Chair of Computer Networks

SensDash Manual

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

Contents

Chapter 1

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.

1.1 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 1.1 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.

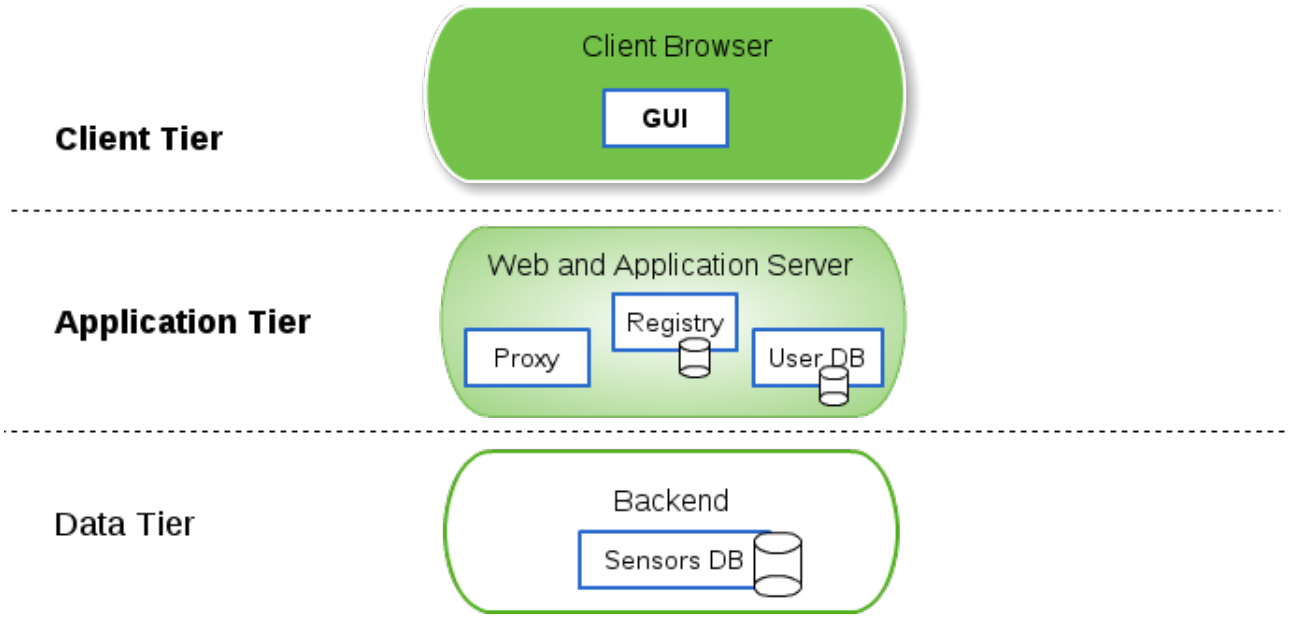

Figure 1.1: 3-tier Architecture

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 have 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 defined by system way and provides an access to the description and data itself through the standardized intreface.

From a historical perspective the three-tier architecture concept emerged in the 1990s from observations of distributed systems[**?**] (e.g., web applications) where the client, application and data tiers ran on physically separate platforms. Nowadays, MVC and similar model-view-presenter (MVP) are used as separation of concerns techniques. 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. Thus, to ensure highly adaptive GUI independently from a data and application tiers, MVC pattern come 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.

1.2 Client Tier

The client tier or another name is the presentation tier is a layer which user can directly access from any type of portable device. In the Section 3.2 was proved necessity to implement web based portable application, which a user can access 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.1.2), 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.

1.2.1 Web-based GUI Composition

Figure 1.2 presents a simple content layout that have to be presented on a web-page in order to satisfy all possible user requirements. It contains:

- *Main five 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 become possible to get real-time streaming data. Users that have an admin

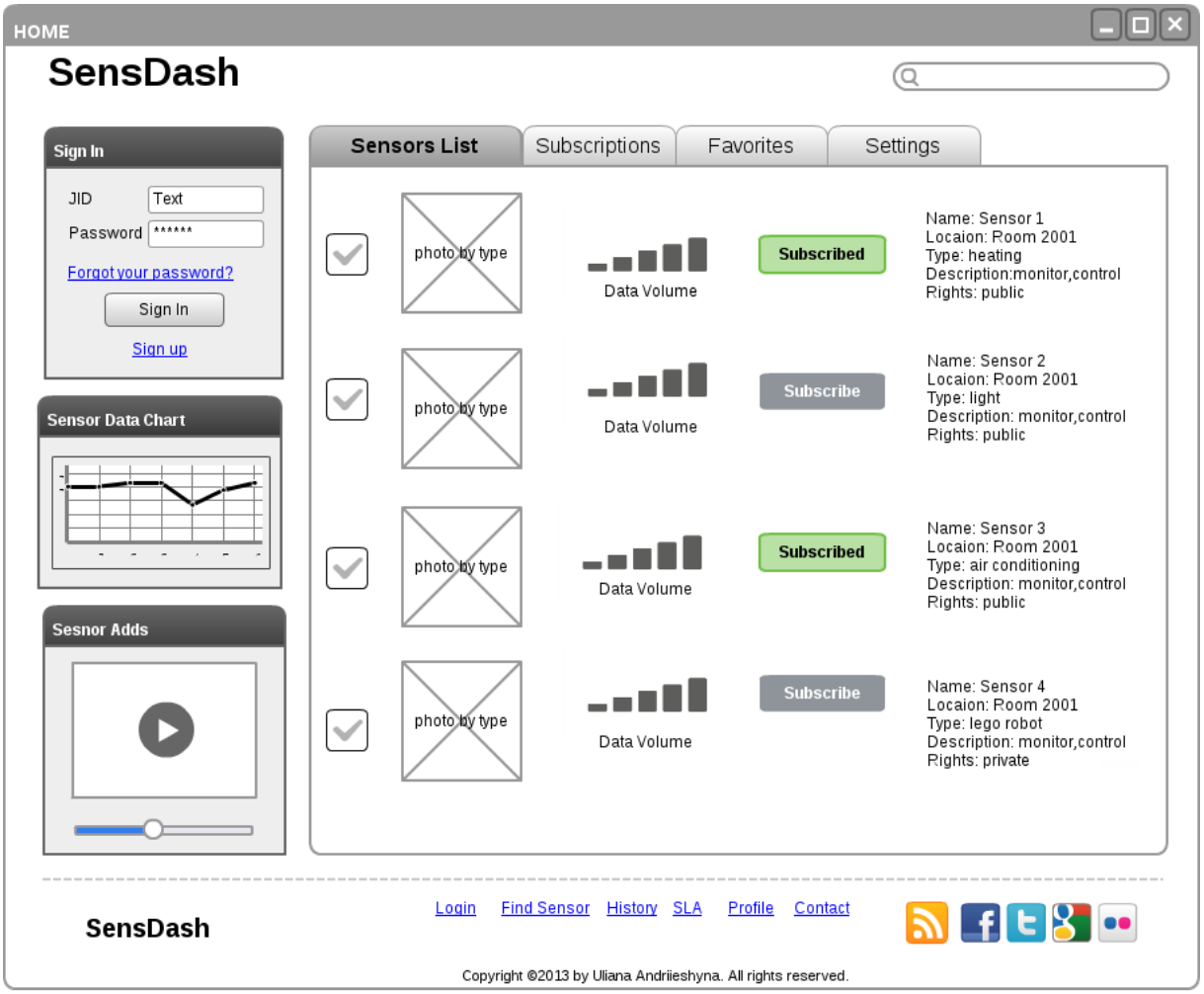

Figure 1.2: GUI Mockup

rights receives an opportunity to manage sensors. Simple user without privileges, receives an opportunity to get statistic and information from sensors and to maintain his/her own account data.

