Karlsruhe Institute of Technology

Design of a Linked Data-enabled Microservice Platform for the Industrial Internet of Things

Bachelor Thesis
by

Ferdinand Mütsch

Chair of Pervasive Computing Systems/TECO
Institute of Telematics
Department of Informatics

First Reviewer:	Prof. Dr. Michael Beigl
Second Reviewer:	Prof. Dr. Hannes Hartenstein
Advisor:	Andrei Miclaus

01/05/2016 – 31/10/2016

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, den 31. Oktober 2016

Zusammenfassung

Während der aktuelle Trend in Richtung hochgradig digitalisierter Smart Factories für die Fertigungsindustrie beträchtliches Potential zur Steigerung von Leistungsfähigkeit, Flexibilität und Produktivität birgt, tun sich Unternehmen im Allgemeinen noch immer schwer, entsprechende Technologie einzuführen. Ein Kernproblem ist der Mangel an einheitlichen, standardisierten Lösungen, die auf dem Hallenboden ohne spezifisches Expertenwissen und einen hohen Zeit- und Kostenaufwand integriert werden können.

Im Hinblick darauf präsentiert diese Arbeit sowohl Architektur, als auch konkrete Implementierung einer Internet of Things Softwareplattform mit Fokus auf technologische Einheitlichkeit und unkomplizierte Integration und Benutzung. Als Richtlinie hierfür wird in Kooperation mit Industriepartnern ein praxisnaher Anwendungsfall erarbeitet. Des Weiteren wird präsentiert, wie universelle Webtechnologie gewinnbringend mit neusten Software-Design Trends, mächtigen Techniken der Maschine-zu-Maschine Interaktion und allgemein verständlichen Konzepten im Bereich User Experience kombiniert werden kann. Dabei wird ausführlich auf Struktur der Software, Möglichkeiten zur Echtzeitkommunikation und Machine-zu-Maschine Interaktion, sowie auf einheitliche Datenintegration und Benutzerfreundlichkeit eingegangen. Am Ende des Prozesses steht eine fertige Softwarelösung als Proof-of-Concept, sowie eine Sammlung von Vorschlägen und Best Practices zur Integration von smarter Technologie auf dem Hallenboden. In einer abschließenden Evaluierung wird die Leistungsfähigkeit und Tauglichkeit der Softwareplattform für bestimmte praktische Anwendungsfälle untersucht. Zudem werden abschließend noch offene Fragen und weiterhin benötigte Entwicklungsschritte bis hin zu einem fertigen Produkt aufgezeigt.

Abstract

While recent trends towards highly digitized, smart factories entail substantial chances for manufacturing companies to boost their performance, flexibility and productivity, the industry commonly struggles to adopt suitable technology. One major problem is a lack of uniform, standardized solutions, which could be integrated to the shop floor without the necessity of highly specialized technical expert knowledge and a large amount of planning and restructuring.

Addressing that problem, this work proposes an architecture design as well as a concrete implementation of a Internet Of Things software platform, which mainly focuses on technological uniformity and ease of adoption and usage. As a guideline, a real-world use case elaborated in cooperation with industry partners is presented. Further on, it is shown, how general purpose web technology can be combined advantageously with recent architectural-, operational- and cultural trends in software design, powerful machine-to-machine interaction techniques and common user experience concepts. In-depth thoughts on software structure, just-in-time communication, machine-to-machine interaction, uniform data integration and user experience are conducted to finally obtain a working proof-of-concept software alongside recommendations and best practices for adopting smart technology on the shop floor. An eventual evaluation investigates the designed platform regarding both performance and suitability for particular real-world scenarios and recommends further endeavor to be conducted towards achieving an actual product.

Contents

1. Introduction

1.1 Motivation

According to recent studies, companies are given a chance to increase their performance, flexibility and productivity by around 40 % when properly applying digitization [1]. As a result, the topic of digital transformation towards smart factories is of rapidly growing concern among the industry. One major challenge is to interlink people, cyber-physical systems (e.g. smart machines and sensors) and resources and deal with the increasing amount of just-in-time data exchanged between them. This is referred to as building an Industrial Internet Of Things (IIoT). Although many solutions are coming up in that domain, their business-integration is cumbersome. On the one hand, neither technical- nor business-process-related standards have established. On the contrary, the adoption of new technology, especially in the context of software development, currently necessitates extensive technical knowledge and therefore is considered a hurdle not to be overcome by most companies. While businesses recognize the high value of an IIoT, it is still necessary to endeavor on large, sluggish software projects to tap into that potential. Addressing these problems, this work proposes a distributed software architecture for integrating data in smart factories, that focuses on flexibility, performance and simplicity. In the course of this, we also present best practices for promoting the digital transformation by introducing state of the art software development practices and well-aligned business processes to the manufacturing industry. The approach outlined in this work allows industry leaders in manufacturing to boost their IT capabilities, obliterating barriers such as high up-front planning and -costs and the risks of rigid IT infrastructure. This eventuates in higher overall performance, flexibility and productivity.

1.2 Structure

The present work is structured as follows. Chapter 2 provides an introduction to fundamental terminology, concepts and background knowledge. The reader will be made familiar with challenges, chances and enablers of digital transformation in the manufacturing industry. Relevant concepts of software architecture, as well as recent trends in the field of modern software development life-cycles are presented.

On a more technical level specific technologies used in our architecture are explained, including state of the art web-development- and Linked Data techniques including several benefits and downsides.

Chapter 3 displays a specific, real-world use case to which our solution might be applied to. This authentic scenario, elaborated in cooperation with industry partners, served as a guideline during our design process and points out which specific problems our software is able to solve.

Chapter 4 describes the actual architecture including possible alternatives and discussions on each of them, weighing their respective benefits and downsides with regard to our use case. Software components, design- and communication patterns and their consequences are pointed out. Additionally, it will be made clear which user experience considerations were made in order to follow our design goal of simplicity.

In chapter 5, the abstract architecture is implemented. The reader will understand how architectural concepts result in actual software components, as well as the accompanying technological decisions made. Furthermore, she will be provided a technical discussion on the trade-off between our goals of flexibility and performance.

Chapter 6 exposes an overview of related work in the fields of both IIoT software solutions and architectural methods. The reader will understand similar approaches, their characteristics, benefits and downsides, in how far they differ from our proposed software and how to integrate given technology to form our solution.

Chapter 7 evaluates our architecture in terms of performance while picking up on the design alternatives from chapter 4 in a comparison. Furthermore, insights derived from presenting our platform architecture to industry partners are exposed. These insights are indicative of potential real-world customers' concerns on the adoption of a platform like the one presented in this work. An overall discussion of the suitability of our platform to achieve the goals introduced in 1.1 is conducted.

Finally, in chapter 8, the key points of this work are summarized and an outlook to the scientific- and economical relevance of this work as well as to potential future work is given.

2. Background

2.1 Industry 4.0 and the Digital Transformation

2.1.1 Overview and Terminology

The term ”Industry 4.0” originates from a future project of the German government with the aim to strengthen and increase the German manufacturing industry's competitiveness through the application of innovative information- and communication technology. Since this work focuses on industrial scenarios, there is a need to introduce these related concepts. Industry 4.0 means the fourth industrial revolution. After having introduced water- and steam-powered manufacturing plants in the first, electricity-powered large-scale production in the second and electronics, lean production and basic IT in the third industrial revolution, the fourth one is mainly characterized through heavily connected, distributed, internet-based IT-systems and the extensive use of large-scale data [1]. Centrally controlled processes should be shifted to decentralized and more autonomous ones. The final goal is a more productive and efficient value chain with better performance and higher flexibility. Outside of Germany all of this is often basically referred to as a ”digital transformation”, ”Smart Manufacturing” or the ”Industrial Internet”. Consequently, these terms will be used interchangeably with Industry 4.0 in this work.

2.1.2 Characteristics

Manufacturers that have undergone the fourth industrial revolution have the following characteristics.

1. Supporting a high degree of product customization
2. Flexible and efficient production up to a batch size of one
3. Comprehensive integration of all stakeholders into their business processes.
4. Combination of products and services, so called hybrid products (e.g. to sell ”mobility” instead of vehicles) [1]

2.1.3 Components

As stated by Hermann, Pentek and Otto, Industry 4.0 consists of three major components: the Internet of Things - or in the context of manufacturing more precisely referred to as the Industrial Internet of Things, cyber-physical systems (CPS) and smart factories. The latter are enabled through the combination of the IIoT and CPS. [2]

"The IoT allows things and objects, such as RFID, sensors, actuators, mobile phones [...] to interact with each other and cooperate with their neighboring 'smart' components to reach common goals" [3]. Such a common goal is, for instance, to improve product quality. An example use case could be a machine analyzing a part's chemical constitution while processing it and publishing the information to another machine further in the supply chain. That machine can, based on the given data, precisely adjust its settings according to the material. Every participant in an IoT is uniquely identifiable and accessible.

The second component are CPS. CPS are "integrations of computation and physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa." [4] A CPS usually, but not necessarily, is able to store and analyze data, has sensors and actuators and is network capable [5]. In the above example use case, both machines can be considered CPS, since they have at least a sensor and an actuator and communicate through participating in an IoT.

The third component are smart factories. In smart factories people, things and resources are connected and exchange data to autonomously draw conclusions and perform decisions. Formally "the Smart Factory is defined as a factory that context-aware assists people and machines in execution of their tasks [...] based on information coming from physical and virtual world. Information of the physical world is, for instance, position or condition of a tool, in contrast to information of the virtual world, such as electronic documents, drawings and simulation models." [6] Smart factories also enable for smart products, which "know their production history, their current and target state, and actively steer themselves through the production process by instructing machines to perform the required manufacturing tasks and ordering conveyors for transportation to the next production stage." [7]

2.1.4 Design Principles

According to Hermann, Pentek and Otto, there are six design principles in Industry 4.0 [2]. These principles support companies in identifying and implementing Industry 4.0 scenarios.

Interoperability: the ability of cyber-physical system (i.e. work-piece carriers, assembly stations and products), human and smart factories to connect and communicate with each other via the IoT [...]

Virtualization: a virtual copy of the Smart Factory which is created by linking sensor data (from monitoring physical processes) with virtual plant models and simulation models

Table 2.1: Industry 3.0 vs. Industry 4.0 from an IT-perspective

	Today (3rd indus. rev.)	Tomorrow (4th indus. rev.)
Production scheduling	Centralized	Decentralized (CPS, Cloud)
Control technology	Monolithic	Open web standard (Cloud)
Data processing	Staggered	Real-time information
Software licensing	Upfront licensing costs	Pay-per-use / Pay-as-you-go
Software management	On-premise software suites	Cloud-Apps (SaaS)

Decentralization: the ability of cyber-physical systems within Smart Factories to make decisions on their own

Just-in-Time Capability: the capability to collect and analyze data and provide the derived insights immediately

Service Orientation: offering of services (of cyber-physical systems, humans or Smart Factories) via the Internet of Services

Modularity: flexible adaptation of Smart Factories to changing requirements by replacing or expanding individual modules

We used these principles as a guideline when designing the subsequent architecture and endeavored to map each of these six aspects to out software components.

2.2 Semantic Web and Linked Data

Industry 4.0 scenarios entail a high degree of homogeneity resulting from diverse human and machine actors. Furthermore, as highlighted in section 2.1, the high communication needs result in a very high volume of data exchanged between actors. These requirements come with challenges of their own. One of them is the fact that machines are completely unaware of the data they are transferring. They cannot interpret the meaning of a datum, but only deal with plain character sequences and numbers (or even just bits, viewed from a lower level perspective). When designing software for autonomous smart factories, this is not sufficient anymore, because at some point a machine has to know which data it needs to fulfill a task and where to get it. A human user browsing the web acts like this: he starts with an initial page, reads the information, interprets the links, gets an idea of what might be behind those links, clicks one based on his interpretation and arrives at a new page containing other links. Computers usually cannot act this way. Therefore, some kind of semantics need to be introduced to make data machine-understandable. "Semantics broadly means a shared meaning of terms that represent a particular domain. The goal is, in other words, that several communities establish terms with a shared meaning, so that the resulting representations can be communicated, accessed, integrated and processed in applications." [8] In an industrial scenario, ontologies covering the respective business domain can be built up to facilitate the various smart actors on the shop floor to get a common understanding of the data they are exchanging. Similar to a human, these smart machine actors could then interpret each others' data, what eventually results in more autonomous communication among them.

2.2.1 Ontologies

Ontologies are a way to add semantics to domains that require shared meaning. An ontology basically is a common vocabulary declaring domain-related terms. Every time a term is used with reference to an ontology, all consumers - be them a humans or machines - can look up its meaning in the shared ontology.

In the context of the Semantic Web, an ontology defines the concepts and relationships used to describe and represent an area of concern. Possible relationships and possible constraints on the defined concepts are classified in the ontology. In order to associate a concept to a given ontology, namespaces can be used. An example usage of a namespace would be ”ex:vehicle”. Every actor using the ”vehicle” term from the ontology related to the ”ex” namespace is referring to the exact same ”vehicle” concept. Although ”ex:vehicle” still is just a plain string, the key is that it is commonly used by everyone when talking about vehicles.

2.2.2 RDF

Semantic data usually is expressed using the Resource Description Framework (RDF) data model. RDF data is graph-based and does always consist of triples. A triple, in turn, consists of three terms - namely subject, predicate and object - and describes a resource.

Figure 2.1 shows an example RDF triple. In the example ”http://example.org/vocab#Tim-Berners_Lee” as the described resource is the subject, ”http://example.org/vocab#hasWorkplace” is the predicate and ”http://example.org/vocab#W3C” as another resource is the object. In this case, the object would probably be included in further triples as subject and therefore link to further resources (the objects of those triples) again. A large graph of data can be built up like this.

Figure 2.1: RDF triple example

2.2.3 JSON-LD

RDF data can be serialized using multiple formats. Common ones are Turtle, N3, RDF/XML, RDFa or JSON-LD. In this work we will make use of JSON-LD for the most parts. It basically looks like traditional JSON, but additionally declares a few keywords to more easily represent RDF concepts.

Snippet 5.1 shows a JSON-LD serialization of the example triple from figure 2.1.

Listing 2.1: Example JSON-LD serialization

```
{
  "@context": {
    "ex": "http://example.org/vocab#"
  },
  "@id": "ex:Tim-Berners_Lee",
  "ex:hasWorkplace": {
    "@id": "ex:W3C"
  }
}
```

2.2.4 SPARQL

A way to persist RDF data is to use graph-based databases or so called ”triples stores”. To query and retrieve data from such triple stores, a standard query format named ”SPARQL” is commonly used. The de-facto standard query language for RDF data is similar to what SQL is for relationally structured data.

2.2.5 Linked Data Principles

As described above, a large information graph can be built up by interlinking resources with the aid of RDF triples. This is the foundation of what Tim-Berners Lee has described as Linked Data. Linked Data uses Semantic Web concepts, primarily RDF, in conjunction with web architecture to consolidate heterogeneous data from multiple sources. Tim-Berners Lee professes that ”the semantic web is not just about putting data on the web. It is about making links so that a person or machines can explore the web of data. With Linked Data, when you have some of it, you can find other, related, data.” [9] This web of data is also referred to as Web 3.0, in which, in contrast to Web 1.0 and 2.0, not (HTML) documents are the central resources of interest, but data is. Information on a specific resource might be contained in multiple documents, but is combined through semantics to one set of data. Such an abstraction has potential to be of great convenience when having to dynamically integrate heterogeneous data from many different IoT participants within an industrial scenario as we face one.

Tim-Berners Lee proposed four principles of Linked Data.

1. Use URIs as names for things / resources.
2. Use HTTP URIs (URLs) to make these names dereferenceable to enable people to look them up.
3. When someone looks up an URI, provide useful information using standard formats.
4. Include links to other URIs so that further resources can be discovered.

In chapter 4.5 we describe how Linked Data can be used to introduce a high level of flexibility to an IoT software platform and pick up on these principles again as a central guideline for the design of our software components and their communication.

2.3 Design Patterns for Distributed Architectures

Recent trends in software engineering brought up two new architectural patterns for distributed applications: Microservice-based- and serverless architectures. Both of them aim at designing very modular, fine-grained software with focus on great flexibility and scalability. These characteristics as well as those concepts’ recent gain in popularity among the industry makes them worth to be considered an option for our target platform architecture.

2.3.1 Microservices

The Microservice pattern is an architectural style for enterprise application suites, which very fundamental idea is to split up a heavyweight monolithic application into multiple independent, usually smaller, stand-alone parts. A rather technical definition of Microservices is given by Fowler: "The Microservice architectural style is an approach to developing a single application as a suite of small [fine-grained] services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies." [10] Also important in the context of Microservices is the principle of bounded contexts, originally proposed by Evans [11]. Fundamentally, an application or service with a bounded context is of very high cohesion and rather isolated from surrounding services in terms of business functionality. Since Microservices themselves usually have very limited functionality, so called mashups are built by wiring together multiple Microservices to build more complex functionality from them.

Based on remarks from Fowler [10] in combination with our own experiences we ascertained Microservices to generally feature the following characteristics.

