Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Tatu Paronen

A web-based monitoring system for the Industrial Internet

Master's Thesis
Espoo, April 13, 2015

Supervisor:	Professor Jukka K. Nurminen
Advisor:	Jouni Aro M.Sc. (Tech.)

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Tatu Paronen		
Title: A web-based monitoring system for the Industrial Internet			
Date:	April 13, 2015	**Pages:**	viii + 99
Major:	Data Communications Software	**Code:**	T-110
Supervisor:	Professor Jukka K. Nurminen		
Advisor:	Jouni Aro M.Sc. (Tech.)		

In the shift towards an Industrial Internet, programmable logic controllers, used to drive processes in industrial facilities, are more intelligent and capable of mutual communication and independent decision-making. Essentially, these controllers are also directly connected to the Internet, which opens new possibilities for the implementation of process control and monitoring systems.

The goal of this thesis is to assess the suitability of web technologies for the development of supervisory control and data acquisition systems. The assessment is performed by developing a web-based monitoring system and comparing it to other solutions currently in the market. The available features and usability are used as the criterion for the comparison. The OPC UA standard for industrial information modeling and data transfer is used as the Industrial Internet platform when developing the application.

The practical part of the thesis is concerned with the development of a generic OPC UA web client. The web client serves as a framework for the development of web-based supervisory control systems that take advantage of the standard data modeling features of OPC UA. Modern web technologies enable the implementation of periodic updates and reusable user interface components, and a uniform service interface transfers the application state with service requests. A central theme in the thesis is the integration of software components written in the Java and JavaScript programming languages.

The implementation shows that web techniques can be used to implement both request/response based and event-based information transfer between logic controllers and end-user devices. The standardization of reusable web components makes the development of web-based industrial monitoring systems more attractive by allowing to combine functionality and style definitions into uniform building blocks. Finally, incompatibility issues between OPC UA sessions and the RESTful architecture were identified and assigned as subjects of future work.

Keywords:	OPC UA, Industrial Internet, Java, JavaScript, REST
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

Aalto-yliopisto
Perustieteiden
korkeakoulu

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Tatu Paronen		
Työn nimi:			
Web-pohjainen valvomojärjelmä teolliseen internetiin			
Päiväys:	13. huhtikuuta 2015	**Sivumäärä:**	viii + 99
Pääaine:	Tietoliikenneohjelmistot	**Koodi:**	T-110
Valvoja:	Professori Jukka K. Nurminen		
Ohjaaja:	DI Jouni Aro		

Teollisen internetin suuntauksessa teollisuuslaitoksia ohjaavat ohjelmoitavat logiikkapiirit ovat aikaisempaa älykkäämpiä, kyeten keskinäiseen kommunikointiin sekä itsenäiseen päätöksentekoon. Oleellisesti nämä logiikat ovat myös suoraan kytkettyinä internetiin, mikä avaa uusia mahdollisuuksia niin prosessiohjauksen kuin valvomojärjestelmienkin toteuttamiselle.

Tämän diplomityön tarkoituksena on arvioida web tekniikoiden soveltuvuutta valvomojärjestelmien kehitykseen. Arviointi perustuu työn tuloksena toteutettavan web-pohjaisen valvomoratkaisun vertaamiseen muihin markkinoilla tarjolla oleviin ratkaisuihin. Vertailukriteereinä ovat ratkaisujen ominaisuudet ja käytettävyys. Teollisen internetin alustana toteutuksessa käytetään teollisen tiedonsiirron ja tiedonmallinnuksen standardia OPC UA:ta.

Työn tuloksena toteutetaan OPC UA standardiin perustuvan web-pohjaisen valvomo-ohjelman prototyyppi. Prototyyppi käsittää kehikon teollisen tiedon lukemiselle ja ohjauskomentojen lähettämiselle OPC UA osoiteavaruuteen tuotujen tietomallien kautta. Nykyaikaiset webtekniikat mahdollistavat jaksottaisten tapahtumien sekä uudelleenkäytettävien valvomokomponenttien toteutuksen. Yhtenäinen palvelurajapinta perustuu sovelluksen tilan siirtämiseen palvelupyyntöjen yhteydessä. Lisäksi työn keskeisenä teemana on Java- ja JavaScript ohjelmointikielillä toteutettujen kirjastojen yhteensovittaminen.

Toteutus osoittaa, että web tekniikoilla on mahdollista toteuttaa sekä kyselyihin että tapahtumiin perustuva tiedonsiirto ohjelmoitavien logiikoiden ja päätelaitteiden välille. Uudelleenkäytettävien web komponenttien standardointi tekee teollisten valvomojärjestelmien toteuttamisesta entistä houkuttelevampaa mahdollistamalla toiminnallisuuden ja tyylimäärittelyiden pakkaamisen itsenäisiksi kokonaisuuksiksi. Keskeisenä ongelmana jatkokehityksen kannalta todetaan tilallisten OPC UA palveluiden yhteensopimattomuus tilattomiin web rajapintakutsuihin.

Asiasanat:	OPC UA, Teollinen internet, Java, JavaScript, REST
Kieli:	Englanti

Acknowledgements

I thank supervisor Professor Jukka Nurminen for his insightful advice on how a graduate thesis should be composed. Much of my gratitude goes to the people at Prosys PMS Ltd for providing a comfortable work environment, with special thanks to advisor Jouni Aro, whose extensive knowledge of OPC UA was at my disposal during the writing of this thesis. I would also like to thank my family for their continuous support, and especially my wife who encouraged me to write during the most difficult moments.

Espoo, April 13, 2015

Tatu Paronen

Abbreviations and Acronyms

API	Application Programming Interface
CORS	Cross-Origin Resource Sharing
DCS	Distributed Control System
DI	Dependency Injection
DOM	Document Object Model
ERP	Enterprise Resource Planning
HMI	Human Machine Interface
HTTP	Hypertext Transfer Protocol
ICS	Industrial Control System
IIoT	Industrial Internet of Things
JNI	Java Native Interface
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MES	Manufacturing Execution System
MVC	Model-view-controller
NPM	Node Package Manager
OPC UA	OPC Unified Architecture
PLC	Programmable Logic Controller
REPL	Read-Eval-Print Loop
REST	Representational State Transfer
RTU	Remote Terminal Unit
SCADA	Supervisory Control and Data Acquisition
SOAP	Simple Object Access Protocol
SPA	Single-Page application
SSE	Server-Sent Events
TSA	Thin Server Architecture
URI	Uniform Resource Identifier
VPN	Virtual Private Network
XHR	XMLHttpRequest

Contents

Chapter 1

Introduction

1.1 Background and motivation

Industrial corporations employ control networks for transferring data and control messages between different levels of the corporate hierarchy. These control networks are used for automating the industrial process. On the lowest level, a large number of various field devices including different kinds of sensors and actuators collect data on the industrial process. Different levels in the corporate automation hierarchy, all the way up to the management levels, use this data to improve the process and to make intelligent business decisions. In order to ensure successful process execution, control room operators need a means to monitor and control the industrial network. Supervisory control and data acquisition (SCADA) systems are often used to provide such a Human-Machine Interface (HMI) for the plant operators.

Traditionally SCADA systems have been implemented as native desktop applications. However, during the last decade software development in general has seen a shift towards mobile and web applications. This development has been partly motivated by the nearly ubiquitous adoption of mobile devices and the high speed Internet connections available today. Web applications in particular have shown many advantages over native applications, including straightforward cross platform support, ease of development, and faster development cycles [33]. Web applications in turn are increasingly hosted in computing clouds, which offer higher scalability, reliability and flexibility compared to hosting the applications on dedicated servers. All in all, the developments in cloud computing and areas such as sensor networking are paving the road for the highly anticipated Internet of Things (IoT), which among other things is expected to yield considerable savings to industrial plants by allowing a higher level of fine-tuning in the industrial processes.

Many automation system vendors are already releasing their own cloud-based SCADA solutions. Hosting the SCADA system in the cloud allows SCADA system manufacturers to sell their products as Software as a Service (SaaS). Industrial companies however might be reluctant to give access to their industrial data to a third-party service provider. As such it may be beneficial for the company to host the SCADA application in its own private cloud. Another advantage of web-based SCADA systems is their portability, as the users need only a standard web browser to access the service. On the other hand the service provider can ensure portability of the service implementation by using a platform independent programming language such as Java. In order to gain the performance benefits of multiple computing cores, web applications must be programmed to execute concurrently in multiple threads or processes. Multi-threaded code is however inherently more difficult to write and prone to hard-to-find bugs. As a simpler alternative, the asynchronous event driven Node.js programming model offers a straightforward way to write scalable web applications. Several frameworks already allow to mix server-side Java and JavaScript code, enabling a new hybrid class of web applications.

1.2 Objectives and scope

The objective of this thesis is to evaluate the applicability of web technologies to the development of industrial monitoring applications for the Industrial Internet. The focus is on finding a way to bridge the gap between browser-based operator interfaces and industrial devices. This topic partly grew out from the need of Prosys PMS Ltd to evaluate the use of its OPC UA Java SDK in the development of applications for the Industrial Internet. As such the OPC Unified Architecture (OPC UA) communication protocol was chosen as the implementation platform, with the Java programming language as the implementation language.

The practical part of this thesis has to do with the development of a generic OPC UA web client. The goal is not to develop a complete SCADA HMI system, but rather evaluate and demonstrate how such a system could be built given the platform and programming language constraints. Nevertheless, the generic client could be used as a starting point when developing complete supervisory control systems. The generic client implementation is evaluated and compared with other web-based SCADA systems. Finally, based on the experiences gained throughout the development effort, the applicability of web technologies to the Industrial Internet shall be assessed.

The objectives of the thesis can be summarized in the following four research questions:

1. What are the requirements for the generic OPC UA web client?
2. How can the generic OPC UA web client be implemented?
3. How does the generic OPC UA web client compare to other solutions?
4. Are standard web technologies compatible with the Industrial Internet?

The first question examines the requirements for the generic client. The requirements analysis serves as a basis for the implementation of the generic client.

The second question is related to the implementation of the generic client. It is concerned with the specific technology choices, architecture, interfaces between different components, and other technical decisions related to the implementation.

The third question has to do with the evaluation of the finished implementation in relation to the existing solutions on the market. The solutions are compared based on their features and usability, with the focus on SCADA solutions that utilize standard web technologies.

The last question assesses the overall applicability of web technologies to the development of applications for the Industrial Internet, by reflecting on the experiences gained during the development of the generic client.

1.3 Research methods

This thesis has two parts: the theoretical part and the technical part. The theoretical part is conducted mainly as a literature study. Industrial networks and industrial user interfaces have been discussed in detail by for example Galloway et al. [27], Hollifield et al. [40], and Heimbürger el al [32]. The book OPC Unified Architecture by Mahnke el al. [45] and the OPC UA specifications serve as the main literary sources regarding OPC UA. In addition to the official language specifications, there is a large number of both printed and online resources for Java and JavaScript, including books by Horstmann [41] and Crockford [12], respectively. The history of rich Internet applications has been covered in detail by Casteleyn [11]. The development of purely web-based monitoring system using standard web technologies has been studied by Teliö [62], and OPC UA stack and framework development has been covered by Hennig et al. [34] and Freund [26], respectively.

The technical part involves creating a generic OPC UA web client application prototype. The requirements for the application are determined based on the application profiles of OPC UA. In addition, existing OPC UA client applications and web-based SCADA solutions serve as a reference point when determining the requirements for the application.

1.4 Structure of the work

The structure for the rest of this thesis is as follows. First, Chapter 2 gives a brief introduction to industrial networks, industrial operator interfaces and the Industrial Internet. Chapter 3 then presents the OPC UA standard for industrial data transfer and information modeling, focusing on OPC UA services (UA Part 4 [51]) and information modeling concepts (UA Part 3 [50]), which form the theoretical foundation for OPC UA application development [45]. Chapters 4 and 5 end the background part of the thesis by examining relevant backend and frontend technologies, respectively. Chapter 6 surveys the existing solutions providing OPC UA connectivity from the web browser, and forms the basis for the requirements analysis performed in Chapter 7. The finished web client implementation developed in accordance to the specified requirements is then discussed in Chapter 8. This chapter outlines the development process and the design decisions that were taken. Chapter 9 evaluates the success of the implementation and compares it to the existing solutions presented in Chapter 6. Alternative approaches and future work are also discussed. Finally, Chapter 10 concludes the thesis.

Chapter 2

Industrial networks

When developing software, it is important to understand the context in which the software solution is to be deployed. This chapter presents a brief introduction to the control networks that can be found in industrial facilities such as chemical processing plants, and manufacturing plants. First, Section 2.1 describes how the industrial processes can be monitored and controlled by the use of an automation system. Section 2.2 expands the discussion to the user interfaces that the plant operators use to interact with the automation system. Finally, Section 2.3 concludes the chapter by discussing the emerging Industrial Internet and how it will affect the development of automation systems.

2.1 Industrial control systems

An industrial plant includes all the machines, systems, structures and people that are needed to perform an industrial manufacturing process [32]. The process comprises all the actions that are taken to reach the business goal of manufacturing a product. The ISA-95 standard [16] specifies integration between the organization's industrial process and business management. It defines a model with three organizational levels: enterprise resource planning (ERP), manufacturing execution system (MES), and the automation system.

Automation systems, referred to as industrial control systems (ICS) for the rest of this thesis, can be divided into two categories: distributed control systems (DCS) and supervisory control and data acquisition (SCADA) systems. Traditionally DCS have been used in local and continuous closed loop processes, such as chemical processing, which have high requirements for reliability, whereas SCADA systems have been used in geographically scattered processes where the monitoring and control activities take place in multiple remote field sites [27].

Industrial plants use control networks to pass information from the field devices on the factory floor level all the way up to the business and management levels of the corporation. The application of industrial networks spans many industries including but not limited to discrete manufacturing and building automation [27]. In comparison to conventional information networks, industrial control networks are often characterized by their deeply hierarchical architecture and strict requirements for fault tolerance, determinism and real-time data transfer [27].

Industrial controllers, referred to as remote terminal units (RTU), control processes at field sites by implementing a control loop. The control loop attempts to maintain a process variable within its assigned limits. The control loop works by connecting inputs and outputs of a controller to sensors and actuators of process instruments. The controller periodically reads the current process information from the sensors, and based on the perceived state it sends commands to the actuators in order to maintain a desired state of the process. The interaction between the actuators and the physical process in turn creates disturbances which are captured by the sensors.

2.2 Human-machine interface

The human machine-interface (HMI) is a combination of displays, input devices and software, which provides the plant operator with services to monitor and control the industrial process. A good HMI enhances the operator's situational awareness, and allows the operator to respond to abnormal situations in a timely manner [40]. The HMI should be carefully designed to ensure safe, efficient and cost-effective operation of the industrial plant [40].

Part 5 of the ISO 11064 set of standards [17] specifies guidelines for the design of HMI systems. It defines a layered approach for structuring the information that is presented on HMI display. Everything that is displayed on a display screen constitutes a page. The page can be further divided into windows. One window can cover the whole page or multiple windows can be displayed at the same time. Finally, windows are composed of elements such as icons and labels, which are used as the basic building blocks of user interface components.

According to Heimbürger et al. [32], a page can show a general overview of the process by utilizing user interface components such as process diagrams, or it can be task specific, showing only the process data and controls that are needed to perform a specific task. Furthermore, the process data includes current values fetched through periodical updates, alarms and notifications, and historical data. Besides displaying process data, the page should contain controls which an operator can use to control the process.

The HMI implements navigation by inserting links between display pages. Display pages are typically organized in a hierarchical structure. The starting point for designing an HMI can be for example a process diagram, which identifies different components of the process, and the relations between them [32]. Processes can be further divided into sub-processes and the operator can be provided with the ability to drill down to specific parts of the process to gain more detailed information [32].

Alarms and notifications are an important part of any industrial automation system. Alarms are exceptional process states that the operator should handle immediately, while notifications are only informative in nature, and describe the normal operation of the process [32]. Alarms demand the immediate attention of the operator, and as such should be paid attention to when designing an HMI. HMIs use sounds and color to inform the operator, and should contain information about the reason of the alarm, the originator, and the priority associated with the alarm. The HMI should allow the operator to easily access this information and properly handle it. The hierarchical organization of display pages allows events to be propagated all the way from the individual components to the process view.

Besides current measurement values, HMIs allow the operator to view value history of variables in the form of trends. The ability to read history data is useful when diagnosing problems with the process. Access to historical data also enables the creation of reports, either periodically or on-demand. Many HMIs allow operators to simply drag-and-drop variables to a trend and see how different variable values relate to each other [32].

In addition to these, automation systems can have a variety of other features that make managing the process easier [32]. For example, the system may have communication tools which plant operators can use to communicate with colleagues who are potentially located in a different countries. The HMI may also provide contextual information which helps the operator to make decisions for example in case of an emergency. Supporting tools may also be integrated to help new operators to get familiar with the automation system. Most automation solutions also come with tools for designing and configuring the HMI for the given industrial process. All in all, the introduction of new technology creates new possibilities for optimizing the industrial process. One such development is the Industrial Internet, which is predicted to revolutionize the way how industrial organizations do business.

2.3 Industrial Internet

Today's industrial plants, such as manufacturing and waste processing plants, are increasingly connected to the global Internet. This development is driven by the large number of field devices found in today's industrial facilities, and a need to further lower the operating costs and improve the efficiency by remotely controlling and monitoring those devices, as well as by automating their functions. On the other hand, the developments on the Industrial Internet of Things (IIoT) are promising new, flexible ways to access and utilize the large amount of data that industrial facilities are already collecting from their field devices. On the supervisory control level, the ubiquity of mobile devices and their increasing network bandwidths are changing the way in which supervisory control systems are composed. Control room operators and field operators alike can now use low cost smart phones and tablets to monitor and control process instruments.

A report by the Research Institute of the Finnish Economy (ETLA) [43] discusses the impacts of Industrial Internet on the Finnish economy. It highlights several enabling factors for the Industrial Internet: availability of cheap smart products, ubiquity of today's Internet, processes becoming more global, and the combination of cloud technologies with big data and data analytics. It also suggests that today's young generation is better at using IT, and may more easily adapt to technological change.

According to the predictions in the report by ETLA, industrial plants will be increasingly automated. As reasons for this increase in automation the report lists higher safety, efficiency, lower cost, and more efficient process management. As a consequence, the responsibility of plant operators lies increasingly in the handling of the big picture, as the automation system becomes increasingly more capable of independent decision making and operation.

According to General Electric [19], the Industrial Internet has the potential to yield significant increases in industrial productivity, efficiency and performance. General Electric defines Industrial Internet as consisting of three main factors: intelligent machines, data analytics and people. Firstly, physical machines will become more intelligent and increasingly interconnected, have a higher number of sensors, and are controller by more sophisticated control software. Secondly, the data generated by the machines will be analyzed by analytics software based on deep domain knowledge and predictive algorithms, revealing new information about the system. Thirdly, the Industrial Internet will connect people more closely with each other, the physical system and the data it produces, allowing for greater productivity.

Industry 4.0, a project by the German government, emphasizes a flexible and customizable production of future's industrial plants [43]. The goal is to optimize production by making production systems more intelligent, and capable of communicating with one another. This is achieved by applying data analytics on the large amount of data that is already being generated by industrial instruments. Intelligent production systems allow for more efficient material utilization, higher energy savings, and a bigger flexibility.

Despite the increasing adoption of open Internet standards in the factory setting, the reality is that on the factory floor level there typically exists a large number of instruments from different vendors, each speaking their own vendor specific language. However, standardization efforts exist, one example being the OPC UA specification, discussed in the next chapter.

Chapter 3

OPC Unified Architecture

This chapter presents the OPC UA standard which defines a platform independent framework for industrial data transfer and modeling. Section 3.1 begins by giving an overview of the specification. Section 3.2 continues by focusing on the data transfer capability, while Section 3.3 discusses the data modeling aspects. Next, Section 3.4 describes OPC UA services and how they are mapped to the other parts of the specification. Section 3.5 discusses the application profiles defined in OPC UA, and how they promote interoperability between different automation vendors. Section 3.6 lists the available tools and frameworks that can be useful when developing OPC UA applications.

3.1 Overview

OPC Unified Architecture (OPC UA) is a set of specifications defined and maintained by the OPC Foundation which serve as a standard for industrial data transfer and data modeling [45]. It is meant to replace the preceding OPC Data Access (OPC DA), OPC Historical Data Access (OPC HDA), and OPC Alarm & Events (OPC A&E) standards, which provide a standardized interface for information transfer between SCADA systems and automation devices. OPC DA specifies mechanisms for reading, writing and monitoring data residing in industrial devices. OPC HDA specifies how OPC clients can access the historical data and events of an OPC server. OPC A&E defines the management of industrial alarms and notifications. However, OPC has a number of limitations that have motivated the definition of the new OPC UA standard. These limitations include its reliance on Microsoft's COM and DCOM protocols for communication, and the lack of a standard security model. The problems associated with the standard OPC specifications are further highlighted by Mahnke et al.

Frejborg et al. list several advantages of using OPC UA over the old OPC protocol. The listed advantages include platform independence and interoperability, flexible communication and security schemes, use of open standards, and a well-defined and extensible architecture [25]. The paper goes on to present different real-world use cases of OPC UA, such as using OPC UA security instead of setting up a VPN, easily extensible information models through vendor specific extensions, the possibility to develop native applications both for mobile devices and web browsers [26], and the use of a unified communication protocol and common information models on all levels of the organizational hierarchy. For example, components like individual PLCs could be exposed to standard OPC UA clients as individual OPC UA servers, enabling straightforward information flow all the way from the control network level to the production and business management levels of the organization.

3.2 Data transfer

The various OPC UA communication stacks implement the client-server communication protocol defined in the OPC UA Service Mappings specification [52]. For data transfer, the specification defines the UA TCP protocol, which is a transport protocol built on top of TCP and optimized for industrial data transfer. In addition, OPC UA supports HTTPS and SOAP over HTTP, which may be useful in situations where the use of UA TCP is restricted by the organization's firewall policies. The transmitted data is serialized either by using OPC UA's own binary encoding format or by encoding it in XML. The OPC UA transport profiles, along with the security protocols and encodings they use, are depicted in Figure 3.1. A client and a server communicate with each other by exchanging messages. Server to server communication is also possible if one of the servers additionally implements the client functionality.