- *Sensor icon* defines what is the current type of sensor, e.g. light, temperature, heating, robot lego battery status etc. It helps easily, even in seconds, understand and catch what is the main function of a sensor in the list, especially helpful wiil be to use famous vendor's icon.

- *Availability or unavailability* of a data source. User can subscribe only to the online services. If some services become offline it will be automatically marked as inactive and after refreshing of the page will be deleted from the list of available sensors. As soon as new data will be sended, 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 have 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 self-define how possible to get such type of streaming data according to his/her available 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 very 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 1.3. 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 log in to the dashboard, he/she can explore all available sensors. If he/she already the user of a dashboard all his/her preferences will be loaded from a server automatically and appear in respective tabs.

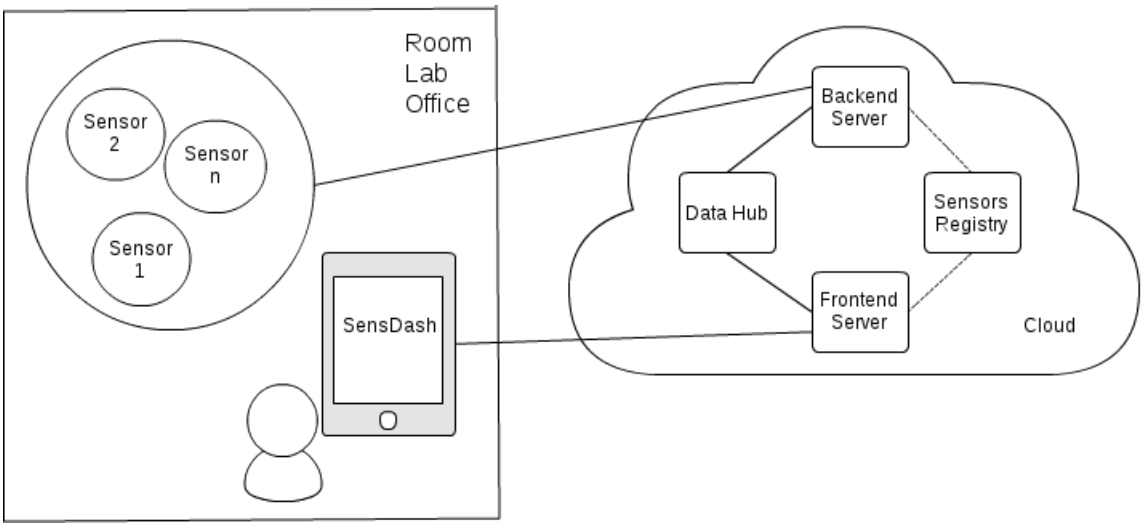

Figure 1.3: Use Case

1.2.2 JavaScript MVC

As 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 1.4, takes place common Model-View-Controller pattern. Where a 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.

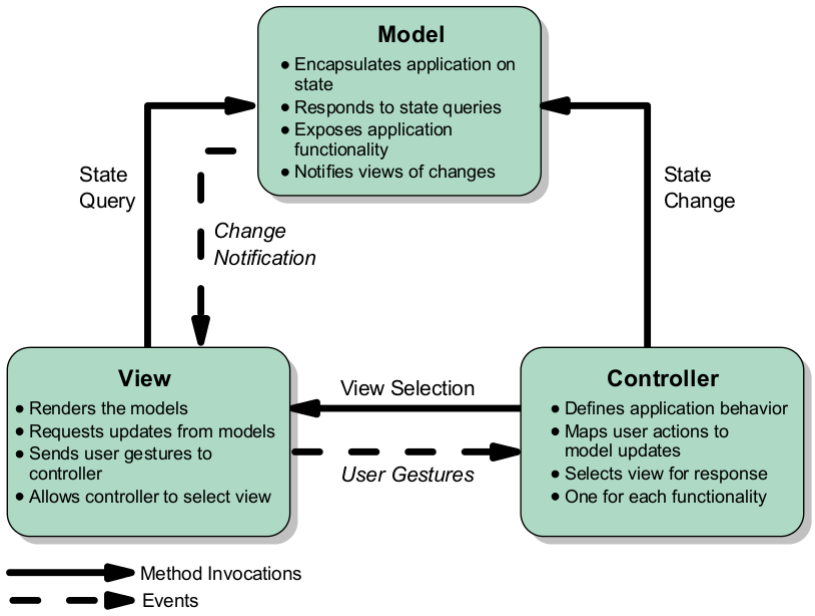

Figure 1.4: MVC Pattern

Not necessary to follow the MVC pattern strictly. The idea of all the patterns is to separate Model, View and Logic that hooks the two behind which is the controller for the best separate of concerns.

1.3 Application Tier

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

Application tier consists all logical modules: Web server, Registry, Data Hub and Web-based Frontend. All these modules connect to each other as shown on the Figure 1.5.