Componentization via Services . Every functionality is implemented as a self-contained service that can be invoked independently. The services are components to be used as building blocks to mash up a more complex, higher-level service.

Decentralized governance and data management. Each service is deployed, executed and monitored as an independent, standalone application with a separate data source and does not rely on any kind of single central control instance.

Organized around Business Capabilities. According to Conways's Law[1] a system's design will inevitably be congruent to the structure of the organization it was built by. Therefore Microservices are designed in a way that each service corresponds to a specific business function. Development teams are rather cross-functional and separated along business capabilities instead of by software components' technical function (UI, logic, database, ...). For instance, an e-commerce scenario would probably comprise (among others) Microservices for purchase, billing and shipping, each of them owned by a distinct team.

High cohesion, loose coupling. A Microservice is responsible for exactly one assignment (e.g. billing), while there is no intersection in terms functionality with any other service in the ecosystem (Bounded Context). In principle, things that are often getting changed together are likely to be used as a single service. This requirement also includes that a service does not strongly depend on any other service in a way that it could not fulfill its own task if the dependent service becomes unavailable. The communication between services is generic in a way that a service could easily be replaced by an equivalent one without having to adapt any other service. Usually, this is achieved by exposing a uniform interface, typically a REST API in the context of web services.

[1] Conway's Law: Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

Polyglott stack. Due to the requirement of loose coupling using uniform interfaces, services may (but not necessarily have to) be realized heterogeneously using a variety of technologies (programming language, framework, database, ...), since the services are completely self-contained applications.

Designed for failure. Both the services itself as well as the network, over which the communication between multiple services usually happens, is considered unreliable. Services are built tolerant in a way that they still continue to work if related services or the network fail. In many scenarios, such as Netflix' Simian Army [12], this is realized by implement the CircuitBreaker pattern proposed by Nygard [13].

2.3.2 Microservices vs. SOA

Microservices are not to be confused with service-oriented architectures (SOA) by any means. Although both patterns rely on services as a central concept, they are completely opposite approaches. Combining details stated by Richards [14] with our own perceptions, we drew up Table 2.2 as a comparison between Microservices and SOAs in view of some technical key aspects.

Table 2.2: Microservices vs. service-oriented architectures (SOA)

	Microservices	SOA
Interface type	Resource-based (REST)	Method-based (WSDL, SOAP)
Coupling	Uniform interface	Contract-based
Application type	Distributed	Monolithic
Request processing	Silo-based, self-contained	Component sharing
Composition	Choreography	Orchestration (central management)
Mediation	API layer	Messaging middleware
Invocation	Event-driven	Context-/data-aware

Figure 2.2 illustrates silo-based processing in comparison to component sharing and figure 2.3 illustrates service-orchestration in comparison to service-choreography.

Certainly both approaches feature various pros and cons. While (dis-)advantages of Microservices are discussed in chapter 4.2.2.1, SOA is not elaborated in greater detail in this work.

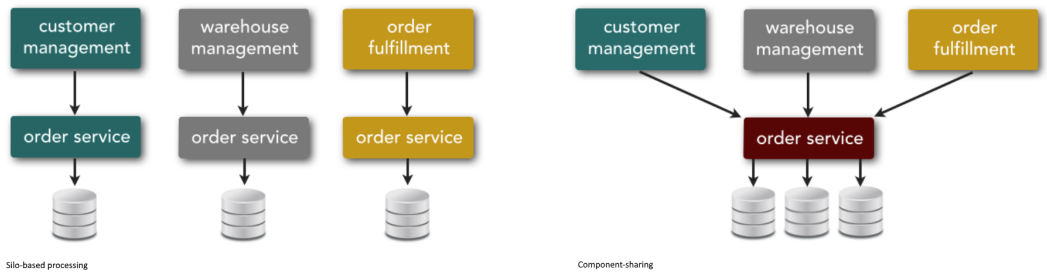

Figure 2.2: Silo-based processing vs. component-sharing [14]

Figure 2.3: Choreography vs. orchestration [14]

2.3.3 Serverless

"Serverless is to operations what IaaS was to hardware." [15] More specifically, "serverless refers to a cloud architectural design pattern that abstracts servers away to the point that developers have little to no direct interaction with them." [16]

In traditional software development, the software engineer had to be acutely aware of the concept of a server. The essence of the serverless trend now is the absence of the server concept during development. As a consequence, it aims at improving efficiency during development by shifting competence to the cloud. It also adds further potential in terms of scalability and flexibility due to its fine-grained, distributed and functional style.

Actually the term "serverless" is somewhat misleading. Of course there has to be any kind of servers in the end, but their role has changed significantly. Instead of supporting a specific application, clusters of servers provide a generic execution environment for any number of applications. More precisely, these applications usually are actually just simple, event-triggered functions comprising only a few lines of code. For that reason, serverless often is also referred to as "Functions as a Service". The functions are typically hosted using a cloud service such as AWS Lambda[2] or Webtasks[3]. The main difference in contrast to Microservices relates to granularity. Even though both patterns split up an application into multiple, separate pieces, a Microservice represents an entire business capability, while a serverless function only performs one single task. To pick up on the example of an e-commerce application again, a Microservice would probably comprise the entire billing process, whereas with serverless there might be a hosted function that only generates an invoice PDF file. Similar examples can be thought of in an industrial context, where such a function could, for instance, be responsible for transforming a particular measurement value or the like.

[2]https://aws.amazon.com/lambda
[3]https://webtask.io/

3. Manufacturing Shop Floor Scenario

Since this work takes place in the context of the ScaleIT research project[1], we are provided with knowledge input from actual industry partners. A global manufacturer of industrial sensor technology located in Germany - pseudonymized referred to "SensorCompany" - gave us an example use case as a starting point to be used as a guideline when designing the architecture. Fundamentally, SensorCompany aims to move away from having partly unstructured and partly statically schemed data, a central, monolithic data integration layer and single, large databases towards more of a smart factory. The reference use case relates to one specific step during the production cycle of sensor chips, namely the equipping. In this step, a machine takes raw circuit boards as input, places several electronic components on it and afterwards bakes the board for several minutes at medium-high temperature. After that the equipped board is passed through a quality assurance step, in which the correct position of the newly placed components is verified with a precision in the magnitude of micrometers. This verification is called optical quality assessment (OQA), because it makes use of high-performance visual detectors. These OQA machines are an essential part of our use case, since they are sensors act as our main data sources. The OQA sensors' primary raw output is a plain bit vector which then is converted into a JSON format by a Python script running on the sensors. Besides some meta data the JSON object contains two objects with 64 key-value pairs, so called features, each. These features basically represent the respective components' shift in x- and y-direction on the board as well as their twisting. More detailed meaning of these data is not relevant for us at this point. Listing 3.1 gives an example dataset produced by OQA sensors.

Listing 3.1: Example dataset output by OQA sensors

```
1 {
2   "Timestamp": "2016-04-23T18:25:43.511Z",
3   "Materialnummer": "2132419",
4   "Aenderungstand": "AYZ1",
5   "Lage": "1",
6   "Prozessschritt": "UZT_AS_1",
```

[1] http://scale-it.org

```
  "Datenklasse": "q",
  "BoardUniqueId": null,
  "PanelUniqueId": null,
  "SeqNo": null,
  "Bauteil-Id": "C1-1",
  "Bauteil-Typ": "C0603",
  "Analyse-Modus": "OLAM",
  "Fenster-Nr": "1301",
  "Feature-Flag": "2",
  "Bediener": "FM",
  "Features": [
    {
      "Feature0": "1",
      "Feature1": "0",
      "Feature2": "250",
      "Feature3": "1",
      "Feature4": "0",
      "Feature5": "5555",
      "Feature63": "1"
    },
    {
      "Feature0": "1",
      "Feature1": "0",
      "Feature2": "330",
      "Feature3": "1",
      "Feature4": "0",
      "Feature5": "366",
      "Feature63": "0"
    }
  ]
}
```

SensorCompany intends the data to be persisted on the one hand and to be published to arbitrary further IT systems within the enterprise on the other. Examples for further processing of OQA data reach from just-in-time visualization on tablets over big data analytics throughout to generating on-demand production reports. In order to ensure data integration and -compatibility, SensorCompany requires the data to be enhanced with semantic annotations, referring to both process-specific and more common, industry-wide vocabularies. A key ontology to be used in this context is the Sensor Network Ontology (SSN)[2]. In cooperation with SensorCompany, we annotated the OQA data using SSN in combination with a custom OQA-related ontology provided by SensorCompany. An annotated example indicating the x-shift is depicted in Listing 3.2. It is serialized in JSON-LD format.

Listing 3.2: Annotated dataset for OQA x-shifts

```
{
  "@context": {
    "ssn": "http://purl.oclc.org/NET/ssnx/ssn#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "oqa": "http://scale-it.org/oqa#",
    "observationResult": {
      "@id": "oqa:observationResult",
      "@type": "@id"
    },
    "observationResultTime": {
      "@id": "ssn:observationResultTime",
      "@type": "xsd:dateTime"
    },
    "observedBy": {
      "@id": "ssn:observedBy",
      "@type": "@id"
    },
    "observedProperty": {
      "@id": "ssn:observedProperty",
      "@type": "@id"
    },
    "featureOfInterest": {
```

[2] http://purl.oclc.org/NET/ssnx/ssn#

```
23        "@id": "ssn:featureOfInterest",
24        "@type": "@id"
25      },
26      "observedPart": "oqa:observedPart",
27      "observedStep": "oqa:observedStep",
28      "analysisMode": "oqa:analysisMode",
29      "materialNumber": "oqa:materialNumber",
30      "dataClass": "oqa:dataClass"
31    },
32    "@graph": [
33      {
34        "@id": "http://localhost:3000/observations/185",
35        "@type": "ssn:Observation",
36        "featureOfInterest":"oqa:shift-feature",
37        "observationSamplingTime":"2016-04-10T18:00:00",
38        "observedProperty" : "oqa:shift",
39        "observationResult": "http://localhost:3000/observations/185/sensor-
              output",
40        "observationResultTime": "2016-05-18T12:55:27.954Z",
41        "observedBy": "http://localhost:3001/OQA_SMD407/x-shift",
42        "analysisMode": "oqa:OLAM",
43        "observedPart": "http://scale-it.org/parts/C0603",
44        "observedStep": "UZT_AS_1",
45        "materialNumber": "382758",
46        "dataClass": "Testdata"
47      },
48      {
49        "@id": "http://localhost:3000/observations/185/sensor-output",
50        "@type": "ssn:SensorOutput",
51        "ssn:isProducedBy": "http://localhost:3001/OQA_SMD407",
52        "ssn:hasValue": "http://localhost:3000/observations/185/result"
53      },
54      {
55        "@id": "http://localhost:3000/observations/185/result",
56        "@type": "qudt:QuantityValue",
57        "ssn:hasValue": {
58          "@type": "xsd:integer",
59          "@value": "-17"
60        }
61      }
62    ]
63  }
```

Figure 3.1 depicts SensorCompany's objective of a possible scenario. It consists of several OQA machines (multiple production lines) that produce annotated data to be injected to our platform. The platform comprises containerized, self-contained software parts to take delivery of these data. These software parts can have arbitrary purposes like the ones mentioned above. Furthermore there should be an integration of these data into their manufacturing execution system (MES) and a time-series database (TSD).

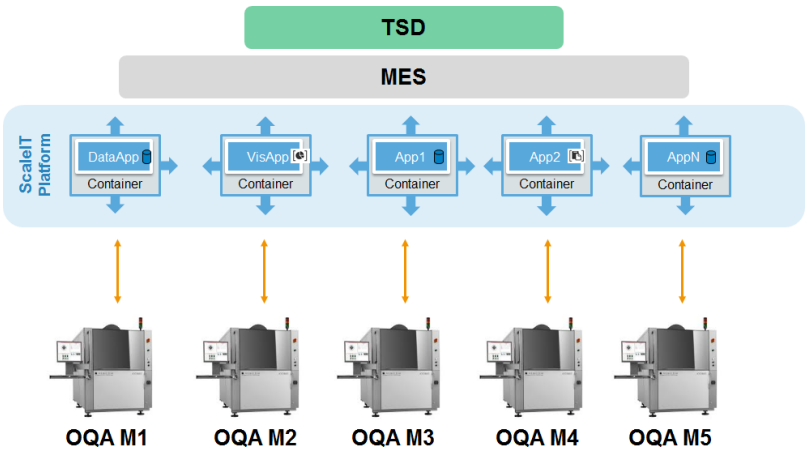

Figure 3.1: SensorCompany's objective of a target scenario

4. Design

Remembering the overall goal of this work as stated in chapter 1.1, we attempt to design a software platform that helps leveraging the digitization's potential in increasing manufacturers' performance, flexibility and productivity.

In order to achieve that goal, the architecture designed in this work aims at building scalable, dynamic software systems. The key focus heavily lies on loose coupling between distributed software components. The components should be as self-contained and independent as any possible in order to make them replaceable and reusable with a minimal need for configuration effort as well as to support a heterogeneous and polyglot stack among them. To still allow for high performance, the communication among components needs to be well-conceived and build using suitable technologies and implementation patterns.

To cover these aspects, we need to focus on various areas. We first formulate a set of requirements and go into challenges related to fulfilling them. Second, we identify required components as well as guidelines and principles on how to realize them and present a mapping to our particular use case. After that we specify how interfaces and communication between components should look like. Doing so we are weighing up multiple options. Finally we outline how the moving parts are being integrated and go into some details on development and deployment.

4.1 Overall Requirements

We define the following requirements, which our architecture has to fulfill in order to achieve the stated goals. These requirements coincide with the Industry 4.0 design principles presented in chapter 2.1.4 for the most parts.

Uniformity. Our architecture is supposed to be built upon existing, common technology. Rather young areas of research tend to employ specialized, custom-built, non-standard protocols. Such include MQTT, CoAP and AMQP in the context of the IoT and OPC-UA and several fieldbuses on the shop floor. Although these custom protocols usually might be better suited for a particular domain of use cases at

first sight, their major drawback lies in adoption and integration. On the one hand, companies usually do not occupy the specialized know-how on such technologies. On the other hand a seamless integration with further, higher-level system will prove difficult.

Just-in-Time Capability. Sensor data usually is emitted in intervals. Such intervals reach from hours or even days to seconds or milliseconds. Applications responsible for monitoring on the shop floor or actuators at machines are likely to require data just-in-time as it emerges to make ad-hoc decisions. Hence, our platform is required to deliver data from a large amount of producers without a time lag.

Ease of Use. We want our platform to require only a bare minimum of technical knowledge to use it. Workers on the shop floor should be able to implement a certain scenario on our platform without a need to deal with software- and network configuration. At best case, deploying a new service instance should be done in a one-click fashion. At the same time it should be as easy as possible for developers to extend our platform without getting deep into implementation details.

Data-Centricity. Our target industry scenarios include heterogeneous and mostly unstructured data. Nevertheless, we want to be able to easily integrate and consolidate these. Since information derived from that data is the companies' most valuable good [17], it should be easy to retrieve and query, just as there was a single large "data-lake" [17]. Users should not have to think of applications or devices but only of data.

Scalability. To fit arbitrary use cases within any size of organization, the developed platform needs to easily scale vertically as well as horizontally. This also includes stability and the absence of a single point of failure. It further includes hardware efficiency in order to run software components on less capable devices like a Raspberry Pi, as they are usual in the IoT.

Fitness Function

In addition to our custom defined requirements, we construct a decagonal fitness function as presented by Ford and Parsons [18]. It includes ten predefined aspects of software architectures, where we rated each of them with a score between zero and five to state the respective aspect's importance for our architecture. The given scores reflect our personal objective rather than being underlain by formal mathematical computations.

The characteristics stated by the fitness function do not completely match our custom ones defined precedingly, but certainly there is an intersection. Our requirement of "just-in-time capability" corresponds to the fitness function's "low latency" in combination with the "high throughput" aspects. "Ease of use" can be mapped one-to-one to "usability." Also our "data-centricity" requirement can be considered related to usability, since it eventually aims at perceivability for the user. However, the "high availability" dimension includes x- and z-axis scalability, but is not completely sufficient, since it neglects y-axis scalability. Furthermore, our "uniformity" requirement cannot be mapped to the fitness function at all. Nevertheless can the presence of such a fitness function be valuable during development to make sure the architecture is evolving in the right direction.