In order to securely send messages using the chosen transport mechanism, OPC UA defines a security model which is based on communication over a secure channel. The secure channel secures the communication between the client and the server by the use of public key cryptography. The messages can be encrypted for privacy, and signed to ensure that the sender information and content have not been altered on the way. A client can have multiple connections to different servers and servers are expected to handle multiple client connections simultaneously. Therefore, all communication between a client and a server is associated with a session, which encapsulates the

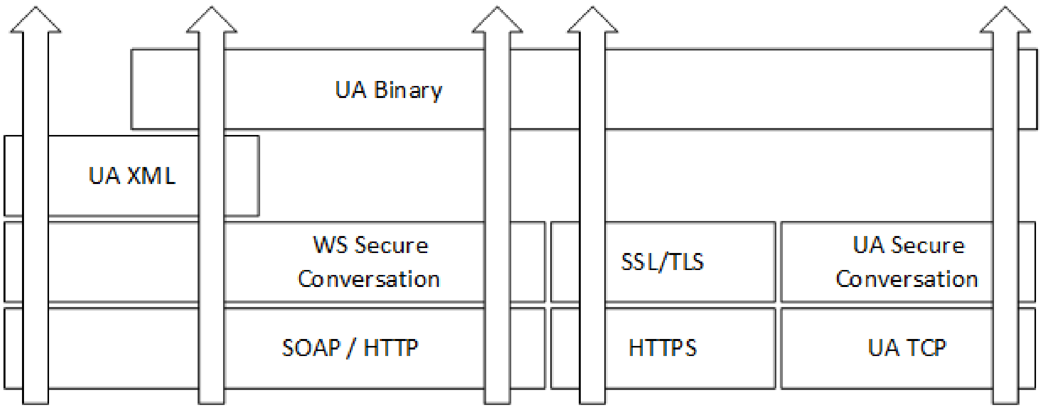


Figure 3.1: The OPC UA transport profiles. Adapted from Mahnke et al [45].

communication context within a secure channel.

3.3 Information modeling

An OPC UA server exposes data through the address space. The address space is a connected network of nodes. The nodes are used to represent data in an organized way that is easily accessible to the client. For example, nodes can represent real objects such as thermometers or valves, or they can be used to organize the address space or to store type definitions. The connections between nodes are called references. A reference represents a directed relationship between two nodes. For example, a *Device* node can have a reference *HasTypeDefinition* to a node called *DeviceType*. Each node belongs to a specific *NodeClass* and has a set of standard defined *attributes*. These attributes are determined by the NodeClass, and they represent the node's data values. The most important attribute a node has, aside from the NodeClass itself, is its *NodeId* which uniquely identifies the node, and is used in service calls by the client. Other standard attributes include *BrowseName* and *DisplayName*, and the optional *Description* attribute. The *HasTypeDefinition* attribute refers to the NodeId of a type definition in the address space.

The address space of an OPC UA server effectively forms a mesh network. While the specification allows a server to organize its address space by using hierarchical references, a client can choose to browse the address space in a number of ways by following the multitude of named references that nodes are allowed to have between each other.

Service Set	**Description**
Discovery	Finding server endpoints and their security parameters
Secure Channel	Creating and maintaining secure channels
Session	Authentication and session management
Node Management	Modifying the structure of the address space
View	Browsing and creating views in the address space
Query	Finding information in the address space
Attribute	Reading and writing attributes
Method	Allows clients to call server-defined methods
Subscription	Managing subscriptions used to hold monitored items
Monitored Item	Monitoring objects for events and data changes

Table 3.1: The Service Sets defined in UA Part 4 [51].

The address space model is the meta model of OPC UA, and it serves as a basis for all other information models. OPC UA already defines some information models such as the base information model. These information models and the address space model can be extended to define new domain specific information models. The advantage of the uniform base model is that an OPC UA client can rely on the OPC UA semantics and type information provided by the server to automatically process the data it gets from the server.

3.4 Services

OPC UA services define the interfaces that OPC UA clients and servers use to communicate with each other. The services are organized into service sets based on the type of facilities that they provide. The client interacts with the server by invoking services. It typically does this by sending a request message to the server and receiving the consequent response message from the server. Messages are passed asynchronously so service requests are by definition non-blocking. Alternatively a client can subscribe to events originating from nodes in the server's address space. Events can be used to provide real-time data updates to the client, and they are also ideal for implementing process alarms and notifications.

OPC UA Part 4 [51] specifies abstract service interfaces which are independent of the transport protocols and programming languages used to implement them. This allows the underlying technologies to evolve and new technologies to be adopted while maintaining the same service interfaces. It also means that all applications use the same interfaces regardless of the technologies that have been used to implement them. OPC UA Part 6 specifies how technologies map to the service interfaces. The actual implementation of the services is left as the responsibility of OPC UA stacks. One of the design principles for services has been to create a small number of generic services which can handle various types of parameters.

3.5 Application profiles

One of the design goals of OPC UA is that it should be usable in hardware of different storage capacity and computing power, ranging from small embedded devices to applications running in the cloud. Profiles, defined in OPC UA Part 7 [53], enable application developers to choose which parts of the specification they want to support. An OPC UA profile is a collection of features that a conformant OPC UA application has to support. A profile consists of facets, which in turn can be broken down to conformance units, representing specific features [45]. The features that a server should support are defined in terms of both facets and complete profiles, while client applications need to only choose the individual facets they want to support. Facets are categorized to client facets and server facets based on the application type. Additionally, clients and servers have to support transport profiles and security profiles, which define the supported protocols.

Conformance units form the test suite which is used to verify that an OPC UA application conforms to the OPC UA specification [45]. OPC Foundation provides certification to OPC UA applications, and passing the tests forms the basis for receiving the certificate. Because of profiles system integrator can make sure that the chosen OPC UA products conform to the specification and are interoperable. The logical functional units formed by profiles can be used as a basis when designing the functional requirements of an OPC UA application.

3.6 Tools and frameworks

An OPC UA communication stack acts as a layer between the OPC UA application software and the networking hardware. The OPC Foundation provides several stack implementations, which can be used with different programming languages.

Developing OPC UA applications against a stack takes considerable effort, so various companies are now providing software development kits (SDK). The SDKs facilitate and speed up development of OPC UA applications by providing a set of high level interfaces, modules and tools. By using an SDK an application developer does not need a deep understanding of the OPC UA specification. As such a number of SDKs have emerged, that facilitate the development of OPC UA applications by abstracting away the complexities of the OPC UA standard.

Prosys has developed an OPC UA Source Development Kit (SDK) [58] for Java, which facilitates the development of OPC UA client and server applications. The SDK sits on top of OPC Foundation's OPC UA Java stack, and provides a higher level of abstraction for accessing the various OPC UA services. The Prosys OPC UA Client [39] and Prosys OPC UA Simulation Server [8] are based on this SDK.

The increasing adoption and a number of successful applications [25] demonstrate the suitability of OPC UA for industrial data communications and information modeling.

Chapter 4

Backend development

Enterprise web applications, such as those found in industrial facilities, are often large and complex systems faced with challenges related to scalability, reliability and security. This chapter discussed backend development in two popular [63] programming languages, Java and JavaScript. In the context of this thesis, the term backend refers to the web server, while the term frontend refers to the web browser acting as a user interface.

First, Section 4.1 presents an up-to-date survey on web server development using the Java programming language. Section 4.2 then gives an overview of the JavaScript programming language and presents an alternative way of building web applications by using a server-side JavaScript framework, a technique which has gained much traction recently. After reviewing some popular server-side JavaScript frameworks, the chapter continues to present the state of the art of technologies supporting intermixed Java and JavaScript code. Finally, Section 4.3 ties everything together by taking a look into web services technologies for exposing the server's functionality in the form of reusable services, readily consumable by web browser based client applications.

4.1 Java Enterprise Edition

4.1.1 Overview

The name Java covers both the programming language and the set of platforms supporting it. When talking about plain Java, the Java Standard Edition (Java SE) is typically meant. Java SE consists of the Java language, runtime environment and APIs, along with tools that aid in development and deployment of Java applications. Java SE 8 was released in early 2014, and

```java
package hello;

import javax.faces.bean.ManagedBean;

@ManagedBean
public class Hello {

    final String world = "Hello World!";

    public String getworld() {
        return world;
    }
}
```

Listing 4.1: A simple JavaBeans class.

includes several new features such as support for lambda expressions and a new DateTime API [41].

Java Enterprise Edition (Java EE) extends Java SE with a collection of Java-based technologies for addressing typical needs encountered in enterprise Java application development, such as object persistence and messaging [56]. Java EE 7 is the current latest version of the specification, but the specification for Java EE 8 is already being worked on. Java EE supports the development of multi-tiered software systems spanning from client applications all the way to the organization's information systems and offers solutions to the mid-level tiers.

Java EE also specifies a web profile, which is targeted at web application developers and narrows the specification down to a smaller set of available APIs, helping to reduce the runtime footprint of web application deployments [54]. The web profile specifies a list of required components that the implementations must provide. The most important components include the specifications for Servlets, JavaServer Pages and JavaServer Faces, along with the components that they depend on such as the Expression Language.

In order to address issues related to the re-usability of software components written in Java, the Java EE specification introduced the JavaBeans technology. JavaBeans is a standardized way to design reusable Java components called beans. Any Java object that fills certain three criteria is considered to be a bean. First, the object should have a parameterless constructor. Second, the object should correctly implement the *Serializable* interface. Lastly, the object's fields should be private and accessible through getters and setters only. A bean is composed of three features: properties, methods and events. Listing 4.1 presents an example of a bean.

Java has provided a way to run Java applications in the web browser since its initial release. This has been achieved in the form of Java Applets. A Java Applet is a Java application that can be embedded to a website by using the <applet>[1] HTML tag. The browser downloads the Java application bytecode and executes it locally in a sandbox, which implements certain restrictions such as preventing the applet from accessing the local file system. Alternatively, since Java 5.0 applications can be installed and run directly from a web site by using the Java web start technology. Recently, a third option has become available as JavaFX applications can be embedded on web sites, enabling two-way communication between the application and the web browser's document object model (DOM).

Integral part of the Java EE specification is the Servlet API, which resembles an applet but with the exception that it is executed on the server side [55]. The servlet acts as a bridge between the web server and applications running in the Java programming language. The paths which the servlet should handle are configured in a *web.xml* file. The servlet container is a component which receives requests from the web server, encapsulates them in request and response objects, and forwards them to specific servlet components based on the servlet's configuration.

The Servlet does not concern itself with how the response is generated. For this reason Java EE employs JavaServer Pages (JSP) which can be thought of as its template engine. JSP pages work by implementing the servlet interface. They are compiled into servlets either in advance or automatically at deployment time. JSP pages can generate HTML responses independently or they can be used as the view component in an MVC framework. In the latter case a separate servlet component is used as the controller while JavaBeans objects are used to represent the data model. In addition to standard HTML elements, JSP views can use special JSP tags. These reusable tags perform some specific task and are organized into tag libraries. In addition to the JSP Standard Tag Library (JSTL), several custom tag libraries exist for JSP, such as the jQuery UI taglib.

Web applications developed in Java EE must conform to a specific directory structure. The application directory is required to have a *WEB-INF/* directory which includes a *web.xml* configuration file, along with *lib/* and *classes/* directories, for external libraries and application specific class files, respectively [14]. The application can be deployed as a directory, or it can be packaged as a Web Application Archive (WAR) file.

[1]Tags <*embed*> and <*object*> can also be used but with varying browser support.

4.1.2 Deployment platforms

Java EE requires the implementation of the Servlet API. Frameworks implementing the Servlet API are called Servlet Containers, or alternatively Servlet Engines. The following list presents two popular servlet containers.

Tomcat is an open source, servlet compliant web server which is widely used for serving Java web applications. It implements the Servlet and JSP specifications, and newer releases have added support for the WebSocket specification. Tomcat by itself does not support features such as JSF, but support for JSF and other Java EE features can be added by bundling it with TomcatEE.

Jetty is a light-weight, extensible servlet engine and web server which can be embedded into software to add web server functionality. Some software using Jetty include Google App Engine, and the Spark web application framework. Like Tomcat, Jetty includes support for WebSocket.

The Servlet API is however only a subset of the whole Java EE specification, and more larger deployment platforms called application servers implement additional Java EE features. The following list presents two widely used open source application servers.

TomcatEE is an open source extension for the Tomcat servlet container. It adds support for the complete Java EE web profile by integrating various other Java EE components developed by Apache.

Glassfish by Oracle is an open source application server supporting Java EE 7. It serves as the reference implementation for Java EE, and as such is among the first to implement new Java EE specifications. Commercial support for Glassfish was discontinued in November 2013 in favor of the WebLogic application server.

Servlet containers and full featured application servers provide a standardized way to build Java web applications. Application servers certified against Oracle's test suite provide large corporations with the confidence that the deployment platform conforms to Java EE standards. On the other hand, development in Java EE relies heavily in XML configuration, and the relationships of different components in the Java EE platform can quickly become confusing and hard to debug. Java-based web development frameworks ease web development by allowing developers to concentrate on the application specific code, and require less configuration. Deployment also becomes easier and applications can be more easily deployed on different platforms.

```
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Facelets Hello World</title>
    </h:head>
    <h:body>
        #{hello.world}
    </h:body>
</html>
```

Listing 4.2: A simple Facelets view.

4.1.3 Development frameworks

There exists a wide variety of Java-based web development frameworks. This chapter focuses on frameworks that utilize the model-view-controller (MVC) architectural pattern, as it is the most common pattern used by web application frameworks. MVC frameworks in general can be divided into roughly two categories based on how they handle incoming HTTP requests: component based frameworks and action based frameworks [56].

Component based frameworks abstract away the underlying request/response cycle [9]. Instead, applications are developed by combining components which are inserted in the view declaration in the form of custom tags. Component based frameworks also hide the underlying HTML, CSS and JavaScript.

JavaServer Faces (JSF) [56] is a component-based MVC framework that is also based on the Servlet definition and originally used JSP as its view rendering system. The use of JSP is nowadays deprecated in favor of the new Facelets view declaration language. JSF defines a new FacesServlet component which is used as the controller. Being a servlet, it receives requests from the servlet container and based on the request generates a response to the requesting client. By abstracting the specific web technologies such as HTML and CSS from the development process, JSF enables developers not familiar with web technologies to write web applications.

JSF defines a collection of components which map to tags specified in a tag library. The components represent user interface elements which are connected to server-side objects that have methods for rendering the component and encoding and decoding its data for transmission. JSF also supports the saving of user interface state across the otherwise session-less HTTP protocol. Listing 4.2 presents an example of a Facelet definition. The example demonstrates how basic Java beans can be bound to the Facelet, allowing to access their properties by using the Expression Language (EL) syntax.

Being part of the Java EE specification, JSF is a popular choice among component based web development frameworks. There are also other component based frameworks such as Tapestry, which is very similar to JSF. However, there exists a large number of Java-based web development frameworks that do not endorse the component-based approach. Such frameworks can be classified as action based frameworks.

Action based frameworks expose the underlying HTTP request and response as objects. The request object can be read for headers, data, and parameters, while the response object can be assigned to set headers and response data. Action based frameworks are useful when access to the request/response cycle is required. They are also more easier for frontend developers and designers to work with, as they do not hide the HTML, CSS and JavaScript files [9]. Following is a list of five popular action based frameworks.

Play is a web application development framework written in Java, and programmed by using the Java programming language. Play Framework is built on top of the Akka framework, which uses the Actor model to help in development of concurrent Java applications.

GWT by Google is a RIA framework that compiles Java source code to JavaScript. Both Eclipse and IntelliJ IDEA provide support for GWT. The main difference to the other alternatives presented here is that the application code is run mainly on the client-side.

Vaadin is a commercially supported, open source web application development framework supporting the creation of rich Internet applications (RIA). Vaadin uses the GWT framework for client-side rendering. Like GWT, Vaadin features a large collection of reusable user interface components. Unlike GWT, however, it emphasizes server-side processing.

Spark is a modern request/response framework supporting Java 8 and based on Jetty. Compared to the other alternatives presented here, Spark is more light-weight and designed after the popular Sinatra framework. Spark may be more suitable alternative for smaller projects that do not need all the enterprise features of Java EE.

Spring is a popular Java EE compatible web development framework which competes with pure Java EE stack. Java EE standardizes industry proven technologies and as a result new features tend to be introduced to the Spring framework more earlier.

4.1.4 JavaScript support

The Java Development Kit 8 (JDK 8) was released to developers on March, 2014. In addition to new features such as Lambda Expressions, it includes the Nashorn JavaScript engine which replaces the old Rhino JavaScript engine. Compared to Rhino, Nashorn achieves significant performance benefits by utilizing a new JVM instruction for supporting dynamic languages [41]. At the time of writing this, Nashorn implements the ECMAScript Edition 5.1 language specification, but future releases are planned to support ECMAScript 6. Not only does Nashorn enable the execution of JavaScript within Java, but it also makes it possible to instantiate and use Java classes within JavaScript code. This in turn allows JavaScript developers to take advantage of Java's features and libraries written in Java.

The implications of using Java and JavaScript together include code reuse, lower entry barrier, easy prototyping, faster development cycles and the possibility to add scripting support to Java applications. The benefit gained from reusing existing Java and JavaScript libraries should not be overlooked. For example, JavaScript code can use the *BigInteger* Java class to handle large integers and avoid overflows, and Java code can utilize for example JavaScript based testing frameworks such as Mocha for increased flexibility and to enable testing of the user interface. The lower entry barrier means that developers knowing one of the two languages can leverage their existing skills when developing against libraries written in the other language. Prototyping and iterative development become easier and faster, as less time is spent on compiling the source code. Scripting support can make applications more flexible to use, as users can write their own scripts to extend the base application and automate repetitive tasks.

4.2 JavaScript

4.2.1 Overview

JavaScript is a prototype-based, dynamically typed programming language, which has originally been run in browsers as an interpreted language, but has in recent years gained popularity as a server-side programming language. Nowadays JavaScript engines typically perform just-in-time (JIT) compilation in order to speed up code execution. For example the V8 engine, developed by Google to power its Chrome web browser, compiles JavaScript straight to native machine code. Mozilla's Firefox on the other hand starts by interpreting the code, but gradually performs JIT compilation on those parts of the code that are encountered often.

Despite its name, JavaScript is not directly related to Java. Java is a statically typed language which is compiled to an intermediate format called bytecode, understood by the JVM, whereas JavaScript is a dynamically typed language which is usually compiled during the execution. Furthermore, JavaScript does not have classes, but instead relies on prototypes to handle inheritance.

JavaScript is an implementation of the ECMAScript standard. At the time of writing this thesis the latest ECMAScript version is 5.1, released in 2011, but version 6 is expected to be released in early 2015.

There are two kinds of data types in JavaScript: primitives and objects. The five primitive data types are *number*, *string*, *boolean*, *null* and *undefined*. All numbers in JavaScript are represented as 64 bit double precision decimal numbers. An object can can be thought of as a map. It is a collection of properties that can be referenced either by using dot notation or with brackets. Aside from the primitive types, everything in JavaScript is represented as objects. For example, the array data type is an object with array semantics, including a length property and methods for performing array manipulations. In JavaScript, even functions are objects.

Functions are a fundamental part of JavaScript, and because they are objects, they can be assigned and passed as a parameter to other functions. This makes functional programming in JavaScript possible [12]. Functions are important for understanding scope in JavaScript. JavaScript has function scope as opposed to block scope. By default variables belong to the global scope, and creating a function creates a new scope. Furthermore, nested functions create closures, whereby the inner function closes on the outer function's variables. Closures are useful because the enclosed information becomes private to the enclosing function, and stays available even after the outer function has returned. A peculiar feature of JavaScript's function scope is hoisting: variables declared within a function are automatically hoisted to the top of the function definition, no matter where they are declared.

In addition to hoisting, JavaScript has several other features that can confuse beginners [12]. First, variables declared without the *var* keyword become global, so developers should make sure to always prepend the *var* keyword. Second, JavaScript has automatic semicolon insertion, which means that in certain cases the use of semicolon is optional, but recommended. Failure to do so may result in weird bugs.

JavaScript is a single-threaded programming environment, which means that the language does not expose an API for developing multi-threaded applications [46]. Instead, JavaScript relies on messaging to handle concurrency. Operations that require concurrency are handled by generating events.

In the web browser environment a developer can add interactivity to a static HTML page by assigning event handler functions to different DOM elements. For example, a developer can bind a click handler to a *div* element. Supposing that a user clicks the div, a message is dispatched and added to the event loop. The JavaScript runtime goes through the event loop, and on each event, calls the associated event handlers. This continues until all messages in the event loop have been processed. As a consequence, callback functions should return quickly. Otherwise they risk blocking the event loop. Blocking the event loop is undesirable especially in the browser environment, because it means that the browser will not be able to handle other events, such as those generated by a window resize.

JavaScript's inheritance model is prototype-based. All objects inherit directly or indirectly from *Object.prototype*. On access to an object's property, if the property is not found in the object itself, it is looked up in its prototype. If the property is still not found, the lookup continues through the prototype chain until is is found or until the whole chain has been checked. Prototypes are assigned during instantiation of an object. To create a new instance, a constructor function is called with the *new* keyword. The *new* keyword does two things. First, it creates an empty object instance and sets the *this* context variable to point to the newly created instance. Second, it assigns the *prototype* property of the constructor function to the *__proto__* property of the newly created object. The *__proto__* property is important since it is referenced when searching for properties in the object's prototype. The constructor function can use *this* to set properties and functions on the object. Alternatively, properties can be assigned directly to the constructor's prototype. This allows new instances to inherit properties. Moreover, this way the properties are assigned only once, no matter how many objects are instantiated [15]. The *prototype* can be manually set to create complex inheritance hierarchies between objects.