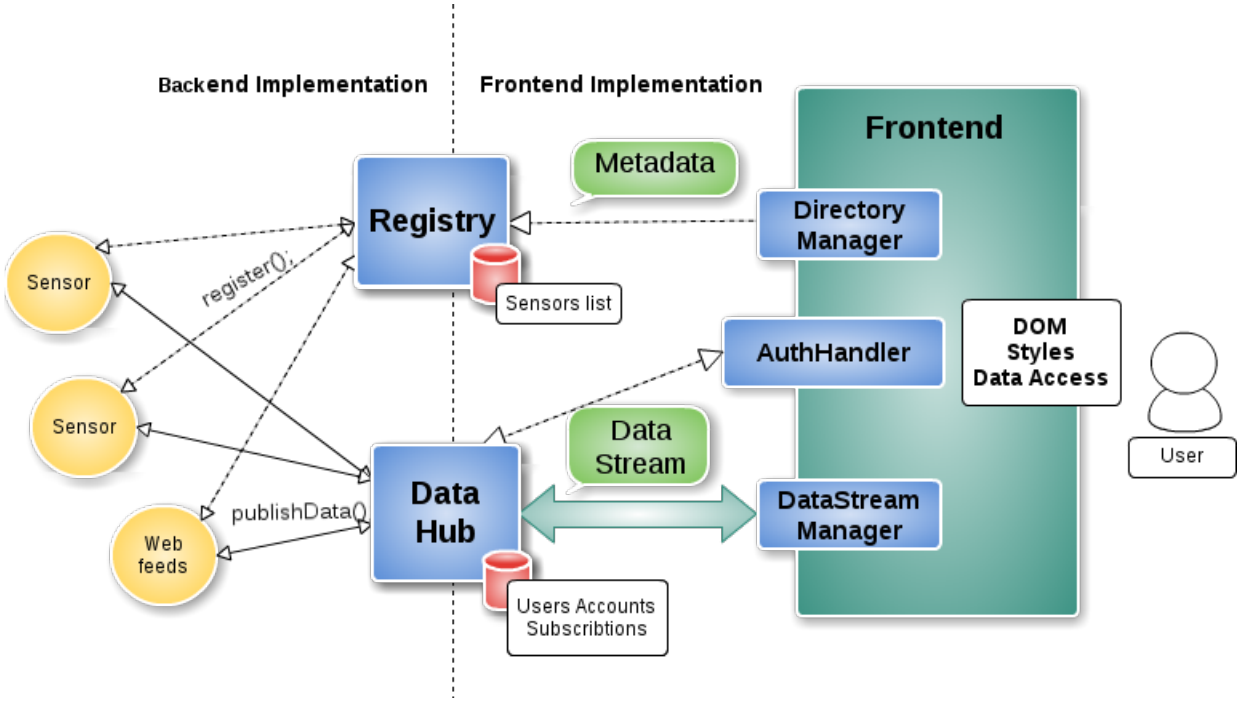

Figure 1.5: System Architecture

The System Architecture is splitted to a backend and frontend, that gives an overview of how these two parts interconnected. 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. To the backend responsibility relate an implementation of Registry and Data Hub modules.

1.3.1 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.

1.3.2 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.

1.3.3 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 1.5 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 1.6. Such architecture decouples sensor specific interface and interface between frontend and backend.

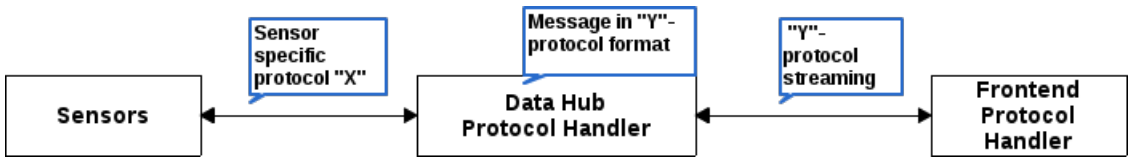

Figure 1.6: Protocol flow

In order to satisfy all aforementioned requirements, two major protocols have been found: *XMPP*[**?**]: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. 1.7[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/`

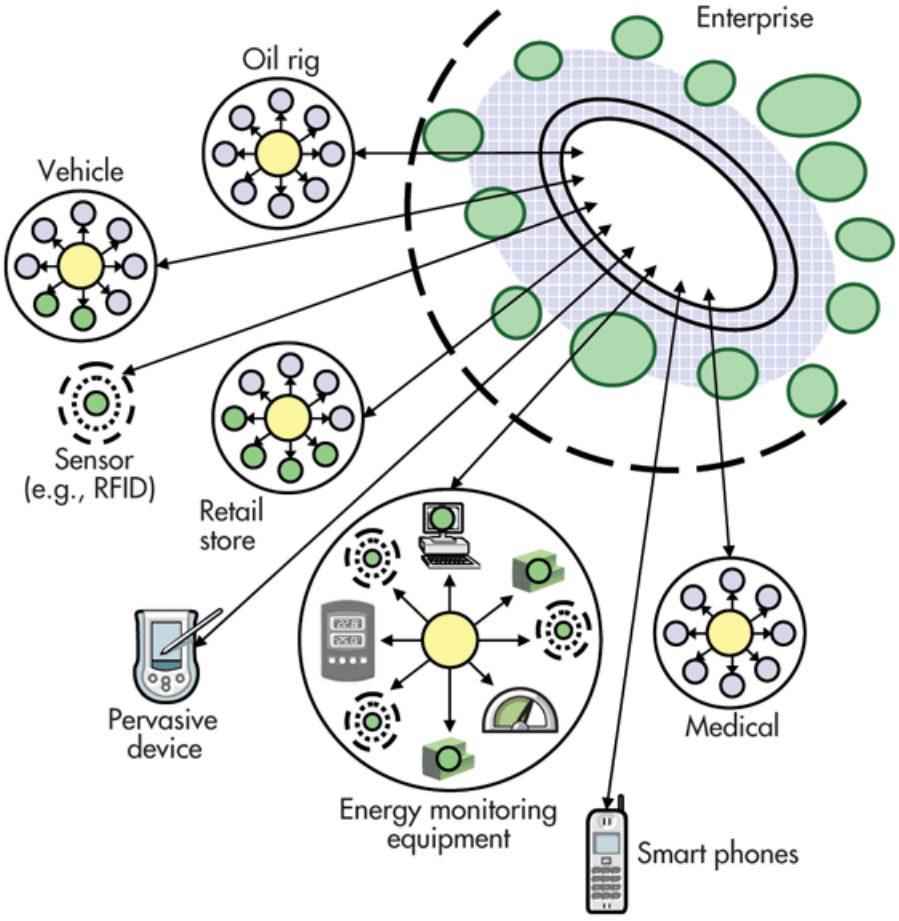

Figure 1.7: Message Queue Telemetry Transport

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. 1.8[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`

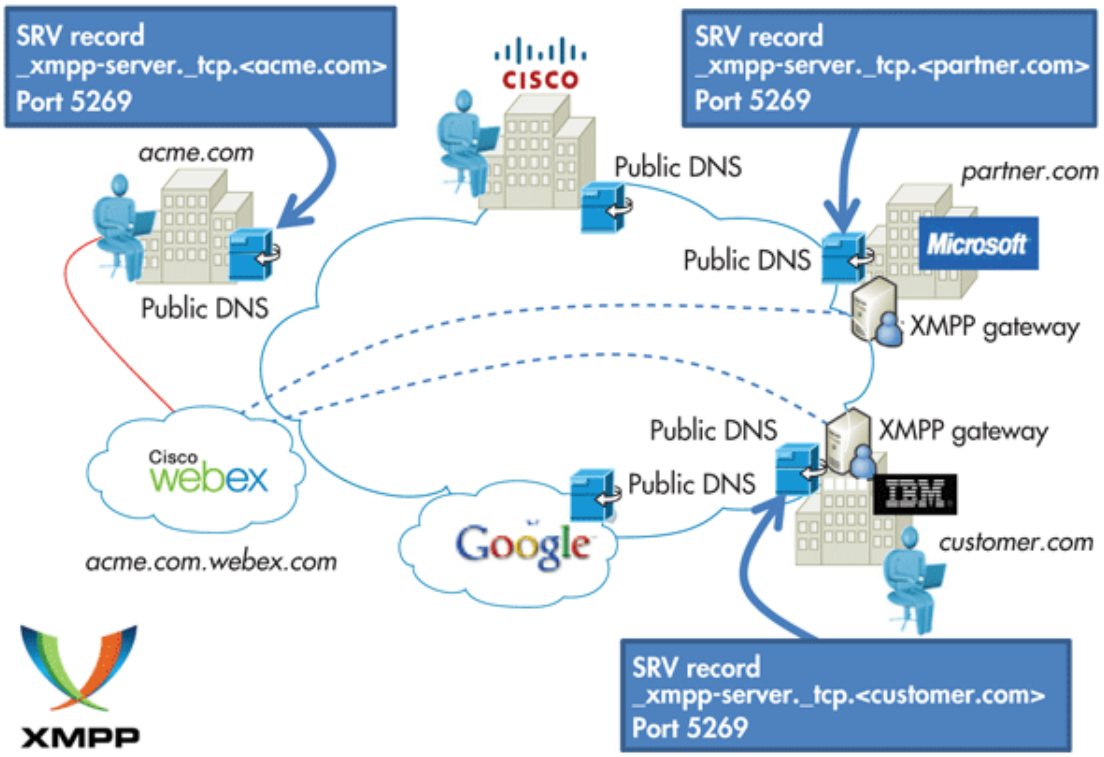

Figure 1.8: The Extensible Messaging and Presence Protocol (XMPP)

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[?].

