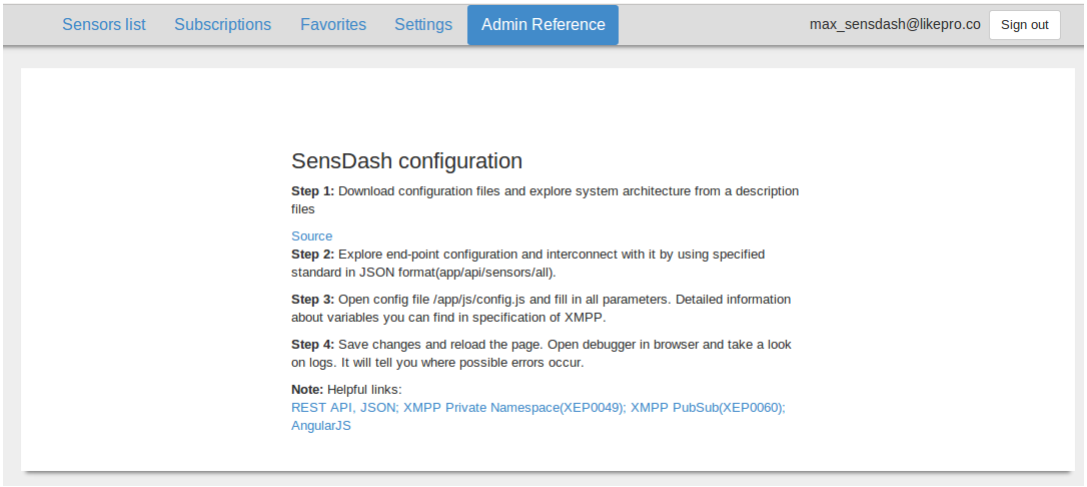


Figure 5.15: Administrator Tab

**Log In** to the system is allowed only with valid JID and password, registered within XMPP servers. Since XMPP network is highly distributed, it does not metter where exactly the JID belongs to, as long as appropriate permissions are granted within sensor rooms. However, it is important to be aware which XMPP server should carry MUC room and be responsible for managing real-time data of a sensor.

First, user credentials are checked with regular expression on the frontend side, and than sent to XMPP server. Once XMPP server authenticates the user, a session is saved in the browser using cookies. Next time when the user will open a SensDash, session from cookies will automatically log him/her into the system.

**Preferences Persistance** is implemented using XEP0049 and stored on the XMPP server (implementation details available in section 5.2.3). When a user signs in to the

system for the first time, the application logic creates an empty account and initiates empty containers for subscriptions and favorites. Once a user subscribes to any resource, this resource automatically appears in Subscriptions Tab and added to the subscriptions map, saved on the XMPP server. The same is valid for favorites and Favorites Tab. Next time when the user will log in to SensDash, all saved subscriptions, favorites and other private data will be automatically loaded and appear in corresponding tabs. Examples od subscriptions map and favorites array for Max are presented in the Appendix B.

**SLA Updates**. A hardware or software sensor may have an owner, vendor or provider with whom a user has to sign a service-level agreement (SLA). The Frontend is responsible for retrieving SLA description and its "last time update" for the user. Once user has accepted the sensor's SLA, he will receive a real-time data till the moment when SLA will be changed by provider. As soon as SLA "last time update" in the list of subscribtions of user will no longer match the SLA "last time update" provided by the Registry through the Web API, user will be automatically unsubscribed from corresponding sensor and notified via popup alert window.

**Search Bar** was made by using frontend-based model filtering, implemented with Angular.js filter concept and directly binded to template. It performs real-time sorting and filtering based on metadata of sensors in all registries. It is fast very straightforward for end-user.

**Data Streaming** is the point where Data Hub joins the system as a part of backend. All data streaming works through XMPP extensions. In presented example with ambient temprature in INF3084 a MUC extension (XEP0045) was used for retrieving real-time data through BOSH service. But since generic frontend should support maximum of approaches, PubSub extension (XEP0060) support was also provided. Strophe.js was used as a client-side XMPP library; some code was refactored to be MVC-structured.

SensDash logic was built using AngularJS MVC skeleton (Appendix D). This enabled two-way data binding between controllers and templates, presenting updates from data stream in real-time. Multiple endpoints in sensor metadata provide a ground for building redundant data streams, which have been used for adding an extra layer of reliability and/or security.

Incoming messages are dispatched to appropriate handlers based on sensor type and ID, this data is cached in a hashmap for instant propagation of updates. Message handler services have been implemented with an aim to be easily pluggable, so that generic frontend can be easily extended with new types of sensors.

### Functional characteristics of a sensor

In the Section 4.4 3 main functional characteristics were clarified, that grant sensor with properties based on a system architecture. Since XMPP server was introduced as an essential part of the Data Hub, XMPP MUC rooms have been used as one of possible realization of end-points (Fig. 5.16). Application logic automatically calculates a number of available end-points for every sensor, based on Registry metadata and shows it on the main page, as shown on Figure 5.17. The rules of how the reliability level was calculated is described in the Section 4.4, Subsection "Sensor Functional Characteristics" (Appendix C).