The *this* keyword is often a source of confusion. This is because the value of *this* varies based on where a function is called. If a function declared in the global scope is called, *this* is set to *undefined*. If an object's method is called, *this* is set to the object on which the method was called. Finally, functions such as *apply* and *bind* can be used to explicitly set the value of *this*.

In his book, Crockford [12] summarizes some of the strengths and weaknesses of JavaScript. As the useful features he lists things like first class function objects, dynamic objects, object based inheritance, and object and array literals. As the bad parts he lists the dependence on global variables, automatic semicolon insertion, two different kind of equality operators and problems related to decimal precision.

4.2.2 Server-side JavaScript

Web applications usually perform I/O operations synchronously and handle multiple concurrent users by spawning new worker threads to handle each HTTP request. However, business applications often need to handle a large amounts of data. This data can include for example customer records residing in a database. These I/O operations are slow and make up most of the time that the web server spends while serving each individual request. Thus serving a large number of concurrent users can severely impact the response time of the server. Furthermore, having to execute each request in a separate thread complicates application design and makes it easier to introduce hard-to-find bugs.

Originally JavaScript has been used almost exclusively as a scripting language in web browsers. JavaScript has allowed web developers to create dynamic web sites with enhanced user experiences and higher levels of user interaction. During recent years however, increasing number of JVM-based JavaScript engines and frameworks have emerged. High-performance JavaScript engines have made it possible to develop server software in JavaScript. Unfortunately JavaScript engines have been tied to the specific browser environments they have been running in. In order to take advantage of JavaScript on the server side, a runtime which does not depend on the browser environment is required. There are several ECMAScript compliant standalone JavaScript runtimes that can be used as embedded components. The following list presents four alternative open source JavaScript runtimes:

V8 is an open source JavaScript engine written in C++. It was developed by Google to power the Chrome web browser.

DynJS is an open source JavaScript engine written in Java. It adds JavaScript scripting capability to Java applications.

Rhino is an open source JavaScript engine written in Java. It comes bundled with Java SE 6 and Java SE 7.

Nashorn is a JavaScript engine written by Oracle. Since Java SE 8, it is the default JavaScript engine that comes bundled with Java.

Node.js is a server side JavaScript environment that uses V8 as its JavaScript engine, and a C library called LibUV as a platform independent abstraction layer for handling asynchronous I/O operations. The public API of Node.js is written entirely in JavaScript. Listing 4.3 shows a simple HTTP server written in Node.js and demonstrates some powerful features of the JavaScript

```javascript
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
```

Listing 4.3: A simple HTTP server using Node.js [47].

language. First, the *createServer* function takes a callback function as a parameter, taking advantage of the fact that JavaScript functions are first class objects. Second, the callback function creates a closure so it is able to access variables in its parent scope. Third, the HTTP module uses a chaining pattern where consequent methods operate on the same *http* object. This is possible because the methods of http module always return the same object instance that they operate on.

The Node.js programming model emphasizes asynchronous and non-blocking I/O operations. It achieves asynchronous and non-blocking I/O by utilizing an event loop [61]. The event loop runs in a single thread and synchronously processes a queue of events. For each event it calls the callback function associated with that event, if one exists. Node.js delegates I/O operations to LibUV, which takes care of polling for I/O and notifies the main event loop when data is available. Using the event loop ensures that only one callback is executed at a time, and that callbacks run to completion without interruption.

The single threaded programming model of Node means that by default a single Node.js application instance can utilize only one CPU core [61]. In order to scale the application up to using multiple cores the application needs to be forked into multiple processes. This in turn requires that incoming connections are evenly load balanced between the processes. Writing Node.js applications that run on multiple cores is facilitated by Node's cluster module, which allows the master process to spawn worker processes that all listen on the same server port. There is no shared state between the worker processes so all communication between them has to go through the master process.

Node.js has built-in support for file I/O, networking by using TCP, UDP, and HTTP protocols, data streams, child process management, and SSL and HTTPS based security [61]. In addition to the core modules, Node.js has a CommonJS based module system, which features a large repository of user contributed modules. The modules range from full stack application development frameworks to highly focused libraries. The module system of Node.js remedies the problems that JavaScript has with its dependence on global scope [12] by allowing developers to organize their code into reusable components that share only their public interface. The module system follows a convention of placing module dependencies in a directory called *node_modules* in the root directory of the module. Modules are searched by starting from the module's own *node_modules* directory and recursively ascending through the parent directories until the module is found. Node.js comes bundled with a package manager called *npm* which facilitates the dependency management.

4.2.3 Java-based frameworks

In the recent years, there has been a great amount of interest in mixing Java and JavaScript code. As a result, several JVM-based server-side JavaScript frameworks exist today. In addition to JVM-based approaches, there is exists even a Node.js module for allowing Node.js applications to import Java classes [20]. This section focuses on the JVM-based approaches, because they also support running Node.js applications from Java, allowing for a larger number of different use cases. Four different alternatives to running Node.js applications on the JVM are considered: Avatar, Nodyn and Trireme.

4.2.3.1 Avatar

Avatar.js is an open source project which seeks to implement the Node.js platform APIs on the JVM [3]. Instead of Google's V8 it uses the Nashorn JavaScript engine included in JDK 8, and uses the Java Native Interface (JNI) to provide bindings to LibUV. Furthermore, a separate project by Oracle called Project Avatar integrates Avatar.js with Java EE by combining it with several other Java projects such as Jersey for creating RESTful web services and Tyrus for creating WebSocket services [57]. Together they form a framework for easily creating web services that utilize REST, Web Sockets and Server-Sent Events. It additionally includes a client-side web application framework which allows the developer to bind Avatar services to client-side view models, and supports templates based on EL expressions. By including Avatar.js, Project Avatar allows the developer to use Node APIs, thus enabling the use of the wide array of available node modules. In addition to Node modules the Nashorn JavaScript engine makes it possible to use any Java libraries, such as the Java Date API [13].

One of the advantages of using Avatar is that it is developed by Oracle, and thus has the potential of making its way to being part of standard Java EE ecosystem in the future. Avatar offers an easy way for implementing web services endpoints which integrate with Java technologies such as the Java Messaging Services (JMS), and the possibility to easily spawn new Java

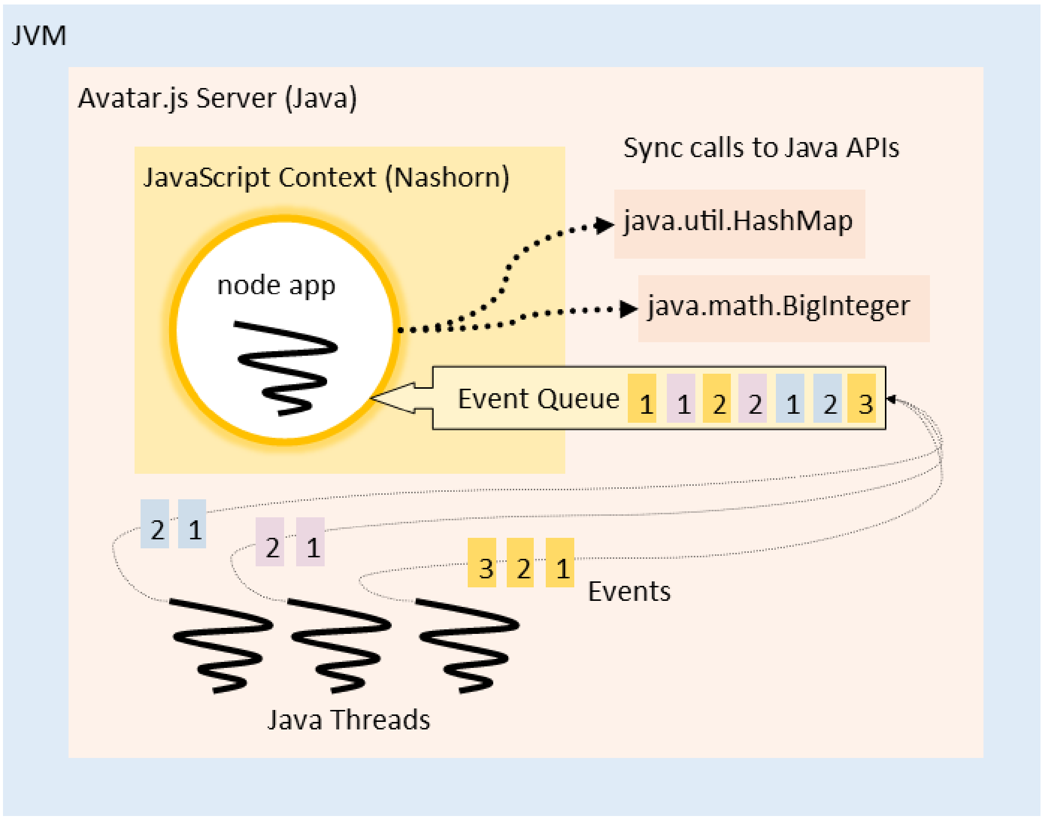


Figure 4.1: The architecture of Avatar.js [3].

worker threads for CPU intensive tasks. The full stack of Avatar views and services therefore presents a Java EE compatible alternative to technologies such as the JSF. Avatar allows to develop JVM-based Node.js applications without the need to write and compile any Java code. Instead, script files are passed as parameter to the included avatar-js.jar file. Executing avatar-js.jar without arguments initiates a Read-eval-print loop (REPL) which facilitates experimenting by reading and evaluating JavaScript line-by-line from the standard input.

The biggest disadvantage of Avatar is that in addition to the JVM, it also depends on a platform dependent avatar-js library and the Glassfish application server. Moreover, service oriented web applications can be developed in plain Java EE, so one may question the need to add yet another programming layer on top of standard Java EE just to accommodate an additional programming language. In addition, the hybrid programming model may be difficult to reason because of the different semantics of Java and JavaScript.

For example, the same source file may contain references to variables that are defined in either of the two programming languages, raising the need to handle variables in different ways based on the programming language context they have been defined in.

4.2.3.2 Nodyn

Nodyn [48] is a framework similar to Project Avatar. Like Project Avatar it enables running Node.js applications on top of the JVM. However, it does not include an implementation for a service layer. Instead of Nashorn, it uses DynJS as its JavaScript engine.

One of the main advantages of Nodyn is that it is based on the Vert.x platform. Vert.x [64] is an open-source, JVM-based software development framework which enables the development of web applications combining multiple different programming languages. The supported programming languages include for example Java, JavaScript, Python and Ruby. Vert.x uses a module based architecture which allows developers to package applications into modules. Since the APIs of Vert.x are asynchronous, concurrent programs are written as if they were single-threaded. Similar to Node.js, Vert.x depends on message passing for increased scalability. The project website states that Vert.x takes care of automatically utilizing available CPU cores.

In Vert.x, modules can be programmed in several different languages. Modules written in different languages communicate via a distributed event bus. Several different modules can use the same event bus to communicate with each other via messaging. The event bus is not limited to only the server, but a JavaScript client API is also available to hook to the same event bus. While Vert.x itself does not implement the Node API, a separate project by RedHat called Nodyn implements the Node API on top of Vert.x. By default, Vert.x uses the Rhino JavaScript engine, but an experimental module which adds support for Nashorn is also available.

Nodyn extends Vert.x in two ways. First, it implements the Node.js API. Second, it adds support for loading NPM modules by incorporating the npm-jvm module loader for the JVM. Nodyn leverages the message passing mechanism of Vert.x and unlike standard Node.js uses Vert.x instead of LibUV to implement its event loop. Nodyn comes with a package manager that can install both Vert.x and Node.js modules.

The polyglot nature of Vert.x has the potential of allowing Nodyn applications to be easily integrated with other Vert.x modules, possibly written in different programming languages. Vert.x enforces a modular architecture where modules communicate via message passing. This kind of architecture enables software composition and helps to remove hard dependencies between components, thus facilitating testing. Message passing itself allows applications to easily scale out to multiple processing units. Like Avatar, Nodyn also features a REPL for conducting small experiments.

The disadvantage of Nodyn is that it currently exists only in its source repository, as the initial release has not yet been published. As a consequence, it is still severely lacking in its documentation, with only a small number of examples residing in its public repository.

4.2.3.3 Trireme

Trireme is another implementation of the Node.js API on the JVM. It is set out to implement the platform dependent parts of Node.js in Java. Unlike Avatar, it uses the older Rhino JavaScript engine. Trireme is geared mainly towards executing Node.js scripts within Java applications. Trireme supports running several Node.js scripts concurrently, each in its own execution environment. Trireme creates a sandbox that can control whether the scripts have access to underlying OS resources.

Trireme is a welcome addition to the available alternatives for a JVM-based Node.js implementation. Its main advantage compared to the other alternatives is that it is designed to be easily embeddable to existing Java projects. Java versions from SE6 upwards are supported. This is in contrast to Avatar which requires Java 8 or later, and is tied to a Java EE compatible application server. The only external dependency besides the JVM and the Rhino JavaScript engine is the SLF4J logging framework. An added benefit of Trireme is the sandbox model which can restrict untrusted scripts from accessing specific OS resources.

As of now Trireme still uses the old Rhino JavaScript engine, which is considerably slower than Nashorn in executing JavaScript code. There are however plans to support alternative JavaScript engines in the future. Similar to Nodyn, writing Java code is required in order to run Node.js applications on the JVM.

4.3 Web services

4.3.1 Overview

Web services allow to decouple the frontend user interfaces from the service backend. However, to enable this reuse, a uniform and well documented service interface is required. Client-server applications usually employ a request-response oriented communication model. In this model, a client sends a request message to the server, along with request parameters. Upon receiving the request, the server parses the request and does the necessary processing to generate a response. Finally, the server sends the response to the requesting client. The request-response model is appropriate when the client needs data from the server on demand.

Another model the client and server can use is publish-subscribe. In this model the client can subscribe to specific events from the server. The client does this by registering a listener for certain events or topics. The server on the other hand can send messages to interested parties by publishing certain events, or messages with a specific topic. Based on the topic or event type, subscribers are able to filter only those messages they are interested in. Some publish-subscribe implementations allow the subscriber to send a reply back to the original publisher. These systems allow the publisher to specify a callback that should be called, with the reply message as parameter, when the reply message arrives. In comparison to the request-response model, publish-subscribe model is better suited to event based systems with high real-time requirements, because it allows the client to receive low overhead, server-initiated push notifications as they are generated.

Both request-response and publish-subscribe models are useful for implementing information exchange between HMI clients and OPC UA servers. Web applications support the request-response model natively by using the Hypertext Transfer Protocol (HTTP). It has also been possible to implement web applications that perform real-time data exchange, but up until now it has required web browsers to resort to expensive polling and long-polling techniques, whereby the browser continuously keeps requesting the same resource, with varying frequencies. In recent years however, HTML5 has standardized new real-time, event-based messaging systems for use in web applications. First, there is the Server-Sent Events (SSE) specification, which enables a one-way event stream from web servers to client web applications. Second, the WebSocket interface enables an efficient two-way communication channel between clients and servers, serving as a basis for both request-response and publish-subscribe style web applications. The following sections provide an overview on these technologies, listing their advantages and disadvantages, and discusses their applicability in industrial monitoring systems.

4.3.2 REST

Nowadays, HTTP APIs are often inspired by the REST architectural style, introduced by Fielding in his dissertation [23]. REST is a technology independent architectural style for designing distributed hypermedia systems for the web. Fielding defines REST in terms of the following constraints:

Client-server The client-server constraint separates data presentation concerns from data storage concerns, improving the overall scalability and portability of the architecture.

Stateless The stateless constraint dictates that session state should be maintained at client-side, and that clients should not depend on server-maintained state when issuing requests. Thus a request should have all the information necessary to understand it. This constraint simplifies the architecture of the server, and improves its scalability and recoverability.

Cache The cache constraint states that the server should inform the client whether response data may be cached for later use. The advantage of using a cached value is that performance is increased by reducing the number of requests to the server. The disadvantage is that a client may use a cached value that has been expired.

Uniform interface The uniform interface constraint means that the service implementation should be able to evolve independent of its public interface. To achieve this, resources should be uniquely identifiable, and it should be possible to manipulate them through their representations, which in turn should be self-descriptive and present the current valid transitions in terms of resource identifiers.

Layered system The layered system constraint ensures that clients need not know past the first layer of the system, simplifying system architecture and allowing the intermediate layers to perform for example load balancing and caching on passing requests.

Code-on-demand The optional code-on-demand constraint grants servers with the flexibility to respond to requests with client-executable program code, which relieves the client from implementing the respective features themselves.

The REST architectural style consists of the following three elements: components, connectors, and data. REST components are the user agent as the sender of the original request, origin server as the final destination for the request, and proxies and gateways as intermediary components forwarding and potentially manipulating the requests and responses [23]. Connectors are the interfaces that components use to communicate with each other. They abstract away the details of resource access and expose the resources as representations. Finally, the data element is the actual data that is exchanged between components. It includes resources and resource identifiers, resource representations, and control data. A resource is an abstract concept signifying a set of information that may change over time. Resources can be referenced with a unique resource identifier. Representations encompass the current or desired state of a resource at a specific time. Representations serve as the medium by which resources may be manipulated. Representations can have metadata associated with them, such as the time of last modification. The actual resources may also have metadata, such as information on the available alternative presentation formats.

In the recent year the implicit hypermedia constraint of REST has gained special attention. This constraint dictates that resource representations should reflect the application state by presenting the valid state transitions. In earlier implementations of the REST architectural style, this constraint was often overlooked and off-the-band information about the interface was used instead, resulting in a high level of coupling between services and clients consuming them. The hypermedia constraint is closely related to the semantic web, which attempts to introduce structure and semantics to the information found in the Internet, and by doing so enable machines to understand and process the information. There are several ongoing efforts to define a standardized hypermedia format including for example JSON-LD, HAL, and Siren.

4.3.3 WebSocket

WebSocket is an application level protocol which allows a web browser to initiate bidirectional, real-time connections to remote web servers. It does so by dividing communication into handshake and data transfer phases. The actual data transfer is designed to have a low overhead, requiring only a framing of two bytes in addition to the payload to differentiate between binary and UTF-8 encoded strings [18]. WebSocket is an independent protocol but it uses the HTTP Upgrade header to request an upgrade to a WebSocket connection. It is designed to work with the existing web infrastructure. First, it can use the standard HTTP and HTTPS ports, which allows it to bypass firewalls and integrate with the existing web infrastructure. Second, it is able to handle intermediaries such as proxy servers.

The WebSocket API [36] specifies the interface which WebSocket clients use to communicate with remote WebSocket servers. WebSocket is supported by the recent versions of all major web browsers, including Chrome, Firefox, Internet Explorer, Opera and Safari. In addition, many WebSocket server implementations developed in different programming languages are available, so developers do not generally need to develop their own implementations.

In comparison to plain HTTP-based techniques, WebSocket's main benefit is that it uses a persistent connection to exchange real-time data. Reusing the same connection results in lower overhead compared to HTTP, where each HTTP request and response carry a large overhead in HTTP headers [21]. This overhead accumulates quickly if messages are exchanges frequently. In addition, the data transfer has very low overhead with only 2 bytes of framing added to each data frame.

A WebSocket server can protect itself against malicious requests by checking the Origin header field [21]. If the value of the Origin header field is different than expected the server may reject the request. However, the server should be careful when trusting the authenticity of the Origin header field, as untrusted clients, or malicious extensions in trusted clients may spoof the Origin field.

4.3.4 Server-Sent Events

Server-Sent Events (SSE) is a technique used to deliver unidirectional push notifications from server to clients [37]. In practice most web browsers supporting SSE do so over the HTTP protocol but SSE specifies a generic interface that may be extended to add support for dedicated push notification protocols. In order to receive server-sent push notifications, a client opens an HTTP connection to the server. On the web browser level push notifications generate normal DOM events. Where HTTP long-polling issues a new GET request after each consecutive response it gets, SSE maintains the same HTTP connection for receiving events. SSE events support four different fields: event, data, id and retry. If one entry point is used for many different kinds of events, the *event* field can be used to differentiate between them. The default event type is *message*. The *data* field should specify the payload of the message. The *id* field should include an incremental ID which the client can use to keep track of the last event it has received. Finally, the *retry* field may be used to define the reconnection time in milliseconds.

The advantage of SSE in comparison to WebSocket is that it is much easier to implement both on the client side and especially on the server side. The server only needs to be capable of sending HTTP responses with *content-type* set to *text/event-stream*, and adhere to a simple message format. SSE also keeps track of the last event ID received and supports automatic reconnection on connection failure. From security's point of view, the server is relieved from the burden of validating user input since the data stream is unidirectional. The disadvantage of SSE is that in only supports sending events with

UTF-8 encoded payloads, so it is not well suited to transmitting binary data. One more limitation comes with the HTTP protocol, which limits the number of simultaneous connections the web browser can have to a web domain. As a work-around, the specification proposes the use of shared web workers in order to share one EventSource instance between multiple open pages belonging in the same domain. The specification also addresses the negative effects a constant event stream may have on the battery consumption of mobile devices. As a solution the specification outlines a connectionless push mode wherein the client may offload connection management to a proxy server [37].

Chapter 5

Frontend development

This chapter discusses the development of industrial HMI by using standard web technologies. The emphasis is on modern web technologies falling under the HTML 5 umbrella. First, Section 5.1 gives an overview of rich Internet applications, describing how common HMI functionality can be implemented with modern web technologies. Section 5.2 describes how the emerging web components specification enables modular, reusable HMI components in a standardized way. Finally, Section 5.3 presents some popular client-side JavaScript frameworks available today.

5.1 Rich Internet applications

Rich Internet Applications (RIA) are web applications that provide a user experience similar to that of a native desktop application. RIA web applications implement a thin-server architecture (TSA), wherein the server merely serves data to the clients. This is in contrast to thin-client applications, which are designed to minimize independent processing and depend heavily on the server to render HTML and to perform application logic. Thus, thin-client applications are characterized by frequent communication over the network, whereas thick-client applications need network resources only occasionally.