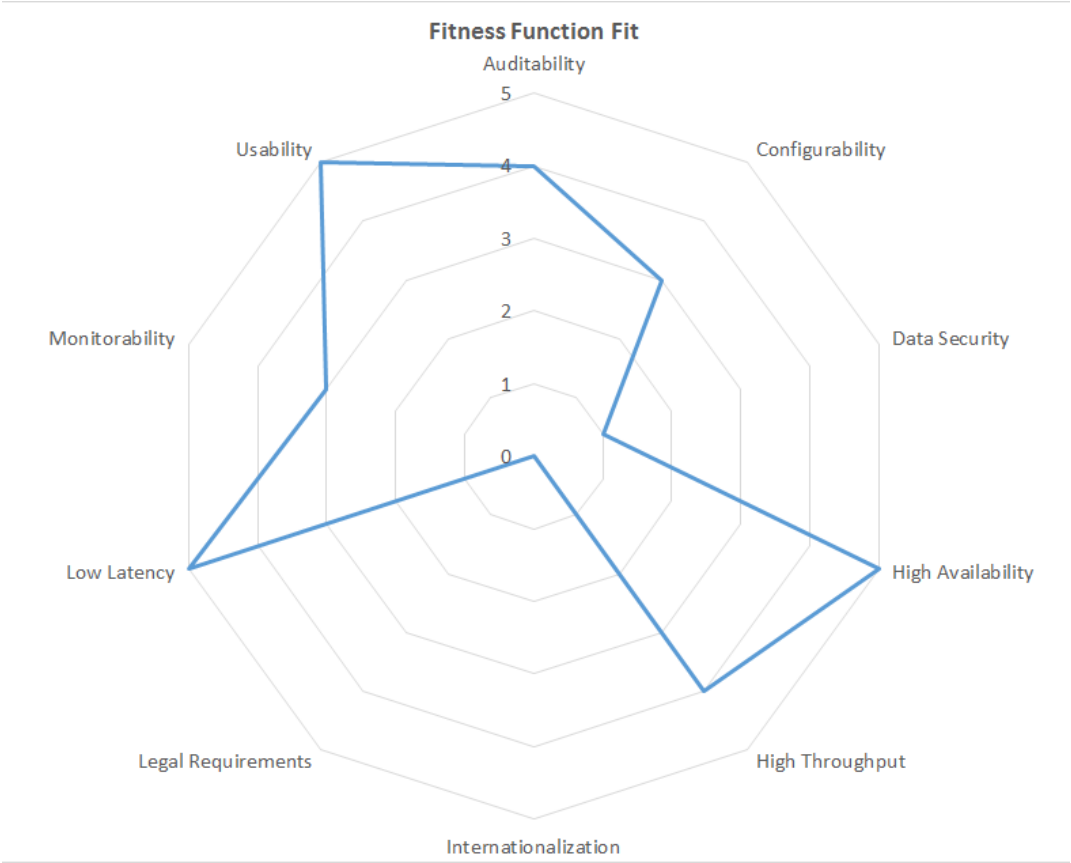

Figure 4.1: Architecture Fitness Function

Readers might wonder why three aspects are almost completely neglected. While we do not want security to be an afterthought, it is beyond the scope of this work to create a secure architecture. However, the design choices made and the technologies used offer great potential regarding security. Since we are using web technology, existing, established standard web-security mechanisms can be applied for the most parts. Therefore we are confident that our reference architecture works as a well-suited foundation that can be extended with security components without major modifications. Additionally, the Fraunhofer Institute carries out deep research on security within an "Industrial Data Space" [17]. Legal requirements and internationalization are scored with zero, because our platform does not aim to be included to a public-facing web-service, but addresses closed, inner-organizational scenarios.

4.2 Architecture

4.2.1 Web Technology

Depending on the context, the most widely used TCP/IP based application protocols in the industrial IoT are MQTT, AMQP, CoAP. However, we find that especially the uniformity requirement is difficult to attain using any of them. As a result, the chance of a rapid adoption of the designed platform is lowered, whereas its integration effort would increase. For the same reason we also want to avoid the implementation of a custom protocol. Instead we choose to follow the suggestions of Guinard and Trifa [19] and build the entire architecture based on web technology.

HTTP acts as both central communication- and application layer protocol. Interfaces are intended to be designed in a RESTful way. Employing web-technology comes with the great benefit that it is probably the most widely accepted technology on the internet and characterized by uniformity and simplicity. Both the actual data transfer as well as the basic CRUD operations on resources are already defined and commonly agreed on when using REST. Instead of explicitly having to know a service's methods or entering a contract, client applications only need to consume the service's REST interface. According to Guinard and Trifa [19], a uniform interface is the most important constraint for integrating devices into a web of things. "First, using a uniform interface such as that defined by HTTP minimizes the coupling between components, which helps us design more scalable and robust applications. Second, we can use a web-like mindset to design applications: markup languages, event- based browser interactions, scripting languages, URLs, and the like. Third, HTTP traffic on port 80 is the only protocol that's always permitted by most firewalls. Fourth, it makes it easy to hide low-level protocol details behind simple, high-level abstractions, which promotes openness, programmability, and reusability of services and data regardless of how they're actually stored or encoded." [19]

However, HTTP is not sufficient for all IoT use cases. Especially, HTTP makes use of the request / response communication pattern (active communication), which requires every data emission to be triggered by a separate, preceding request. This does not allow for bi-directional data exchange between server and client and especially prohibits to subscribe to event-triggered data sources, such as sensors. There is a need to extend basic HTTP by mechanisms that permit the publish / subscribe communication pattern (passive communication). Existing techniques include Websockets and WebRTC for fully bi-directional communication alongside HTML5 server-sent event (SSE), long-polling and webhooks for server-to-client data flow. Guinard and Trifa, who had realized the exact same problem with purely req / res based HTTP when designing their similar platform approach, employed Websockets [19]. However, we do not consider this the most suitable options due to the reasons presented in the context of a discussion on these different technologies in chapter 5.3. Even with both active and passive communication, there still is a downside with a general-purpose protocol like HTTP. Performance and communication overhead are expected to succumb those of a highly customized, optimized and potentially binary protocol. The concrete impact of this performance gap is outlined in chapter 7.2 as well as are a few approaches to counteract it up to a certain extent.

4.2.2 Distributed Architectures with Microservices

After having agreed on using web technology - while not using the web solely as a transport protocol but making applications and especially physical devices an integral part of it - the next point to give thoughts about is the actual application-level architecture. The Internet of Things is massively distributed by nature, as are our target scenarios. Physically separated participants, such as sensors and actuators at machines, workstations, mobile phones and tablets, autonomous vehicles and tools on the shop floor and smart products need to cooperate. Their interaction has to be coordinated effectively.

Firstly, as a consequence to the distributed nature of our target scenarios, building a single monolith obviously is not an option. We need to employ a distributed architecture pattern that is suited for rather large and complex systems. Basically there

are two major approaches to realize distributed, yet scalable designs: centralized and decentralized architecture patterns with their respective major representatives SOA and Microservices. Centralized approaches favor authority over autonomy, orchestration over choreography and configuration (WSDL descriptions) over convention (REST). Decentralized ones do the other way round. Microservices are usually small, self-contained software components with a uniform interface that each fulfill one domain-specific task. SOA services, on the contrary, are more heavyweight, strictly described and usually rely on a central integration layer in the form of a message bus. While chapter 2.3.2 basically clarifies the differences between SOAs and Microservices, the present section compares both approaches from a Microservice point of view. Advantages of Microservices - as discussed in the following - correspond to SOA's drawbacks and the other way round.

4.2.2.1 Advantages

Microservices feature the following advantages in view of our target scenarios.

Fault isolation & resilience. Due to the fact that Microservices are deployed completely independent from each other, they are less likely to cause a system failure when going down. Usually they are deployed as containers following the Service-per-Container pattern (see chapter 5.5.3 for a discussion on different Microservice deployment patterns).

Technology diversity. With Microservices, multiple languages, development frameworks and data-storage technologies can be mixed up. This allows to adopt technology more quickly on the one hand and to pick the most perfectly suitable stack for a specific use case on the other. In combination with HTTP as a uniform interface, the services still remain compatible. There is no restriction to a single technology like SOAP or JMI.

Reusability. Since Microservices are totally self-contained and independent from a central authority, they can easily be reused in multiple scenarios or cloned several times within the same scenario. There is nothing more to do so than spawning a new container each time. The authors of "The Art of Scalability" [24] refer to this as scalability along the x-axis or "scale by cloning" (see figure 4.2), other literature speaks of vertical scalability. Also z-axis scalability [20] can be archieved through the enhanced usability. Z-axis scalability basically means to have redundant service instances (but usually with a disjoint data set) where the traffic is load-balanced to according to pre-defined rules (see figure 4.3).

Small footprint. Since Microservices are responsible for limited scope of tasks within a domain, they are much more lightweight than monolithic systems. Having a smaller memory- and CPU footprint and less dependencies they are also ready to run on small hardware, such as one-chip computers, devboards and sensors.

Organizational alignment. Reinforcing modular structure results in strong modular boundaries and functional decomposition of Microservices. One the one hand this allows for more efficient development and operation through clear separation of competences. On the other hand it enables to scale horizontally or along the y-axis as referred to by Abbott [20].

Suitable application scope. Richards states that Microservices are - in contrast to SOA architectures - most suitable for "well-partitioned web-based systems rather than large-scale enterprise-wide systems", which have "few shared components and ones that can be broken down into very small discrete operations." [14] This just describes the average scenario we are facing. Additionally, according to Posta, Microservices even leverage their potential if designed event-driven - which is the recommended way. [21] Since our target scenarios intensively include events, such as threshold exceeding, machine collapse, production stage completion and more, this appears to be another plus.

Ease of deployment. "A one-line change to a million-line-long monolithic application requires the whole application to be deployed in order to release the change." [22] This does neither apply to Microservices nor to SOAs. In both cases, changes to a single service can be deployed independently of the rest of the system and therefore allow for faster deployments. Since Microservices are more lightweight than SOA services, there is an even increased advantage.

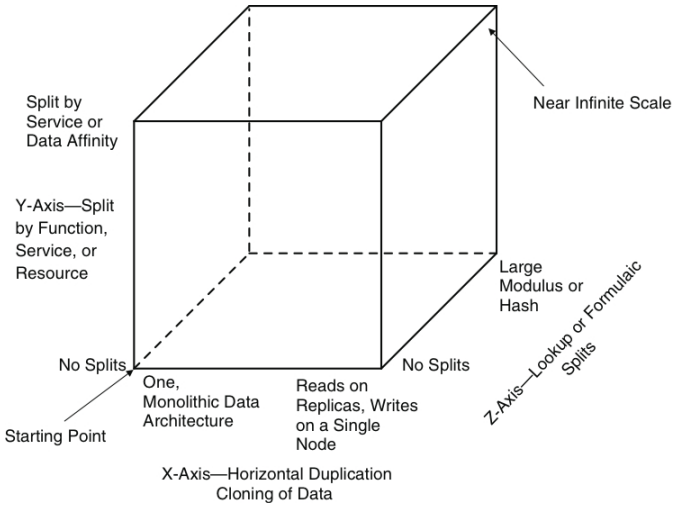

Figure 4.2: Scalability cube [20]

4.2.2.2 Drawbacks

Of course, Microservices also have their drawbacks in comparison to SOAs.

No contract decoupling. Richards describes contract decoupling as follows. "Contract decoupling is a very powerful capability that provides the highest degree of decoupling between service consumers and services. Imagine being able to communicate with a service using different data in a message format that differs from what the service is expecting — that is the very essence of contract decoupling." [14] As depicted in figure 4.4, contract decoupling allows a client to send a format different from what the service expects (in terms of both data serialization and -content). Basically content decoupling is enabled through a mapping between given input- and required output properties. Since Microservices lack a mediator (middleware) by nature, it is hard to define such a mapping globally. Chapter 4.5 shows how we address to solve this problem to a certain extent using Linked Data principles.

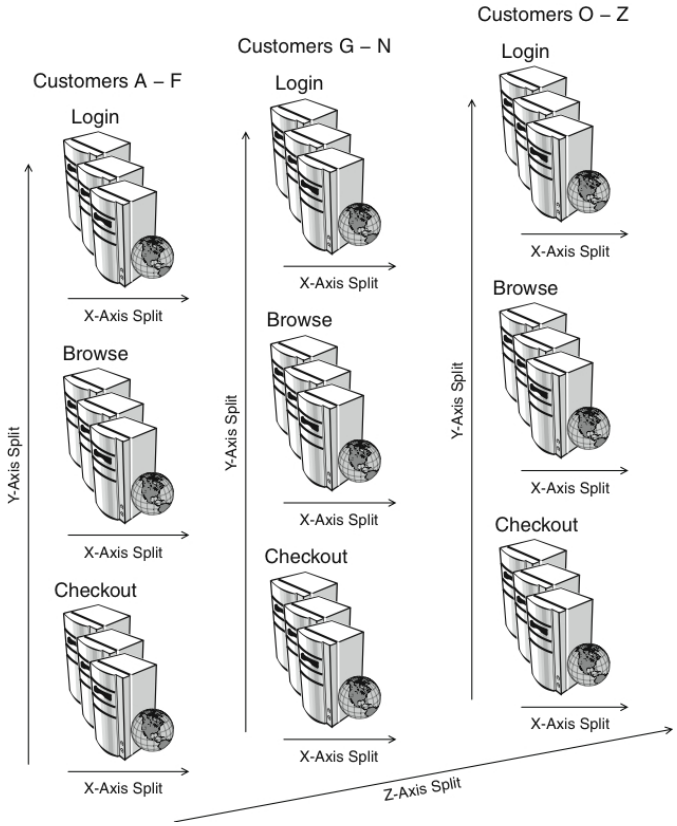

Figure 4.3: Three axis scalability - Split example [20]

Operational complexity. Many operation teams are required to manage the large amounts of services and related infrastructure, that are being redeployed regularly. As explained in chapter 5.5.3.2, the DevOps culture can help to counteract this.

Eventual consistency. Maintaining strong consistency is difficult for a distributed system in general. While SOA systems rely on ACID transactions that imply consistency at transaction level, things are more complex with Microservices. Because delay between numerous distributed service calls may become too high and systems or connections are considered more likely to fail, Microservice system rely on BASE transactions [14]. Applications using BASE transactions strive for eventual- rather than transactional consistency. Every single autonomous service is responsible for ensuring its own eventual consistency. The use of EventSourcing [23] can help with this.

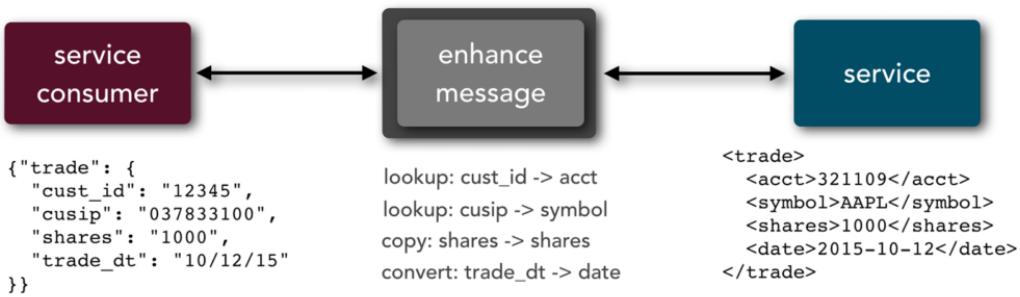

Figure 4.4: Contract decoupling example [14]

(Distribution).* Distributed systems are harder to program and more likely to break since remote calls are slow and unreliable and consistency has to be maintained across multiple subsystems. Furthermore, they always imply a network overhead.

* However, the last point is to be laid at the door of distributed systems in general and not specifically related to Microservices. Obviously it cannot be bypassed, since we are facing a distributed system.

4.2.2.3 Culture and Prerequisites

Although this work has a strong technical focus, there are still some rather non-technical considerations to be made in order to facilitate real-world adoption of our platform.

In practice, implementing the Microservice style involves more than building a highly modular service landscape along business domains. If applied properly, practicing Microservices implies a whole cultural change within the organization. Based on the works of Fowler [24] and Posta [25] we present a set of organizational prerequisites that have to be accomplished in order to truly benefit from the new architectural style.

Cross-functional teams. Chapter 2.3.1 explained that a key characteristic of Microservices is to have a bounded context that corresponds to a specific business domain. As an inevitable consequence to Conway's Law, there is a strong need to also align the development teams with these bounded contexts. Instead of having a UI team, a service team and a database team (functionally separated), the competences have to correspond to the service's business capabilities. Moreover, every team should own all relevant knowledge from development over testing throughout to production and maintenance, since "[b]ad behavior arises when you abstract people away from the consequences of their actions." [25] Therefore it is a very common best practice to adopt a DevOps culture when doing Microservices, where a team is responsible for its service's entire life-cycle. Figure 4.5 by Wilsenach [26] provides a good overview over the key points of DevOps.

Well-sized teams. According to Hackman and Coutu, the growth in the number of communication links (which are potential points of failure) between team members when adding more people follows this equation: (n * n-1) / 2 where n is the number of team members. [27] Therefore, according to Posta, as a rule of thumb a development team should have at maximum as many members as two pizzas could feed (two-pizza team) [25]. Usually, but not necessarily, agile development techniques are introduced along with Microservices to make team collaboration effective.

Rapid provisioning and deployment. Since Microservices-based systems require a large amount of infrastructure components, an organization should ensure that provisioning is not more than a matter of few hours [24]. At best case, provisioning and deployment are done completely automated by fully embracing continuous delivery [28] principles. Such include Infrastructure as Code (IaC) [29] techniques, automated deployment pipelines (including performing automated tests) and multiple environments. It is also highly recommendable to use Microservices in combination with containerization - usually using Docker - to both speed up provisioning and facilitate isolation.