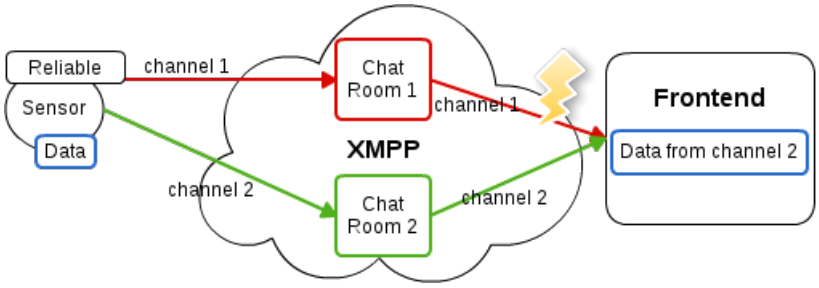


Figure 5.16: Data secure transfer using Chat Rooms

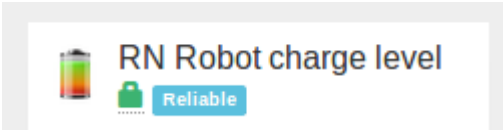


Figure 5.17: GUI identification of security and reliability level

## 5.4 Summary

This chapter presents implementation details of the generic frontend concept and proposes first working prototype, which is shown by using convincing scenario in the Evaluation section 5.3. The use case was based on subscribing to ambient temperature sensor from the room INF3084 (hardware sensor) and IT news Feed (software sensor).

At the begining of the chapter, development tools, dependencies and environment were discovered and presented: JavaScript and its jQuery extensions as a programming language, AngularJS together with Bootstrap to bind application logic with UI, and Strophe.js – to implement the XMPP interface and its extensions. Web API interface was defined used to retrieve available sensors metadata from Registries by sending an HTTP GET request to each. Authentification, private data persistance and sensor data stream handling was implemented by using XMPP BOSH protocol, XEP0049 and XEP0045 extensions, using Strophe.js library. In order to connect, disconnet, authorize, save, load, initiate connection through XMPP all these methods and functions were declared based on AngularJS directives, services and controllers.

All used technologies, protocols, libraries and methodologies were gathered in order to implement a working prototype. A summary overview of all mentioned components and tools are shown on the Figure 5.18.

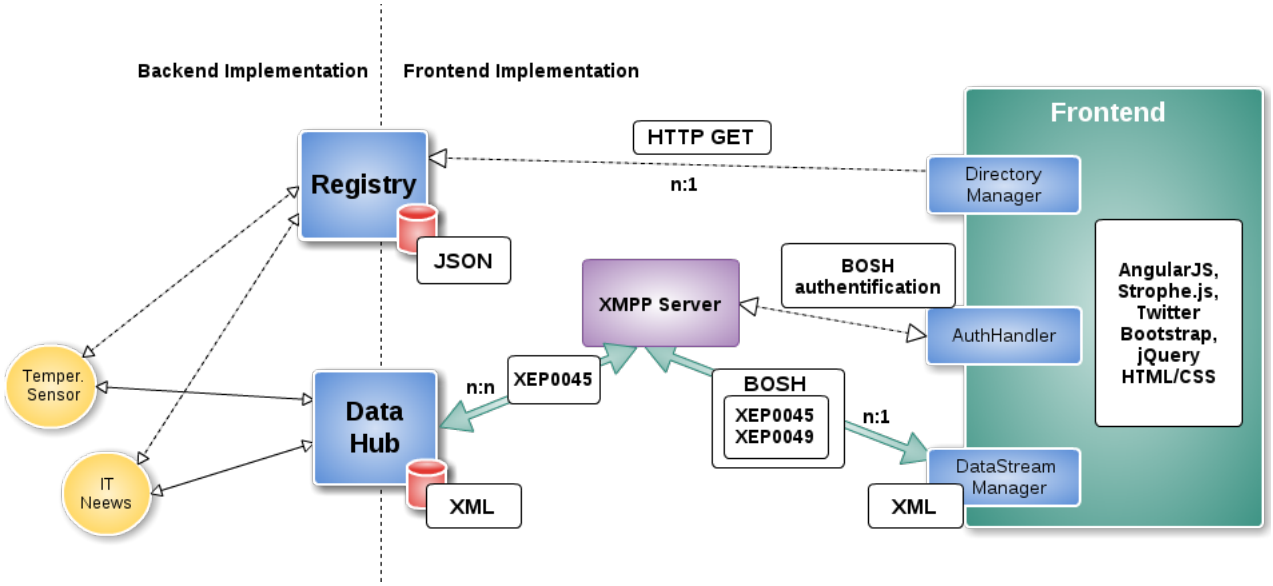


Figure 5.18: Integrated Implementation Architecture

# Chapter 6

# Conclusions and Outlook

## 6.1 Conclusions

This chapter contains the summary of the presented thesis and provides a brief overview of each chapter. The presented thesis consists of six chapters including the current one. Overall answers to the addressed researched questions are given. The chapter is concluded with future work proposals and descriptions.