A systematic mapping study conducted by Casteleyn el al. [11] on the current state of RIA research suggests that new RIA research might be increasingly more often identified as standard web research, instead of RIA specific research. They propose that such change might be the result of new RIA enabling technologies, such as widgets, rich media types, and client-side storage, being added under the HTML5 standard. Among the considered research papers, the study also distinguishes three main drivers for RIA: the need for better interaction capabilities, richer user interfaces, and better responsiveness. They conclude that these goals can be achieved with asynchronous communication, distribution of operations between clients and servers, and the use of specialized RIA toolkits.

5.2 Reusable components

Component-based design is by no means a new concept to software engineering. Desktop applications have long taken advantage of this design pattern for added structure, minimized code reuse, and easier maintenance. Web standards in contrast have had poor support for the component based design paradigm. Until recently, web browsers have served mainly as thin-clients, with remote servers handling everything from database access to rendering HTML. These servers have served the thin-clients with pre-rendered HTML. The HTML has contained a relatively small amount of JavaScript to update the dynamic content of the web site using XHR requests.

Nowadays a wide variety of JavaScript frameworks exist for the frontend. Most of these frameworks enable a developer to encapsulate HTML, CSS and JavaScript into reusable components. There are varying solutions for implementing reusable components for web applications, which has called for a new standard that could unify existing approaches and allow components in one framework to be used in other frameworks.

The Web Components specification, which is being standardized by W3C and WHATWG, attempts to solve this problem. The specification consists of four independent parts: HTML templates, custom elements, shadow DOM, and HTML imports [66]. The first one, HTML templates, was later moved to be part of the core HTML5 specification.

HTML templates are reusable fragments of HTML, CSS, and JavaScript, that can be declared within an HTML document by using <template> tags, and accessed from JavaScript through the browser DOM [6, 38]. Templates are not rendered by default, nor do they incur side-effects such as resource fetching or script execution. In order to render a template, it first must be imported to the document. This is achieved by using the *document.importNode()* API method, which clones a node from another document, allowing it to be inserted into the DOM of the current document.

The custom elements specification allows page authors to define reusable, custom HTML elements [4, 28]. By default all custom elements inherit from the *HTMLElement* object, but elements can also extend standard elements such as <p> or other custom elements. The power of custom elements lies in the fact that they can encapsulate HTML markup, CSS styles and JavaScript code into a reusable component. The *document.registerElement()* API may be used to declare custom elements. In order to add functionality to the element, a developer attaches handlers to different element life cycle events.

Shadow DOM is a new way to encapsulate HTML markup, CSS styles and JavaScript code within a DOM element, and hide the implementation details from the user of the element [7, 29]. A special shadow root may be created for a DOM element by using the *myElement.createShadowRoot()* API. An element with a shadow root, called a shadow host, is rendered based on the DOM tree of the shadow root. The content of the shadow host itself does not get rendered. The main advantage of shadow DOM is that it allows an element to hide its internals from developers using the element. Styles defined within a shadow root are applied in the context of the shadow root and do not leak outside. The shadow boundary also protects the internal styles of the element from style definitions external to the shadow root. This encapsulation avoids conflicting *class* and *id* attributes, and the developer can be assured that attributes of a custom element do not conflict with the other attributes of the page.

The HTML Imports specification adds support for importing markup, style definitions, and scripts from external HTML documents [5, 30]. However, the imported content does not automatically get inserted into the DOM of the importing document. Instead, the import statement returns a reference to a DOM document object. This object has a property called import, which may be referenced in order to access the content of the imported file. Scripts in the imported file have access to the parent document, and can therefore embed markup to the parent document. The only content that is processed automatically, affecting the parent document, are inline style definitions and scripts. Inline style definitions are applied straightly to the importing document, and scripts within the import are executed in the context of the parent document, in sequential order, but without blocking the processing of the main page. This implies that imports can be used to run scripts asynchronously relative to the main page. HTML imports can be used to bundle different types of resources, such as scripts, style definitions, and images together. However, the normal cross-origin policy restrictions apply also to HTML imports, so remote file imports should support CORS.

5.3 Frameworks

Exposing data through a uniform service interface has the benefit of decoupling client implementations from the service interface. Thus the role of the service is reduced from rendering complete hypertext documents to serving domain specific data to the clients. For client application development this separation implies an unprecedented degree of freedom as web application development takes a step closer to resembling native application development. As a result, web application developers can benefit from new kinds of tools and frameworks that speed up the development process by adding structure to client-side applications, and by providing solutions to commonly encountered problems.

In order to make a distinction to libraries, this section uses the term library for small and focused software tools, that do not necessarily provide structure to the project as a whole. In contrast, the term framework is used to refer to collections of different functionality, usually provided in the form of software modules. In general, frameworks offer more functionality than libraries, but they often present more restrictions on the way the software is written.

The large number of available JavaScript frontend frameworks requires to survey the existing alternatives. This section gives an overview on some of the popular alternatives for building single page applications (SPA) in order to point out the similarities and differences between them, in addition to the advantages and disadvantages of them. Four different frameworks were evaluated: Backbone, Ember, Angular, and React. The following list presents a summary of the evaluation.

5.3.1 Backbone

Backbone [1] is a light-weight frontend framework which adds structure to client-side JavaScript projects by providing them with the missing structure. This is achieved by specifying a way to create models and collections that reflect the state of REST resources and allow for syncing between client and server, views whose representation is bound to models via change events generated by the models, and a router class for assigning request URIs to actions and events. One of the design principles of Backbone is to remain light-weight and flexible to work with, while providing only the most essential functionality. It also purposefully avoids two-way data binding, claiming that it is rarely needed in practice. Two-way data binding is a central feature of many other frontend frameworks.

5.3.2 Ember

Ember [44] is a frontend framework designed for building large Internet applications. The design principle of Ember is to provide developers with best practices solutions to common problems, while letting the developers concentrate on business specific problems. The drawback is that developers are more tied to the development style of the framework. Coding in Ember follows convention, so developers can save up on source lines by naming things in a specific way. Ember provides a two-way mapping between the browser URI and the state of the application, effectively allowing the URIs to uniquely identify individual application states. Whereas Backbone leaves the choice of view technology to the developer, Ember comes integrated with the Handlebars template language. The Ember team claims to have adopted best practices from native application development, thus allowing for single-page applications (SPA) that resemble native applications.

5.3.3 Angular

Angular [35] is an SPA framework from Google. It has built-in support for dependency injection (DI) which facilitates testing and allows to compose applications from loosely coupled components. The architecture of Angular resembles the MVC architecture. Models in Angular are achieved via objects called scopes. Scopes are organized in a hierarchical structure within the application, and serve as the context against which expressions are evaluated.

Models in Angular are simply variables and methods assigned to different scopes of the application. The value of scopes may be watched, and a change handler may be assigned to the watched expression. This is the basis of how data binding works in Angular. Method invocations in Angular generally cause the application to switch to the internal execution context of Angular, containing a digest loop. The digest loop evaluates the watched expressions in the current scope, and schedules the evaluation of expressions that should be performed asynchronously. This loop is repeated until an algorithm based on dirty checking does not detect any more changes in the watched expressions.

The view component of Angular are is HTML, coupled with directives, which declare custom HTML elements and attributes. Directives help to encapsulate functionality by linking it to different DOM elements. The preferred way to use directives is to perform all DOM manipulation in them. Directives may create new scopes, which may either prototypically inherit the parent scope or create a new isolate scope. Directives can add watches to variables within their scope.

Controllers in Angular are classes, or constructor functions, which are used to manage scope objects. Their intended use is to associate the scopes with data and methods. Each controller has its own scope that is injected to it via a constructor parameter. Controllers may also be injected with services for accessing data from different sources. Services encapsulate reusable functionality between controllers, including access to web services, local storage, application configuration and constant definitions.

In order to implement two-way data binding, the views in Angular have to be compiled. This compilation is performed on the initial page load, and consists of two distinct phases: compile and link. The compile phase creates a special linking function by traversing the DOM of the web application. It matches HTML elements to directives based on the configuration of the directives. Next, the synchronization between the template and its scope is achieved by creating a link between them. The link is created with the linking function which assigns watches on the scope.

5.3.4 React

React [59] is a frontend JavaScript framework developed by Facebook. It implements the view component of a client application, and can be used in conjunction with other libraries and frameworks, such as Backbone for data models and routing, and jQuery for manipulating the DOM. React does not differentiate between view models and the templates representing them. Instead, HTML is closely tied to the component definitions. In order to facilitate the writing of HTML in JavaScript, React introduces its own JSX syntax extension to JavaScript, which is a syntax resembling HTML. React introduces its own light-weight representation of DOM called virtual DOM. React has an efficient algorithm for calculating differences between two versions of the virtual DOM, which upon change to the underlying view models, allows it to quickly render only the changed parts of the view. Virtual DOM also enables React views to be rendered on the server side. Similar to Ember and Angular, React emphasizes the use of reusable view components, which are also set to support the upcoming Web Components specification [59].

Unlike Ember and Angular, React does not embrace the concept of two-way data binding, even though it can be achieved. Instead, React encourages one way data flow, where data flows from the parent components to their child components. This unidirectional data model behind React is called Flux [24]. There are three main components in Flux: the dispatcher, stores, and views. Views present the current state of the application and serve as the interface for a user to interact with the application. Views generate events on user action, and these events are transformed to actions, which are sent to the dispatcher. The dispatcher queues actions and only dispatches an action after the previous action has completed. The dispatcher sends actions to stores, which update the data model based on the data passed along the action. Finally the view gets notified of a data change and updates itself based on data it fetches from the store. If the change in the data model triggers more actions, these actions are sent to the dispatcher, completing the one-way data flow. The unidirectional data flow of Flux makes the application logic easier to understand. Since Flux is an architectural style, rather than a framework, it can also be applied outside React.

5.3.5 Discussion

In addition to the above frameworks, a new wave of frameworks is starting to take advantage of the new cutting edge features of HTML5, support for upcoming ECMAScript edition 6 and 7, and the Web Components set of specifications currently in the works. These frameworks directly target the newest ECMAScript specifications by implementing temporary replacements for those language features that are not yet implemented by web browsers.

In practice all of the listed frameworks possess many similarities. Perhaps the main similarity between the frameworks is that every framework seems to have adopted the Web Components mindset, and are providing their own ways to implement reusable web components. Many frameworks are promising support for the Web Component specification when it finally arrives to mainstream use. In addition, the syntax for declaring view models is largely the same. Each framework is also well documented. All in all, the features of the frameworks greatly overlap, making the comparison at a glance more difficult. However, by taking a brief look at the available frameworks provides valuable insight into the state of modern web development and industrial best practices.

Despite the many similarities, differences do exist, and the choice of framework is largely dependent on use-case specific requirements. Backbone is suitable for projects where a light-weight and industry-proven framework is required. Backbone does not govern how the project should be structured, but provides the basic tools to do so. Additional libraries are easily introduced to the project as the only hard dependency of Backbone is the Underscore.js utility library. Ember is a large framework which attempts to solve problems often encountered in the development of web applications, and presents a more dogmatic way to the development of web applications, depending on convention over configuration in order to reduce boilerplate code and to allow the developers to focus on the application specific logic.

There are several advantages to adopting the thin-server architecture. The main benefit is that application logic can be moved from the server to the client, reducing the workload of the server, allowing it to stay stateless, and thereby enabling it to scale horizontally. Another advantage is the possibility to share data models between the client and the server, as is demonstrated by for example the Ember.js framework. Other advantages include for example the ability to see changes in real-time by utilizing file watchers, and the possibility to use the same programming language both on the frontend and the backend.

There are also some disadvantages to having a thick client. Firstly, the application code can be publicly seen by anyone who has access to the web application. Depending on the use case, this may be undesired. Secondly, JavaScript is loosely typed so the script engine or integrated development environment (IDE) will not warn the developer about conflicting data types. The dynamic nature of the language also makes the creation of tooling support, such as debuggers, more challenging. Finally, with an increasing number of user agents running in mobile devices nowadays, moving application logic to the client-side has the adverse effect of increasing the battery usage on already power-constrained mobile devices.

In terms of tools and frameworks, the vibrant nature of the current JavaScript ecosystem provides many alternatives for new projects to choose from. On the downside, the choice of a frontend framework has become serious investment to organizations, which may prefer the choice of a well maintained framework backed by a large corporation. The JavaScript ecosystem is characterized by a high churn rate and in general there seems to be a high overlap between different frontend frameworks, as was evident in the list above. Frontend developers are also constantly faced with the need to supplement missing browser features with their own implementations in order to support older browser versions. This problem has been partly remedied with the introduction of automatic update systems in modern web browsers.

With the emergence of the Industrial Internet, it should be evident that the importance of web technologies is only set to increase in the coming years. Many solutions for developing native desktop and mobile applications using web technologies exist, including Phonegap, Ionic, and React Native for mobile application development and Node Webkit for desktop application development.

Chapter 6

OPC UA in a web browser

Several web-based SCADA solutions supporting OPC UA already exist on the market. Many of these solutions depend on client-side browser extensions or locally installed software components such as Java [62]. Among the available solutions that provide OPC UA connectivity from a web browser, three different categories were identified: pre-rendered user interfaces (Section 6.1), web APIs (Section 6.2), and native stacks (Section 6.3). This chapter gives an overview on existing solutions that provide OPC UA connectivity to browser-based clients. Section 6.4 compares the solutions in relation to the features they offer. The focus of the survey is on solutions that are driven by standard web technologies. The results of this chapter serve as a starting point when deriving the requirements for the web client application.

6.1 Pre-rendered user interfaces

In the traditional client-server approach of developing web applications, the web server renders complete hypertext documents, and serves them to the client. Pre-rendering in this context means the act of generating web pages in the form of HTML, which the client can consume, rather than predictive pre-rendering of documents by the web browser [65]. The advantage of rendering web sites on the server side is that it reduces the workload on client-side, and allows for minimal to no JavaScript executed by the web browser. This type of architecture is commonly referred to as the thin-client architecture, in contrast to the thin-server architecture presented earlier.

The thin-client architecture is well suited to the development of industrial HMI applications, as has been demonstrated by the configurable, purely web-based SCADA systems developed by Prosys PMS Ltd. For instance, one such system consists of a web server communicating with a remote SOAP server in order to provide data to its clients [62]. The architecture of the system consists of four components: the web browser, presentation layer, service layer, and OPC UA servers. The architecture resembles a typical client-server architecture with the exception that the presentation and services represent individual layers which are both separate from the client running in a web browser. This separation has the benefit of effectively decoupling data access from the business logic, and the business logic from the presentation logic. The separation of the presentation logic also has the benefit that it renders the web application searchable by web crawlers.

Another example of the thin-client architecture is Groov, developed by Opto 22 [31]. Groov is a toolkit for developing monitoring and control systems for mobile devices of different display sizes. It advocates the use of high performance HMI design guidelines presented by Hollifield et al. [40]. Groov comes with two components: a user interface designer web application and a server which can be deployed as an embedded device or as a service running in the Windows operating system. The user interface designer enables an operator to build scalable operator interfaces by dragging and dropping touchscreen ready components such as charts and indicators on a canvas, and binding those components to the data found in the address space of possibly multiple OPC UA servers. The Groov server on the other hand performs the tasks of a web server and an OPC UA client, performing as a proxy to forward service requests between the operator interface and the OPC UA servers. Finally, the operator interfaces are loaded on to the server and viewed with a web browser. A web browser initially loads the whole user interface from the server, but subsequent partial updates are performed via XHR requests. These partial updates along with client-side caching increase the performance perceived by the user. By utilizing native web technologies, Groov is able to function on a wide variety of devices as long as those devices feature a modern web browser. Operator interfaces composed in Groov are rendered on client-side, which opens the possibility to package them as native mobile applications by using Phonegap.

One more example is the Atvise SCADA system [2], which delivers functionality that is similar to Groov. It includes an editor for creating web-based operator interfaces, along with a default set of user interface components. Similar to Groov, the user interface components are implemented using Scalable Vector Graphics (SVG) which allows them to scale to different display sizes and layouts.

6.2 Web APIs

Web-based application programming interfaces (API) are a common way to implement re-usable web services which can be easily consumed by various end devices connected to the Internet. The web API approach effectively can hide the implementation details of the OPC UA communication and allows to develop an interface that is customized to the needs of the client applications. Web APIs also allows for service composition and can be utilized by a large number of end devices including smart phones, tablets and desktop computers.

HyperUA by Projexsys is one example of the web API approach. It is a JVM-based server application providing a REST compliant HTTP interface to OPC UA servers. It works by mapping OPC UA services to REST collections and resources. HyperUA complements the OPC UA security model with HTTPS and its own implementation of user roles and privileges.

The service conforms to the hypermedia constraint of REST, which means that after the initial API entry point has been discovered, all subsequent requests can be made to the links found in the response. The set of available links depends on the state of the requested resource and represents the valid state transitions. The advantage of conforming to the hypermedia constraint is that the state of a resource can be discerned from its representation. As a result, any changes to an URI other than the API endpoint becomes transparent, as the actual resource and links to it are separated. The API of HyperUA is traversed by applying CSS selectors on hypermedia responses.

HyperUA generates unique URIs for the nodes of the address space. Accessing these URIs with a web browser displays information about the corresponding node. An example application would be to attach QR codes to the machines of an industrial plant. These QR codes would point at specific nodes in the UA address space, providing the operator with detailed information about the machine and its state.

The current version of HyperUA supports the OPC UA Data Access, Historical Access and Alarms and Conditions specifications. Future plans for HyperUA include for example support for server-sent push notifications and cloud integration [42].

6.3 Native stack

Generally, web-based monitoring systems implement OPC UA connectivity via the use of an intermediate server, which serves both as a web server and an OPC UA client. However, most browsers support executing JavaScript code, so there is no reason why a standard web browser could not directly communicate with an OPC UA server, provided that an OPC UA stack for JavaScript is available. Indeed there exist efforts at introducing OPC UA to client-side and server-side JavaScript alike.

The JasUA stack by Hennig el al. [34] is currently the only OPC UA stack written in JavaScript, and usable in the web browser. It implements OPC UA binary encoding and UA secure conversation directly in the browser. Motivated by the recent improvements in the performance of JavaScript engines, the authors evaluated the performance of the implementation, claiming to have achieved sufficient response times for the stack to be usable. The benefit of this approach is that it does not require the installation of any external plugins. Instead, a web browser capable of running JavaScript code is sufficient. The authors reported problems with being able to support only a single OPC UA server due to cross-origin restrictions, and the failure to achieve a permanent secure channel, due to HTTP requests always creating a new socket connection. As a solution to these problems the authors proposed the use of a proxy server to bypass the cross-origin policy, and the use of the WebSocket protocol to create a reliable secure channel.

The JSUA framework by Freund et al. [26] enables the development of OPC UA client applications in pure JavaScript. It is an SDK level implementation of the OPC UA specification, based on the JasUA stack, and implements a hybrid stack profile which combines the binary encoding of the binary profile with the HTTP(S) transport and SSL/TLS security of the web service profile. The JSUA framework is currently limited to HTTP(S) connections, but according to the authors this could change in the future as a raw socket API becomes standardized.

An alternative to the JasUA stack is presented by Node OPC UA [60], which is a Node.js implementation of the OPC UA stack. Node OPC UA enables the creation of OPC UA clients and servers in the asynchronous and non-blocking way of Node.js. Node OPC UA is based on Node.js so it is tied to the server-side, and not strictly in the scope of web browser based implementations. Nevertheless, it serves as an example of the applicability of JavaScript to both frontend and backend side OPC UA application development.

6.4 Discussion

In this chapter, three different approaches for accessing OPC UA data from a web browser were examined, along with existing real world use cases. The main difference between the different architectures lies in the way workload is distributed between clients and servers. The most conventional way of serving pre-rendered pages to the client is the most flexible approach in terms of client CPU processing, as client interaction can be achieved with little to no client-side JavaScript code. The thin client architecture along with web services reduces CPU usage on the server side by moving the responsibility for rendering web pages to the clients. The client can use the web service to communicate with the system and the API of the service creates an abstraction layer between the client-side rendering and server-side business logic. For added flexibility, web applications may issue XHR requests on the service APIs to dynamically update their data models. Client-side interaction with web services is further facilitated by client-side JavaScript frameworks, which abstract away the interaction with the web services, providing a simple interface to query data.

Following the hypermedia constraint of the RESTful architectural design has many benefits. For example, user agents without JavaScript support and without off-the-band information about the API, such as search engine bots, can traverse the API. Finally, it was observed that the web browser is not a limiting factor, and that direct connectivity from the web browser to the OPC UA servers is possible through the use of JavaScript and the HTTP(S) protocol supported by the OPC UA specification. At the time of writing this thesis, there two JavaScript frameworks for OPC UA. One of these is a proprietary framework which works in web browsers, while the other one is an open-source Node.js module.

The different solutions reviewed in this chapter demonstrate the opportunities introduced by web-based monitoring of industrial data. Each approach has its advantages and disadvantages, and the choice of technology should be made depending on each project's specific requirements.

Chapter 7

Requirements

This chapter lays down the specific requirements for the web client. Section 7.1 begins by identifying the problem, the potential users, and needs of the users. Section 7.2 presents the different use cases that the web client should support. Section 7.3 discusses the general requirements associated the monitoring system as a whole. Section 7.5 determines the requirements for the user interface, and Section 7.4 completes the analysis by deriving the requirements for the backend system from the requirements of the HMI.

7.1 Overview

The plant floor of an industrial facility usually encompasses a large number of interconnected devices. Performing routine maintenance and pinpointing cause of system failures is costly and time consuming if the automation engineer must pay an individual on-site visit to each device. To make the monitoring of the whole system easier, industrial facilities employ SCADA systems, which allow the automation engineers to oversee the operation of the whole facility from a central location, called the control room. However, the problem with most today's SCADA solutions is that they require the installation of dedicated SCADA software, which must be separately installed on each device that is to be used for monitoring purposes. Furthermore, support for the software is often limited to certain operating systems [62]. The solution is to deploy the monitoring system on the web by using standard web technologies such as HTML5 and JavaScript.