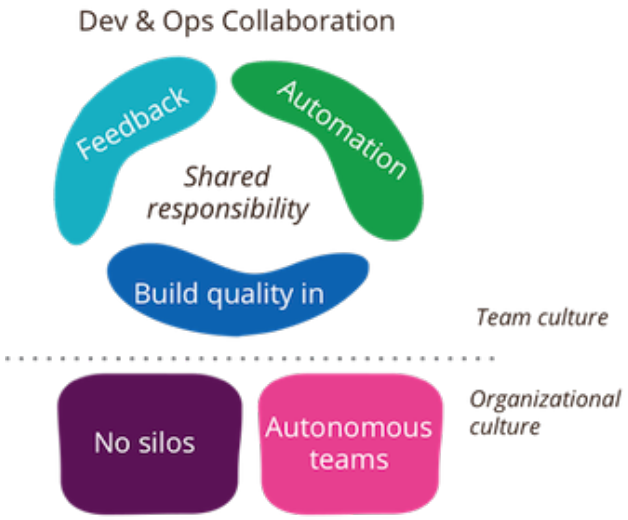

Figure 4.5: Key aspects of a DevOps culture [26]

The more mature organizations are in the adoption of these cultural aspects, the more likely is the success of Microservices.

4.2.2.4 The Serverless Paradigm

The Serverless or Function as a Service (FaaS) architectural pattern introduced in 2.3.3 appears to be promising at first sight, since it leads to a high degree of modularity. However, there are some reasons why it cannot be considered an alternative to Microservices or SOA for us. In the first place, it is too fine-grained. It appears elusive to build up an entire software platform out of such nano-services. Serverless rather turns out to be a solution to be used when implementing a particular service or subset of functionality. Consequentially, the overall, higher-level system consists of services, which internally might make use of Serverless principles. Otherwise, when invariably operating on such a fine-grained level, it is to be expected that communication overhead and orchestration effort would be prohibitive. On the other hand, Serverless is designed to be completely cloud-based. Manufacturers, however, do not wish to rely entirely on cloud infrastructure, thus, Serverless is not feasible without appropriate infrastructure.

4.2.2.5 Conclusion

Weighing benefits and drawbacks of SOA and Microservices we conclude that the latter approach almost perfectly fits our needs. A high degree of modularity, autonomy and isolation, which results in increased fault tolerance, reusability, scalability, as well as facilitation for development and operations and overall flexibility ratify this decision. We are confident that the stated lack of contract decoupling with Microservices can be overcome by introducing Linked Data principles. A potential solution to support eventual consistency across services is the use of EventSourcing, although we consider this a detail not to be implemented as part of this work. Lastly, the fact that large enterprise-scale companies like Otto [30], SoundCloud [31], Netflix and Amazon (cf. [22]) have successfully adopted and propagated Microservices, confirmed our decision.

4.3 Communication

In chapter 4.2.2 we agreed on deploying a Microservice architecture. In such architecture it is common to wire multiple simple Microservices together to achieve a higher-level service of more complex functionality. This is referred to as service composition or -mashup. Apparently, this requires quite a large amount of well-designed communication between the Microservices. To enable this while still maintaining a high degree of flexibility, scalability and stability, it is required to give up any kind of central communication broker that could turn out as a bottleneck or single point of failure. Thus, an enterprise service bus (ESB) or similar concepts are not an option for us. Instead, all data should be exchanged directly between the respective services. Basically this is also referred to as choreographing services in contrast to orchestrating them.

4.3.1 Event Collaboration

Due to increased breakdown resilience and further reasons described by Posta [21], communication within a Microservice architecture should always be driven by events, following the event collaboration pattern. "When we have components that collaborate with each other [...] we commonly think of their style of collaboration as being driven by requests. One component needs information that another has, so the needier component requests it [...] Event Collaboration works differently. Instead of components making requests when they need something, components raise events when things change. Other components then listen to events and react appropriately." [32] With reference to our use case this could be an OQA machine emitting the event that it has stopped working. All other components would receive that event and either ignore it or perform an action tied to it, e.g. generating a small management report and sending a push notification to the shop floor manager's smartphone. The sequence diagrams in figure 4.6 depict the difference between classic request-response-based active communication and passive communication using event collaboration. Jon Udell expressed the differences in an uniform way saying "[request-driven] software speaks when spoken to, event-driven software speaks when it has something to say." [33] "The great strength of Event Collaboration is that it affords a very loose coupling." [32] In fact, an event-producing component does not have to know anything about its consumers, as well as the consumers have no need to know where to get their data from. The consumers are simply provided with all events broadcasted from any producer. Although this is extremely simple in terms of development and deployment, there obviously is a downside: employing event collaboration in that narrow sense implies a tremendous communication overhead that is not affordable for our use cases - especially with respect to the usually small, limited hardware devices.

4.3.2 Passive Communication

To bypass communication overhead accompanied by event collaboration, we slightly modify the concept, while keeping the basic idea. Services are still supposed to communicate via events, but we limit the producers to multicast their events rather than broadcasting it. For this purpose we implement the publish / subscribe pattern. A service explicitly subscribes for events at other services in order to subsequently receive such. Of course this heavily restricts the loose coupling brought by event

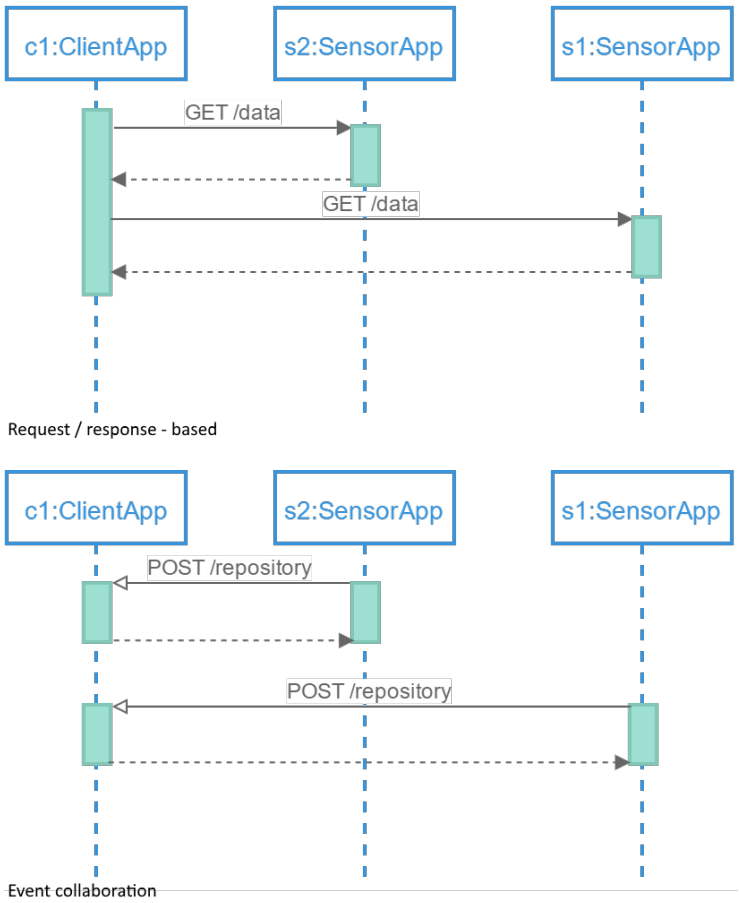

Figure 4.6: Request / response vs. event collaboration

collaboration, since a consumer has to know in advance which producer might potentially send relevant events. However, we will show how to restore loose coupling to a certain extend by introducing dynamic service composition and Linked Data APIs. The ideal way to avoid large communication overhead was to subscribe to certain events, not event producers, just as with an ESB. A service would automatically subscribe to just exactly those events, it has actions implemented for. Unfortunately, our decentralized Microservice environment does not allow such an implementation, since this would inevitably necessitate a central control instance to route the events.

4.3.3 Active Communication

Event-driven, passive communication is suitable for just-in-time cooperation between services. However, it is not always the right choice for all use cases. For instance, assume an end-user-facing frontend web application - a dashboard summarizing and visualizing key production metrics from an operational point of view. The service is invoked ad-hoc, right when the qualified shop floor manager opens the web page in his browser. At this point the application not only needs to receive just-in-time data in form of events, but also depends on historical data stored anywhere in a database. The easiest and only reasonable way is to use classic request / response based communication again. The dashboard service queries another service's API to fetch the data, processes and displays it and additionally subscribes to live-events to also display just-in-time data (passive communication again). Techniques to dynamically

discover the target service to query are presented subsequently. One more thing to note here is that the above use case could also be realized in a truly narrow-sense event-collaborational way. However, we consider this way far too cumbersome and prefer to employ request / response as a second pattern alongside (our adapted version of) event collaboration. Request / response (req / res) communication between services is facilitated by REST APIs, as it is de-facto standard on the web. Consequentially, we require every data-producing service to expose a RESTful-fashioned API.

4.4 Data Integration

As a consequence to having chosen Microservices as the architectural frame another challenge arises. Every service is a potential producer of data and to mash up larger services, that data apparently needs to be combined. The presence of such a large amount of potentially heterogeneous data sources makes integration operose. Especially there is a need for a mechanism to ensure data compatibility between multiple services. To pick up on the dashboard example again, the dashboard might be asked to present a particular type of metrics in a diagram and therefore must know which of the values it is receiving are relevant for that visualization. Tim Berners-Lee basically presented a solution to this when propagating the Linked Data web [9]. As described in chapter 2.2.5, data is virtually combined by annotating it with commonly defined terms of a vocabulary. Doing so introduces semantics to the data. Semantic annotations can help machines get a common understanding of the real-world meaning of data up to a certain extend. We require our services to follow the four Linked Data design principles [9] and mandatorily emit their data enhanced with semantic annotations and structured in a RDF fashion, referring to common vocabularies. Since it is de-facto standard for web services to expose their data in JSON format, we want to keep up on this and use JSON-LD as simple-to-use, easy-to-integrate, JSON-based serialization format for the exposed RDF data. Not only does this help with integrating data but also enhances event collaboration as explained in chapter 4.3.1. According to Petrovic, Burcea and Jacobson, extending pub/sub systems with semantic capabilities is ”an important issue to be studied, because components in a pub/sub system are priori decoupled, anonymous and do not necessarily speak the same language.” [34] Having events consisting of semantically annotated payload data adds meaning to these and enables subscribers to potentially decide autonomously which events might be relevant enough to subscribe for.

4.5 Dynamic Service Composition

4.5.1 The Findability Problem

Chapter 4.3 introduced event-driven communication between Microservices, utilizing the pub/sub and the req/res communication patterns. However, relinquishing a central control instance in order to remain decentralized causes several challenges for the architecture. A set of questions, also referred to as ”The Findability Problem” by Guinard and Trifa [19], need to be answered:

- ”How does one service get to know about another and how to discover a certain functionality?”

- ”What data in which format to send to which endpoint?”
- ”I got a response, but what does it mean and how do i process it?”

To pick up on the example from chapter 4.3, the dashboard service would probably have to subscribe to multiple other services. For instance, it might need to register itself for receiving events at all OQA machines on the shop floor, a service that manages customers' orders and another service capable of performing some kind of analytics. The challenge is to provide the dashboard service with information (ip address, API route, accepted data format, ...) on the targeted services in a dynamic way. We want dynamic service composition. The vision is to have entirely generic (machine-) clients that are capable to autonomously discover surrounding services with a certain functionality, explore their APIs and exchange respective data with them - without the need for any configuration or having to write code. To take a step towards that vision, we introduce semantic descriptions to our APIs.

4.5.2 Hypermedia-controlled Web APIs

One thing we require in order to solve the findability problem is to have self-describing APIs. We need a framework to describe services' interfaces in a machine-processable way, so that clients, in order to consume an API, do not have to be developed specifically against it but in a generic way. WSDL tried to address this for SOA-based services to a certain extend, however, only on a syntactical level and therefore not allowing for truly generic clients. For REST there are hypermedia controls as a first step towards solving this. REST favors convention over configuration and therefore a resource's (as well as its related resources') provided actions accrue from its current state instead of from a comprehensive description. An API using hypermedia controls - or hypermedia as the engine of application state (HATOAS), as commonly referred to - corresponds to the third highest level of maturity in the Richardson Maturity Model (RMM) [35]. With hypermedia controls, a resource includes links to related resources and thus the API can be discovered incrementally. For instance, a sensor's observation resource may include a link to another resource representing the observed part, which in turn may include another link to the corresponding customer's order. Combined with properly implemented HTTP verbs and status codes - which corresponds to the second level in the RMM - a client can discover resources and interact with them automatically.

4.5.3 Semantic Web APIs

However, the pure use of hypermedia controls is not sufficient for all cases either. Clients still need information on the meaning of the linked resources. Having fetched a resource http://example.org/observedPart735 that includes a link to http://example.org/order97 does not tell the client what the linked resource represents. Different from a human browsing the web, a machine is not able to interpret the string and consequently does not know that the linked resource probably refers to a customer order. Some kind of semantics need to be added to the hypermedia links, so that a machine-client looking for customer orders knows that it probably should follow that specific link and no other. We already have introduced semantic annotations in chapter 4.4, where we required all payload data emitted by services

to follow Linked Data principles. Consequently, the payload data itself already is given a ”meaning”. However, when discovering an API, that meaning is unknown to a client until it actually receives some data. Since, apparently, its unaffordable to first wait for some data at every API route to decide whether its useful or not, the semantic annotations need to apply to the APIs hypermedia links, too, resulting in full stack semantic annotations. Furthermore, the terms describing the API itself, e.g. what operation to perform on which route, may also refer to a common vocabulary, so that there commonly understandable. Whereas previous approaches like WSMO or SAWSDL addressed exactly this for SOA services, to the best of our knowledge, there has not been an accepted solution for REST services around, yet. However, a few hypermedia formats for web APIs do exist. We subsequently present them, go into some of their details and do a comparison with regard to their key aspects before choosing one of them to be used for our suggested architecture.

4.5.4 Hypermedia Formats

According to Sporny, a common problem with today's web APIs is the lack of a common standard to use HATEOAS [36]. A few attempts to establish such a standard format include HAL[1], Collection+JSON[2], JSONAPI[3], Web Things[4] and Hydra[5]. Web Things is a format defined by Guinard and Trifa [19], which is a reference IoT platform architecture very similar to ours. Therefore Web Things is specifically suited for IoT applications. Table 4.1 compares these formats with respect to key aspects.

To understand the table, a short explanation of the respective criteria follows.

Embedded resources. This aspect refers to whether the hypermedia format supports to include child resources to parent resources within one request. For example, assume an ”post” resource representing a social media posting, that has a property ”text” as well as a sub-resource ”author”, which again comprises the properties ”name” and ”email”. When a user fetches the article resource, the ”author” property is just another link and he would have to perform another request to get hypermedia controls of the author object. With embedded resources, they would already be included to the parent object. Of course, the API itself needs to implement this functionality first.

Flattening. By default, resources are embedded to other resources recusively. That means if, for instance, one person is referenced from two different resources, the resulting document would contain that person resource twice. With flattening, there would only be one object for this person in the result graph, which then is referenced internally (technically using RDF blank nodes). This decreases redundancy and network overhead and speeds up processing time.

Supported operations. This point is about whether the hypermedia allows to specify a link's destination's supported operations. Given the ”article” resource

[1] http://stateless.co/hal_specification.html
[2] https://github.com/collection-json/spec
[3] http://jsonapi.org/
[4] https://www.w3.org/Submission/2015/SUBM-wot-model-20150824/
[5] http://www.hydra-cg.com/

Table 4.1: Comparison of hypermedia formats

	Hydra / JSON-LD	HAL	Collection+JSON	JSONAPI	Web Things
Embedded resources	Yes	Yes	Yes, with queries	Yes	No
Flattening	Yes	No	No	Yes	No
Supported operations	Yes	Implicitly, with templates	Yes, with templates	No	Yes
Supported properties	Yes	Implicitly, with templates	Yes, with templates	No	Yes
URL templates	Yes	Yes	Yes	No	No
Simplicity	+	+	-	++	++
Ease to adopt in existing APIs	++ with @vocab	+	-	+	—
Vocabulary / schema	Yes	No	No	Yes	No
Support for CURIEs	Yes	Yes	No	No	No
Number of implementations	-	+++	-	–	—
Mimetype	application/ld+json	application/hal+json	application/collection+json	application/vnd.api+json	unspecified

again, if this aspect is fulfilled, the client would also be provided with information on which (HTTP) methods are allowed on the ”author” resource without a further request. The important thing to note here is that this refers to being able to know the operation in advance (before fetching the other resource). For example, the authors of JSONAPI tell that they also support to get a resource's operation by GETing and analyzing the ”Allow” header field. This is exactly not what we mean by ”supported operations”, because with JSONAPI, the resource needs to be fetched first.

Supported properties. This point is similar to the preceding one and refers to the support to view a resource's properties (fields) in advance without the need to dereference it. Apparently this is also accomplished when having embedded resources.