**Chapter 1, Introduction:** the motivation (Section 1.1) of the work arguing the necessity of creation of a generic frontend. A global area of application usage is presented in the Section 1.2, covering infrastructure, tools, protocols and application overview itself. Main research questions and goals that have to be achieved in the thesis are formalized in the section 1.2. The structure of the master thesis (Section 1.4) completes the first chapter.

**Chapter 2, Foundation and Requirements:** gives an overview of a basic requirements to a generic frontend. Starting from main properties in Section 2.1 namely loose coupling, fine-grained structure, multi-user binding, cross-platforming, responsive design and usability. And continuing in Section 2.2 with definition of data sources, typization and separation to a physical and software sensors. It helps to define main channels of retrieved data in which user can be interested in and based on that build a system architecture. An important part of data retrieval, without dependency on data type, is to guarantee a real-time streaming. This approach provides an universal way to derive information services and possible solutions for implementation described and explained in details in the Chapter 5.

**Chapter 3, State of the Art:** has introduced main approaches of building web-based dashboards for different types of sensed data. Section 3.1 provides a list of projects devoted to retrieve sensed data such as: temperature, humidity, traffic sensors. Not only scientific research projects were discovered but also private customer-oriented solutions for Big Data management. In total 13 projects were structured and characterized according to formulated properties of a generic frontend concept.

The section 3.2 consists of main methodologies of a design and implementation of a generic frontend: portal with portles, mashup, native applications and non-browser based systems. As a result, mashup technologies based on browser satisfy all necessary requirements to

create generic frontend for exploring sensor and information services. Based on this decision in the chapter 4, defined system architecture and described responsibility of an every module.

**Chapter 4, Concept:** according to a 3-tier architecture, the first web-based concept for sensor streaming services is to be created. Fine-grained structure provides a clear separation of concerns between different modules of concepts. Client tier contains GUI description and content, Application Tier provides application logic in order to connect backend and Client Tier, and finally, Data Tier describes typization of data in order to easily connect it with Application Tier and visually retrieve it by using Client Tier. Characteristics as reliability, perfomance and level of information security have been assigned to every data souce. Every characteristic relies on a number of end-points responsible for sensors. As a result next most important modules of Application Tier was identified: Registry, Data Hub, AuthHandler.

**Chapter 5, Implementation and Evaluation:** presents details of implementation of the generic frontend concept and proposes first working prototype, which is shown by using convincing scenario in the evaluation section 5.5. At the begining of a chapter, choosen tools and development environment were discovered and presented: Javascript with jQuery extensions as a programming languages, HTML5 and CSS for markup, AngularJS together with Bootstrap as frontend frameworks, Strophe.js for XMPP client interface implementation. Web API was defined and afterwards used to retrieve all available sensors from the Registry by requesting a simple Web-API. Authentification and data streaming handling through the XMPP server was made by using XMPP BOSH standard and XEP0049, XEP0045 extensions. In order to connect, disconnet, authorize, save, load, initiate connection through the XMPP interface all these methods and functions where structured within AngularJS MVC skeleton. As an example of hardware sensor a temperature sensor from INF3084 university room was used, and as software sensor IT news feed bot was used.

## 6.2 Achieved Goals

**Which architecture should have a concept of a generic frontend? How should it be designed in order to provide ease of integration with any backend system?**
The concept has to have as much fine-grained structure as possible. Thus every module has to perform independent task. It provides a possibility to scale and maintain system without influence to another modules.

**What types of data sources have to be retrieved and what is the most universal interface for collaboration between backend and frontend systems? Which protocol can perform real-time data streaming and access historical archives of data streams independently from type of data in it and bind it with user personal preferences?**
The main type of data that has to be retrieved is a real-time streaming data. Based on XMPP interface it becomes not important which concreate type of data has any stream: text, images, array or map of values etc. The technique that can be used for keeping a connection alive and guarantee the information arrival to a client side is http long-polling.

XMPP with BOSH and XEP0049, XEP0060, XEP0049 extensions satisfies all requirements to the system interface.

**Which software components might be applied to the concept in order to be applicable to the most available data sources and platforms? How can be a generic GUI designed and implemented by providing dynamic content retrieval?** Nowadays a variety of different software solutions for a web-based applications are available. Among all of them for the GUI implementation Twitter Bootstrap version 3.0 and HTML5 with CSS were used. A JavaScript library Strophe.js was used for connecting browser with XMPP. And to combine all aforementioned tecnologies together and to make easily extensible and maintainable code AngularJS (MVC JavaScript Framework) was used. With AngularJS it is more convinient to create a dynamic GUI content visualization keeping a canonical structure of the code.