7.2 Use cases

In order to establish a connection, the OPC UA client needs the URI endpoint of an available OPC UA server. It can find this information either by the means of manual user input, or even automatically by using the OPC UA discovery services. Since the OPC UA web client being developed is a proof-of-concept in nature, implementing server discovery is out of the scope of this thesis. After obtaining a valid endpoint URI, the client should be able to use it to connect to an OPC UA server.

A client that has connected to a server should be able to view the server address space and find information in it. The user interface should convey the hierarchical structure of the address space to the user by displaying the address space as a mesh network of UA nodes. The root node constitutes the entry point of the server address space, and as such it should be presented in a way that makes it possible to navigate to each node of the address space. This can be achieved by issuing browse requests and following the references that are returned in response. The user can select a node in the hierarchy and immediately see its details.

After browsing to a node, a user can view its attributes, including its value. The user may also write a new value to the value attribute. Reading an attribute should always display a value that represent its current state at the time of the request.

Reading variable values manually is not practical, especially if there is a large number of nodes values that need to be monitored. The UI should present the user with means to subscribe to value changes. The UI should then automatically update to reflect changes in the subscribed values.

In addition to data changes events, the user should be able to subscribe to events and notifications originating from the OPC UA server. Incoming events should be presented to the user in the order that they were generated. A user should also be able to acknowledge events by calling the appropriate method in the server address space.

Besides reading the current value attribute of a node, the user should be able to fetch value history from the OPC UA server. The user should be able to specify history read parameters such as the date and time range of the data to be fetched. Large responses should be paginated, in order to reduce the processing load on the web browser.

Finally, the user should be able to call server-defined OPC UA methods. Method nodes should be visually distinct in the address space browser, and include controls for specifying method parameters and calling the method.

7.3 General requirements

The web client is targeted at automation engineers who need to gain remote access to the devices connected to an industrial network, and the data residing in them. The web client should allow the user access to remote OPC UA servers by using the web browser of a modern smart phone. The connectivity between a web browser and an OPC UA server provides the browser-based clients with a standardized way to access the data of the automation system.

Since the web application is an application prototype, it is designed with modern web technologies in mind. Supporting old browser versions is thus not required. In the context of this thesis, a modern web browser is taken to mean browsers that support standard HTML5 features such as the EventSource interface for sending push notifications. Web-based applications have many advantages over their counterparts on the desktop. For example, web applications do not require installation. Web applications are also more portable, as they can be used with any client device with a web browser. However, these advantages can only be achieved if the web application does not require external dependencies such as browser plugins or other software installed on the user's operating system.

Requirement 1. The web client should not depend on external software components.

The development of a web based SCADA solution is a reoccurring problem. As such, it would be beneficial to develop a solution that allows for code reuse. By developing the view logic and business logic as separate components, the generic business logic of interacting with OPC UA servers can be reused and only a customized SCADA interface needs to be developed from scratch, when customizing the monitoring system for different use cases.

Requirement 2. The service and presentation layers of the web client should be decoupled from each other.

Table 7.1 lists the OPC UA related functional requirements of the web client, along with the profiles that encompass them. According to the OPC UA specification, the minimum requirement for an OPC UA client is that it should support the Core Client Facet and at least one transport protocol. The Core Client Facet includes essential session management and security features, such as connection handling and encryption.

Requirement 3. The web client should conform to basic OPC UA client facets.

Required feature	**Profile**
Transport protocol	UA-TCP UA-SC UA Binary Profile
Security and session management	Core Client Facet
DataAccess Client Facet	Browsing the server address space
Reading attribute values	Attribute Read Client Facet
Writing attribute values	Attribute Write Client Facet
Subscribing to data changes	DataChange Subscriber Client Facet
Subscribing to events	Event Subscriber Client Facet
Reading historical data	Historical Access Client Facet
Calling server-defined methods	Method Client Facet

Table 7.1: The requirements in relation to OPC UA client profiles [53].

When defining the requirements, a question that arises is how to document them. One way to specify the application's requirements is to write them as unit tests. Writing unit tests has the added benefit of documenting the requirements directly in the code. Many software companies doing agile development also advocate the use of test-driven development for specifying the requirements in the form of desired behavior of the system [10].

Requirement 4. The web client should incorporate unit tests.

The OPC UA specification defines a security model which addresses the information security of OPC UA communication, including issues with authentication, authorization, integrity and confidentiality [49]. However, in the scope of this thesis it becomes important to ensure that the communication between an HMI and the service interface remains secure, in order to maintain proper end-to-end security between HMIs and OPC UA servers.

Requirement 5. The web client should respect the end-to-end security principles of OPC UA.

In order to support varying deployment options, the web client as a whole should be configurable. For the service layer this means that it should be possible to specify in a configuration file the endpoints for the service interface and push notifications. In addition, there should be a separate configuration file for the presentation layer, where the respective API endpoints can be configured. A separate configuration file is needed, so that the client can be configured independent from the service, and a loose coupling can be achieved.

Requirement 6. The web client should be configurable.

7.4 Service requirements

Nowadays, the use of JavaScript on the server-side has become common. However, the large number of different JavaScript frameworks and libraries makes it increasingly more difficult to decide on an appropriate framework. It is difficult to know how long a specific library or component will be supported. This is much unlike the Java EE platform where the future of a particular component can be predicted more easily. Therefore, the service component should be designed to be modular to allow for substituting the libraries later on. For example, the service interface should be defined independently from the module used to realize the web services.

Requirement 7. The source code of the service layer should inhibit a modular structure.

The service layer provides the interface for the presentation layer to interact with the OPC UA servers. The service layer should be decoupled from the presentation layer, meaning that any HTTP-capable client application should be able to use it. The API should be simple, scalable, reusable and discoverable.

Requirement 8. The service should expose a request-response based API which is usable by standard web browsers.

Unlike traditional desktop applications, web applications have to support potentially large number of concurrent users. For the web client this means that it must be possible to maintain multiple OPC UA sessions. Users should only be able to access their own sessions, but not the sessions of other users.

Requirement 9. The service should support multiple simultaneous users.

As a prerequisite to conforming to the OPC UA client facets above, the service layer should be capable of communicating with remote OPC UA servers. This communication should be possible both on-demand by sending OPC UA service requests, but also in real-time by assigning event listeners.

Requirement 10. The service should provide connectivity to OPC UA servers.

The high-level service methods of the OPC UA SDK return OPC UA service response data as Java object references. In order to return the requested data to the web clients, it is necessary to serialize and encode the response data to an appropriate format. This format should be simple and human readable. Simplicity of the format makes debugging the service layer easier, and also facilitates the implementation of web clients that consume the service. It should also accommodate the structure and semantics of plain Java objects.

Requirement 11. The service should serialize OPC UA response data.

In order to conform to the subscription related client facets, the service layer should be capable of sending event notifications to clients, who have subscribed to specific events in the address space. This means that the service layer must be able to send real-time push notifications to web browsers.

Requirement 12. The service should be capable of sending real-time push notifications to web browsers.

7.5 User interface requirements

Upon entry to a website, it is common for the site to ask the user whether a mobile optimized website should be served instead of the typical one. Many organizations maintain a separate mobile-optimized website, instead of having one website that properly adapts to the user's display size. By using a responsive design approach, the style definitions of the website can be parametrized based on the display size of the client device. The main advantage of this approach is that only one version of the website needs to be kept up-to-date. The mobile-first methodology supports this approach.

Requirement 13. The layout of the user interface should adapt to different display sizes.

Existing SCADA toolkits usually come with a set of reusable user interface components for displaying various information about the process being monitored. These components can range from simple indicators displaying for example the monitored values and their limits, to the representation of physical devices such as boilers, valves, and thermometers. The HMI components allow the automation engineer to monitor and control the process by simply interacting with the UI.

Requirement 14. The web client should support reusable user interface components.

In addition to the above requirements, the user interface should incorporate controls that the operator can use in order to realize the use cases described in Section 7.2. In practice this means that the user interface should support the features presented in Table 7.1.

Chapter 8

Implementation

This chapter describes the implementation of the OPC UA Web Client. The goal is to address the requirements set in Chapter 7. First, Section 8.1 addresses the first three requirements by giving a brief overview of the chosen architecture and technology stack. Then, Section 8.2 presents the various technology choices that had to be made in order to realize the web client. Next, the different parts of the implementation are discussed individually, addressing the suitability of the chosen technologies to the project at hand, and describing the challenges faced during the application's development. Section 8.3 describes how the service layer requirements were met, while Section 8.4 describes the user-facing part of the system, the presentation layer, and how its requirements were met. Finally, Section 8.5 discusses the testing and debugging of the web client, Section 8.6 explains the security features of the web client and Section 8.7 presents how the web client can be configured.

8.1 Overview

In the practical part of this thesis two compatible but independently deployable software components were implemented: a web service that exposes a RESTful interface for interacting with remote OPC UA servers, and a simple OPC UA web application that uses this interface to provide an HMI. Together with the OPC UA servers these components form the three-tier architecture portrayed in Figure 8.1. The web application was developed and tested together with the service in order to verify that the overall solution meets the requirements set in Chapter 7. The implementation of the OPC UA servers is out of the scope of this thesis. The monitoring system was tested by connecting the web client to a locally deployed instance of the

Prosys OPC UA Simulation Server. The Prosys OPC UA Java Client and Unified Automation's UaExpert were used as an example and starting point when designing the user interface of the web client. They were also used as a basis when choosing which features to implement, along with the OPC UA specifications and software solutions examined in Chapter 6.

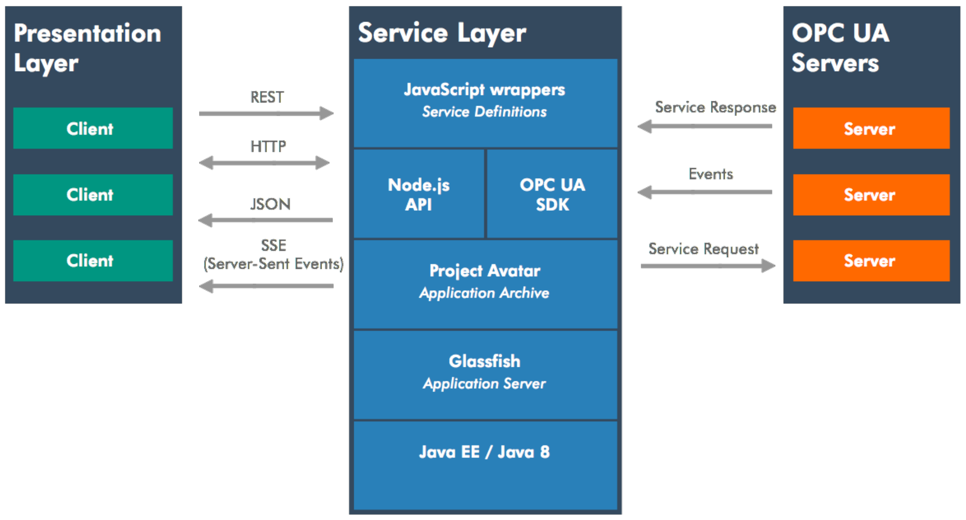


Figure 8.1: The architecture of the OPC UA Web Client.

The web client was designed to support standard web browsers. Aside from the service layer, it does not impose any external dependencies, such as Java, on web browsers. However, it does target modern, HTML5-enabled web browsers. This was done on purpose, because the purpose of this implementation was to serve mainly as a prototype implementation. If desired, the features of the web client that depend on technologies not found in older web browsers can be easily enabled by the use of various polyfills.

The service and view components of the web client are decoupled from each other. This is achieved by using a uniform HTTP interface between the web client and the service. The drawback of the current implementation is that the various API endpoints are hard-coded in the web client. This makes it difficult to introduce changes to the API without breaking support for existing client implementations. From the perspective of loose coupling, a better approach would have been to make the API more easily discoverable to client implementations by following hypermedia formats such as JSON-LD.

The web client conforms to basic OPC UA client facets both on the service layer and presentation layer levels. This is facilitated by the use of the Prosys OPC UA Java SDK, which provides high-level interfaces for interaction with OPC UA servers. The service implementation is thus more focused on application-specific aspects such as session management, response data serialization, and forwarding of real-time events to web browsers. Section 8.4 describes how the web client implements the required client facets.

8.2 Technology choices

The first choice that had to be made was whether the web client should be developed in pure Java or by using one of the emerging polyglot frameworks for the JVM. The latter approach was chosen for a few reasons. First, by using a polyglot platform, the number of programming languages supported by the Prosys OPC UA Java SDK could be increased. Second, this approach would allow Node.js applications to leverage the functionality of the SDK and on the other hand Node.js scripts could be embedded into existing Java-based OPC UA applications. Third, there is currently only one commercially supported JavaScript SDK for OPC UA, so there is a potential demand for an additional JavaScript API to the Java SDK. Finally, there are potential differences in developing the application code in JavaScript, which are worth investigating. As a side note, an alternative solution that was not explored in this thesis is the node-java Node.js module which provides access to Java classes from within Node.js scripts. However, this approach is limited to the embedding of Node.js scripts in existing Java applications.

The decision to write the experimental client application in JavaScript prompted for a choice between the available JVM-based server-side JavaScript frameworks. The evaluated frameworks included Avatar, Nodyn, and Trireme, all of which implement the popular Node.js API. The decision was to go with Avatar to the early availability of a public release. The fact that Avatar is developed by Oracle also affected the decision, because it was deemed at the time that it would most likely be supported in the future and take benefit of the existing Java EE ecosystem. At the time of choosing the framework, the documentation of Avatar, albeit lacking, was the best among the candidates. The potential of a full-stack framework, including both view and service components, was also intriguing.

The third choice was to decide how communication between the web browser and the service layer should be accomplished. Since one of the requirements was to rely on standard web technologies, the most basic need of transferring hypermedia was achieved with the HTTP protocol. The REST architecture was chosen to implement the service API, with HTTP as the transport protocol. The reason to use REST was that when implemented correctly it could make applications more easily scalable to multiple server instances. Additionally, the REST architecture complies with Internet standards and is widely used and supported by clients. Finally, a protocol for transmitting periodic events was required. Two protocols introduced by the HTML5 standard were evaluated: SSE and WebSocket. The SSE protocol was chosen because only unidirectional server-to-client push notifications were required. Specifically, the use cases of the web client did not require a two-directional low latency connection between the client and the server. Another reason for choosing SSE over WebSocket was its simplicity compared to WebSocket.

After fulfilling the requirement of transmitting data between the client and server, it was required to display the data and controls in a client-side user interface. On the most basic level the web browser would request JavaScript files from the web server and execute them locally to add rich-client functionality to the downloaded HTML. The jQuery library is often used to create dynamic web sites that can manipulate the content of the web site and to issue HTTP requests without causing a full page reload. Since a requirement was to implement a rich user interface for the web client, it was justifiable to experiment with existing frontend frameworks for JavaScript. The evaluated frameworks were Backbone, Ember, Angular and React. The decision was to use Angular because it has the potential of catering to Java developers familiar with JSF. The syntax of embedding directives in view templates is very similar to the syntax of Facelets used in JSF. Additionally, in Angular user interfaces are defined declaratively.

The final choice to be made was how the user interface components would be composed. Industrial user interfaces are often component based. One component may be reused to show information from different device instances. For example, a component to display a temperature value could be reused across boiler instances. Three methods were evaluated: Angular directives, React components, and WebComponents. The Angular directives were chosen for simplicity and because they seemed to be the most easiest to integrate with Angular scopes. In any case, using any of the three technologies would separate component declaration from their use, facilitating changes to the underlying technology at a later time if deemed necessary.

8.3 Service layer

The service layer implements a uniform, web-accessible interface for interacting with OPC UA servers. In order to do so, it serves as a proxy between the browser-based HMIs and the OPC UA servers, concurrently fulfilling both the responsibilities of a web server and an OPC UA client. The service layer maps incoming HTTP requests from the web browsers to OPC UA service requests, which it sends to the corresponding OPC UA servers. A server responds to a service request by sending a response message, which the service then forwards back to the requesting clients in response to the original HTTP request.

The service layer is implemented in JavaScript, and it uses the Avatar framework for session management, creation of REST and SSE endpoints, and threading coupled with a inter-thread communication mechanism through messaging. The OPC UA client functionality of the service is implemented by using the OPC UA Java SDK, a feature which was made possible by the underlying Nashorn JavaScript engine.

The service layer is organized into modules which each have their own distinct responsibilities. This separation of concerns enables to test each module individually. Moreover, clear interfaces between modules ensure that functionality of one module can be changed without affecting the others. The most important modules are the main module for starting up the service, wrapper modules for the OPC UA Java SDK, modules that define the service endpoints exposed by the server, and a module for defining listeners to events generated by the SDK.

8.3.1 Service interface

A RESTful architecture was chosen for implementing the service interface. The architecture was chosen because its architectural restrictions promote scalability, simplicity and re-usability through statelessness. REST is a popular architecture among web developers, and maps conveniently to the HTTP protocol supported by web browsers.

The REST interface enables communication between web browsers and the OPC UA client service. It does so by mapping OPC UA services and nodes to RESTful collections and resources. For example, an OPC UA node can be logically represented as a resource, and a browse service request can be realized with a GET request to a collection of node resources. Given below is an extract of the RESTful interface.

/api/sessions{/sessionId} Represents a server connection.

> **GET** Lists all session resources or retrieves a specific one.
>
> **POST** Connects to a server by creating a new session resource.
>
> **PUT** Updates a session resource by replacing it.

DELETE Disconnects from a server by removing a session resource.

/api/sessions/{sessionId}/nodes{/nodeId} Represents a node.

GET Retrieves the root, or a specified node, from the address space.

POST Adds a new node into the server address space.

PUT Updates the structure of an existing node in the address space.

DELETE Removes a node from the server address space.

The services module of Avatar framework follows the popular request-response model. Services are defined by implementing service handlers for the desired URL patterns. The handlers receive request and response objects as parameters and are tasked with constructing and sending a response. The service handlers perform three duties. First, the request object is checked for the user parameter, which tells whether the client sending the request has authenticated. If not, a new session is created and the client associated with that session. Next, the session ID of the current session is fetched and the JsClient instance associated with that session is fetched. Finally, the service handler calls the method of the JsClient instance that corresponds to the requested service, spawning a new operating system thread for it, and immediately returning a promise object. When the operation finishes the promise also gets resolved, and the service handler sets the CORS headers on the response object and sends it.

Since the service interface was designed to be consumed by web browsers, it was only natural that the interface should accommodate the needs of the potential web client implementations. In practice these needs were often found to be in conflict with simplicity of the web service. For example, one question during development was whether the service layer should directly mirror the API of the OPC UA SDK. The initial version took this approach. It was found out that this approach significantly complicated the development of the client component, as several consecutive requests had to be made to the service API to achieve the desired result. For example, the root node had to be fetched first. Then the references of the root node were to be fetched. Finally, the nodes referred to by the root node had to be fetched. Thus, a total of three requests were required in order to acquire the information required to display the initial state of the address space browser. The later versions of the service fixed this issue by complementing the response JSON with embedded resources.

The REST API supports hypermedia traversal by embedding hyperlinks to response JSON. The hyperlinks use a standard template URI syntax, where the client can substitute URI parameters with appropriate values, as shown above. The initial version did not incorporate support for hypermedia traversal, but support was later added by introducing a simple URI based format, where the service simply embeds a uniquely identifiable URI to each resource representation. In practice, this turned out to be easy to achieve as it only required the addition of one additional URI property. These URIs can be referenced to attain the corresponding representations. A resource always includes a URI to itself, but may also include URI references to other resources. These additional URIs signify the available state transitions, and may be used by a client implementation to automatically discover the API. A client may also cache the links and at a later point in time obtain an up-to-date representation of the cached resource, without having to traverse the whole API starting from the API entry point again. This URI based approach was inspired by the public API of Github. An alternative approach that was first evaluated would have been to use an existing hypermedia format such as HAL or JSON-LD, but this would have added extra complexity to the response JSON, without adding any significant benefits, as there currently is no clear consensus on what hypermedia format clients should support.

One problem with the service interface was how history read should be handled. Reading history from an OPC UA server is a potentially slow operation, and responding slowly to a request would degrade the user interface of the web client. The first solution to this problem was to limit the amount of data that could be requested. In practice the service always returned the last hour worth of history data. However, even with this kind of restriction the service would take several seconds to return the history data. This response time was clearly unacceptably long, so the next attempt was to split the history read into multiple separate chunks. Fortunately, OPC UA specifies that services should support continuation points. The continuation points were combined with the deferred.progress API which allowed to easily send progress notifications via the message bus of Avatar to the push service. The push service then sent subsequent notifications with the data as payload, as new chunks from the OPC UA server were received. This allowed the service to respond to the request immediately, notifying the client that history read had commenced. The drawback of this approach was that it complicated the implementation of the history view, because of the added responsibility of handling the push notifications and updating the chart dynamically.

Another problem had to do with how the OPC UA stack performs conversions between service parameters and their string representations. The conversion is important because the SDK must be able to correctly parse the string formatted parameters as they are passed through the REST interface. For instance, NodeIds are composed of three components: namespace index, type and value. Encoding of NodeIds to strings is performed according to the following syntax [52]:

```
ns=<namespaceindex>;<type>=<value>
```

According to OPC UA Part 6 the conversion of NodeIds is done by converting the namespace index and value part to strings, and concatenating them with a semicolon (;). The Internet standard for Uniform Resource Identifiers (URI) however reserves the use of semicolons in URIs, and thereby prevents the use of standard string formatted NodeIds in HTTP requests. Replacing the semicolons with an unreserved delimiter such as a dash (-) was not an option as it could not be guaranteed that a server would not use a specific character in NodeIds. Moreover, deviating from the specification could confuse clients. The solution was to encode the NodeId URI parameter before passing it to the server as part of the URI. This was achieved by using the *encodeURIComponent()* JavaScript function.

8.3.2 Session management

In order to support multiple simultaneous users, each one connected to possibly different OPC UA server, the web client needs to store client related session data somewhere. There are two options. First option is to store all session data on the service layer. This is the approach that was taken by the web client, because it has the benefit of integrating well with the session model of OPC UA, as browser sessions can be directly mapped to related OPC UA sessions. The added benefit is that existing OPC UA access control mechanisms can be extended to the web client. The drawback is that, by storing sessions on the server, the service layer breaks the stateless constraint of the REST architecture. This in turn makes it more difficult to scale the service horizontally, as services need a way to access shared client session information with one another.