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.

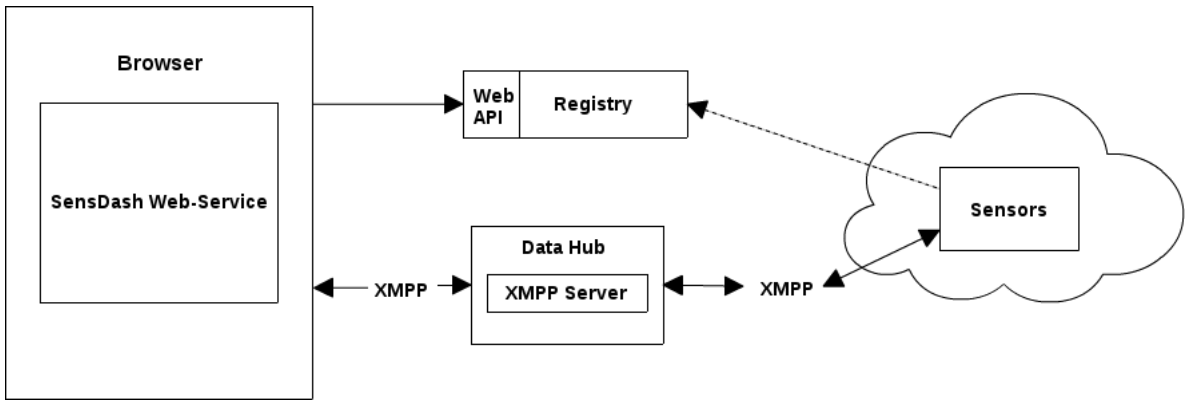

Figure 1.9: Interface

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.

1.3.4 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.

1.4 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 1.10 a). A reliable sensor has two, where first one is primary, and next serves as backup/failover

(Figure 1.10 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

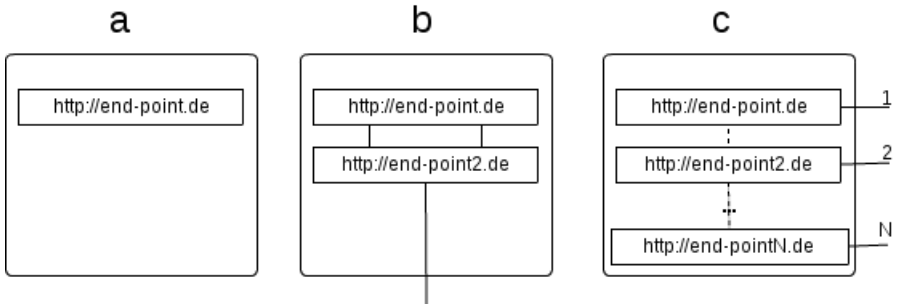

Figure 1.10: Sensor Functional Characteristics

end-points (Figure 1.10 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 1.10 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.

1.5 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 2

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.

2.1 Implementation requirements

2.1.1 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 2.1: 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 2.2: 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/

2.1.2 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 2.3.1. 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 1.3.3, 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[?]. 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 2.1) that allows automatic synchronization of models and views. As a result, AngularJS deemphasizes DOM manipulation and improves testability.

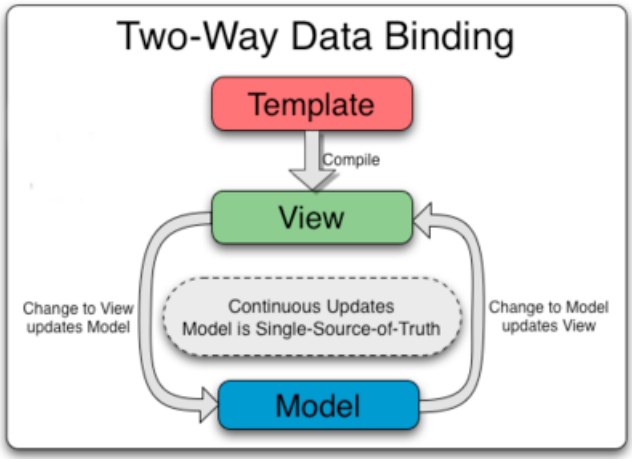

Figure 2.1: Two-way data binding[13]

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 2.2. 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.