URL templates. This aspects refers to the support of the hypermedia format to specify templates for URIs including placeholders for variables / parameters. There might be cases where a URL is not known up-front, but is determined at runtime. This especially occurs when arguments need to get passed in the URL.

Simplicity. This aspect is quite subjective and scores our perceived simplicity of using the respective format as well as the readability and clarity of a document formatted that way. Please note that the scores for this aspect, as well as for ”Ease to adopt in existing APIs” are not quantitatively measured, but represent a subjective perception.

Ease to adopt in existing APIs. This characteristic states how much effort need to be taken to modify an existing API in order to introduce the respective hypermedia format to it.

Vocabulary / schema. This one states whether there is a formal, machine-processable specification of the hypermedia format's properties and terms, e.g. using JSON-Schema.

Support for CURIEs. A CURIE, very basically, is a shortcut. This is especially crucial in the context of Linked Data, where every object is identified by a fully qualified URL. Assume the ”article” resource again, but this time assume that is of type ”https://schema.org/Article”, the ”author” property actually refers to the ”https://schema.org/author” property and the ”text” now became ”https://schema.org/articleBody”. A list of articles in JSON could look like depicted in listing 4.1. With CURIEs, the full URL would be mapped to a shortcut, e.g. ”author” again, so that the actual objects get more readable and contain less overhead, as depicted in figure 4.2. Please note that these listing use pseudo-syntax, which does not correspond to any of the presented hypermedia formats.

Number of implementation. This aspect scores how well the respective format has already been adopted, how many times it could be found used in production and how many open-source programming libraries can be found on the internet.

Mimetype. This is the mimetype that indicates a document to be using the respective hypermedia format.

Listing 4.1: JSON-LD sample without CURIEs

```
[
        {
                "id": "http://example.org/articles/1",
                "type": "https://schema.org/Article",
                "https://schema.org/author": "http://example.org/authors
                    /1",
                "https://schema.org/articleBody": "Some text"
        },
        {
                "id": "http://example.org/articles/2",
                "type": "https://schema.org/Article",
                "https://schema.org/author": "http://example.org/authors
                    /2",
                "https://schema.org/articleBody": "Some other text"
        }
]
```

Listing 4.2: JSON-LD sample with CURIEs

```
{
        "context": {
                "ex": "http://example.org"
                "article": "https://schema.org/Article",
                "author": "https://schema.org/author",
                "text": "https://schema.org/articleBody"
        },
        "article": [
                {
                        "id": "ex/articles/1",
                        "type": "article",
                        "author": "http://example.org/authors/1",
                        "text": "Some text"
                },
                {
                        "id": "ex/articles/2",
                        "type": "article",
                        "author": "ex/authors/2",
                        "text": "Some other text"
                }
        ]
}
```

For our targeted scenarios we consider the aspects ”Embedded resources”, ”Flattening”, ”Simplicity” and ”Ease to adopt in existing APIs” nice-to-have, but not truly important, ”URL templates”, ”Vocabulary / schema” and ”Number of implementations” crucial and ”supported operations”, ”Supported properties”, and ”Support for CURIEs” essential. Not meeting the essential requirements is a disabler and prohibits from the format being used. Considering this ranking in combination with the comparison in table 4.1, Collection+JSON, JSONAPI and Web Things are out. However, due to its simplicity and excellent documentation, especially JSONAPI appears to be a very well-suited, promising hypermedia format for non-semantic API use cases. Hydra and HAL remain for further investigation. HAL’s main benefit is its adoption. It even has become an almost de-facto standard for hypermedia APIs and is supported by a large amount of open-source libraries for a large variety of programming languages[6]. This is a major downside of Hydra - it is still very experimental and lacks of implementation and community support. However, it appears to exactly fit our needs. Hydra as a vocabulary in combination with JSON-LD as the serialization format as a complete solution for describing web APIs, natively using Linked Data techniques, and allows for generic clients [37] as shown by the Hydra Console[7]. According to Thoma, Sperner and Braun, in order to successfully

[6] https://github.com/mikekelly/hal_specification/wiki/Libraries

[7] http://www.markus-lanthaler.com/hydra/console/

integrate sensor networks into enterprise IT (however, note that we are not only facing sensor integration but arbitrary other services, too), it is necessary to have at least technical and desirably also non-functional descriptions of a sensor's service on how to invoke it [38]. Using Hydra provides both in one. For these reasons we pick Hydra as our hypermedia framework of choice.

4.5.5 Discovery

What we achieved by introducing semantics to payload data and semantic-enhanced hypermedia to APIs is that services can consume other services completely autonomous without having to be aware of their respective interfaces. We actually enabled completely generic clients. However, there is one more challenge to be mastered towards solving the findability problem. Given a service's Hydra API, a Hydra-aware client can consume it, but first it even needs to find that API. This is referred to as service discovery. Commonly, services discovery is realized through a central, yellow-pages-like service registry that maintains a list of all available services, their endpoints and capabilities. In the context of WSDL-based services, UDDI is commonly used for that purpose. Usually, a client sends a service request to a registry and receives a list of service advertisements. A request contains information on capabilities the client is looking for. The registry matches the request against a set of advertisements which the services published and eventually returns a subset based on that match. The client then chooses one of these services and consumes it. The service triangle in figure 4.7 illustrates these relations. Basically we face three potential solutions to perform service discovery in a Microservice environment.

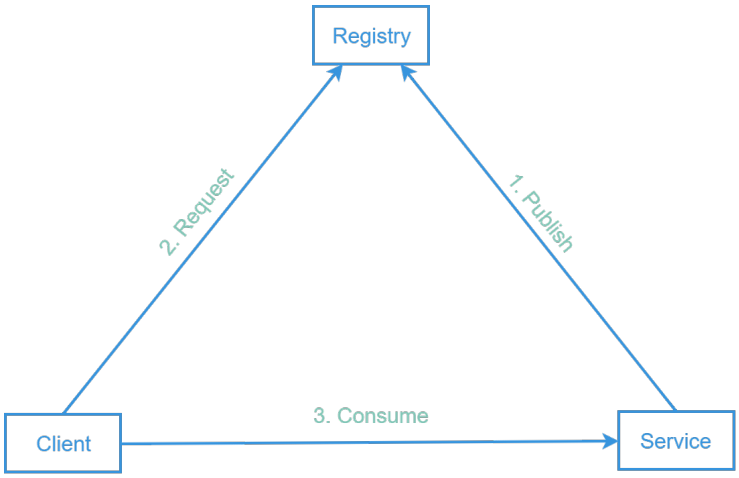

Figure 4.7: Service triangle

4.5.5.1 Self-registration

This approach is proposed for service discovery within the Web Of Things [19] using mDNS for concrete implementation. it is probably the simplest way to do service discovery and works as follows. When a new service comes up and is registered to the network, it usually gets assigned an IP address by a DHCP server. Afterwards it sends a UDP broadcast message containing its local IP and potentially its hostname. The broadcast is sent to every other client in the network, which subsequently can parse the broadcast message and add the newly joined service to their internal list

of available service. However, there are some downsides with this approach. First, problems can occur when having multiple, separate networks within an organization. Second, after a new service having sent the broadcast, all other services might know it, but the new service still does not know any of them. A solution might be that, having received a new broadcast, every other service also emits one, which the new service finally receives. Another solution might be that all services directly contact the new one and introduce themselves. However, either approach results in quite an amount of network traffic and management effort.

4.5.5.2 Client-side discovery

The term "client-side discovery" was coined by Richardson [39], while the underlying concept is well-known and commonly used. This approach uses a central service-registry, however, not as comprehensive as UDDI within SOA-environments. A new service is configured with the registry's URL and registers itself with its IP and hostname when started. All other services have access to the registry, too, and can fetch a list of available services. In contrast to, for instance, UDDI, the registry only has to manage services' location (IP / hostname), but no detailed description, because such can be fetched directly from their respective API descriptions. This approach is very straightforward, but relies on a central point of management. However, such a registry is very lightweight on the one hand and can be deployed highly distributed and redundant on the other in order to minimize the chance of failures or bottlenecks. Therefore the approach is considered far more resilient than self-registration or discovery by traversal, but depends on clients to be coupled to the registry. Popular technologies to realize a Microservice-registry include Netflix Eureka[8] and Consul[9].

4.5.5.3 Discovery by traversal

With this approach a service only needs to be given an initial seed URL which it takes as a starting point to then discover other services' APIs by derefencing hypermedia links. A downside of this approach is that it is not necessarily deterministic. The graph representing the network of services in a particular scenario is directed and not necessarily strongly connected. That means that, on the one hand, there might be nodes with an indegree greater than zero and an outdegree of zero. On the other hand, not every sub-graph is reachable from within any other sub-graph. Consequently, a client could terminate its traversal without having found a suitable service, even if there exists one. Another downside is that the approach implies a large amount of network traffic and many round trips and therefore is expected to be comparatively slow.

4.5.5.4 Conclusion

Since we committed to not having any kind of central control instance in our Microservice architecture, we quit to have a service registry, although the approach would probably be the simplest. Instead we rely on discovery by traversal. Our decision to relinquish a central registry is underpinned by the work of Thoma, Sperner and Braun, according to which a central registry is only suitable for static network

[8] https://github.com/Netflix/eureka
[9] https://www.consul.io

scenarios rather than ad-hoc, self-organizing ones. [38] Furthermore, taking this approach can, to some extent, act as kind of a proof-of-concept for our services' hypermedia controls. However, due to its risks and downsides, discovery by traversal is probably not suitable for production, so we keep the option open to still include a service registry after all. Accordingly, we decide to employ client-side discovery as a second option.

4.5.6 Summary

In this chapter we argued the options and our choice to use semantically annotated hypermedia controls for service APIs and to employ traversal-based, autonomous service discovery as well as client-side service discovery. In combination with respecting Postel's law - which says "be conservative in what you do, be liberal in what you accept from others" - during application design, these techniques allow for dynamic service composition. Consequently, a service is able to find other services that it requires, just-in-time and only given required in- and output parameters. This way, service mashups comprising multiple Microservices can be build up at runtime and with a minimum of configuration effort.

4.6 Apps as Building Blocks

As a concrete implementation of Microservices, we introduce the user experience (UX) concept of "apps". An app is a realization of the more abstract concept of a Microservice and thus a cohesive unit of service for the implementation of a business context. Especially, an app is a self-contained, isolated component with a single responsibility and aims at simplifying deployment and management by providing a high-level abstraction. Everything that has been called a "service" throughout this document now is referred to as an "app". For instance, an app could be:

- a frontend application running in a user's browser, e.g. a business intelligence (BI) dashboard
- an application running on a low-hardware sensor device, processing and publishing observation values
- a backend-only analytics application running on a server that gets queried by user-facing apps
- a smartphone app

Especially, an app can either represent the interface to a physical "thing", since we are facing IoT scenarios, or just be an ordinary software application. Regardless of which of these various shapes an app takes, it still should look uniform from the outside (in terms of its interface and in-/out data) and be able to communicate with any other app. A manager's smartphone app should be able to discover and consume a sensor app, just as the app on an autonomous shop floor vehicle is able to interact with its counterpart on the next machine within the production line. Given the flexibility achieved with the architecture presented in previous sections, such apps can easily be realized.

4.6.1 Compliance Requirements

This chapter presents a set of requirements, guidelines and conventions to be followed when implementing apps for the present platform proposal in order to adhere to the principles stated in previous chapters. The following list combines requirements declared for Web Things, guidelines described by "The twelve factor app" [40] and own considerations. While the requirements stated in the SUBM WoT Model[10] are perfectly suitable for apps on physical things, they do not sufficiently apply to all potential kinds of apps in our target scenarios. The following list is still heavily influenced by them.

Similar to the the SUBM WoT Model[11], we define three levels of requirements. Level 0 requirements are essential and inevitably must be fulfilled, level 1 requirements should be fulfilled and level 2 requirements are optional. Some requirements, marked with a star (*), only apply to non-client apps that expose data. Client apps (only consisting of a frontend) have softer requirements. They correspond to leaves (out-degree of zero, since they can not be queried for data) in a directed graph that represents all possible data flow in a scenario consisting of multiple apps.

4.6.1.1 Compliance Level 0

An app must ...

- ... be at least HTTP/1.1 capable.
- ... be uniquely identifiable and by a URL. *
- ... have a root resource accessible via an HTTP URL. *
- ... expose a REST API. *
- ... accord to the highest level of maturity in the RMM. *
- ... fulfill all four Linked Data principles. *
- ... implement passive communication.
- ... use JSON or JSON-LD as default representation format.
- ... expose RDF-structured data with semantic annotations. *
- ... describe its API using the Hydra vocabulary.
- ... offer a human-facing HTML representation or user-interface.
- ... be stateless. *
- ... explicitly declare its dependencies, instead of relying on system-wide packages.

[10] https://www.w3.org/Submission/2015/SUBM-wot-model-20150824/
[11] https://www.w3.org/Submission/2015/SUBM-wot-model-20150824/

4.6.1.2 Compliance Level 1

An app should ...

- ... have a bounded context.
- ... be designed to be reusable.
- ... run in a container.
- ... be developed and operated by exactly one development team.
- ... comprise its own database. *
- ... have as less configuration as possible.
- ... read its configuration from either environment variables or a web service.
- ... use the technology stack best suitable for its purpose.
- ... be available for installation in an app store.
- ... have a separate code base tracked in revision control.
- ... follow the TolerantReader pattern [41].

4.6.1.3 Compliance Level 2

An app may ...

- ... implement more than one passive communication mechanisms.
- ... have logging.
- ... expose its logs as an event stream.
- ... use secure connections (HTTPS).
- ... support representation formats besides JSON. If so, it must properly implement content negotiation.

4.6.2 App Store

In chapter 4.1 we required our platform to be easy to use. Shop floor employees without a technical background in information technology and software engineering should be able to deploy functionality without instructions. To support this, it is helpful to rely on familiar concepts, such as apps. Also familiar in the context of apps, especially mobile smartphone apps, is an app store. "An app store (or app marketplace) is a type of digital distribution platform for computer software. [...] App stores typically take the form of an online store, where users can browse through these different app categories, view information about each app [...], and acquire the app [...]. The selected app is offered as an automatic download, after which the app installs." [42] Having an app store as a key component of out platform enables users to select among various apps according to their need and install them without knowledge about the technical backgrounds. Figure 4.8 shows a mockup of our objective of an appstore for the present platform.

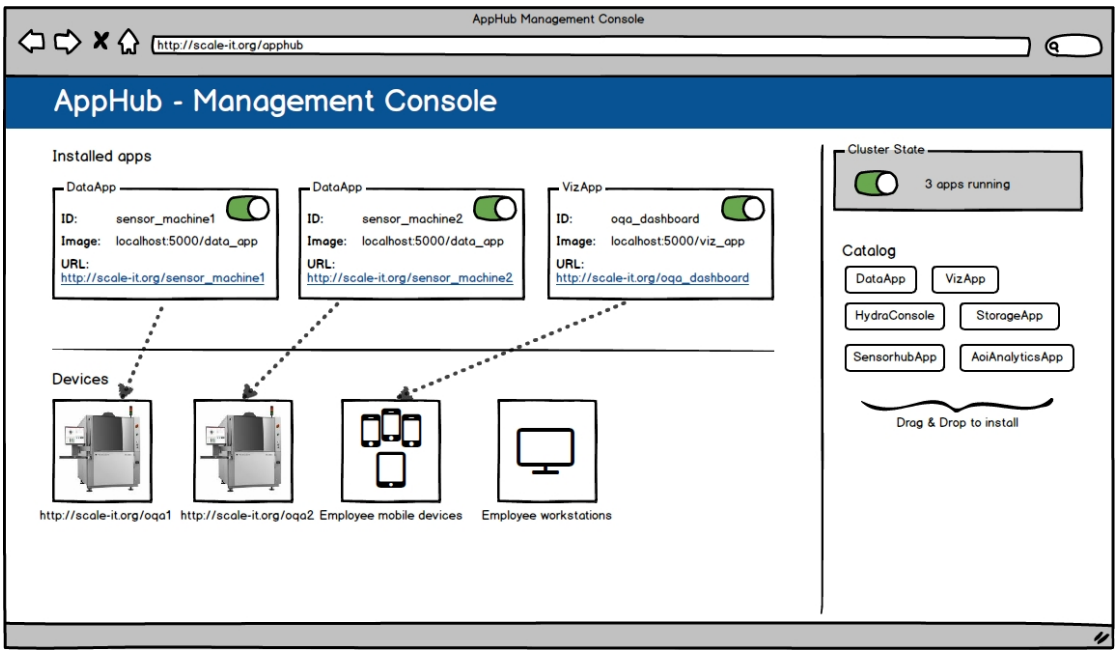

Figure 4.8: AppHub mockup used for installing and deploying shop floor apps

4.7 Conclusion

This chapter stated requirements crucial to the proposed software platform, subsequently discussed various options regarding architectural structure, communication, data integration and cooperation between components and finally proposed respective implementation guidelines. As a conclusion we will pick up on the requirements listed in chapter 4.1 and map them to our architectural decisions.