The other approach would have been to store client related session data directly in the client. This approach is in line with the REST architectural style, because it allows the requests of the client to contain the session data that the server needs to understand the request. It also has the advantage of freeing resources from the server-side and allowing it to freely scale horizontally. Taking this approach would have required additional work, as a stateless abstraction layer between the service interface and stateful OPC UA servers would have had to be devised. This feature is subject to future work.

8.3.3 Retrieval of data

In order to be able to server browser-based clients, the service must be capable of communicating with remote OPC UA servers. The Nashorn JavaScript engine made it possible to use classes of the OPC UA Java SDK within JavaScript code. The SDK consists of a set of jar files that had to be placed in a standard location where the Glassfish application server could find them. Next, the required packages were imported in the top of the source file in similar fashion to Java. After the desired classes were imported and aliased, they could be transparently used in JavaScript code. The syntax of accessing the methods and properties of a Java class is conveniently the same as for native JavaScript objects, but on the other hand it made it difficult to discern Java objects from native JavaScript objects. This problem could be alleviated by improved IDE support, but for now a convention to prefix Java objects with the letter 'j' was assumed.

The service layer uses the Nashorn JavaScript engine to gain access to classes of the Java SDK. Using the proxy objects provided by Nashorn is convenient but requires to keep track of which JavaScript variables refer to Java classes and instances. The service layer's convention is to prefix proxy objects wrapping a Java class with the letter 'j'. For example, an instance of the Java class UaNode is referred to as jUaNode. To further facilitate the development of JavaScript based OPC UA application against the SDK, the service layer defines the JsClient class which encapsulates access to the Java SDK. For added convenience, most of JsClient's functions return promises.

For simplicity, the service uses the synchronous service methods of the OPC UA Java SDK. These methods provided a higher level of abstraction compared to the asynchronous alternatives, and were easier to use. In standard Node.js using synchronous calls would be a bad idea since JavaScript is single-threaded. OPC UA service calls could easily block the event loop and no other requests would get served during the blocking operation's execution. The service layer is able to work around this issue by using Avatar's threading module, which it uses to spawn a worker thread for each new service call. Starting to execute a thread returns a promise object which makes it easy to chain asynchronous operations together. Listing 8.1 shows how the getRootNode function handles the asynchronous method call getNode by returning a promise to the calling function. Replacing the synchronous API calls with the asynchronous alternatives is subject to future work.

The call to *getNode* returns a promise, so it is necessary to also wrap the getRootNode function with a promise. Otherwise the function would return prematurely.

In practice the deferred API was misused in the early iterations of the

```javascript
/**
 * Returns the root of the server address space.
 * @returns {Promise.<UaNode>} The root node.
 */
JsClient.prototype.getRootNode = function getRootNode() {
    var deferred = Promise.defer();
    if (!this.jUaClient.isConnected()) {
        deferred.reject(new exceptions.ClientException("Not
            connected"));
    } else {
        this.getNode(this.rootNodeId)
            .then(function(rootNode) {
                deferred.resolve(rootNode);
            }, function(error) {
                deferred.reject(error);
            });
    }
    return deferred.promise;
};
```

Listing 8.1: The getRootNode method of the JsClient proxy object.

service layer, as it should not be used only for the sake of providing a promise API. Instead, it should be used to provide the promise interface to an existing asynchronous API that does not return promises, but depends on callback functions to handle asynchronous results.

When a client issues its first request on the service API, a new JsClient instance is created and linked to the client's session ID. Subsequent requests continue to use the existing JsClient. The UA sessions exist separately from the browser sessions. While the current implementation forces a one-to-one mapping between a browser session and an OPC UA session, future versions might associate multiple OPC UA sessions with a single browser session, enabling the web client to maintain multiple simultaneous server connections.

The response payload varies in size depending on the service request. For example, browsing the server address space returns the requested node, along with its references. Each reference if returned in its full JSON object representation would take around 1 KB in size, thus a node with a thousand references would take around 1 MB including headers. While HTTP does not limit the content size [22], especially mobile devices with limited computing resources might be too slow to process the response data. For this reason the service layer supports dividing the response into multiple messages, delivered as push notifications. The JavaScript interface to the history read service takes advantage of this feature.

```
/* Importing a single Java class. */
var HashMap       = Java.type("java.util.HashMap");
var ArrayList     = Java.type("java.util.ArrayList");
/* Importing a Java package. */
var ClientImports = new JavaImporter(com.prosysopc.ua.client);
var ClientNodes   = new JavaImporter(com.prosysopc.ua.nodes);
/* Accessing individual Java classes. */
var jUaClient = (new ClientImports.UaClient()).setUri(uri);
var jEndpoints = jUaClient.discoverEndpoints();
```

Listing 8.2: An example of using the JavaImporter class.

In similar fashion to plain Java, using Java classes in Nashorn requires that the containing packages are first imported. According to the Avatar framework source code, the best practice seems to be to import the packages at the top of the file, like in Java. In practice the list of imported packages quickly grows as classes from different packages are required. One of the problems in developing the service layer was managing the growth of Java class dependencies. The task of dependency management was especially challenging as the IDE did not provide any auto completion for importing Java classes. It also did not provide any other help in managing the dependencies, such as warning messages about missing imports. Therefore, the introduction of new Java classes to the JavaScript source code often resulted in confusing runtime exceptions, if the import statement was missing. However, the Nashorn runtime does provide a useful extension to the JavaScript language, which helps to address these problems: the *JavaImporter* class. The *JavaImporter* class allows to import specific packages, creating a scope which may be assigned to a local variable, as shown in Listing 8.2. This is in contrast to having to manually import each individual Java class that is to be used in the JavaScript application.

One problem related to working with the Java classes had to do with exception handling. Initially the service layer used try/catch blocks to handle errors. This worked well when the API calls were performed synchronously, but when the code was made asynchronous with the use of the threading module of Avatar, strange bugs started to occur. The reason for this is that the throw/catch exception handling model does not work with asynchronous functions. Because of the asynchronous processing model, asynchronous function calls are deferred to a later point in time. When this kind of function throws an error, the function that made the call and contained the catch block has already exited. As a result, the error does not get handled and the application will crash. The correct way to handle these errors would have been to pass the potential error as a parameter to the callback function, as is the convention is Node.js.

8.3.4 Serialization

One of the requirements for the web client was to implement serialization between Java and JavaScript objects. The requirements for the serialization format were that it should be simple and human readable, and that it should be capable of representing the data structure of plain Java objects. The JSON data format was chosen because it is natively supported in JavaScript and fills the set requirements.

The next problem was how the actual serialization between Java and JavaScript objects should be performed. Two Java-libraries were investigated for this purpose: Google GSON and Jackson data-binding. Upon evaluation it was discovered that while the serialization worked performed well for simple Java objects, more work have been required to support more complex Java objects with several references to other objects. The Jackson data-binding library depends on annotations on the Java classes. This can be a problem for developers who only have access to the binary version of the Java SDK. GSON on the other hand required to implement special serializer and deserializer classes in Java. This approach presents an interesting subject for future work, but for now implementing the serialization in plain JavaScript was more straightforward. Furthermore, implementing the JSON serialization of SDK classes could be better suited to be done on the SDK level. This way future implementation could take advantage of the shared serialization functionality.

The first iterations of the web client implemented a module for converting Java objects to plain JavaScript objects. The advantage of this approach was that JavaScript objects could then be easily converted to the JSON string format. The converter module separated the task of serialization from the actual model that was to be serialized. The problem with this approach was that in practice the converter implementation was tied to the specific type of object it was supposed to serialize. For this reason the converter was later replaced with JavaScript wrapper classes, which wrapped the Java objects returned by the OPC UA SDK. Encoding the returned objects was then reduced to a matter of calling JSON.stringify on the JavaScript wrapper objects. This method serializes JavaScript objects to the JSON format required by the service interface, stripping all methods from the resulting JSON.

The wrapper classes allowed to use the exact same data model on the server-side and client-side. However, the JSON serialization phase strips all functions from the serialized objects, so it is not possible to use them on the client-side. Nevertheless, this did not turn out to be a major problem, as most of the OPC UA related functionality of the web client is driven by the service interface.

In order to retain special characters in query parameters, all requests sent to the service layer need to have their request URI query parameters encoded before transmission. Thus, it is the responsibility of the service layer to decode all query parameters by using the standard decodeURIComponent JavaScript function. Similarly, the service layer encodes all query parameters in the resource URI embedded in the response data by using the standard encodeURIComponent JavaScript function.

8.3.5 Subscriptions

The timely delivery of up-to-date process data is an essential feature of any monitoring system. In the context of web applications, the conventional way to transmit real-time data to the web browser is to have the client poll the server for new data with frequent intervals. However, polling is not optimal for frequently changing data as it incurs the extra overhead of HTTP request headers. HTML5 introduced the SSE and WebSocket protocols for the delivery of real-time push notifications to browser-based clients. Since only one-way data flow was required for implementing the transfer of OPC UA events, the more simple SSE protocol was used.

Nashorn specifies a simple syntax for extending existing Java classes in JavaScript. The service layer uses this feature to extend the listener classes of the OPC UA SDK with event handling logic. When a new event is received, it is transformed into a JSON object and published to all interested subscribers by using the MessageBus class of the Avatar framework. The push service receives the published events and pushes them to the corresponding clients by using SSE. The advantage of SSE is that it supports automatic reconnection and that an SSE server is considerably easier to implement than its WebSocket alternative. The main disadvantage, aside from being unidirectional by design, is that the HTTP specification limits the number of concurrent HTTP connections on domain basis, which prevents the user from opening the web client in multiple browser windows at the same time. The SSE specification suggests the use of HTML5 web workers to circumvent this problem.

One of the requirements for the web client was that it should take advantage of modern web technologies. While all of the technologies covered in this thesis are well supported by modern web browsers, the availability of fallback strategies is still relevant. For example, the latest version of Internet Explorer is yet to support the Server-Sent Events specification. Moreover, despite the automatic update feature of modern web browsers, considerable number of web browsers in use today represent older releases. For instance, automatic updates may be hindered by restrictive company policies. Such cases present a relatively small but real motivation for fallback technologies, which are commonly referred to as polyfills.

The service sends heartbeat messages to the web client. The absence of heartbeat messages informs the connected web clients that connection to the service has been lost. This allows the user interface to display the connection status to the user. Heartbeats are sent through a separate push endpoint. The drawback is that it does not solve the problem with HTTP proxies that close the connection after a specific timeout [37]. A better approach would have been to send a periodic comment line in the event stream, as suggested in the SSE specification.

One problem encountered with the push notifications was related to the buffering of notifications. The service layer maintains a buffer of outbound events. This is done in order to maintain control of the rate with which messages are pushed out to the clients. The push service uses a shared buffer for outbound events. Events are added to the buffer via a single message bus shared by all service instances and worker threads. Avatar enables the automatic load-balancing between these threads. Running several service threads caused events to be produced from multiple threads simultaneously. These events were then then received by the push handler. Access to a shared buffer from multiple threads resulted in race conditions. The issue was finally solved by implementing a simple critical section around the part of the code which deals with the shared buffer. This approach is not optimal. More work is required to refactor the code so as not to require in the use of shared buffers. This would have the added benefit of better scalability of the application to multiple servers.

8.4 Presentation layer

The presentation layer implements the HMI functionality of the web client, providing operators with access to OPC UA servers. It can be used to manage OPC UA connections, browse address spaces, read and write attribute values, subscribe to data changes and events, and call server-defined OPC UA methods. The user interface has controls such as buttons, lists, and links, which upon user interaction send requests to the service layer. The user interface updates itself based on the response data it receives from the service layer, displaying an up-to-date representation of the servers it is connected to.

The web client implements the thin-server architecture. This means that aside from the initial page load, subsequent user interaction need only generate asynchronous requests for the changing data, without causing a full page reload. The changing data is retrieved either on demand with asynchronous HTTP requests or in real time by subscribing to server-sent push notifications.

One of the requirements set for the user interface was that it should adapt to different display sizes. The user interface achieves this by the use of CSS3 media queries, which allow specifying different style definitions for different display sizes. The use of Bootstrap framework since the beginning of the project made adding mobile support to the web client's user interface slightly easier, as its style definitions support responsive design principles out of the box.

At first, the web application was developed for desktop first. The main benefit that this approach brought was that it was easy to add new features, because the design was not limited by a small screen size. However, converting the desktop optimized user interface to mobile devices was found to be difficult, because of the significantly smaller display size of mobile devices. For this reason a mobile first approach was adopted later on in the development. The advantage of a mobile first approach was that it was much easier to adapt the mobile optimized layout to a desktop environment, than the other way around.

Using Bootstrap allowed to quickly build a relatively good looking user interface with little effort. The drawback of littering the HTML markup with Bootstrap specific classes was that the GUI library was soon tightly coupled with the application. Defining custom styles become more obscure, as it often required overriding existing styles defined by Boostrap. Another drawback was the lack of some more complex GUI components, such as horizontal and vertical splitters, tree views, and editable span elements. These deficiencies were corrected with the addition of external library dependencies, which on the other hand increased the overall size of the user interface, having a negative impact on mobile clients that need to download it from the server.

All of the evaluated frameworks could have been used to create a single-page application, so making the choice between them was not straightforward. Finally, the Angular.js framework was chosen because of its declarative syntax which extends HTML with new tags and attributes, a feature thought to be useful for the creation of reusable SCADA components.

8.4.1 Server connections

The web client enables a user to connect to an OPC UA server. The landing page presents an input field where the user can specify the server to connect to, by entering a standard OPC UA server URI, and clicking the connect button. Optionally the user may enter user credentials to be used when connecting to the server. The input field is automatically populated based on the configuration options of the web client.

The API for managing server connections is as follows:

```
/api/sessions
```

The service interface represents server connections as session resources. This is because a server connection is always associated with a session. A new connection is created by sending an HTTP POST request at the collection of session resources. Similarly, a client can disconnect from a server by sending an HTTP DELETE request.

8.4.2 Address space browser

The address space browser is used to view and navigate the address space of an OPC UA server. A tree view was chosen to visualize the server address space as it is a common and intuitive way to represent the mesh-like tree structure of OPC UA. The custom Angular Tree Control directive was used to provide a graphical representation for the tree view.

The web client uses the following API of the service layer to fetch the references of a node:

```
/api/sessions/{id}/nodes/{nodeId}/references
```

Here *id* is the session ID assigned by the service layer, and *nodeId* is the nodeId of the node for which references are to be fetched. Besides linking to itself, a reference representation contains links to its source and target nodes. In order to reduce round-trips to the service layer, the representations of both source and target nodes may also be embedded directly to the reference representation. The support for embedded representations was found to significantly improve the performance of the client. With embedded resources, fetching references for a node with five references was reduced to a single request, whereas in the old system six separate requests would have been required.

8.4.3 Node details

The details view is used to present detailed information about a selected node, including its attributes and references. The reference list has controls for filtering references by their type. Angular makes filtering data particularly easy by its built-in filters, which can be applied directly in HTML in conjunction with the ng-repeat directive. Figure A.1 shows a screenshot of the node details view with the root node being selected. The header bar allows for customization and features a label which indicates whether the client is currently connected to a server. The current version of the web client supports only one consecutive server connection, but future versions will allow for switching between active OPC UA sessions.

8.4.4 Subscriptions

The subscriptions view presents a tabular view of the current subscriptions and the monitored items associated with them. Subscriptions can be added either by dragging them to the subscriptions view, or by right clicking the specific node and choosing monitor. The controller of the subscriptions view listens for incoming data change events and updates the values of the subscribed nodes which have changed. The service buffers data change notifications and sends a notification every fixed interval. Therefore the notification may contain multiple value updates. Figure A.2 shows a screenshot of the subscriptions view with several monitored data items assigned to a subscription. The current version of the web client supports only one subscription, but future versions will allow the user to choose which subscription a monitored data item should be assigned to.

The web client uses the following API of the service layer to manage subscriptions:

```
/api/sessions/{id}/subscriptions
```

The web client has two views for displaying subscribed data values and events. These are the subscriptions view and the events view. The subscriptions view displays a list of all subscribed variables with their most recent values and timestamps indicating the time when the value was last updated. In addition, it also shows some additional information about the variable, such as its data type and display name. The following API is used to manage monitored data items:

```
/api/sessions/{id}/subscriptions/{subId}/dataitems
```

The events view differs from the subscriptions view by displaying incoming events as they are received from the service layer. A new event is appended to the end of the list regardless of whether an event from the same event source had already been received or not. The current version only shows the events as they are received. Future versions will also allow for filtering the list of events, and support for easily acknowledging the received events by using the standard OPC UA methods defined in the address space. The following API is used to manage monitored event items:

```
/api/sessions/{id}/subscriptions/{subId}/eventitems
```

8.4.5 Historical data

The history view displays value history read from the server in tabular and graph formats. Figure A.3 shows a screenshot of the history view. The amount of history data requested from the OPC UA server can be large. For example, the client may decide to fetch value history data of a counter variable from the last three months. In this case the requested data set is too large to fetch in one go. The solution is to divide the history read request into multiple steps by using continuation points. The service reads a specific number of data points and processes the result, after which it uses the continuation point returned by the server to carry on with the history read. The service uses push notifications to inform the client of its progress, and the client can update its visual representation as more data gets pushed to it.

The web client uses the following APIs of the service layer to issue history read requests. The first collection represents value history, while the second collection represents event history.

```
/api/sessions/{id}/nodes/{nodeId}/values
```

```
/api/sessions/{id}/nodes/{nodeId}/events
```

8.4.6 Server-defined methods

In order to remain RESTful, the service layer represents each method call made by the client as an individual resource. Issuing a GET request on the calls collection would conceptually return a list of all method calls ever issued since the startup of the server. This could be useful for accountability reasons, but is not implemented in the current version. The web client may call a server-defined method by issuing an HTTP POST request on the calls collection. The three parameters expected by the call service are passed as HTTP POST parameters. These are the NodeId of the method, the NodeId of the object on which the method should be called, and an array of input arguments.

The web client uses the following API of the service layer to call server-defined methods:

```
/api/sessions/{id}/nodes/{nodeId}/methods/{methodId}/calls
```

8.4.7 User interface components

Generic user interface components can speed up the development of SCADA HMIs. A generic design should not cater to one specific type of application area, but rather support different use cases, and thereby promote reuse across different projects. The web client provides reusable components in the form of Angular.js directives. An example component is the graph component of the history view, shown in Figure A.3, which is used for displaying trends of various kinds to the operator. The source code of the graph component is presented in Listing B.1.

8.5 Testing and debugging

In order to ensure the correct functioning of the application and reduce the chance for introducing new bugs through regression, it is important to test the application continuously. The application was tested mostly by running it in a web browser and trying different use cases manually. In addition, some unit tests were written with Karma and Jasmine. Karma is a test runner which allows to run tests written in JavaScript from the command line. Jasmine is a test framework which emphasizes specifying the expected behavior of the application.

Regardless of the amount of testing, bugs sometimes find their way into the source code, so a way to debug the code is needed. At the time of writing this thesis, at least Eclipse, Netbeans and WebStorm have added support for debugging Java applications that use Nashorn to embed JavaScript into Java. The debugging process is the same as for normal Java applications with the exception that the debugger stops at breakpoints assigned to embedded JavaScript files. The first attempt was to setup a development environment around Glassfish and the Avatar framework. A Glassfish project complemented by Avatar was created and the web application was deployed in it. In practice debugging the web application in this way turned out to be impossible since none of the IDEs included support for running Glassfish projects on Java 8. The other option for debugging JavaScript running on the Nashorn JavaScript engine was to use the debug module of Node.js. However, this module depends on the debugging capabilities of the V8 JavaScript engine, and is not supported by Avatar.

8.6 Security

The web client supports full end-to-end security. Client-server security between web browsers and the service layer is achieved by using HTTPS, while communication between the underlying system and OPC UA servers is secured with UA Secure Conversation.

One of the requirements was that the user interface should be decoupled from the web service. This requirement includes use cases where the user interface and service are deployed under different domain names. Nevertheless, modern browsers enforce the same-origin policy, which prevents resources with different origins, that is a combination of protocol, hostname and port, from accessing each other's DOM and communicating with XHR. The service layer supports Cross Origin Resource Sharing (CORS) which by setting specific HTTP headers enables it to specify which origins should be allowed access to the resource. Without CORS, it would not be possible to host the presentation layer and the service layer under different domains.

The web client uses a cookie-based authentication scheme. A session cookie is set when the user enters the web application for the first time. Each browser session has its own hash map for storing OPC UA sessions. A new OPC UA session is assigned to the hash map each time a user connects to an OPC UA server. Disconnecting from a server removes the session from the hash map. The hash map is indexed by a server ID local to the browser session.

8.7 Configuration

All the configuration options of the web client are present in a configuration file. The configuration file is a normal JSON formatted JavaScript file. One section of the configuration options is reserved for user interface specific settings such as the name of the application, and the various display options for the graph element. Another section is dedicated for the service configuration, having settings for configuring the service's location and a list of pre-defined OPC UA server endpoints. The configuration options are defined in their own module, providing a clean encapsulation and allowing to easily inject them in the modules that require them.

Chapter 9

Discussion

9.1 Summary and results

The general goal of this thesis was to evaluate the suitability of standard web technologies when developing monitoring applications for the Industrial Internet. The constraints were that the application should be deployed on the OPC UA platform and programmed by using the OPC UA Java SDK of Prosys PMS Ltd. In order to perform the evaluation, a generic OPC UA client application was developed. The results show that it is indeed possible to develop applications for the Industrial Internet using standard web technologies.

Section 9.2 of this chapter presents an evaluation of the finished implementation in terms of requirements met and a comparison to alternative solutions in the market, while also discussing problems faced during development, and how the finished implementation fits in within the Industrial Internet. Section 9.3 then answers the research questions, and Section 9.4 discusses alternative approaches that could have been taken. Finally, Section 9.5 discusses the potential future work.