## 6.3 Future work

To conclude the thesis, suggestions for the future work have been proposed. These suggestions are devoted to improve either the conceptual aspects or the current implementation.

### 6.3.1 Conceptual Aspects

This subsection describes possible improvements on the concept level.

#### SLA

Differentiation of SLA depending on user type/rights. SLA changes brings next questions into a picture:

- How to re-sign SLA? (On the example of SensDash it was done by automatically unsubscribing from a sensor)
- How to notify the frontend and user about changes of SLA that has been already accepted?
- How will system react in case of user not accepting new SLA?
- How can frontend predefine all future changes that can appear according to SLA changes?
- Is it possible to accept one SLA which provides access to many sensors?
- How to handle billing aspects?

The thesis was focused on the core architectural design and development aspects, so while current contracting implementation has not so "user-friendly" design, it demonstrates a possible implementation of the contracting mechanism. The questions of billing, contracting and SLA are still open and left for future work. The billing should use fine-grained approach to make sure that consumers of cloud resources only pay for what they use. Contract phase may be more closely integrated with the SPACE platform by reusing the Contract Wizard component. The SLA should be taken into account during on-the-fly rescaling, as this procedure may result in the capacity change, which in its turn result in pricing changes.

### 6.3.2 Implementation Area

This subsection describes possible improvements and enhancement on the implementation level.

#### Introducing the interaction awareness

In principle, when a user opens an application first time, different information can be interesting to him. Therefore, to introduce a convenient awareness, the following research questions should be investigated:

- What kind of interaction awareness information end users really need?
- Does a user want to configure the received awareness information?
- In case of providing a configuration, what an appropriate visualization and interaction metaphor can be provided?

#### Privacy and security

Session credentials stored via cookies inside any browser on any kind of device can be stollen relatively easy. To avoid this additional possibility of data encryption or more secure authorisation process should be researched.
An option to specify a data source as highly secure if its data would be splitted between two or more end-points have to be researched. Even if the data will be splitted onto 2, 3 or more parts and then combined to one on a frontend side, an encryption system have to be used. Since frontend is web-based, a possibility to debug its hidden login from a web-console of a browser has to be concidered.

#### Collaborative space

In order to perform social collaboration between different users, it would be helpful to have a possibility to rate the data source or leave comments. Users can share their experience, help each other, define hot topics, leave personal rating and help administrator to improve the system itself. Since now, personal preferences such as: subscriptions, favorites, user

data are stored on a XMPP server by using XEP0049 (XML-based private namespace), it is not possible to share personal rating or comments with someone else, because someone else can change everything from a browser console. Thus, another XMPP extension needs to be found, which can provide sharing space with access control.

### Integration with other applications via Internet

The best chance to make an application widely-used is to enhance a list of supported hardware and software sensors. But a lot of systems already propose their own sensors and corresponding app to it. How possible will it be to create an additional module for enhancement that will play a role of retranslator or proxy between two different systems? In which information in such a case a developer of a consumer application would be interested? Would not it be better to specify main topics and propose to choose the most relevant for his/her search at first?

### Resource limitations: energy, bandwidth and computation

Since mobile devices face internal and external resource limitations, the need of differentiation of connection properties is important. For example, location data can be provided using GPS, WiFi, and GSM, with decreasing levels of accuracy. Compared to WiFi and GSM, continuous GPS location sampling drains the battery faster. One approach to this problem uses low duty cycling to reduce energy consumption of high-quality sensors (i.e., GPS), and alternates between high- and low-quality sensors depending on the energy levels of the device (e.g., sample WiFi often when battery level is less than 70 percent). This approach trades off data quality and accuracy for energy.

### Visibility Rules

The definition of roles for every user with according visibility rules was not covered in scope of this master thesis. Assigning a role “administrator”, “backend developer”, “vendor of sensor”, “3d party developer” , “consumer” etc to user groups can significantly expand a graphical interface. Also it makes possible to add new modules such as: statistics of sensor usage/subscriptions, overview of user behaviour, business analysis of user interests, feedbacks, comments and ratings. In the meanwhile, for “administrator” and “backend developer” modules oriented on “sales” or “marketing” can be hidden, in order to leave only technical part of implementation. Such features as receiving notifications about new changes from subscriptions by e-mail or sms can become also very interesting for the user.

# List of Figures

# List of Tables

# Bibliography


[1] S. Suakanto, S.H. Supangkat, Suhardi, and R. Saragih. Smart city dashboard for integrating various data of sensor networks. In *ICT for Smart Society (ICISS), 2013 International Conference on*, pages 1–5, 2013.

[2] Schuster Schill Ackermann Ameling Bendel, Springer. A service infrastructure for the internet of things based on xmpp. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on*, pages 385–388. IEEE, 2013.