Uniformity. We required uniformity in terms of application interfaces and protocols in order to support interoperability and facilitate integration with further systems. Addressing this requirement we decided to employ HTTP as the most uniform and widely spread communication protocol together with commonly-understood communication patterns and self-describing APIs in order to support generic clients in addition.

Just-in-Time Capability. In order to meet the just-in-time-requirement implied by the IoT, we introduced event collaboration and the publish / subscribe pattern. These techniques enable applications to receive relevant data just when it arises.

Ease of use. In order to make our software tangible for less experienced, non-technical users and require a minimum of upfront knowledge, we introduced the commonly understood concept of apps. We required these apps to support dynamic service composition and be highly autonomous to ban configuration effort. Furthermore we proposed the presence of an app store to install / deploy new functionality with just a few clicks.

Data centricity. By employing Linked Data techniques and requiring all data to comprise semantic annotations, we created a capable basis for data integration and abstracted away the actual data source.

Scalability. Scalability is mainly achieved by the use of Microservices that are loosely coupled, designed for failure, organized autonomously and decentralized and can easily be replicated and reused.

5. Implementation

Chapter 2 presented an abstract field of application for the present software platform. Chapter 4 then discussed key architectural questions concerning software structure, communication, data integration and cooperation between components. In this chapter we pick up on the design decisions made previously as well as on the abstract scenario from chapter 2 and combine these thoughts in an actual implementation. Subsequently we implement an example use case as a demonstrator, consisting of several apps as building blocks. We show how exemplary real-world objectives can be realized with these apps and how they are implemented.

5.1 Demonstrative Scenario

Figure 5.1 depicts the example scenario we are going to implement. It is based on the SensorCompany use case outlined in chapter 2 and consists of several components.

OQA machines. Machines performing automatic optical inspection are the major data sources. They emit JSON-serialized data containing information on quality of the currently observed part, alongside some meta data. Since we do not have access to these actual OQA machines, we just emulate them with a script that produces randomized values based in the actual OQA output data structure.

DataApp. A DataApp implements the app concept presented in chapter 4.6 while adhering to its principles. It is intended to run directly on an OQA sensor. A DataApp is responsible for taking the raw sensor output, annotating it (see listing 3.2 in chapter 2) and making it available to be fetched by further apps. Apparently, it is a data-producing app and therefore required to adhere to all principles from 4.6.1. Especially it exposes a REST API to request for data, it supports webhook- and SSE-based subscription mechanisms and includes an HTML representation. An app that subscribes to the DataApp will be pushed the latest OQA observation value as it arises. For simplicity reasons, this demo DataApp also includes the emulator script to produce random OQA data.

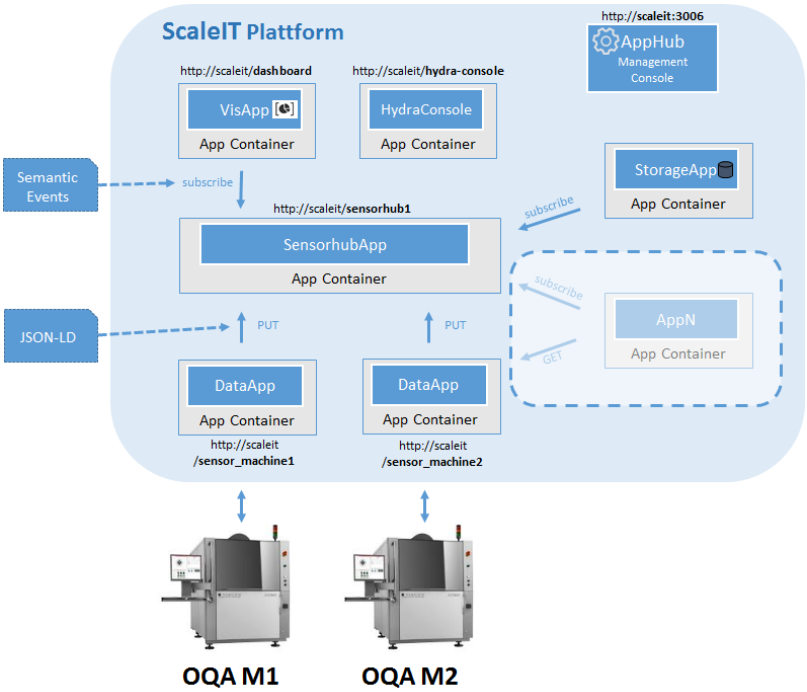

Figure 5.1: Example scenario for OQA use case

SensorhubApp. This app acts as kind of a gateway to DataApps. Usually, one might not be interested in data from only one OQA DataApp, but from all. To not have to take the burden of managing multiple subscriptions to every DataApp, other apps could simply subscribe to the SensorHub app. This unloads both the eventual clients as well as the low-hardware DataApps. However, since we strived for a possibly decentralized architecture, this app is not mandatory at all and only introduced for simplicity. It provides a REST API, two types of subscriptions mechanisms and an HTML representation, too. Further apps can subscribe to semantic events at the SensorHub and get pushed respective data if the event is fired.

VisApp / DashboardApp. This app is a non-producing, end-user-facing frontend app running in a web browser. It can subscribe for particular data or events at procuding apps (e.g. SensorhubApp or DataApp) and then visualize them graphically.

HydraConsole. Another app is the third-party Hydra Console. It is a project maintained by the developers of Hydra and acts as a proof-of-concept for a generic Hydra client. As such it is a fronted application able to browse arbitrary Hydra APIs. Figure 5.2 depicts the HydraConsole user interface browsing a DataApp's API.

StorageApp. Usually, data not only needs to be consumed just-in-time but it needs to be persisted. For this purpose, we included the StorageApp. Internally it is another third-party application, namely the Apache Fuseki RDF triple store projects. It is intended to simply subscribe to all data emitted inside our scenario, persist it and so make it queryable through Fuseki's SPARQL endpoint.

AppHub. This application is special in that it is integral part of the platform and therefore is not an app like the above. Actually it is not an app at all, but a

native application. It implements the app store concept presented in chapter 4.6. It is a graphical, user-facing management interface to deploy and administer apps in the cluster and maintains a list of all available apps. An end-user can open the AppHub in his browser, choose from the collection of apps and one-click instantiate and deploy them. Further technical details are explained subsequently.

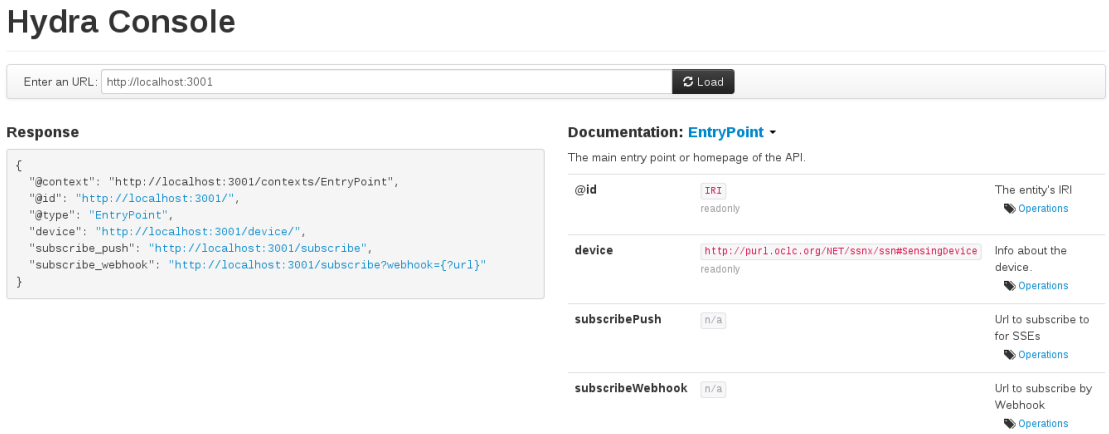

Figure 5.2: Browsing DataApp's API using HydraConsole

5.2 Technology Stack

We choose NodeJS v6.4 as runtime environment for the AppHub and all custom implemented apps (1, 2 and 3). It makes use of the JavaScript programming language and ports it to be used server-sided. Guinard and Trifa consider NodeJS the most suitable programming platform for IoT purposes, right after native C [19]. It offers great flexibility and portability, while yet being performant for web applications due to its asynchronous, single-process event-loop. Additionally, a major benefit is the vibrant open-source community support, offering a large amount of libraries and packages. To implement the HTTP server, the REST API and respective middleware, we included the Express 4 web framework[1].

5.3 Pub/Sub Technologies

In chapter 4.3 we required all of our service to support both active and passive communication patterns, while we left open, which specific technology to use for the latter. We are given several options, which are compared in table 5.1.

5.3.1 Discussion

Given table 5.1 we want to do a short discussion on each of the listed aspects.

Data flow describes whether data can be passed uni- or bi-directional. With bi-directional duplex communication, data can not only be pushed from server to client, but also the other way round. For our purpose, uni-directional communication is

[1] http://expressjs.com/

Table 5.1: Comparison of web-based pub / sub technologies

	Long-polling	Websockets	WebRTC data channel	Web hooks	Server-sent events
Data flow	Simplex	Duplex	Duplex	Simplex	Simplex
Built into HTTP	Yes	No	No	Yes	No
Library overhead	None	Moderate	Extremely high	None	Moderate
Binary payload	Yes	Yes	Yes	Yes	No
Webserver required	Yes	Not necesarily	No	Yes	Yes
TCP connection kept alive	No, closed and re-opened after every response	Yes	Yes	No	Yes, but may time out
Overhead per message	High	Low	Low	High	Moderate
RTT	High for request, low for response	Low	Low	High	Low
Implementation effort	Low	Moderate	High	Moderate	Low

sufficient, since events are only considered to be sent by the server. **"Built into HTTP"** means that no further libraries are required and the technology can be realized with plain HTTP methods. Therefore no **library overhead** is added. This is the case for long-polling and web hooks. Long-polling basically uses a client-server request that is kept open until the server has new data. Web hooks imply server-client requests, while clients expose their own API. Not necessitating a library is good, but actually not crucial at all. The support of **binary payload** is especially essential if advanced, binary-based serialization formats were introduced instead of sending JSON over the wire, because if the communication channel natively support binary data, there is no overhead due to utf-8 encoding in such cases. Whether a **webserver** is **required** or not to use the respective technology is not that important in our case, since all of our apps include a webserver anyway. If the technology **keeps TCP connections alive** over the whole interaction time, less overhead accompanied by connection establishment and handshaking is added and therefore performance is improved. Since we aim for just-in-time communication, this aspect is important to us, as well as the **overhead per message** and the **round trip time (RTT)**, which are directly related to that. The **implementation effort** describes how complicated it is to adopt the respective technology in an application. Long-polling just uses plain HTTP requests and SSEs and Websockets require only a few lines of code to realize subscription, while they are natively supported in any modern browser in addition. WebRTC data channels, since they work peer-to-peer, have need for a signaling- and a STUN/TURN server for relaying and therefore are harder to adopt. Web hooks are restricted in a way that they only work for machine-to-machine communication, since browser apps can not expose an API. Nevertheless, the implementation effort in our case is only moderate, since all backend-sided apps have APIs anyway, which only need to be extended with a further route.

5.3.2 Conclusion

Due to its good just-in-time capability, as well as the low implementation effort, native browser support and standardization through HTML5, we decide to use SSE as a central pub/sub technology for all of our apps. Additionally web hooks are used as a second option for server-to-server pub/sub communication. In contrast to the Web of Things architecture from Guinard and Trifa [19], who realize pub/sub between multiple IoT devices using a central broker, we relinquish such and perform subscriptions directly to those devices.

On the backend side, web hooks do not necessitate any further library for implementation. For SSEs we use the server-event library[2] for sending and the EventSource library[3] for receiving SSEs. On the frontend side (in the browser), SSE do not require external dependencies.

5.4 App Interface Design

Since all producing apps are required to adhere to the same principles, they basically share the same interface implementation, regardless of their eventual purpose. The interface consists of a Hydra-annotated REST API on the one hand and at least an SSE endpoint for subscriptions on the other. All presented producer apps additionally have a webhook endpoint for subscriptions.

[2]https://github.com/kevinmehall/server-event
[3]https://github.com/aslakhellesoy/eventsource

5.4.1 Active and Passive Communication

In chapter 4.3 we introduced active and passive communication, utilizing the req/res and pub/sub patterns. Now, this chapter presents how they actually take place in the apps' implementations. Assuming the DataApp is listening at http://scaleit/sensor_machine1, the content shown in listing 5.1 is returned as a response to a GET request to that URL, alongside a Content-Type header set to "application/ld+json". This returned JSON-LD data is a Hydra description of the DataApp's REST API at its root endpoint. The other apps' APIs are designed analogously. Dereferencing the @context URL brings up the JSON-LD context depicted in listing 5.2.

Listing 5.1: DataApp's root endpoint

```
{
  "@context": "http://scaleit/sensor_machine1/contexts/EntryPoint",
  "@id": "http://scaleit/sensor_machine1/",
  "@type": "EntryPoint",
  "device": "http://scaleit/sensor_machine1/device/",
  "subscribe_push": "http://scaleit/sensor_machine1/subscribe",
  "subscribe_webhook": "http://scaleit/sensor_machine1/subscribe?webhook={?
      url}"
}
```

Listing 5.2: Dereferenced context for DataApp's root endpoint

```
{
  "@context": {
    "hydra": "http://www.w3.org/ns/hydra/core#",
    "vocab": "http://scaleit/sensor_machine1/vocab#",
    "ssn": "http://purl.oclc.org/NET/ssnx/ssn#",
    "EntryPoint": "vocab:EntryPoint",
    "device": {
      "@id": "vocab:EntryPoint/device",
      "@type": "@id"
    },
    "subscribe_push": {
                "@id": "vocab:EntryPoint/subscribe_push",
                "@type": "@id"
        },
        "subscribe_webhook": {
                "@id": "vocab:EntryPoint/subscribe_webhook",
                "@type": "@id"
        }
  }
}
```

This is how apps actively interact with each other. They request other apps' API root, parse and "understand" their Hydra descriptions and possibly perform further requests according to that description. In case of the DataApp, a GET request to the /device route brings up meta information on the sensor device as well as the latest generated observation data as shown in figure 5.3. Again, this corresponds to active communication introduced in chapter 4.3.3.

SSE subscriptions are registered by POSTing to the *subscribe_push* endpoint while webhook-based subscriptions are registered using the *subscribe_webhook* endpoint with a POST. As a result to one of these POSTs, the requesting app will subsequently receive data updates either via SSE or at its webhook. Note that the Hydra annotation describing *subscribe_webhook* points to a URL template, in which the *?url* placeholder needs to be replaced by a webhook url. This flow corresponds to passive communication introduced in chapter 4.3.2 and is depicted in more detail in figure 5.4 for both alternative methods.

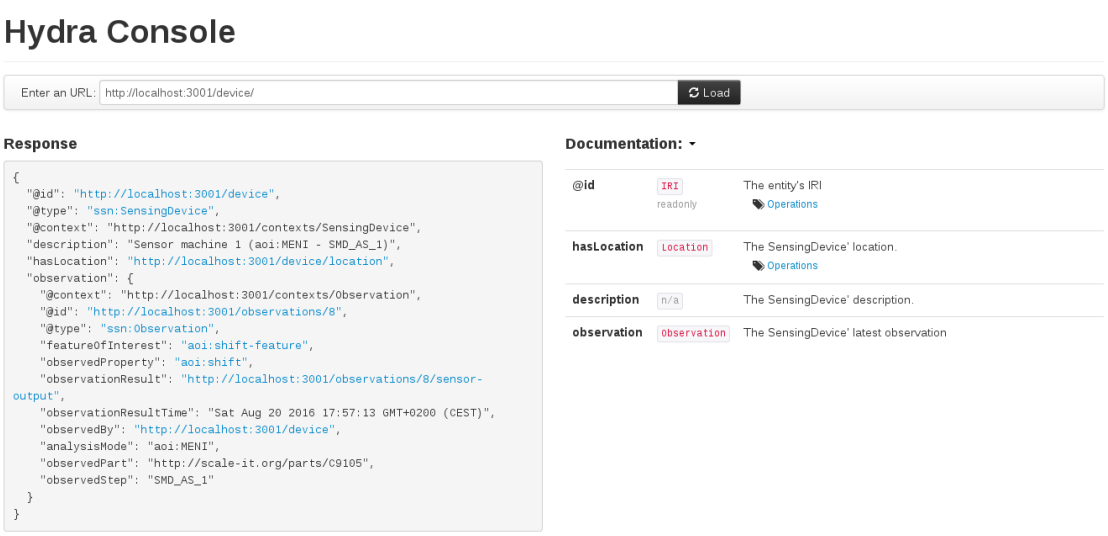

Figure 5.3: HydraConsole showing DataApp's device- and observation information

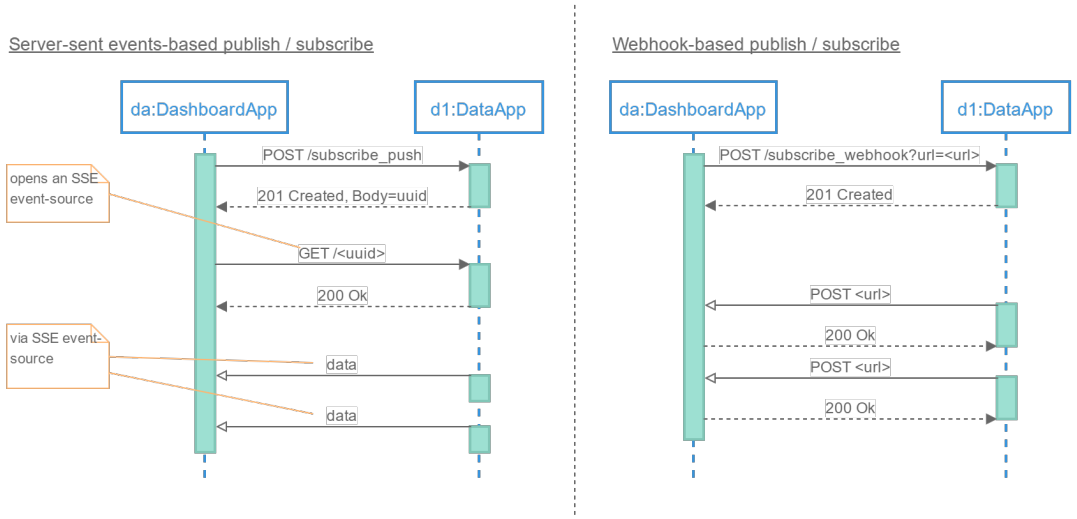

Figure 5.4: Communication flow with server-sent events and webhooks

5.4.2 Self-describing Hypermedia APIs

To get an understanding of how the Hydra-powered, hypermedia-controlled APIs - as proposed in chapter 4.5.2 - work in action, we investigate the DataApp's */device* route opened inside HydraConsole (figure 5.3). In this context, we further analyze the actual payload data emitted by a DataApp to clarify how semantic annotations can help with data integration as proposed in chapter 4.4.