9.2 Evaluation

9.2.1 Meeting the requirements

In Chapter 7, several requirements for the generic client implementation were specified. Fulfilling these requirements was then addressed in Section 8.

Five general requirements were identified: independence of external software components, separation between business and presentation logic, support for basic OPC UA features, end-to-end security, and configurability.

Independence of external software components was achieved by using only those standard web technologies that were natively supported on all target web browsers. The target web browsers chosen were Google Chrome and Mozilla Firefox. Later on it was discovered that Internet Explorer could also be supported by employing a polyfill for the server-sent events feature. The fact that no browser plugins or locally installed components were required demonstrates that rich Internet applications can be developed by depending only on standard web technologies. The client-server, service-oriented architecture of the web client ensured a proper separation of concerns, fulfilling the second requirement. Conformance to basic OPC UA client facets was facilitated by the use of the OPC UA Java SDK. In order to reduce development time, this requirement was changed to providing JavaScript wrappers to the Java interfaces of the SDK. The SDK has been officially certified against the test suite of OPC Foundation, so it is perfectly possible to develop fully conformant clients by using it. It is therefore reasonable to assume that the JavaScript API conform to the client facets to the extent that they replicate the Java API. End-to-end security of OPC UA was extended to web browsers by forcing web browsers accessing the presentation and service layers to use SSL. The configuration of the web client is based on JSON-formatted configuration files that are located at the server hosting the service. This approach works well for small deployments where the effort required to update configuration files is relative low, but a separate configuration backend could be more suitable to larger deployments, where central management of service configuration is desired.

Six requirements were specified for the service layer: modular design, client-server architecture, support for multiple simultaneous users, connectivity to OPC UA servers, serialization of OPC UA data, and the delivery of OPC UA events to the connected clients. The requirement for modularity was met both on the client-side and on the server-side. On the client-side the Angular.js framework encouraged a division of the source code into three main categories: controllers, services, and view directives. Essentially this enabled to define view components independently of their data sources defined in the services. The controllers, encapsulating the applications state within scope declarations, connected data sources to view components. Modularity on the client-side allowed to define data sources once and define those definitions across all controllers. Similarly, code on the server-side was organized into several modules, each with their own distinct responsibilities. For example, all OPC UA communication was performed in the JsClient object which encapsulated the client interface of the OPC UA Java SDK, exposing a JavaScript API. This JavaScript API was then used in the REST and SSE service modules in order to pass request data to OPC UA services and to send responses with the corresponding results of the service calls. The requirement for client-server architecture was met by using HTTP as the transport protocol. Moreover, the decision was made to try to conform to the REST architectural style because of the potential benefits listed in Chapter 4. The HTTP protocol turned out to map easily to OPC UA service calls. However, adhering to the RESTful style was not straightforward with OPC UA, and some of the problems that were encountered are described later in this chapter. For supporting multiple simultaneous users the thread module of Avatar framework was used. This module facilitated the creation of asynchronous API for the JsClient object. Another, better approach would have been to take advantage of the asynchronous method calls of the OPC UA Java SDK, and to provide an asynchronous interface that returns promise objects, or the possibility for configuring a callback. Connectivity to OPC UA servers was achieved by complying to the OPC UA client facets. The serialization of OPC UA data was performed by creating JavaScript wrapper classes for core classes of the OPC UA Java SDK. This way each JavaScript class could provide its own methods for serializing and deserializing between Java and JSON representations of the OPC UA data objects. Performing serialization manually instead of relying on a JSON serialization library such as GSON provided more flexibility and allowed to encapsulate the functionality within the respective JavaScript wrapper classes. Finally, the service layer meets the requirement for the sending of periodic push notifications. Using the SSE protocol proved to be sufficient for the purposes of the web client, with the exception of the problems caused by the HTTP request limit imposed by web browsers.

Finally, two requirements were identified for the presentation layer: responsive layout and reusable user interface components. The Bootstrap framework was used as a starting point for implementing the responsive user interface of the web client. Initially the web client was tested on the desktop. It was later discovered that a mobile-first approach would have been better as fitting all the user interface components into a smaller screen size proved to be difficult. With a mobile-first approach, the mobile-optimized interface could be later optimized for desktop use by using for example CSS media queries and a progressive enhancement methodology. The web client also features some sample web components developed as Angular.js directives. Aside from trivial examples, the syntax for declaring Angular.js directives was rather complex.

9.2.2 Comparison to existing solutions

This section compares the generic client implementation to the solution categories presented in Chapter 6. These categories included pre-rendered user interfaces represented by Groov, web APIs represented by HyperUA and native stacks represented by the JSUA framework. The solutions are compared based on available features and usability.

In relation to the evaluated solutions, the generic client implementation offers a basis for building web-based HMI applications, while also exposing a JSON-based web API for interacting with OPC UA servers. It offers an inexpensive way to expand existing OPC UA Java SDK based OPC UA applications with web capabilities.

Both the generic client and HyperUA expose a web-based API for accessing OPC UA servers. HyperUA is capable of exposing OPC UA services as RESTful services. Its API seems to conform to the hypermedia constraint of the REST architectural style. Resources are represented as HTML can be traversed by using standard CSS selectors. The web client takes a different approach by providing a JSON-based resource format. This format is inspired by the Hypertext Application Language (HAL) format and is capable of representing references between individual resources. Depending on the use case, the more traditional JSON responses may be easier to program against. However, the advantage of the API of HyperUA is that the same HTML5 API can be consumed both by a machine, and by a human. The machine uses standard CSS selectors to navigate the HTML5 response, while the human clicks the links in the HTML5 response. All in all, the REST design principles followed by HyperUA and the web client are likely to garner extra benefits when it comes to scaling the applications horizontally to multiple threads, CPU cores, and ultimately to the cloud.

Out of the evaluated solutions, Groov and HyperUA provide a graphical user interface that can be used in the web browser. While HyperUA only provides a simple HTML5 based presentation of its API, Groov comes with tools for designing and viewing HMI views in web browsers. This comparison focuses on Groov because it features plug-and-play HMI components, which are an essential feature of an HMI toolkit. Groov comes with a collection of user interface components which can arranged in the web-based editor to create an HMI that scales to displays of different sizes. The web client also supports reusable components, but it does not come with a designer application. A similar feature could however be implemented on top of the web client by allowing the user to drag-and-drop address space nodes on a dedicated canvas in the user interface of the web client. Like Groov, the web client does not depend on any external browser plugins or locally installed software, and in the same way as Groov it has been designed to work with various display sizes. Groov seems to rely heavily on scalable vector graphics (SVG) to ensure that graphics elements scale properly on different end-user devices. The web client does not use SVG but instead most graphical elements aside from the address space browser are defined using plain CSS. The advantage is that user interfaces can be more easily designed to work with both large and small display sizes by using CSS media queries.

All solutions employ HTTPS in order to ensure that proper end-to-end security is maintained between the web browser and the OPC UA servers. Both Groov and HyperUA additionally implement their own access control mechanisms based on configurable roles and permissions. The web client depends solely on OPC UA authentication. The advantage of an additional authentication layer is that permissions can be configured independent of the underlying OPC UA servers. For example, web client users may be denied access to a specific server, while still having the permission to access the servers locally on organization premises.

The advantage of the web client compared to Groov and HyperUA is that by developing the monitoring application against the OPC UA Java SDK interfaces, full control over the application design can be retained. This approach of developing directly against the OPC UA APIs means that applications can be more flexibly customized to conform to the case-specific requirements.

The generic client and the other solutions all require an intermediate proxy server for deployment, although in the case of the JSUA framework the web server is only needed for bypassing the same-origin policy restrictions enforced by most web browsers. On the other hand, securing the source code of the browser-based implementation might prove to be an issue in some cases. Moreover, the web server can be seen as an advantage as it provides a centralized aggregation point and allows the service to pre-process data before it sends it to the web browsers. Deployment consists of installing the server software on the target server hardware. Alternative, a pre-installed device ready to be installed in the factory network can be provided, as is already done by Groov. Since the generic client was developed in Java it can be easily deployed on a variety of platforms, including embedded systems.

Among the evaluated solutions Groov is the only one which does not require any programming. Instead, HMI can be developed by merely drag-and-dropping user interface components, and graphically connecting them to the relevant data sources. The drawback is that it is more difficult to customize the HMI to meet specific requirements. HyperUA on the other hand provides the necessary APIs to integrate OPC UA data to the HMI but does not itself come with tools that facilitate HMI development. The web client strikes a balance between these two approaches by on one hand providing the developers with programmatic interfaces to access OPC UA data, an on the other hand defining simple conventions for the implementation of reusable user interface components. The web client does so by depending on the OPC UA Java SDK and the Nashorn JavaScript engine of Java 8 to provide both Java and JavaScript APIs to OPC UA, along with a JSON-based web API, accessible by all web-capable devices. The web client is therefore not restricted to plain JavaScript such as is the case with the JSUA framework. It also provides higher level OPC UA abstractions, likely resulting in fewer number of source lines of code. The downside of the web client is that it requires more programming work to implement HMIs. Moreover, the level of skill required may be higher, as the programmer is required to have knowledge of both Java and JavaScript, whereas in HyperUA only CSS selectors known by web designers are required, and in JSUA only JavaScript knowledge is required.

9.2.3 Problems with the implementation

This section discusses some problems that were faced during the development of the web client, with the purpose of outlining some challenges that can be encountered when developing web-based monitoring applications using the proposed approach. Along with the problem statements, some potential solutions are discussed.

The first problem was related to the mixing of Java and JavaScript code when developing the web client. It was quickly noted that the mixing of Java and JavaScript code made it more difficult to reason about the flow of the program execution. Essentially, the solution was to encapsulate all code using the OPC UA Java SDK in its own JavaScript wrapper modules, which were then referenced and used in plain JavaScript modules. This separation has the added benefit of allowing to reuse the JavaScript wrappers also in other projects.

The second problem was that the Avatar framework deployed the service layer written in JavaScript as a Java servlet which had to be re-compiled in order to see the changes reflected in the web browser. Therefore the benefit of not having to compile JavaScript when testing changes to the service layer was lost, which made iterative development more difficult.

The third problem was that the Server-Sent Events used to push event notifications from the server to the web browsers does not support running the web application in multiple tabs of the web browser at the same time. Currently, a new EventSource object is assigned to each individual event the server might generate. The number of consecutively allowed HTTP connections to the same server, governed by the HTTP protocol, is thus quickly exceeded. This problem was partly solved by sharing the same EventSource between server events, but multiple tabs could still open too many HTTP connections. A better solution, suggested by the protocol specification [37], would be to either create a single shared worker instance for the browser tabs and use it to share the same EventSource between browser tabs. Alternatively, the SSE protocol could be replaced with a protocol such as WebSocket, which in most modern browsers has a separate, typically much higher connection limit than standard HTTP. This however would add unneeded complexity to the web client, as high performance two-way communication is not strictly required by the current implementation. An alternative, WebSocket based implementation, could be considered in the future if a higher performance implementation were to be deemed necessary.

The fourth problem was associated with the performance of the web client. There was a performance bottleneck in the web client that was found in the round-trip times between the client and the service. The measurements performed using the Chrome developer tools reported round-trip times of several seconds depending on the size of the request. This is likely due to the implementation of the handling logic which upon loading the references of a node also loads the referenced nodes. Thus, the overall performance could be improved by separating the loading of references from the loading of the node objects. Another way to increase the performance of browsing the address space would be to pre-emptively cache frequently requested nodes on the service side. In relation to the round-trip times the rendering times were found to be negligible, averaging around 1 millisecond per requested node. Nevertheless, the overall performance could be improved also on the client-side by performing client-side caching of the JSON serialized nodes, references and subscriptions to the local storage of the web browser. Client-side caching could speed up the initial loading of the user interface because seldom changing top level nodes could be loaded from cache. Storing application state locally would also enable the user to browse the address space in offline mode. Operations such as the creation of new nodes in the address space could be stored and executed when a connection to the Internet becomes available. It would also allow the user to continue from a previous state after closing the web browser.

The fourth problem was associated with debugging the application. The dynamic nature of JavaScript made debugging the application more difficult, because the IDE was not able provide as much support and error checking, compared to for example the strongly typed Java language. One way to ease the development of JavaScript applications that rely on strongly typed Java classes, would be to use an intermediate, typed language that compiles to

JavaScript. Such languages include for example TypeScript, CoffeeScript, and Flow.

The final, most fundamental problem with the chosen approach was associated with the scalability of the web client. The web client maps browser sessions to OPC UA sessions. OPC UA sessions are stateful, so the server has to maintain them across requests. As described in Section 4.3, storing client sessions on the server-side leads into scalability issues, as sessions need to be shared between service instances. Due to this constraint, the web client is incapable of supporting multiple service instances, let alone multiple service threads executing on the same host.

There are a few ways to solve this issue. First, the load balancer can be configured to always forward requests with a specific session ID to the same server. The problem with this approach is that it does not evenly distribute the load between different servers. Moreover, this setup alone would not allow multiple service threads to run on the same server, as each would need access to the same session state. The second approach is to implement synchronization of browser sessions between servers. The drawback of this approach is that the synchronization would create extra overhead and it would be difficult to ensure session consistency between servers. The third approach would be to move the browser sessions from each individual web server to a shared database. This approach would eliminate the need to perform synchronization between web servers, but the database server could turn into a performance bottleneck if the number of web servers became high. It would also add an extra layer to the architecture of the service, thus making it more complex and increasing round-trip times between web browsers and OPC UA servers. Finally, in some cases could be possible to reuse OPC UA sessions. In this model a pool of OPC UA sessions shared by all browser sessions would be created for each server. In practice this approach would be the most difficult one to implement because of the need to control concurrent access to shared session resources. It would also not be possible to rely on the OPC UA session for application state, because the OPC UA session used to communicate with the server could be different across subsequent requests. OPC UA authentication would also not be possible because anonymous connections would be required.

In practice, due to the stateful nature of OPC UA, it is difficult to adhere to the REST architectural principle of not maintaining client state on the server-side. In this regard the server-side JavaScript frameworks lose in comparison to the purely client-side JavaScript implementations, capable of maintaining direct OPC UA connections between the web browser and OPC UA servers.

9.2.4 Web technologies and the Industrial Internet

When choosing the right tools and frameworks for implementing an industrial web application, it is highly desirable that the chosen frameworks will continue to be supported in the future. This was also an important criterion for the web client. However, the choice between different tools and frameworks is made more difficult by the rapid pace of change within web development libraries. The two main frameworks chosen to implement the web client, Avatar.js and Angular.js, were chosen based largely on the fact that large companies developed them. It was presumed that these frameworks would be well supported in the future. Unfortunately this presumption proved to be wrong with Angular.js announcing a new version of the framework without backward compatibility, and Avatar.js being later discontinued by Oracle. Therefore it is important that emphasis is placed on choosing future proof technologies when developing web applications for the Industrial Internet. This is especially true for software projects which have a long life-span and require long-term support, such as industrial automation systems.

The recent efforts in the standardization of new web technologies show that the way modern web applications are developed today is slowly starting to converge with the way native desktop applications are developed. The last years have seen a vast increase in client-side frameworks and software development methods and tools originally conceived for native desktop programming are slowly being introduced to the web world. Tools such as task runners, build systems, package managers, source code verifiers, unit testing frameworks, continuous integration systems, profiles, debuggers, optimizers, and minifiers are helping developers to produce code of better quality more efficiently. Web technologies are even used to power traditional desktop applications. For example, the JavaFX Scene Builder allows to define customized user interfaces with the use of CSS. A project called node-webkit allows to create native desktop applications by using HTML, JavaScript and CSS. More and more focus is also put into enabling desktop like features on the web browser. For example, cutting edge technologies such as raw sockets, native data binding, web components, WebGL, and WebRTC are already enabling developers to write native-like applications for the web browser.

9.3 Answering research questions

The first question, "What are the requirements for the generic OPC UA web client?" was answered in Chapter 7, which listed the functional and non-functional requirements for the web client. An important requirement was that the client should be generic enough to be easily reusable in different use cases.

The second question, “How can the generic OPC UA web client be implemented?”, was answered in detail in Chapter 8. The web client was implemented by using a service oriented client-server architecture, where a REST-like HTTP interface allows multiple concurrent clients to use the same uniform interface for interacting with OPC UA servers. The integration with OPC UA Java SDK was facilitated by the use of the Nashorn JavaScript engine which allowed to access all the capabilities of the SDK from JavaScript.

The third question, “How does the generic OPC UA web client compare to other solutions?”, was answered in the previous section. Due to its flexible nature, the proposed architecture compares well against the other solutions, with the benefits of an industry proven SDK and full control over the whole application stack.

This chapter evaluated the web client in relation to the other solutions, discussing both its advantages and shortcoming. To answer to the final question, “Are standard web technologies compatible with the Industrial Internet?”, the finished application shows that modern web technologies provide developers with sufficient tools to develop efficient, feature-rich and usable applications for the Industrial Internet.

9.4 Alternative approaches

This section discusses some alternative approaches that could have been taken when developing the web client.

First, it could have been wiser to use the plain Avatar.js library instead of project Avatar. The advantages of Avatar, easy creation of service endpoints coupled with integration with the JMS, quickly eroded as compared to the difficulties that a service container layer brought to the development of the application. For example, deploying a simple debugging environment proved to be difficult as the service would not run at all without a full Java EE application server. Furthermore, the only supported deployment platform was Glassfish, and choosing a heavy weight application server for one light weight application instance could be difficult to justify to customers. One important feature of Project Avatar was the capability of spawning Java worker threads from JavaScript code. This was facilitated by the utility functions provided by the framework, which return promises. The threading module greatly simplified the code of the service layer as it was possible to use straightly the synchronous versions of the OPC UA SDK method calls. On the other one could argue that using multiple threads in a Node.js application beats the purpose of using Node.js in the first place, and that standard JavaEE servlet technologies would fit the purpose better.

In addition to the Avatar framework, several other JVM-based Node.js platforms were discussed in this thesis. The most obvious alternative approach would have been to use only the Avatar.js component of the Avatar framework. The main benefit of this approach would have been that it would have removed an extra layer from the implementation. Thus it can be argued that debugging the application could have become easier, as the implementation would have been simpler.

Second, an entirely different approach would have been to use the polyglot Vert.x framework, and altogether abandon the idea of integrating the SDK classes straightly with the application code written in JavaScript. Instead, a separate Java verticle containing the SDK would have been implemented, alongside with a client verticle written in JavaScript, or any other language supported by Vert.x. There is even a project called Nodyn which aims to implement the Node.js API on top of Vert.x, allowing applications to leverage the Vert.x APIs and run existing Node.js scripts on top of Vert.x. This approach presents a direct replacement for Avatar.js. It could also be worth exploring projects such as node-java, which is a Node.js module that adds support for invoking Java methods from Node.js applications. In order to test the portability of the service code, it could also be tested with a different Java-based Node.js API implementation such as Nodyn.

9.5 Future work

Due to the emergence of the Industrial Internet, web-based industrial monitoring applications have increasing potential in the future. Industrial Internet is often characterized by the increasing connectivity between industrial devices and people operating them. Consequently operators need better tools for monitoring and controlling the devices, and these tools must be usable on-the-go with normal end-user devices such as mobile phones, tablets, and laptops. This section outlines some potential future work associated with the web client implemented as part of this thesis.

First, the view component that was developed was only a demonstration of what can be possible when the declarative syntax of HTML, custom DOM elements and Angular.js directives were mixed together. Future versions could leverage this technology in order to provide more complex UI components, such as gauges and thermometers, which the user can drag-and-drop on the HMI. The upcoming standards for reusable web components [66] could be adopted to create future-proof components which are compatible with most frontend frameworks.

Continuing with reusable components, the thesis by Boström [8] discussed the implementation of custom and re-usable JavaFX user interface components for the OPC UA product line. These components could potentially be used also in the web client by running them in the embedded execution mode of JavaFX. The web application could then communicate with the components by using a JavaScript API provided by JavaFX. Similarly, JavaFX provides embedded applications with access to the host web application. Developing web-ready components using JavaFX could thus be an interesting subject for future work.

Second, one key theme in the shift to the Industrial Internet is that machines are becoming increasingly connected to each other through both physical and wireless links, more intelligent through data mining, machine learning and advanced heuristics, and more capable of independent decision making. Industrial plants however generate huge amounts of data and higher levels of semantics need to be assigned to the data in order to better analyze, understand, and leverage it to meet various business needs. Adding support for a linked data format such as the JSON for Linking Data (JSON-LD) could thus be a subject for future work.

Third, the HTML5 standard offers many new technologies which could be used to add support for offline usage to the web client and are thus worth exploring. One of these technologies is local storage, which allows browsers to store data locally, either across requests, or also across subsequent browser sessions. This way the web client could maintain its state even after it has been closed and reopened. The local storage could also be used for caching references to interesting OPC UA nodes. For example, the address space could maintain the state of opened folders and selected nodes.

Fourth, in the future, the web client should be expanded to support more OPC UA features. Some examples for missing features, or features that need more work, include parametric history reads, event handling, and server discovery.

Fifth, the service layer makes it possible to reference individual nodes of the address space by their node ID. However, unique URI references are supported only on the service layer, and there is currently no way to refer to specific application states in the browser address bar. In the future this could be solved by taking advantage of the HTML5 History API.

Sixth, the performance issues related to the address space browser should be addressed in future versions. The possible remedies include on one hand the partitioning of large requests to a number of smaller requests in order to reduce round-trip times, and on the other hand replacing the dirty-checking Angular.js views with for example React views in order to minimize rendering times.

Finally, the web client could be turned into a native mobile application by using for a example Phonegap. Alternatively, a user interface consuming the same service API as the web client could be developed by using for example the Ionic framework.

Chapter 10

Conclusions

This thesis set out to evaluate the applicability of web technologies to the development of monitoring applications for the Industrial Internet. The evaluation was carried out by implementing a generic web client application for industrial monitoring and control, and comparing it to existing monitoring solutions on the market. The OPC UA standard for industrial data transfer and information modeling was used as the target platform for the application.