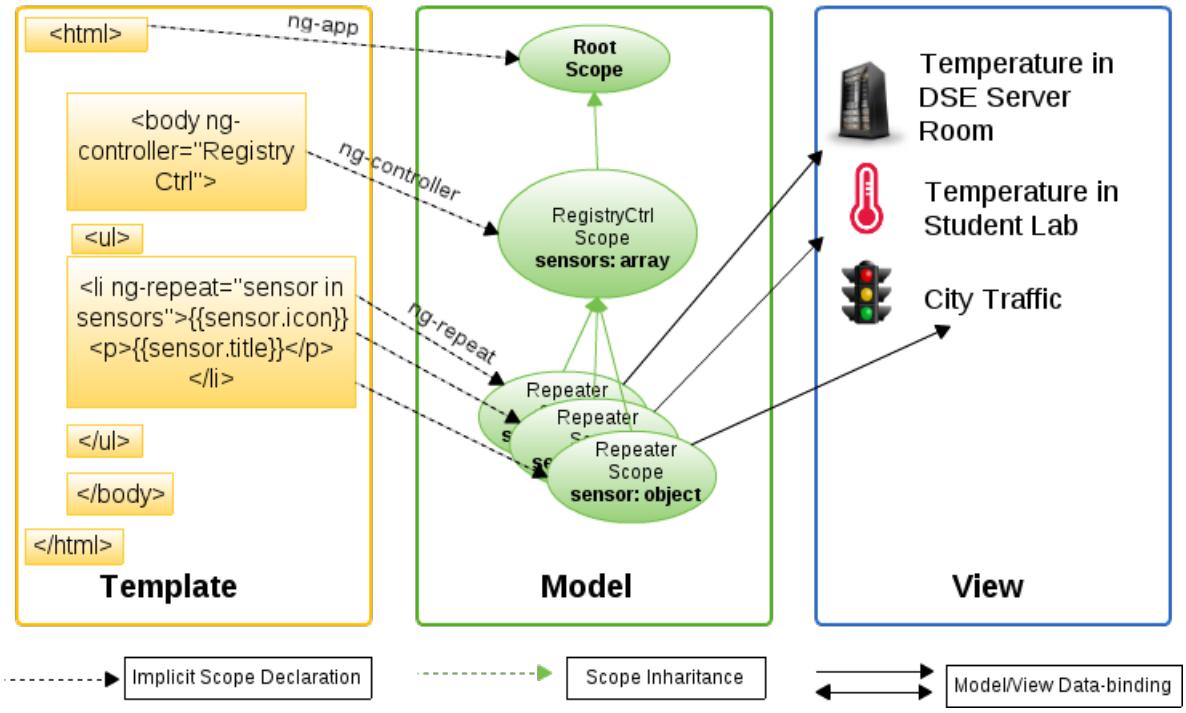

Figure 2.2: Template Model View

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 2.1.

```
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 2.1: 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 2.1. Next, listing 2.2 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 2.2: Registry Controller

2.1.3 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 2.3):

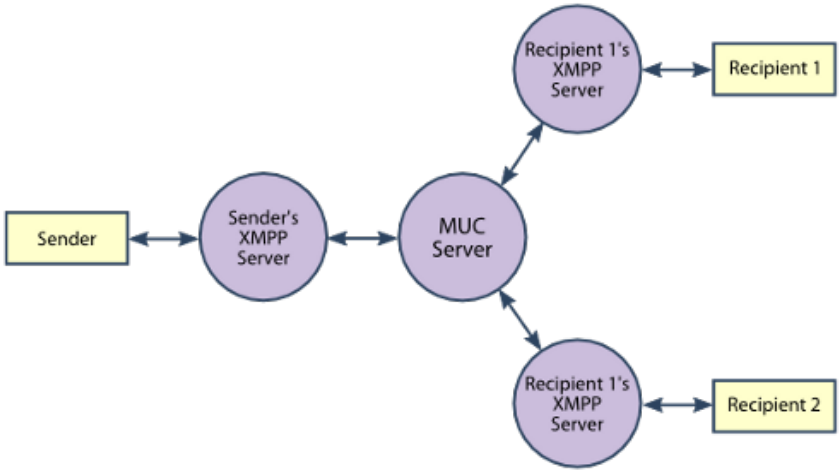

Figure 2.3: MUC System structure

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 2.3:

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

Listing 2.3: 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 2.4.

```
<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 2.4: 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 2.5, 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 2.5: 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 [?].

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 2.4.

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 2.6, 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`

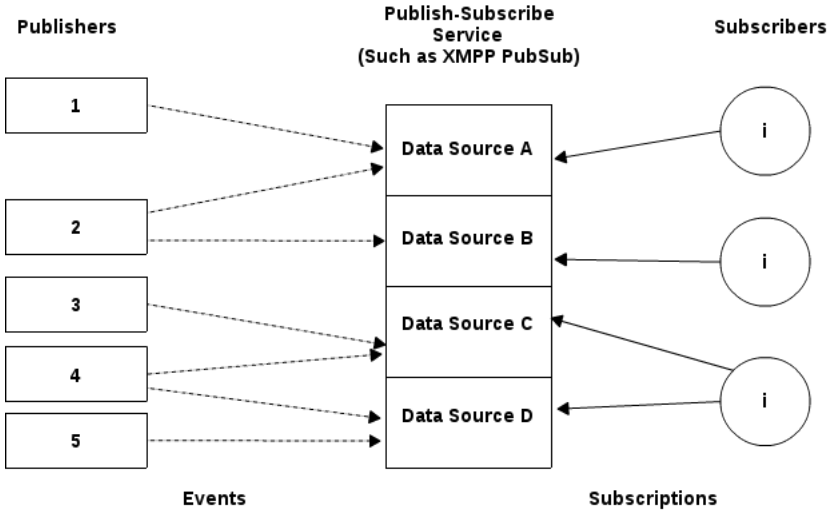

Figure 2.4: General Event Subscription

```
<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 2.6: 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 2.7, 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 2.7: 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

Figure 2.5 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.

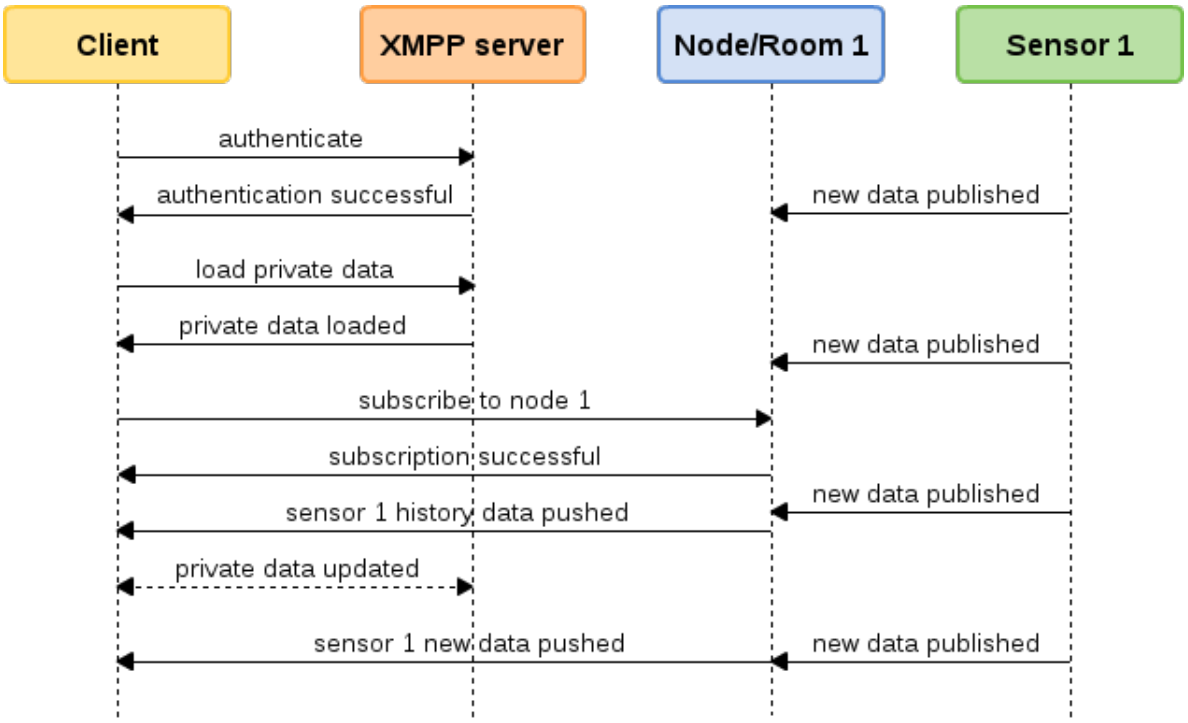

Figure 2.5: Service PubSub

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 2.3.

	Publish-Subscribe	**Multi-User Chat**
Advantages	• Pubsub extension is generic, assuming nothing about the subscribers. • Pubsub nodes and subscriptions are arranged in a tree-based hierarchy. • Events can be published as notifications or as full payloads, and the subscriber can choose which is most appropriate. • Retrieval of the publishing history is built in and fine grained. • 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. • Presence handling is built in to MUC at a low level. • Common moderation, administration and privelege features are supported. • MUC has many implementations, both of clients and of servers. • 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. • Support for Pubsub features varies in quality and depth, e.g. no tools for node creation and configuration. • No special handling of presence built in. • Tooling for pubsub node creation and configuration is lacking. • 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. • There is no way to organize chat rooms except as a flat hierarchy. • There is no way to share configurations or participation across collections of rooms. • Unlike pubsub, MUC implementations have a lot of edge cases in order to be user friendly and robust.

Table 2.3: 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.

2.2 Interface Implementation

General architecture, as desined in the concept chaper, is shown on image 2.6

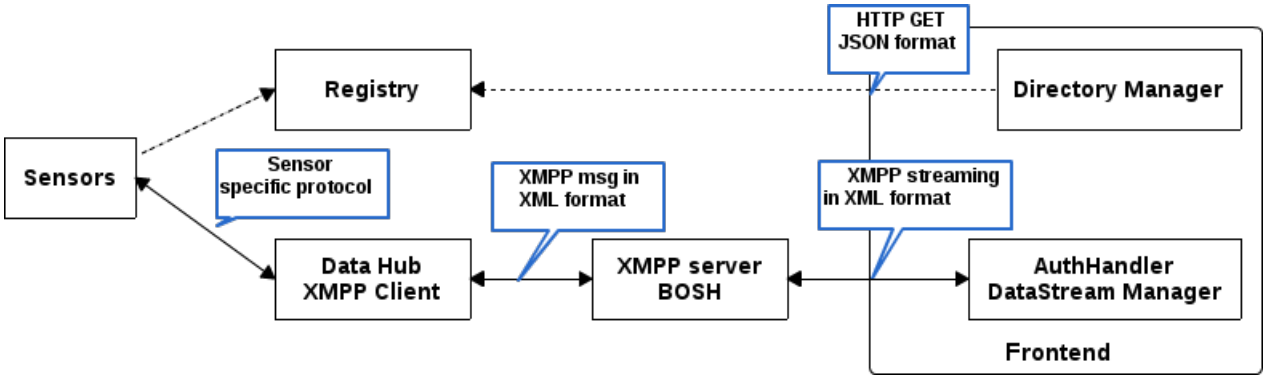

Figure 2.6: System Interfaces

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.

2.2.1 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 2.8. 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 2.8: 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.

2.2.2 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 2.9. 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 2.9: 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 2.10:

```
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 2.10: 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 2.7). 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.

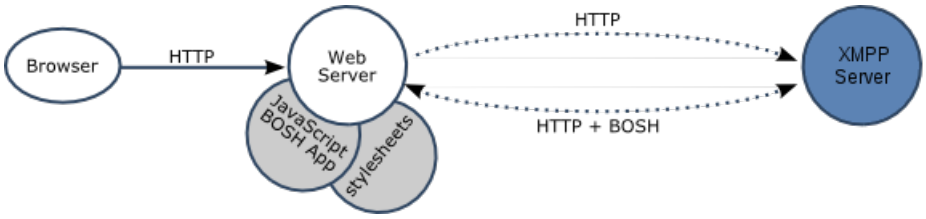

Figure 2.7: XMPP BOSH

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 2.11):

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

Listing 2.11: 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 2.12 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 2.12: Simplified MUC handler

2.2.3 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 2.8.

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 2.8: 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 2.13 and 2.14.

```
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 2.13: 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 2.14: 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 2.15. 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 2.15: Snippet of Save/Load preferences to a private namespace

2.3 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 2.9, 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 2.9.

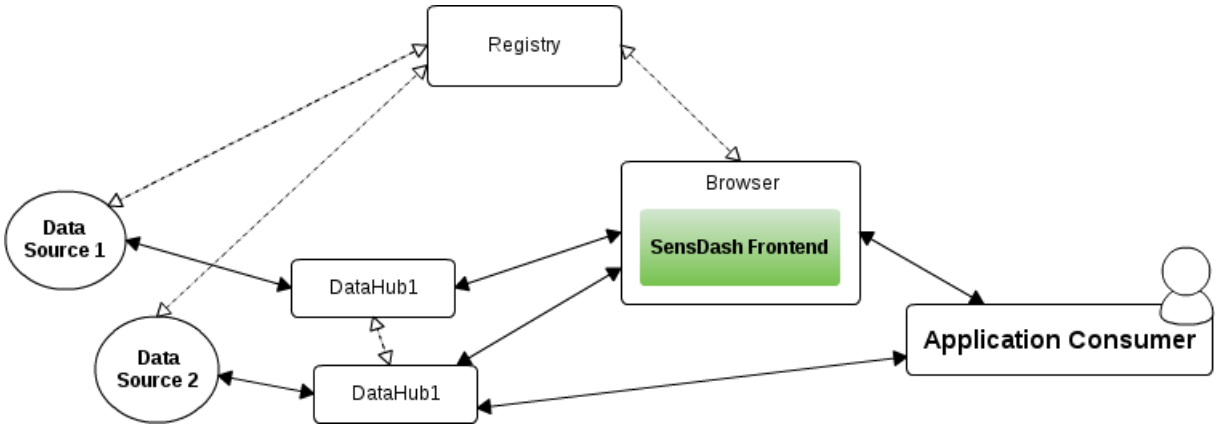

Figure 2.9: System Architecture Scheme

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 2.16.