[3] Xianfeng Song, Chaoliang Wang, Masakazu Kagawa, and Venkatesh Raghavan. Real-time monitoring portal for urban environment using sensor web technology. In *Geoinformatics, 2010 18th International Conference on*, pages 1–5. IEEE, 2010.

[4] Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle. Sensorcloud: Towards the interdisciplinary development of a trustworthy platform for globally interconnected sensors and actuators. *arXiv preprint arXiv:1310.6542*, 2013.

[5] Wikipedia. Front and back ends — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Front_and_back_ends&oldid=572495173`, 2013. [Online; accessed 15-November-2013].

[6] William A Firestone. The study of loose coupling: Problems, progress, and prospects. 1984.

[7] Erwin Danneels. Tight–loose coupling with customers: the enactment of customer orientation. *Strategic Management Journal*, 24(6):559–576, 2003.

[8] Wikipedia. Responsive web design — wikipedia, the free encyclopedia, 2014. [Online; accessed 2-February-2014].

[9] Zoe Mickley Gillenwater. Examples of flexible layouts with css3 media queries. In *Geoinformatics, 2010 18th International Conference on*, page 320, Dec 15, 2010.

[10] Nick Pettit. Beginner's guide to responsive web design, Aug 8, 2012.

[11] ETHAN MARCOTTE. A list apart, March 03, 2009.

[12] Wikipedia. Usability — wikipedia, the free encyclopedia, 2014. [Online; accessed 2-February-2014].

[13] JAKOB NIELSEN. Usability 101: Introduction to usability, 2012.

[14] Fabian Schomm, Florian Stahl, and Gottfried Vossen. Marketplaces for data: an initial survey. *ACM SIGMOD Record*, 42(1):15–26, 2013.

[15] Suman Nath, Jie Liu, and Feng Zhao. Sensormap for wide-area sensor webs. *Computer*, 40(7):90–93, 2007.

[16] Suman Nath, Jie Liu, and Feng Zhao. Challenges in building a portal for sensors world-wide. In *First Workshop on World-Sensor-Web*, 2006.

[17] Xiaogang Yang, Wenzhan Song, and Debraj De. Liveweb: A sensorweb portal for sensing the world in real-time. *Tsinghua Science & Technology*, 16(5):491–504, 2011.

[18] Schuster Daniel, Ronny Klauck, Michael Kirsche, Renzel Dominik, and Istv an Koren. Federated access to smart objects using xmpp. 2013.

[19] Josef Spillner, Marius Feldmann, Iris Braun, Thomas Springer, and Alexander Schill. Ad-hoc usage of web services with dynvoker. In *Towards a Service-Based Internet*, pages 208–219. Springer, 2008.

[20] Inc. Open Geospatial Consortium. A cloud design for user-controlled storage and processing of sensor data. In *Sensor Web Enablement Architecture*, pages 13–30. Open Geospatial Consortium, Inc., 2008.

[21] Institute for Software and Multimedia-Technology Technical University of Dresden. Vicci, 2012.

[22] M. Franke, C. Seidl, and T. Schlegel. A seamless integration, semantic middleware for cyber-physical systems. In *Networking, Sensing and Control (ICNSC), 2013 10th IEEE International Conference on*, pages 627–632, 2013.

[23] René Hummen, Martin Henze, Daniel Catrein, and Klaus Wehrle. A cloud design for user-controlled storage and processing of sensor data. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 232–240. IEEE, 2012.

[24] Josef Spillner, Johannes Schad, and Stephan Zepezauer. Personal and federated cloud management cockpit. *Praxis der Informationsverarbeitung und Kommunikation*, 36(1):44–44, 2013.

[25] Diego Calvanese, Magdalena Ortiz, Mantas Simkus, and Giorgio Stefanoni. Reasoning about explanations for negative query answers in dl-lite. *Journal of Artificial Intelligence Research*, 48:635–669, 2013.

[26] Ian Horrocks Ernesto Jiménez-Ruiz, Bernardo Cuenca Grau. Is my ontology matching system similar to yours? In *8th International Workshop on Ontology Matching*, 2013.

[27] Ralf Möller, Christian Neuenstadt, Özgür L. Özçep, and Sebastian Wandelt. Advances in accessing big data with expressive ontologies. In Thomas Eiter, Birte Glimm, Yevgeny Kazakov, and Markus Krötzsch, editors, *Description Logics*, volume 1014 of *CEUR Workshop Proceedings*, pages 842–853. CEUR-WS.org, 2013.

[28] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. big'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM, 2008.

[29] Daniel Su Kuen Seong. Usability guidelines for designing mobile learning portals. In *Proceedings of the 3rd international conference on Mobile technology, applications & systems*, page 25. ACM, 2006.

[30] Jin Yu, Boualem Benatallah, Fabio Casati, and Florian Daniel. Understanding mashup development. *Internet Computing, IEEE*, 12(5):44–52, 2008.