The finished application consists of two parts, a service layer and a presentation layer. The service layer exposes stateful OPC UA services as stateless HTTP services conforming to the REST architectural style, and the presentation layer consumes this service in order to provide a graphical human-machine interface (HMI) for plant operators. The analysis part of the thesis brings up some key issues in the development of web-based OPC UA applications.

The first finding was that the stateful OPC UA service calls cannot be easily mapped to the stateless web resources of the REST architectural style. The reason for this is that the service has to maintain connection to OPC UA servers across HTTP requests, and client sessions are tied to the OPC UA sessions. Several ways to circumvent this problems were discussed in the thesis, including the implementation of a shared pool of OPC UA connections. Solving this problem could increase the scalability, because new service instances could be deployed independent of one another.

The second finding also related to service scalability was that the hypermedia constraint of REST could be applied also to HTTP services that expose OPC UA services, and that at least one such system has already found its way to the market. The benefit of passing application state in web requests is that the server can be scaled out more easily, as new service instances can be added without concern for sharing the client sessions between them. In short, according to the hypermedia constraint of REST, the request itself should embody all the information that the server needs to process it.

The third finding was that asynchronous API are useful because they allow to handle requests quickly, allowing for a better scalability to a large number of users. Therefore all I/O handling should be performed asynchronously and long taking synchronous processing should be avoided.

The fourth finding was that the new web specifications for reusable web components and server-sent push notifications are among the key technologies enabling a new kind of web-based industrial monitoring applications to emerge. The difference between what is currently possibly only with native desktop applications as compared to web applications is actively being narrowed by the web community.

The fifth finding was that the service layer can benefit from a polyglot application architecture, where multiple languages are tightly integrated, capable of utilizing the libraries and language features of one another. Essentially this is true when pairing languages that are sufficiently different with each other. In this thesis, JavaScript wrappers were developed for the client API of the OPC UA Java SDK. The experience was that in the absence of compile-time type restrictions, features could be implemented more easily and more quickly. However, developing the application code comes with a trade-off between easier debugging through type checking, and faster iterations through less stringent restrictions of a dynamic language.

The sixth and final finding was that the enormous number of different tools and frameworks for the JavaScript programming calls for a careful consideration when it comes to bootstrapping a new web application project. The great number of tools and frameworks also means that most of the new projects will no longer be used in the years to come. Choosing the wrong tools for the job can become costly later on if the maintainer of the tool decides to pull support for it.

The results of this thesis serve as a groundwork for further investigations on web applications for the Industrial Internet. Future versions of the web client will extend the support for OPC UA client features. In the future the web client may be deployed in a cloud in order to provide an aggregating point for multiple process controllers. The data residing on the OPC UA servers could then be easily accessed by using the service layer, and the presentation layer could be deployed for debugging purposes.

Bibliography

[1] Ashkenas, J. Backbone.js, 2015. http://backbonejs.org/. Accessed 6.2.2015.

[2] Atvise - HMI and SCADA in pure web technology, 2014. http://www.atvise.com/. Accessed 12.11.2014.

[3] Avatar.js, 2014. https://avatar-js.java.net/. Accessed 7.1.2014.

[4] Bidelman, E. Custom elements, Dec 2013. http://www.html5rocks.com/en/tutorials/webcomponents/customelements/. Accessed 2.2.2015.

[5] Bidelman, E. Html imports, Dec 2013. http://www.html5rocks.com/en/tutorials/webcomponents/imports/. Accessed 2.2.2015.

[6] Bidelman, E. HTML's new template tag - standardizing client-side templating, Dec 2013. http://www.html5rocks.com/en/tutorials/webcomponents/template/. Accessed 2.2.2015.

[7] Bidelman, E. Shadow DOM 101, Dec 2013. http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom/. Accessed 2.2.2015.

[8] Bjarne, B. JavaFX based OPC UA simulation server. Master's thesis, Aalto University, Espoo, Finland, 2014.

[9] Burns, E. Why another MVC?, 2014. http://www.oracle.com/technetwork/articles/java/mvc-2280472.html. Accessed 22.1.2015.

[10] Cao, L., and Ramesh, B. Agile requirements engineering practices: An empirical study. *Software, IEEE 25*, 1 (Jan 2008), 60–67.

[11] Casteleyn, S., Garrig'os, I., and Maz'on, J.-N. Ten years of rich internet applications: A systematic mapping study, and beyond. *ACM Trans. Web 8*, 3 (jul 2014), 18:1–18:46.

[12] Crockford, D. *JavaScript: The Good Parts.* O'Reilly Media, Inc., 2008.

[13] Delabassee, David. Project Avatar: Server side JavaScript on the JVM. Slides of a talk given at QCon London, http://bit.ly/1MJTt7j. Accessed 7.1.2014, mar 2014.

[14] Downey, T. *Guide to Web Development with Java.* Springer London, 2012.

[15] ECMAScript Language Specification. ECMA-262 5.1, Ecma International, Geneva, Switzerland, jun 2011. http://www.ecma-international.org/ecma-262/5.1/. Accessed 3.4.2015.

[16] Enterprise-control system integration. ANSI/ISA 95, The International Society of Automation, Durham, NC, USA, 2013.

[17] Ergonomic design of control centres – part 5: Displays and controls. ISO 11064, International Organization for Standardization, Geneva, Switzerland, 2008.

[18] Estep, E. Mobile HTML5: Efficiency and performance of WebSockets and Server-Sent Events. Master's thesis, Aalto University, Espoo, Finland, 2013.

[19] Evans, P. C., and Annunziata, M. Industrial internet: Pushing the boundaries of minds and machines. General Electric White Paper.

[20] Ferner, Joe. Node-Java: Bridge API to connect with existing Java APIs, 2015. https://github.com/joeferner/node-java. Accessed 7.3.2015.

[21] Fette, I., and Melnikov, A. The WebSocket protocol. Standards track, Dec 2011. https://tools.ietf.org/html/rfc6455. Accessed 22.1.2015.

[22] Fielding, R., and Reschke, J. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. Standards track, Jun 2014. https://tools.ietf.org/html/rfc7230. Accessed 3.4.2015.

[23] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, 2000. AAI9980887.

[24] Flux: Application architecture for building user interfaces, 2015. http://facebook.github.io/flux/. Accessed 7.3.2015.

[25] Frejborg, A., Ojala, M., Palonen, O., and Aro, J. OPC UA connects your systems - top 10 reasons why to choose OPC UA over OPC. In *Finnish Society of Automation, Biannual seminar, 2013* (2013).

[26] Freund, M., Martin, C., Braune, A., and Steinkrauss, U. JSUA - an OPC UA JavaScript framework. In *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on* (Sept 2013), pp. 1–4.

[27] Galloway, B., and Hancke, G. Introduction to industrial control networks. *Communications Surveys Tutorials, IEEE 15*, 2 (Second 2013), 860–880.

[28] Glazkov, D. Custom elements. Editor's draft, W3C, Dec 2014. http://www.w3.org/TR/custom-elements/. Accessed 3.4.2015.

[29] Glazkov, D., and Ito, H. Shadow DOM. Working draft, W3C, Jun 2014. http://www.w3.org/TR/shadow-dom/. Accessed 3.4.2015.

[30] Glazkov, D., and Morrita, H. HTML imports. Working draft, W3C, Mar 2014. http://www.w3.org/TR/html-imports/. Accessed 3.4.2015.

[31] Groov, 2014. http://groov.com/. Accessed 12.11.2014.

[32] Heimbürger, H., Markkanen, P., Norros, L., Paunonen, H., Savioja, P., Sundquist, M., and Tommila, T. *Valvomo- suunnittelun periaatteet ja käytännöt.* Suomen automaatioseura. Helsinki., 2010.

[33] Heitkötter, H., Hanschke, S., and Majchrzak, T. A. Comparing cross-platform development approaches for mobile applications. In *WEBIST 2012 - Proceedings of the 8th International Conference on Web Information Systems and Technologies, Porto, Portugal, 18 - 21 April, 2012* (2012), pp. 299–311.

[34] Hennig, S., Braune, A., and Damm, M. JasUA: A JavaScript stack enabling web browsers to support OPC Unified Architecture's binary mapping natively. In *ETFA* (2010), IEEE, pp. 1–4.

[35] Hevery, M., and Abrons, A. Angularjs - Superheroic JavaScript MVW framework, 2015. https://angularjs.org/. Accessed 6.2.2015.

[36] Hickson, I. The WebSocket API. Candidate recommendation, W3C, Sep 2012. http://www.w3.org/TR/websockets/. Accessed 3.4.2015.

[37] Hickson, I. Server-Sent Events. Proposed recommendation, W3C, Dec 2014. http://www.w3.org/TR/eventsource/. Accessed 3.4.2015.

[38] Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Doyle, E., O'Connor, E., and Pfeiffer, S. HTML5 - a vocabulary and associated APIs for HTML and XHTML. Recommendation, W3C, Oct 2014. http://www.w3.org/TR/html5/. Accessed 3.4.2015.

[39] Hiltunen, T. Java based OPC UA client development. Master's thesis, Helsinki University of Technology, Espoo, Finland, 2012.

[40] Hollifield, B., Oliver, D., Nimmo, I., and Habibi, E. *The High Performance HMI Handbook*. PAS, 2008.

[41] Horstmann, C. S. *Java SE 8 for the really impatient*, 1st ed. Addison-Wesley, 2014.

[42] HyperUA - Projexsys, 2014. http://projexsys.com/hyperua/. Accessed 14.8.2014.

[43] Juhanko, Jari and Jurvansuu, Marko and Ahlqvist, Toni and Ailisto, Heikki and Alahuhta, Petteri and Collin, Jari and Halen, Marco and Heikkilä, Tapio and Kortelainen, Helena and Mäntylä, Martti and Seppälä, Timo and Sallinen, Mikko and Simons, Magnus and Tuominen, Anu. Suomalainen teollinen internet – haasteesta mahdollisuudeksi: Taustoittava kooste, 2015. http://pub.etla.fi/ETLA-Raportit-Reports-42.pdf. Accessed 8.1.2015.

[44] Katz, Y., and Dale, T. Ember.js - a framework for creating ambitious web applications, 2015. http://emberjs.com/. Accessed 6.2.2015.

[45] Mahnke, W., Leitner, S.-H., and Damm, M. *OPC Unified Architecture*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[46] Mozilla. *JavaScript Guide*, 2015. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide. Accessed 23.1.2015.

[47] Node.js, 2015. http://nodejs.org/. Accessed 25.1.2015.

[48] Nodyn: Node.js API for the JVM, 2015. http://nodyn.io/. Accessed 7.3.2015.

[49] OPC Foundation. *OPC Unified Architecture Specification, Part 2: Security Model*, aug 2012. Version 1.02.

[50] OPC Foundation. *OPC Unified Architecture Specification, Part 3: Address Space Model*, aug 2012. Version 1.02.

[51] OPC Foundation. *OPC Unified Architecture Specification, Part 4: Services*, aug 2012. Version 1.02.

[52] OPC Foundation. *OPC Unified Architecture Specification, Part 6: Mappings*, aug 2012. Version 1.02.

[53] OPC Foundation. *OPC Unified Architecture Specification, Part 7: Profiles*, aug 2012. Version 1.02.

[54] Oracle. *Java Platform, Enterprise Edition (Java EE) Specification, version 7*, 2013. http://download.oracle.com/otndocs/jcp/java_ee-7-fr-eval-spec/index.html. Accessed 8.12.2014.

[55] Oracle. *Java Servlet Specification, version 3.1*, 2013. http://download.oracle.com/otndocs/jcp/servlet-3_1-fr-eval-spec/index.html. Accessed 8.12.2014.

[56] Oracle. *JSR 366: Request for Java Platform, Enterprise Edition 8 (Java EE 8) Specification*, 2014. https://jcp.org/en/jsr/detail?id=366. Accessed 3.12.2014.

[57] Project Avatar, 2013. https://avatar.java.net. Accessed 7.1.2014.

[58] Prosys OPC UA Java SDK - Prosys OPC, 2014. http://prosysopc.com/products/opc-ua-java-sdk/. Accessed 24.3.2014.

[59] React: A JavaScript library for building user interfaces, 2015. http://facebook.github.io/react/. Accessed 7.3.2015.

[60] Rossignon, Etienne. Node OPC UA, 2014. https://github.com/erossignon/node-opcua. Accessed 25.1.2015.

[61] Teixeira, P. *Professional Node.js: Building Javascript Based Scalable Software*, 1st ed. Wrox Press Ltd., Birmingham, UK, 2012.

[62] Teliö, A. Implementing configurable browser-based SCADA system for OPC UA data. Master's thesis, Aalto University, Espoo, Finland, 2013.

[63] TIOBE programming community index, 2014. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Accessed 31.3.2014.

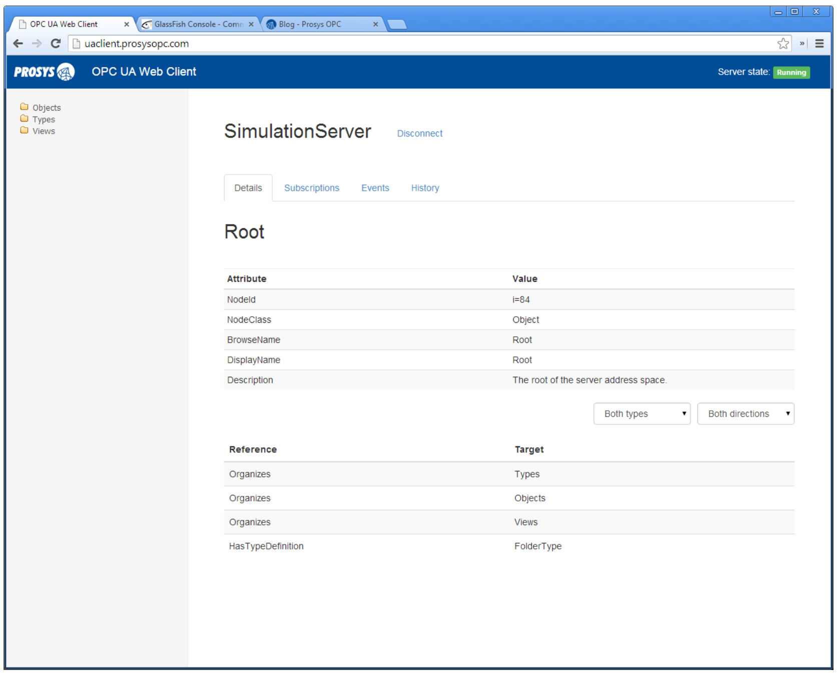
[64] Vert.x, 2015. http://vertx.io/. Accessed 7.3.2015.

[65] Web developer's guide to prerendering in Chrome, 2012. https://developers.google.com/chrome/whitepapers/prerender. Accessed 30.11.2014.

[66] Webcomponents.org, 2015. http://webcomponents.org/. Accessed 15.1.2015.

Appendix A

Screenshots

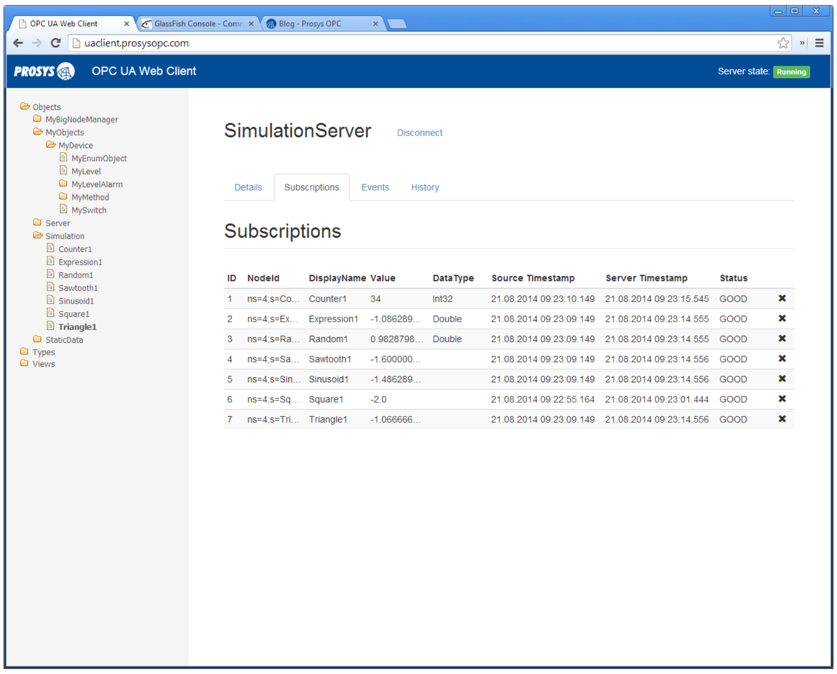


Root

Attribute	Value
NodeId	i=84
NodeClass	Object
BrowseName	Root
DisplayName	Root
Description	The root of the server address space.

Reference	Target
Organizes	Types
Organizes	Objects
Organizes	Views
HasTypeDefinition	FolderType

Figure A.1: Screenshot of the node details view.



SimulationServer

Subscriptions

ID	NodeId	DisplayName	Value	DataType	Source Timestamp	Server Timestamp	Status	
1	ns=4;s=Co...	Counter1	34	Int32	21.08.2014 09:23:10.149	21.08.2014 09:23:15.545	GOOD	✖
2	ns=4;s=Ex...	Expression1	-1.086289...	Double	21.08.2014 09:23:09.149	21.08.2014 09:23:14.555	GOOD	✖
3	ns=4;s=Ra...	Random1	0.9828798...	Double	21.08.2014 09:23:09.149	21.08.2014 09:23:14.555	GOOD	✖
4	ns=4;s=Sa...	Sawtooth1	-1.600000...		21.08.2014 09:23:09.149	21.08.2014 09:23:14.556	GOOD	✖
5	ns=4;s=Sin...	Sinusoid1	-1.486289...		21.08.2014 09:23:09.149	21.08.2014 09:23:14.556	GOOD	✖
6	ns=4;s=Sq...	Square1	-2.0		21.08.2014 09:22:55.164	21.08.2014 09:23:01.444	GOOD	✖
7	ns=4;s=Tri...	Triangle1	-1.066666...		21.08.2014 09:23:09.149	21.08.2014 09:23:14.556	GOOD	✖

Figure A.2: Screenshot of the subscriptions view.

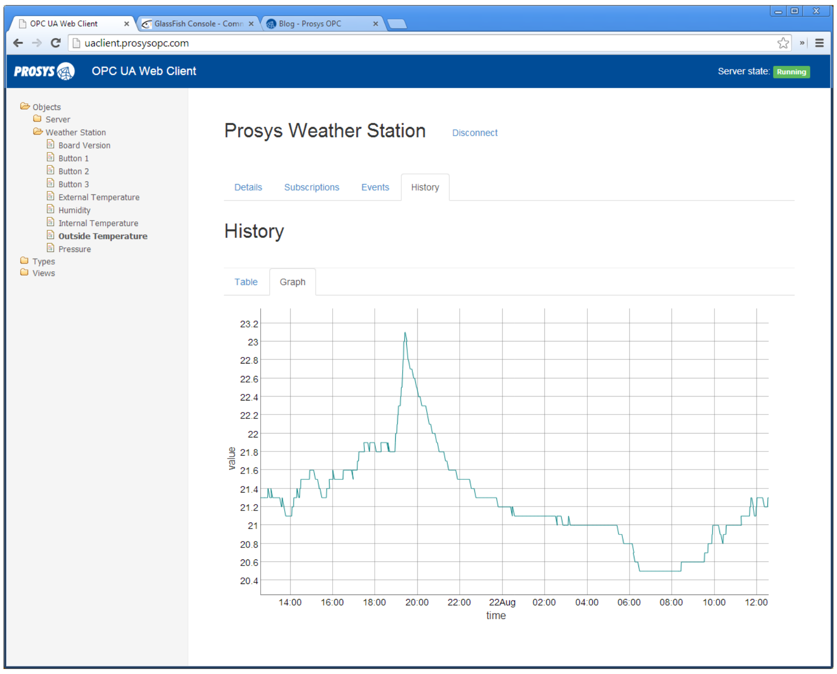


Figure A.3: Screenshot of the history view.

Appendix B

Source code

```
/**
 * A graph component for Angular.js.
 * Uses the Dygraphs charting library.
 */
uaclientDirectives.directive('uaGraph', [function() {
    return {
        restrict: 'E',
        scope: { data: '=', opts: '=', active: '=' },
        template: "<div class='graphComponent'></div>",
        link: function(scope, elem, attrs) {
            var graphData;
            var graph;
            scope.$watch('data', function(points) {
                if (points.length == 0) return;
                if (typeof graph === 'undefined')
                    graph = new Dygraph(elem.children()[0],
                        scope.data, scope.opts);
                graphData = points.map(function(point) {
                    return [new Date(point[0]), point[1]];
                });
                graph.updateOptions({'file': graphData});
                graph.resize();
            });
            scope.$watch('active', function(value) {
                if (value === true)
                    graph.resize();
            });
        }
    };
}]);
```

Listing B.1: The graph component used in the history view.

```javascript
/**
 * A gauge component for Angular.js.
 * Uses the C3.js charting library.
 */
uaclientDirectives.directive('uaGauge', [function() {
    return {
        restrict: 'E',
        scope: { data: '=' },
        template: "<div class='gaugeComponent'></div>",
        link: function(scope, elem, attrs) {
            var gauge = c3.generate({
                bindto: '#'+elem.attr('id'),
                data: {
                    columns: [
                        ['data', scope.value]
                    ],
                    type: 'gauge'
                },
                color: {
                    pattern: [
                        '#FF0000', '#F97600',
                        '#F6C600', '#60B044'
                    ],
                    threshold: {
                        unit: 'value',
                        values: [30, 60, 90, 100]
                    }
                },
                size: {
                    height: 128
                }
            });
            scope.$watch('data', function(value) {
                chart.load({
                    columns: [['data', value]]
                });
            });
        }
    }
}]);
```

Listing B.2: A gauge component.