```
1 {
2     "sensormuc": {
3         "type": "AMBIENT_TEMPERATURE",
4         "format": "short",
5         "location": {
6             "countryCode": "DE",
7             "cityName": "Cottbus",
8             "latitude": 51.076834,
```

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

Listing 2.16: 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 2.17 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 2.17: 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.

2.3.1 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. 2.10) or auto-created by XMPP registration software.

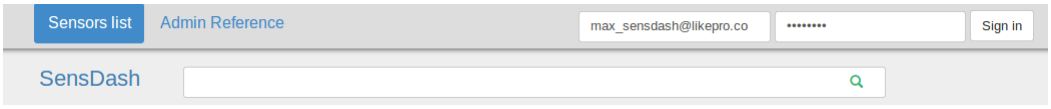

Figure 2.10: Log in to the SensDash

Step 2: After successfully completing the authentification process, Max can start browsing, searching and filtering sensors, as shown on the Figure 2.11.

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 2.12) 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.

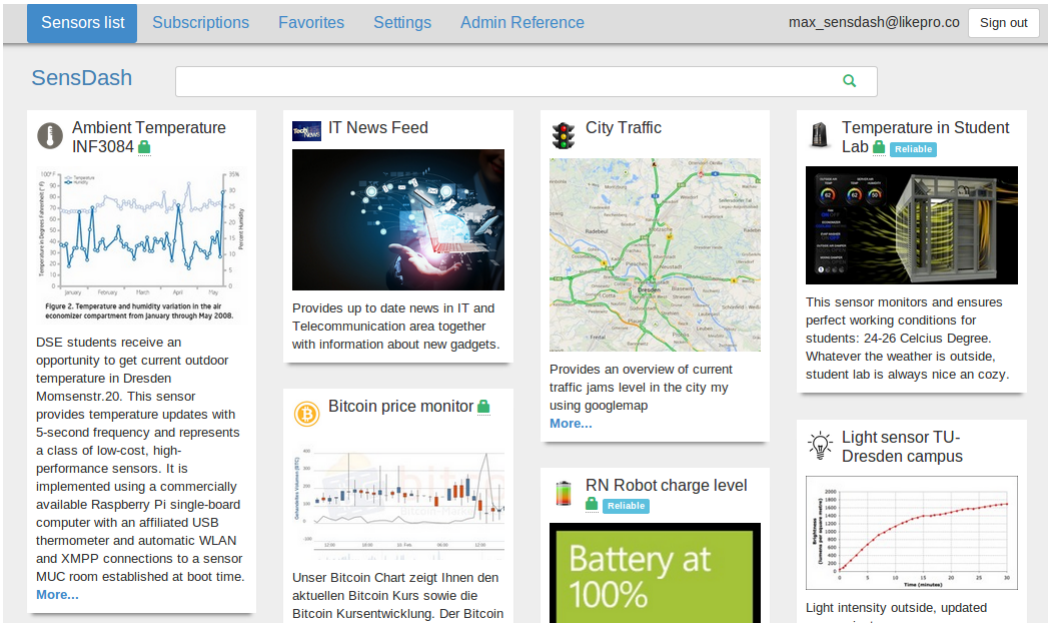

Figure 2.11: List of available sensors

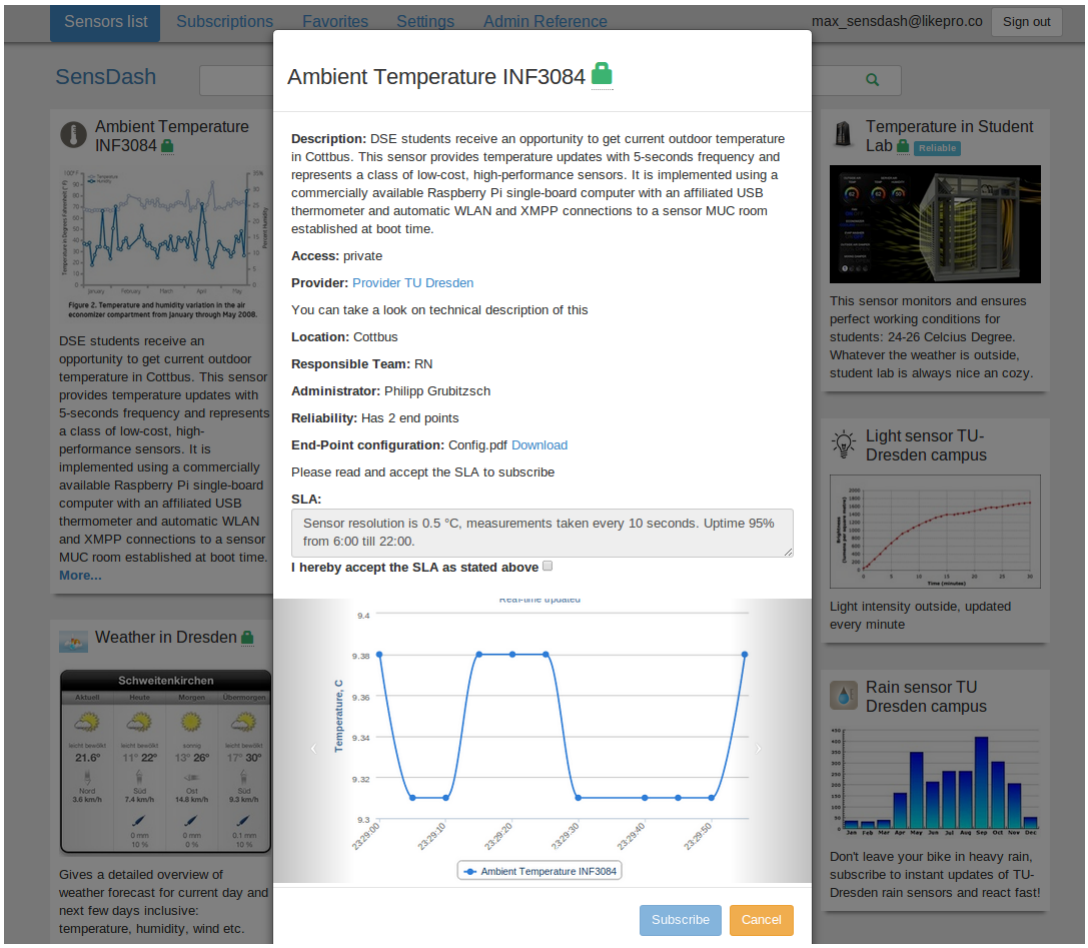

Figure 2.12: Personal Modal of a Sensor

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.

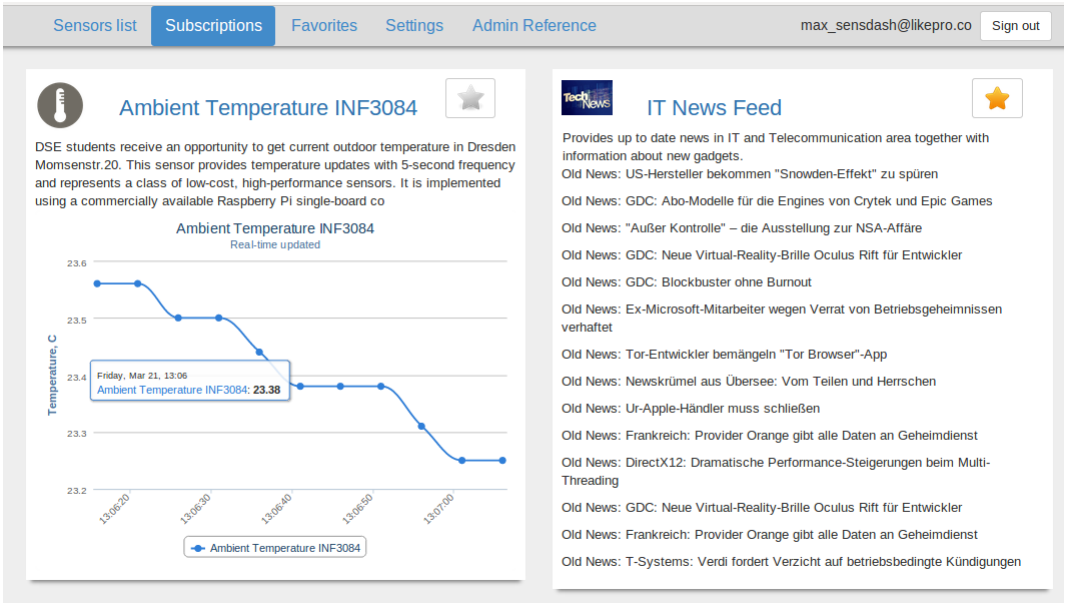

Figure 2.13: Subscriptions list

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 2.13). 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 2.14.

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. 2.15), where he can find all available implementation detailes of a SensDash, e.g. used XMPP services and extensions, interface workflow etc.

2.3.2 SensDash Implementation

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

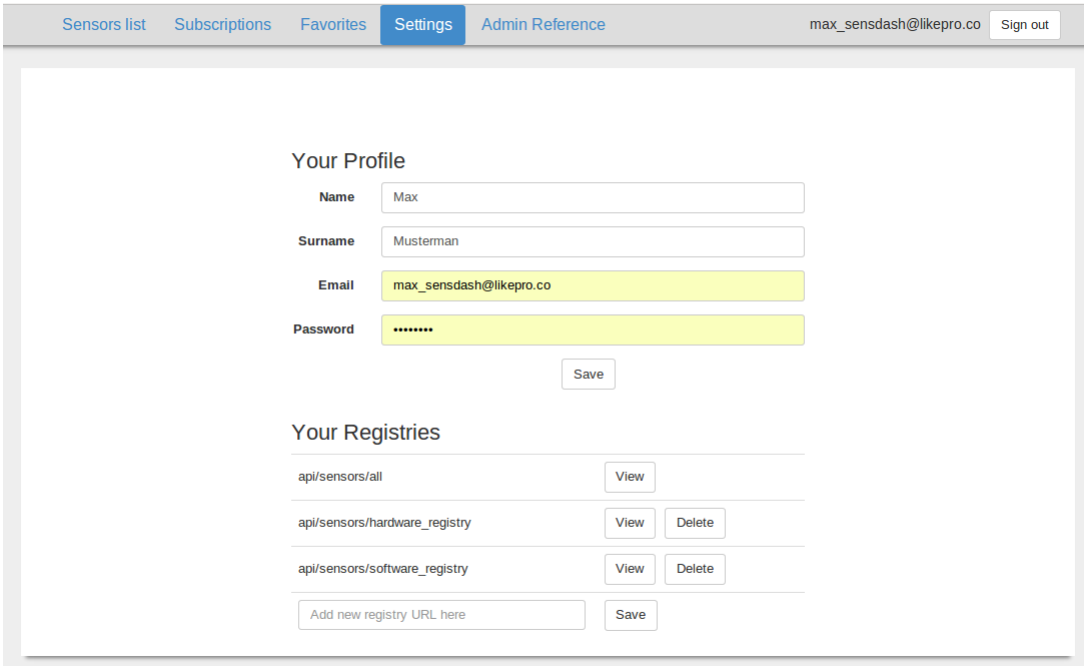