[31] Rabiu Ibrahim. Framework and model design for higher education mash-ups. In *Computer & Information Science (ICCIS), 2012 International Conference on*, volume 2, pages 938–943. IEEE, 2012.

[32] Volker Hoyer, Katarina Stanoesvka-Slabeva, Till Janner, and Christoph Schroth. Enterprise mashups: Design principles towards the long tail of user needs. In *Services Computing, 2008. SCC'08. IEEE International Conference on*, volume 2, pages 601–602. IEEE, 2008.

[33] Sean Brydon Ed Ort and Mark Basler. Mashup styles, part 1: Server-side mashups, May, 2007.

[34] Sean Brydon Ed Ort and Mark Basler. Mashup styles, part 2: Client-side mashups, August, 2007.

[35] Michael Michael Thomas Bolin. *End-user programming for the web*. PhD thesis, Massachusetts Institute of Technology, 2005.

[36] Ian Hickson and David Hyatt. Html5: A vocabulary and associated apis for html and xhtml. *W3C Working Draft edition*, 2011.

[37] Wikipedia. Multitier architecture — wikipedia, the free encyclopedia, 2013. [Online; accessed 1-December-2013].

[38] Jack Moffit. *Professional XMPP programming with JavaScript and jQuery*. Wiley Publishing, Inc., Indianapolis, Indiana, 2010.

[39] XMPP Standards Foundation. Xmpp extensions, 1999.

[40] Wikipedia. Angularjs — wikipedia, the free encyclopedia, 2014. [Online; accessed 19-February-2014].

# Appendix A

# Registry JSON Standard

In the Section 5.3 *Evaluation* only some part of a Regisry interface standard was introduced. The metadata which describes sensor and needed for dynamically build whole content and functional handlers, are shown in the Listing A.1, by using example of an ambient temperature sensor. Type of data provided by this sensor is a map of temperature values, which is updated every 5 seconds.

```
[
    {
        "id": "30",
        "title": "Ambient Temperature INF3084",
        "availability": true,
        "last_update": "2014-04-03T11:14:34.000+02:00",
        "description": "DSE students receive an opportunity to get current
            outdoor temperature in Dresden Momsenstr.20. This sensor
            provides temperature updates with 5-second frequency and
            represents a class of low-cost, high-performance sensors. It
            is implemented using a commercially available Raspberry Pi
            single-board computer with an affiliated USB thermometer and
            automatic WLAN and XMPP connections to a sensor MUC room
            established at boot time.",
        "sla": "Sensor resolution is 0.5 C, measurements taken every 10
            seconds. Uptime 95% from 6:00 till 22:00.",
        "sla_last_update":"1395005996",
        "icon": "img/icon/temper_outside.png",
        "preview": [
            "img/sensor_images/preview/id_1_3.png",
            "img/sensor_images/preview/id_1_2.png",
            "img/sensor_images/preview/id_1_1.png"
        ],
        "picture": "img/sensor_images/temper.jpg",
        "access": "private",
        "provider_name": "Provider TU Dresden",
        "location": "Dresden",
        "provider_www": "http://www.inf.tu-dresden.de/",
        "dev_details": "true",
        "responsible_team": "RN",
        "administrator": "Philipp Grubitzsch",
```

```
24              "EPConfig": "config30.pdf",
25              "end_points": [
26                  {
27                      "type": "muc",
28                      "name": "xmpp://inf3084@conference.mobilis-dev.inf.tu-
                            dresden.de",
29                      "pwd": null
30                  },
31                  {
32                      "type": "muc",
33                      "name": "xmpp://inf3086@conference.mobilis-dev.inf.tu-
                            dresden.de",
34                      "pwd": null
35                  }
36              ],
37              "type": "chart",
38              "template": {
39                  "subtitle": {
40                      "text": "Real-time updated"
41                  },
42                  "yAxis": {
43                      "title": {
44                          "text": "Temperature, C"
45                      }
46                  },
47                  "series": [
48                      {
49                          "data": [],
50                          "name": "Ambient Temperature INF3084"
51                      }
52                  ],
53                  "title": {
54                      "text": "Ambient Temperature INF3084"
55                  },
56                  "chart": {
57                      "type": "spline"
58                  },
59                  "xAxis": {
60                      "labels": {
61                          "rotation": -45
62                      },
63                      "type": "datetime"
64                  },
65                  "plotOptions": {
66                      "area": {
67                          "turboThreshold": 20
68                      }
69                  },
70                  "credits": {
71                      "enabled": false
72                  }
73              }
74          }
```

```
]
```