As described in chapter 5.1, the HydraConsole is a generic client that is able to consume any API only given its Hydra description. We use HydraConsole to browse the DataApp's API and are provided with meta information on operations and respective in- and output parameters for each route / resource. As shown in figure 5.3, the DataApp exposes basic information about itself (id, type, description, location) as well as the latest observation data. All exposed data include semantic annotations. For instance, the DataApp represents a ssn:SensingDevice, where "ssn" is mapped to the sensor network ontology (http://purl.oclc.org/NET/ssnx/ssn#) inside the context located at "http://localhost:3001/contexts/Observation". In ad-

dition to the actual payload data, HydraConsole also displays which operations to perform at which route. For instance, in figure 5.3, a click on "http://localhost:3001/observations/8/sensor-output" reveals that this route only supports a "GET" operation. If a route additionally supports write operations (like POST), Hydra would automatically render HTML input fields for the respective parameters required to POST data to that route. Hydra extracts these information from the so called "Hydra vocab", which is referenced by "vocab" at the entry point route (see figure 5.2). The vocab is another JSON-LD document containing all information about the Hydra API. Figure 5.3 depicts an excerpt from the DataApp's Hydra vocab.

Listing 5.3: Excerpt from DataApp's hydra vocab

```
[
  {
    "@id": "ssn:SensingDevice",
    "@type": "hydra:Class",
    "supportedProperty": [
      {
        "property": {
          "@id": "vocab:hasLocation",
          "@type": "hydra:Link",
          "label": "hasLocation",
          "description": "The device's location",
          "domain": "ssn:SensingDevice",
          "range": "vocab:Location",
          "supportedOperation": [
            {
              "@id": "_:location2_retrieve",
              "@type": "hydra:Operation",
              "method": "GET",
              "label": "Retrieves a Location entity",
              "description": null,
              "expects": null,
              "returns": "vocab:Location",
              "statusCodes": [ ]
            }
          ]
        },
        "hydra:title": "hasLocation",
        "hydra:description": "The SensingDevice' location.",
        "required": true,
        "readonly": false,
        "writeonly": false
      },
      {
        "property": "so:description",
        "hydra:title": "description",
        "hydra:description": "The SensingDevice' description.",
        "required": true,
        "readonly": false,
        "writeonly": false
      },
      {
        "property": {
          "@id": "oqa:observation",
          "@type": "rdfs:Property",
          "label": "observation",
          "description": "The device's latest observation",
          "domain": "ssn:SensingDevice",
          "range": "ssn:Observation"
        },
        "hydra:title": "observation",
        "hydra:description": "The SensingDevice' latest observation",
        "required": true,
        "readonly": false,
        "writeonly": false
      }
    ],
    "supportedOperation": [
      {
        "@id": "_:info_retrieve",
        "@type": "hydra:Operation",
```

```
        "method": "GET",
        "label": "Retrieves the info.",
        "description": null,
        "expects": null,
        "returns": "ssn:SensingDevice",
        "statusCodes": []
      }
    ]
  },
  {
    "@id": "ssn:Observation",
    "@type": "hydra:Class",
    "subClassOf": null,
    "label": "Observation",
    "description": "An observation entity.",
    "supportedOperation": [],
    "supportedProperty": [
      {
        "property": {
          "@id": "ssn:observationResult",
          "@type": "hydra:Link",
          "label": "observationResult",
          "description": "The observation result",
          "domain": "ssn:Observation",
          "range": "ssn:SensorOutput",
          "supportedOperation": [
            {
              "@id": "_:result_retrieve",
              "@type": "hydra:Operation",
              "method": "GET",
              "label": "Retrieves the result.",
              "description": null,
              "expects": null,
              "returns": "ssn:SensorOutput",
              "statusCodes": []
            }
          ]
        },
        "hydra:title": "observationResult",
        "hydra:description": "The observation result",
        "required": null,
        "readonly": true,
        "writeonly": false
      }
    ]
  }
]
```

This fundamentally is what we referred to as hypermedia controls in chapter 4.5.2. The Hydra descriptions augment API resources with information on how to process them as well as on related resources and their actions. The fact that Hydra itself is an ontology even makes these descriptions ”semantic” hypermedia controls. All terms to describe an API are commonly defined and therefore understandable by every client that has the Hydra ontology on hand. For instance, every operation at every API is given the class ”http://www.w3.org/ns/hydra/core#Operation”, or ”hydra:Operation” in short. These hypermedia controls combined with the semantic payload data can enable clients to autonomously browse services and discover data. As an example, assume a client-side dashboard application that was told by the user to visualize any data related to OQA shifts that were created using the MENI analysis mode. Using the traversal-based discovery method from chapter 4.5.5.3 the client would start at some seed URL and traverse the respective service's API. It would look for resources that contain ”http://scale-it.org/oqa#shift” as value to a ”http://purl.oclc.org/NET/ssnx/ssn#observedProperty” key and http://scale-it.org/oqa#MENI” as value to a ”http://scale-it.org/oqa#observedStep”. As usual with Linked Data, every service publishing data on such OQA shifts would annotate it with reference to exactly that ontology. Therefore data from all kinds of sources

could easily be integrated. Having arrived at a service's API, the client could walk recursively through all links (either using a breadth- or depth-first algorithm), which especially may also include links to resources from other services. Continuing like this potentially leads the client to all linked services (apps) available within the company, until it finally finds the data it is looking for. Given that service's hypermedia controls (in this case especially the supported operations), the client knows how to automatically subscribe to the discovered service (e.g. using the *subscribe_push* endpoint depicted in listing 5.1) to be provided with updates in the future.

5.5 Containerized Deployment

Microservices aim to be highly isolated by nature and so should be their deployment. In order to support resilience, high cohesion, a polyglot stack and good vertical scalability, we need a way to deploy our apps in a self-contained manner. A common solution in the Microservice context is to use containerization, usually with Docker[4]. Basically containerization is an efficient way of virtualizing an application's environment - somehow similar to virtual machines, except without the overhead of having to run a complete separate operating system on virtualized hardware. Containers fundamentally rely on the host system's kernel and directly access its hardware through a hypervisor, but still allow to run a completely independent system within the container itself.

5.5.1 Advantages

Efficient virtualization. Containers combine the benefits from running an application natively on the host machine and inside a complete virtual machine (VM). Performance and overhead are by far better than with a VM, however, containers are still separated from the host operating system (OS). Especially, they can not access each other by default on the one hand and can have completely heterogeneous technologies on the other. Figure 5.5 compares VM- to container architecture.

Isolation. Containers are completely isolated from each other as well as from native application running on the hOSt. This high degree of isolation concerns security, stability and heterogeneity. To give an example for heterogeneity, assume one container that contains a Java application with an Ubuntu 16.10 operating system, OpenJDK 1.8, Apache Tomcat 8.0, a DB2 database and an Apache2 proxy at port 80, while another container on the same host could run a NodeJS app in combination with a Redis database and an nginx proxy at port 80. However, no clashes in terms of dependency versions, data storage directories or network port usage occur. Concerning stability, assume the Java virtual machine (JVM) crashes for some reason in one container, then all other containers' Java applications still keep running perfectly.

Rapid development. Due to their bounded context, Microservices usually are relatively small, managed by exactly one development team and have no or at least few dependencies on other components or teams. As a result, Microservices can be released much faster than large software systems. In the Microservice culture it is common to have very short development cycles in which the service gets improved

[4]http://docker.com

incrementally. At the end of each cycle is an incremental release. But these short development cycles also require for a suitable deployment technique. If a component's new feature has been finished, but the component can only be deployed in combination with rebuilding an entire monolith or at least re-configuring a large host system, productivity suffers. Using containerization, a new container image for the new release is built and simply spawned up. This is a main reason why containerization aligns perfectly with Microservices.

Rapid provisioning. In contrast to a VM, a container is build and started in almost no time, since it contains a minimum of overhead. Multiple instances of a service can be created by just cloning containers within few seconds, while the memory usage of having multiple, equivalent containers is nearly constant.

Environment specificity. With containers, a developer team can precisely specify the environment for its application. Former problems like that the production host system runs a different version of the language runtime than the developers' machines or the test system are not present anymore. Developers can specify the environment to be set up within the container in a configuration file delivered along with all other application code, so an environment is perfectly tied to an application.

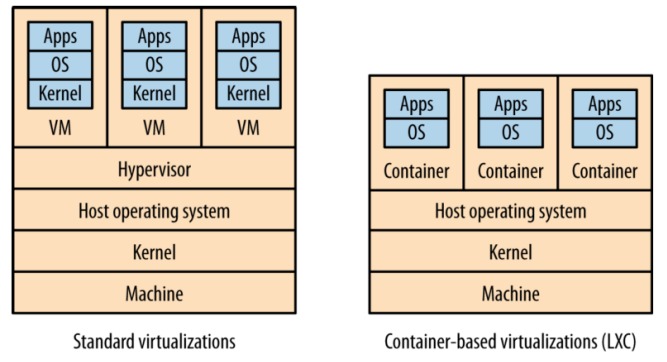

Figure 5.5: VM- vs. container virtualization [22]

5.5.2 Disadvantages

Kernel specificity. A disadvantage of containers compared to traditional VMs is that due to every container's dependence on the host's kernel, traditionally only operating systems within the same family can be used, i.e. there was no way to run a Windows environment inside a Docker container (which natively runs on Linux) by now. However, recent efforts of the Docker community are trying to overcome this problem to a certain extent by integrating Docker natively with Windows Server[5].

Image storage overhead. Containers are build from multiple layers of images, e.g. one layer includes OS derivation specific files, another includes the NodeJS runtime etc. Therefore building an application into an image needs storage space, since the image does not only contain the actual application, but also environmental dependencies. Although this overhead is magnitudes less than with VMs, it still requires a central image registry with large capacity.

[5]https://www.docker.com/microsoft

5.5.3 Deployment Strategy

In view of the advantages described above, we decide to rely on containerization, for which we use Docker. Docker is our containerization software of choice because of its wide acceptance and vast library- and tool support. Another aspect to give thoughts about is not only the deployment technology, but also the strategy. Basically it includes thoughts on how and where to deploy.

5.5.3.1 "Where" to deploy

Concerning the "where", Evans presents multiple deployment patterns (depicted in figure 5.6 and 5.7) [22]. The multiple-services-per-host (MSPH) pattern corresponds to a n:1 relation between Microservices (or more specifically apps in our case) and physical host machines, while the single-service-per-host (SSPH) corresponds to a 1:1 relation. Such physical machines include traditional servers as well as micro-devices (such as sensors), mobile devices (smartphones, tablets) and workstations. Which pattern to choose depends fundamentally on the host machine's hardware. When facing a large, multi-core server with multiple gigabytes of RAM, it is more efficient to deploy multiple services to one host, especially because an operations team's effort linearly scales with the amount of machines they have to manage. However, when facing a Raspberry Pi as target device (e.g. for deploying our DataApp to), it usually might be better to follow SSPH, because it might even reach its capacity with one app. However, these patterns do not entirely fit our container deployments. Therefore we extend these patterns by introducing the following four additional ones, which describe the relationship between apps and containers, as well as between containers and hosts.

1. 1:1:1 - A container comprises exactly one application and is exclusively hosted on one host machine.

2. n:1:1 - A container comprises multiple apps and is exclusively hosted on one host machine.

3. n:m:1 - A container comprises multiple apps and a host machine hosts multiple containers.

4. 1:n:1 - A container comprises exactly one application, but a host machine hosts multiple containers.

We exclude (2.) and (3.) from being used in our scenarios, because they heavily contradict the purpose and best-practices of both Microservices and containers. We recommend (1.) to be used for lightweight devices, such as sensors and other "things" and (4.) to be used for backend applications, such as analytics apps or the like. However, having exactly one app per container is not contrary to having multiple containers per app (a cluster) to get better scalability. Frontend-only apps - which do not include any backend themselves at all and therefore are just static HTML pages, optionally with CSS' and JavaScripts - may be hosted in either of these patterns, depending on the use case.

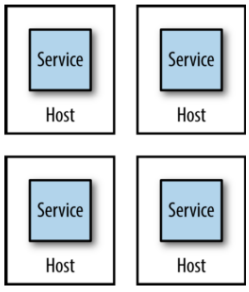

Figure 5.6: Single-service-per-host deployment [22]

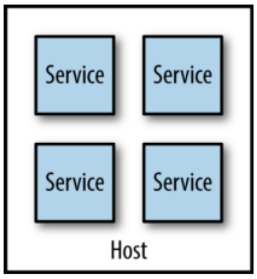

Figure 5.7: Multiple-services-per-host deployment [22]

5.5.3.2 ”How” to deploy

The question about how to deploy apps exceeds the scope of this work for the most parts, since it does relate rather to the organizational culture than to the applications' architecture. However, we want to give recommendations. First and overall, we heavily suggest to stick to the Microservice fashion of having short development cycles with many releases. This perfectly aligns with agile development techniques, however, they are not mandatory. With this goal in mind, we suggest to set up a automatic delivery pipeline, facilitated by tools like Jenkins. A simplified associated flow could be as follows. A developer team checks in its latest code changes of an app into version control and pushes them to the remote repository. A continous integration (CI) server is notified about these changes, pulls the code, automatically runs included unit tests against it, builds a Docker image, pushes that image into a repository and notifies a client-script on the development-stage host machine about the new image. The script pulls the new image, spawns up a new container from it and notifies the CI server to run end-to-end tests against it. If all tests were successful, the team gets notified. Finally, on the team's behalf, another script on the production-stage host machine pulls the image, too, and spawns a new, production-ready container from it. Additionally, its very recommendable to adopt a DevOps culture to support strong cooperation between developer- and operations teams to enhance productivity by making an app's team responsible for the app over its entire lifecycle.

5.5.4 Summary

In this chapter we presented how our apps are deployed in an efficient, flexible and reliable way that aligns with other Microservice best-practices. We presented how the app concept is facilitated by the use of containers and how virtual containers should map to physical hardware. Finally, we gave a set of recommendations on how

to integrate our platform with inner-organizational software development. Figure 5.8 shows a deployment recommendation for the scenario introduced in chapter 5.1.