Figure 2.14: Settings Tab

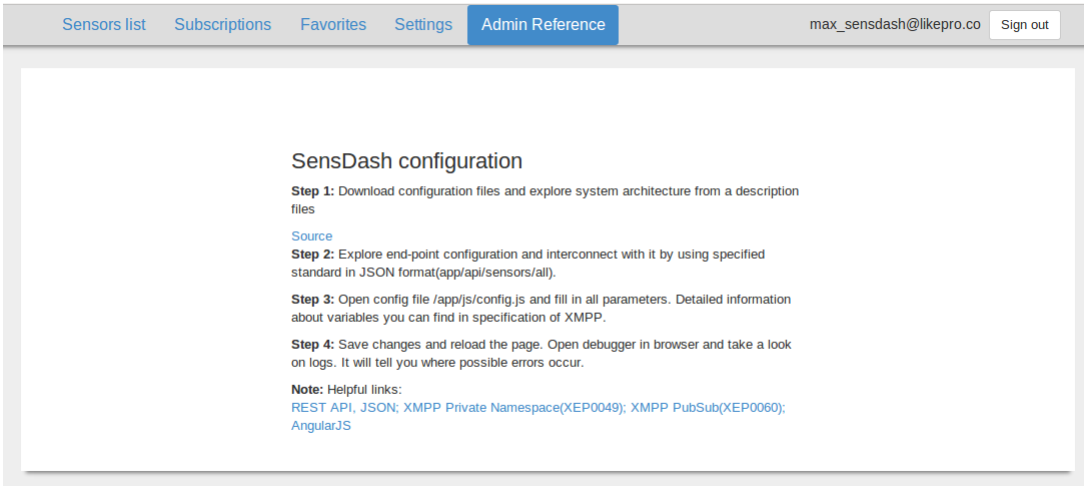

Figure 2.15: Administrator Tab

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 2.2.3). 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. 2.16). 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 2.17. The rules of how the reliability level was calculated is described in the Section 4.4, Subsection "Sensor Functional Characteristics" (Appendix C).

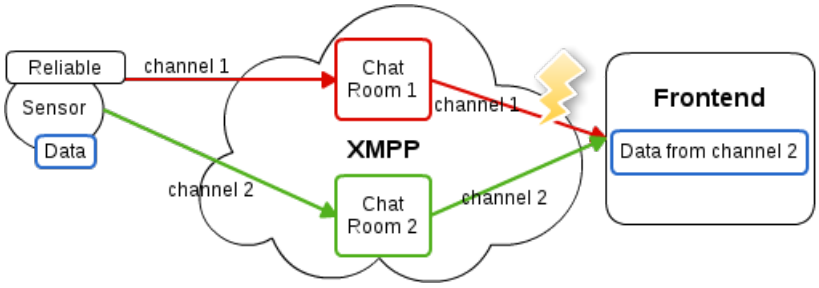

Figure 2.16: Data secure transfer using Chat Rooms

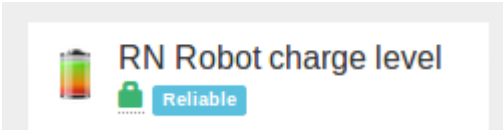

Figure 2.17: GUI identification of security and reliability level

2.4 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 2.18.

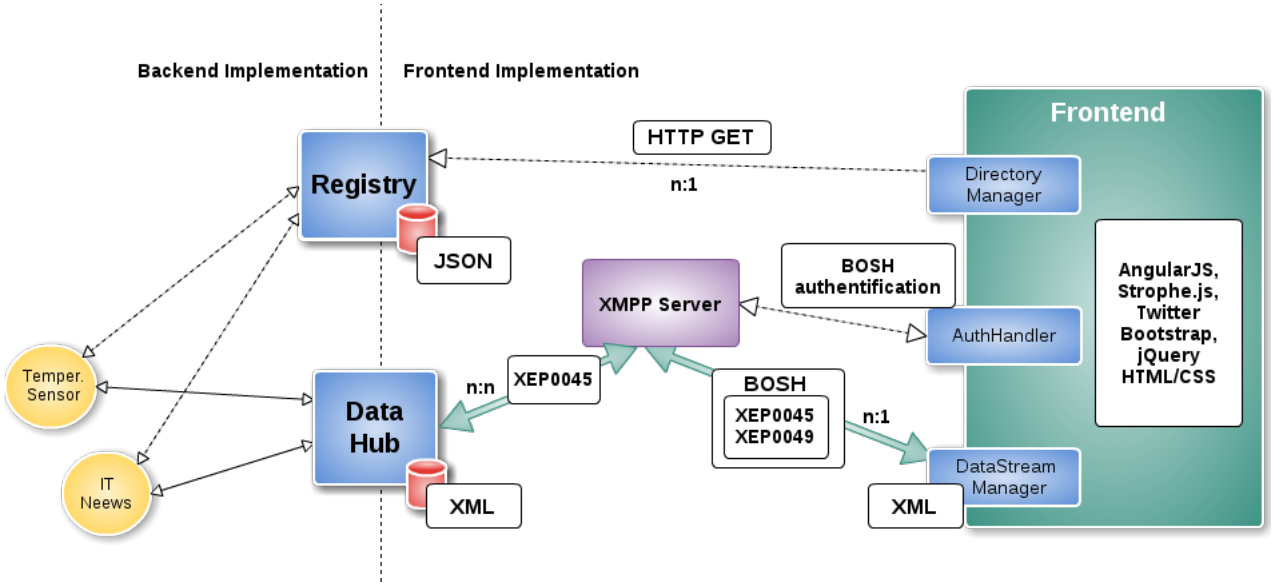

Figure 2.18: Integrated Implementation Architecture

Appendix A

Registry JSON Standard

In the Section 5.3 *Evaluation* was introduced only some part of a Regisry interface standrad. The whole metadata which is needed from sensor in order to dynamically build whole content and functional handlers are shown in the Listing A.1, on the example of 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 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 in the subscriptions map:

```
{
    "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 of a user the his/her personal favorites are saved in array, which consists only ids of sensor: ["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 defined in Registry as "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 skeleton:

```
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 functions: 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, given in a Registry.