Listing A.1: JSON Description Format

For software sensor which produces text messages, a standart metadata descriptions are shown in the Listing A.2

```
[
    {
        "id": "20",
        "title": "IT News Feed",
        "availability": true,
        "last_update": "2014-04-02T11:14:34.000+02:00",
        "description": "Provides up to date news in IT and Telecommunication
           area together with information about new gadgets.",
        "sla": "",
        "sla_last_update": "1395663569",
        "icon": "img/sensor_images/itIcon.jpg",
        "picture": "img/sensor_images/itNews.jpg",
        "preview": "",
        "access": "public",
        "provider_name": "TU Dresden",
        "provider_www": "http://www.inf.tu-dresden.de/",
        "location": "Dresden",
        "dev_details": "true",
        "EPConfig": "config20.pdf",
        "end_points": [
            {
                "type": "muc",
                "name": "xmpp://testraum@conference.mobilis-dev.inf.tu-dresden
                   .de",
                "pwd": null
            }
        ],
        "type": "text",
        "template": ""
    }
]
```

Listing A.2: JSON Description Format for Software Sensor

# Appendix B

# XEP0049 saving user preferences

In the Section 5.3.2 *SensDash Implementation* the functionality to save and retrieve personal preferences of a user by using the XMPP private XML-based namespace are shown below:

```
{
    "20": [
        "0": {
            "handler_id": "20",
            "handler_type": "text",
            "name": "xmpp://testraum@conference.mobilis-dev.inf.tu-dresden.de"
                ,
            "pwd": null,
            "sla_last_update": "",
            "type": "muc"
        }
    ],
    "30": [
        "0": {
            "handler_id": "30",
            "handler_type": "chart",
            "name": "xmpp://inf3084@conference.mobilis-dev.inf.tu-dresden.de",
            "pwd": null,
            "sla_last_update": "1395005996"type: "muc"
        },
        "1": {
            "handler_id": "30",
            "handler_type": "chart",
            "name": "xmpp://chat1@conference.likepro.co",
            "pwd": null,
            "sla_last_update": "1395005996",
            "type": "muc"
        }
    ]
}
```

But in comparison to subscriptions map, favorites are saved in array which consists only ids of sensors: ["20", "30"].

# Appendix C

# AngularJS and Reliable Sensor Functionality

In the Section 5.3.2 *SensDash Implementation* the one of the main functional characteristic of a sensor, namely reliability was implemented.

Firstly, the list of end-points in Registry (attribute: "end_points") have to be sorted based on their priority on a server side, example are in the Appendix A. Than, SensDash checks if the room is not empty and also if it has someone except of a user. The part of a code below are shown below:

```
1  check_room: function(full_room_name, end_points) {
2              if (!xmpp.endpoints_to_handler_map[full_room_name]) {
3                  return;
4              }
5              var ep = xmpp.endpoints_to_handler_map[full_room_name];
6              if (ep.participants.length == 0) {
7                  console.log("Room is empty, endpoint rejected: " +
                       full_room_name);
8                  for (var i = 0; i < end_points.length; i++) {
9                      // find invalid room in all end_points list, delete it,
                           and call subscribe again
10                     if (end_points[i].handler_id == ep.handler_id) {
11                         xmpp.unsubscribe(end_points[i]);
12                         end_points.splice(i, 1);
13                         xmpp.subscribe(end_points);
14                     }
15                 }
16             } else {
17                 console.log("Room is not empty, endpoint approved: " +
                       full_room_name);
18             }
19         }
```

This functionality enhance the basic list of xmpp-based functions and called for xmpp subscriptions(joining the room).

# Appendix D

# AngularJS factory for XMPP Connection

In the Section 5.3.2 *SensDash Implementation* in order to handle all requests which goes through the XMPP, was implemented next abstract methods which helps to invoke XMPP via Strophe.js:

```
1  sensdash_services.factory("XMPP", ["$location", "Graph", "Text", function ($
       location, Graph, Text) {
2      if (typeof Config === "undefined") {
3          console.log("Config is missing or broken, redirecting to setup
               reference page");
4          $location.path("/reference");
5      }
6      var BOSH_SERVICE = Config.BOSH_SERVER;
7      var PUBSUB_SERVER = Config.PUBSUB_SERVER;
8      var PUBSUB_NODE = Config.PUBSUB_NODE;
9      var xmpp = {
10         connection: {connected: false},
11         endpoints_to_handler_map: {},
12         received_message_ids: [],
13         // logging IO for debug
14         raw_input: function (data) {
15             console.log("RECV: " + data);
16           },
17         // logging IO for debug
18          raw_output: function (data) {
19              console.log("SENT: " + data);
20          },
21         connect: function (jid, pwd, callback) {
22             xmpp.connection = new Strophe.Connection(BOSH_SERVICE);
23             xmpp.connection.connect(jid, pwd, callback);
24         //     xmpp.connection.rawInput = xmpp.raw_input;
25         //     xmpp.connection.rawOutput = xmpp.raw_output;
26         },
27         subscribe: function (end_point, on_subscribe) {
28             xmpp.endpoints_to_handler_map[end_point.name] = end_point;
29             var jid = xmpp.connection.jid;
```

```
30              if (end_point.type == "pubsub") {
31                  xmpp.connection.pubsub.subscribe(
32                      jid,
33                      PUBSUB_SERVER,
34                      PUBSUB_NODE + "." + sensor_map,
35                      [],
36                      xmpp.handle_incoming_pubsub,
37                      on_subscribe);
38                  console.log("Subscription request sent,", end_point);
39              } else if (end_point.type == "muc") {
40                  var nickname = jid.split("@")[0];
41                  var room = end_point.name.replace("xmpp://",'');
42                  xmpp.connection.muc.join(room, nickname, xmpp.
                        handle_incoming_muc);
43                  on_subscribe();
44              } else {
45                  console.log("End point protocol not supported");
46              }
47          },
48          unsubscribe: function (end_point, on_unsubscribe) {
49              var jid = xmpp.connection.jid;
50              if (end_point.type == "pubsub") {
51                  xmpp.connection.pubsub.unsubscribe(PUBSUB_NODE + "." +
                        end_points);
52                  on_unsubscribe();
53              } else if (end_point.type == "muc") {
54                  var room = end_point.name.replace("xmpp://",'');
55                  xmpp.connection.muc.leave(room, jid.split("@")[0]);
56                  on_unsubscribe();
57              }
58          },
59          find_sensor: function (message) {
60              var endpoint_name = message.getAttribute('from');
61              //to get to the end_points.name and add "xmpp://"
62              endpoint_name = "xmpp://" + endpoint_name.replace(/\/\w+/g, '');
63              if (endpoint_name in xmpp.endpoints_to_handler_map) {
64                  var handler_id = xmpp.endpoints_to_handler_map[endpoint_name].
                        handler_id;
65                  var handler_type = xmpp.endpoints_to_handler_map[endpoint_name
                        ].handler_type;
66                  return {"id": handler_id, "type": handler_type};
67              } else {
68                  // endpoint not found in subscriptions map
69                  return false;
70              }
71          },
72          handle_incoming_muc: function (message) {
73              var sensor = xmpp.find_sensor(message);
74              if (!(sensor)) {
75                  // sensor not found
76                  return true;
77              }
```

```
78              var text = Strophe.getText(message.getElementsByTagName("body")
                    [0]);
79              if (sensor.type == 'text') {
80                  if (typeof text == "string") {
81                      Text.updateTextBlock(text, sensor["id"]);
82                  } else {
83                      console.log("Message is not a Text");
84                  }
85              } else if (sensor.type == 'chart') {
86                  try {
87                      text = text.replace(/&quot;/g, '"');
88                      var msg_object = JSON.parse(text);
89                      var data_array = [];
90                      console.log("JSON message parsed: ", msg_object);
91                      //creating a new array from received map for Graph.update
                            in format [timestamp, value], e.g. [1390225874697, 23]
92                      if ('sensorevent' in msg_object) {
93                          var time_UTC = msg_object.sensorevent.timestamp;
94                          var time_UNIX = new Date(time_UTC).getTime();
95                          data_array[0] = time_UNIX;
96                          data_array[1] = msg_object.sensorevent.values[0];
97                      } else {
98                          data_array = msg_object;
99                      }
100                     console.log(data_array);
101                 } catch (e) {
102                     console.log("message is not valid JSON", text);
103                     return true;
104                 }
105                 if (Array.isArray(data_array)) {
106                     Graph.update(data_array, sensor.id);
107                 }
108             }
109             return true;
110         },
111         handle_incoming_pubsub: function (message) {
112             if (!xmpp.connection.connected) {
113                 return true;
114             }
115             var server = "^" + Client.pubsub_server.replace(/\./g, "\\.");
116             var re = new RegExp(server);
117             if ($(message).attr("from").match(re) && (xmpp.
                    received_message_ids.indexOf(message.getAttribute("id")) ==
                    -1)) {
118                 xmpp.received_message_ids.push(message.getAttribute("id"));
119                 var _data = $(message).children("event")
120                     .children("items")
121                     .children("item")
122                     .children("entry").text();
123                 var _node = $(message).children("event").children("items").
                        first().attr("node");
124                 var node_id = _node.replace(PUBSUB_NODE + ".", "");
125
```

```
126                 if (_data) {
127                     // Data is a tag, try to extract JSON from inner text
128                     console.log("Data received", _data);
129                     var json_obj = JSON.parse($(_data).text());
130                     Graph.update(json_obj, node_id);
131                 }
132             }
133             return true;
134         }
135     };
136     return xmpp;
137 }]);
```

Which has main methods: XMPP.connect(), XMPP.subscribe()(In case of MUC chat room it means join the room), XMPP.unsubscribe()(For MUC chat rooms it means leave the room), XMPP.handle_incoming_muc(), XMPP.handle_incoming_pubsub(). Last two functions handles MUC or PubSub connection, based on the type of end_point, defined in the Registry.