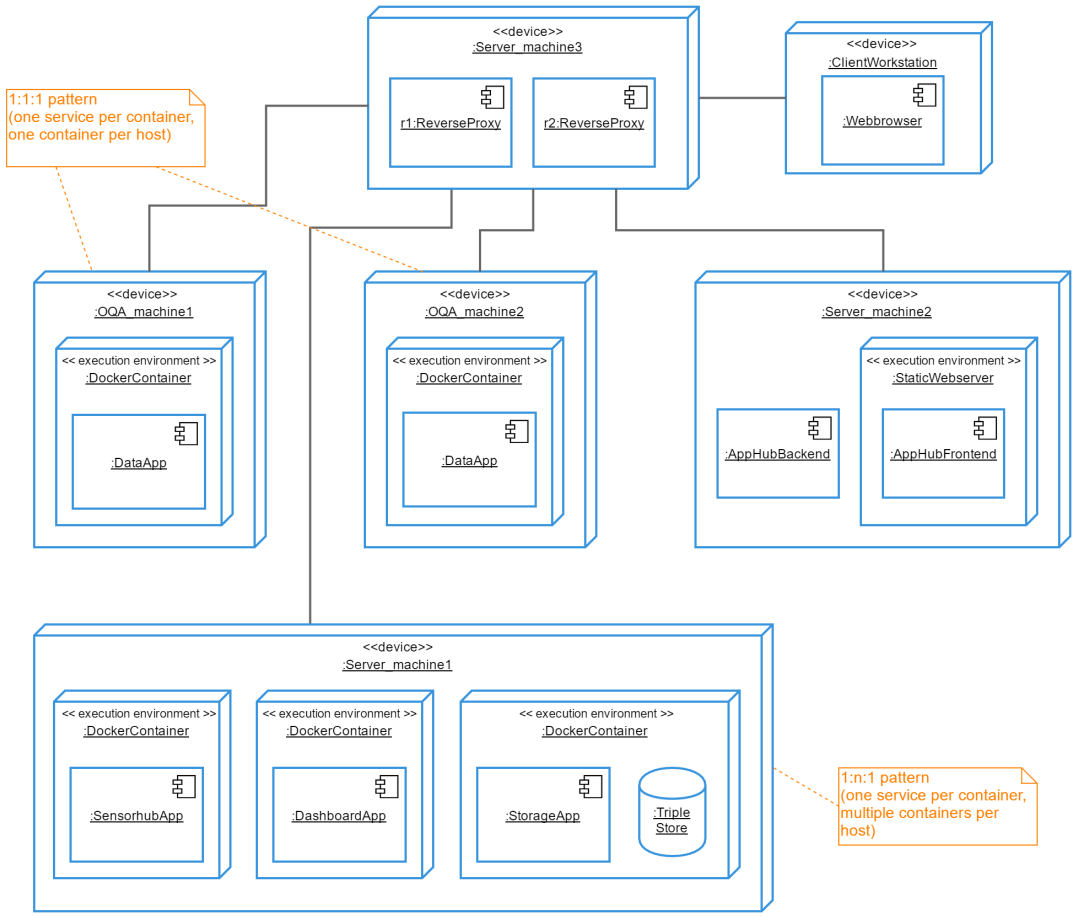

Figure 5.8: Deployment diagram for our scenario

5.6 App Store

After having presented how apps, especially their interfaces, are implemented to satisfy our requirements one the one hand and how they technically get deployed into production on the other, there is one more accompanying topic to cover. While the preceding chapters represented a rather technical view, we take an end-user perspective and present the implementation of an app store now. As described in chapter 4.6.2, an app store brings off good user experience by abstracting from technical details and providing the end-user with familiar concepts. The app store - which takes the shape of the so called ”AppHub” in this case - is a web application to manage all deployed apps, including adding new or removing old ones. Just like in an appstore for smartphone apps, the user is able to choose from a variety of different software pieces and then install them without any technical background knowledge. In contrast to the applications presented previously, the AppHub does not inevitably have to be an app in our understanding, but can also be installed natively. Since it is used to install new apps, which are containers, it needs access to the host machine's Docker daemon. This can either be achieved through the recently introduced *–privileged* flag[6]. In the above scenario, the AppHub is run

[6]http://obrown.io/2016/02/15/privileged-containers.html

natively. Furthermore, the AppHub is considered an integral part of out platform, besides apps, Linked Data and more.

5.6.1 Frontend

The AppHub was implemented two-fold, consisting of a frontend and a backend. The frontend comprises the user interface, which the end-user interacts with to perform the actions described in chapter 4.6.2. It mainly written in AngularJS and HTML5. Figure 5.9 shows a screenshot of the AppHub UI, in which a cluster with 3 apps is installed and running. Figure 5.10 a screenshot of the catalog of available apps in the AppHub. The AppHub introduced the term of a ”cluster”, which basically is the set of all deployed app in the user's scope. A user who is willing to install new functionality would typically open the AppHub under a well-known URL in his browser and bring up the app catalog first. He would browse the available app images to find the desired functionality and add the respective app to the cluster with two clicks. The app is now represented by an icon with connectors. Afterwards he would assign the app to the target device, which he aims the app to be run at, using a commonly understood Drag & Drop fashion. The AppHub abstract from actual servers and ip-addresses and ports, so that the user drags an app to where he intuitively expect it to run. For instance, a frontend-web-app is not actually deployed to a smartphone technically, but to a server that hosts its content. However, the user does not have to care about such details. Also does the AppHub automatically detect incompatibility, e.g. an OQA DataApp can not be deployed to a smartphone, since it depends on an actual data-producing OQA sensor. Consequently the user could not drag the DataApp to a smartphone. After having assigned the app to a machine, the user can connect the app to other apps, meaning that he declares them to be able to exchange data. Finally, the user would start up the cluster. At this point, the AppHub frontend serializes the entire cluster including its apps and their connection to JSON format and requests the backend via HTTP. What it technically means to connect apps and start a cluster is explained in section 5.6.2.

5.6.2 Backend

The backend performs the actual actions requested by the user through the frontend. It is a NodeJS-based application running on a Linux host system that mainly consists of an HTTP server providing a REST API and an interface to the docker- or docker-compose daemon (since we decided to rely on Docker technology in chapter 5.5). The AppHub makes use of docker-compose, which is a tool to define multiple docker containers and their relations and dependencies in a single configuration file and then start up and link all containers automatically and in proper order with a single command. Typically, when a new request with a serialized cluster arrives from the frontend, the AppHub first creates a new docker-compose.yml file. Afterwards it executes ”docker-compose up” to start the cluster defined in that file. During this process, container images for the respective apps are pulled from a central, organization-internal registry. Then, containers are instantiated from these images and links among them are established. Additionally, a container with an nginx webserver is started and linked to all containers. It acts as a reverse proxy to the apps. The major benefit with having a reverse proxy is that multiple apps are made accessible within a single domain namespace by routing requests not directly to the apps, but through a webserver in front of them. For instance, assume a simple

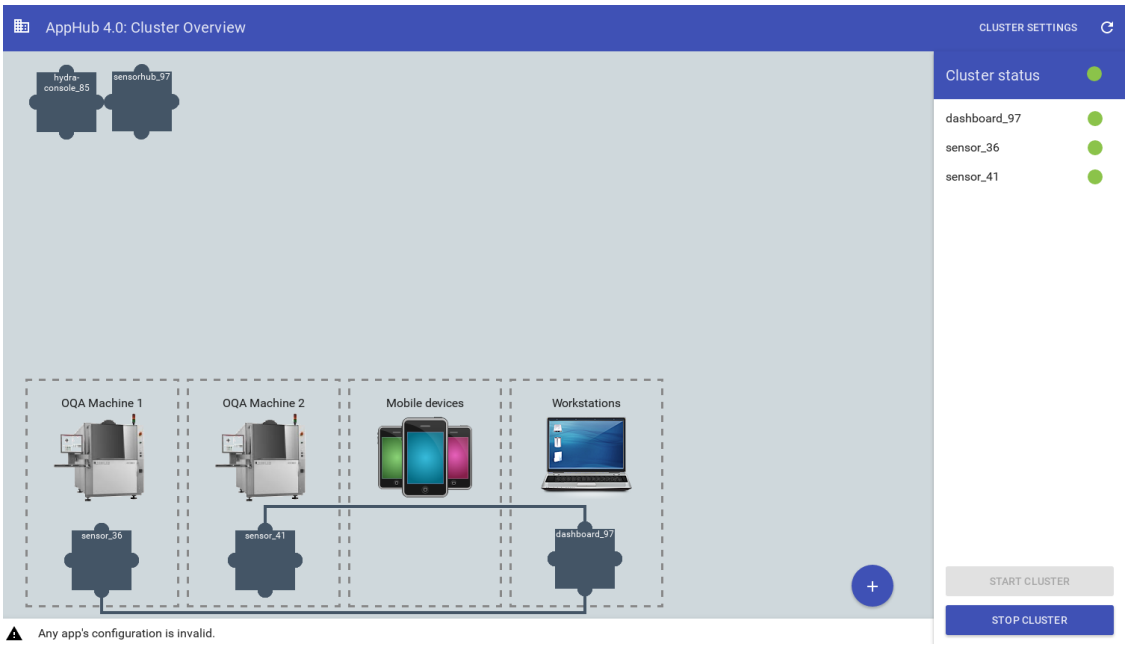

Figure 5.9: AppHub user interface

scenario where two DataApps are listening on http://localhost:3000 and http://localhost:3001. The reverse proxy makes them accessible through http://localhost/app1 and http://localhost/app2, where app1 and app2 correspond to the respective apps' id. With appropriately adjusted DNS resolution, it could even be, for instance, http://scale-it.org/app1. This abstracts from the apps' actual physical location and technical details, such as the exposed port. Other benefits with a reverse proxy are advanced caching- and load balancing capabilities. In short summary, the following happens in background, when a user starts a cluster, he clicked together at the frontend.

1. A docker-compose based cluster is put together with all the other apps in the cluster plus a reverse proxy.
2. The chosen apps' Docker images are pulled from an internal registry.
3. Those images are instantiated to containers and linked to each other.
4. Containers are started and made accessible through the web, exposing their proper ports to the host machine.

One last thing to clarify in the context of the AppHub backend is what it means to "link" containers. This term is two-fold in our case. First, Docker containers are linked using the Docker DNS system. This means that a container is made accessible for another container via its IP. Additionally, in our case, an "–extra-hosts" entry is set, so that a container can access another one not only by its plain IP but by a qualified URL like "http://scale-it.org/the_other_container". This is necessary to enable apps to reach other apps. Second, besides the previously described technical linking, we further introduce a linking on application level. An app A linked to an app B is told to, firstly, add B to its list of seed URLs for traversal and to, secondly,

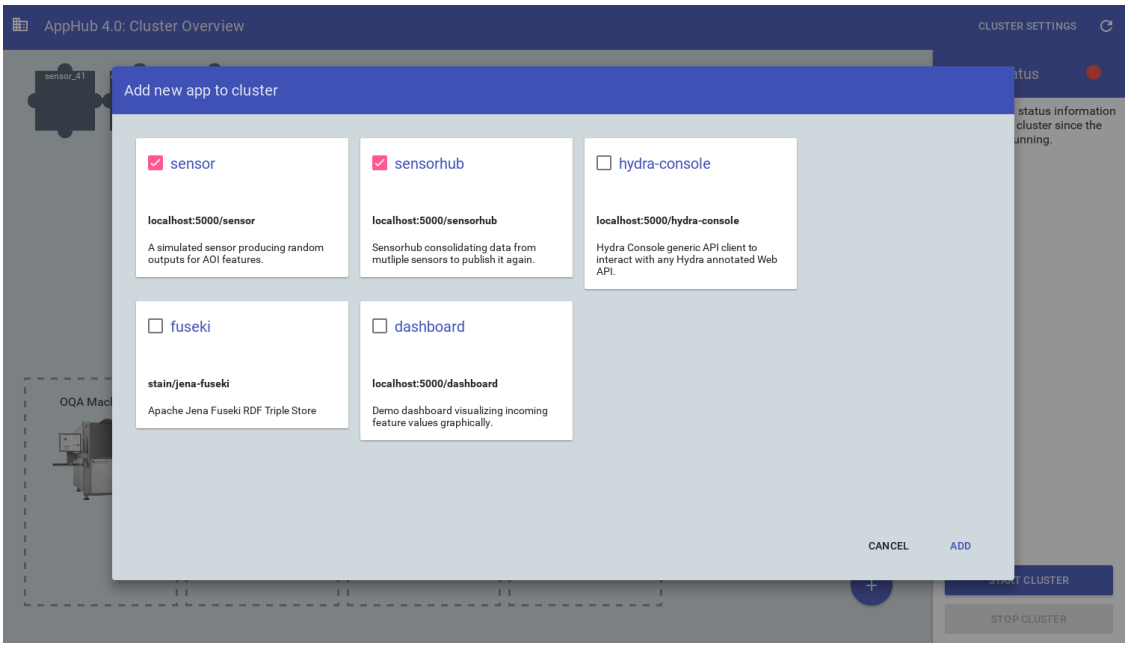

Figure 5.10: AppHub user interface displaying catalog

automatically subscribe to it. As a result, the user would not have to explicitly enter app A and configure it to subscribe to app B. Instead, A receives B's events automatically.

5.7 Conclusion

In this chapter we will pick up on the app requirements stated in chapter 4.6.1 and check our apps against those. Doing so will clarify how the requirements have been mapped to the actual software components. For most of the requirements, their satisfaction can easily be reconstructed. Those, which are not that trivial are explained in short detail.

5.7.1 App Requirements

5.7.1.1 Compliance Level 0

An app must ...

- ... be at least HTTP/1.1 capable. ✓(All apps are web-apps and at least HTTP/1.1 capable. HTTP/2 support is planned, but currently lacks of supported libraries.)
- ... be uniquely identifyble and by a URL. ✓(All producing apps are at least identifyable through the URL combination of http://host:port and optionally also within a single namespace, like http://scale-it.org/app1.)
- ... have a root resource accessible via an HTTP URL. ✓(All producing apps expose the Hydra EntryPoint at their root.)
- ... expose a REST API. ✓

- ... accord to the highest level of maturity in the RMM. ✓(All producing apps are described with Hydra, which especially is a hypermedia format and therefore fine with the RMM level 3.)
- ... fulfill all four Linked Data principles. ✓(Since all producing apps are at least identifiable through the URL combination of http://host:port, the first two principles are fulfilled. The fourth is fulfilled, because we use hypermedia. The third is fulfilled because every resource exposes semantically annotated RDF data in JSON-LD.)
- ... implement passive communication. ✓(All producing apps have at least SSE and additionally also webhook subscription endpoints.)
- ... use JSON or JSON-LD as default representation format. ✓
- ... expose RDF-structured data with semantic annotations. ✓
- ... describe its API using the Hydra vocabulary. ✓
- ... offer a human-facing HTML represenation or user-interface. ✓
- ... be stateless. ✓(Truely RESTful service, as we face them, are designed stateless by nature.)
- ... explicitly declare its dependencies, instead of relying on system-wide packages. ✓(As usual with NodeJS, all dependencies are declared inside the package.json file and can be installed dynamically.)

5.7.1.2 Compliance Level 1

An app should ...

- ... have a bounded context. ✓
- ... be designed to be reusable. ✗(While the VizApp and the SensorhubApp reusable, the DataApp is not completely, since it specifically applies to OQA machines).
- ... run in a container. ✓
- ... be developed and operated by exactly one development team. (We can not make an assertion on this, because it depends on the organization which finally uses this platform in production.)
- ... comprise its own database. ✓(Those of our apps, that need to persits data (StorageApp), uses a own database, namely a triple store.)
- ... have as less configuration as possible. ✓(The only configuration currently needed is the app's id and port to listen on.)
- ... read its configuration from either environment variables or a web service. ✓(Id and port and passed via environment parameters in all of our apps.)

- ... use the technology stack best suitable for its purpose. ✓(According to Guinard and Trifa, NodeJS is well-suited for IoT scenarios [19]. Further, a triple store apparently is the best choice to store RDF data.)
- ... be available for installation in an app store. ✓
- ... have a separate code-base tracked in revision control. ✓
- ... follow the TolerantReader pattern. ✓(For instance, our apps can use only CURIEs as identifier in semantic data as a fallback, if the context is not available for any reason.)

5.7.1.3 Compliance Level 2

An app may ...

- ... implement more than one passive communication mechanisms. ✓(All producing apps support SSE and webhooks).
- ... have logging. ✗
- ... expose its logs as an event stream. ✗
- ... use secure connections (HTTPS). ✗
- ... support representation formats besides JSON. If so, it must properly implement content negotiation. ✗

5.7.2 Overall Requirements

The presented app requirements were developed based on the overall architecture requirements described in chapter 4.1 and are fulfilled by all of our apps. We further investigate shortly to which extend the overall requirements are met, too, while identifying their key enablers.

Uniformity. The key enablers of uniformity are the use of web technology, the consequently uniformly defined apps and the self-describing Hydra APIs allowing for generic clients.

Just-in-Time Capability. Just-in-Time capability is facilitated through the use of publish / subscribe mechanisms and event collaboration. However, we did not investigate if our platform's technical performance is sufficient for just-in-time communication, too. This aspect is covered in detail in chapter 7.

Ease of use. Ease of use is mainly facilitated through the introduction of the AppHub, which abstracts from any underlying technical details as well as through the requirement for every app to have a human-readable web representation.

Data-centricity. Data-centricity is heavily facilitated through the introduction of Linked Data techniques which boost data integration over heterogeneous sources.

Scalability. Y-axis scalability is mainly facilitated through the Microservice concept including bounded contexts. X- and Z-axis scalability is facilitated by the use of containerization. However, we have not performed load tests or the like to get quantified metrics on X- and Z-axis scalability. This goes beyong the scope of the present work.

5.7.3 Architecture Stack

Following "The Web of Things" [19], our platform architecture can be seen as consisting of several layers, too. Figure 5.11 is strongly influenced by the work of Guinard and Trifa [19] and depicts these layers and their respective technology options in summary.

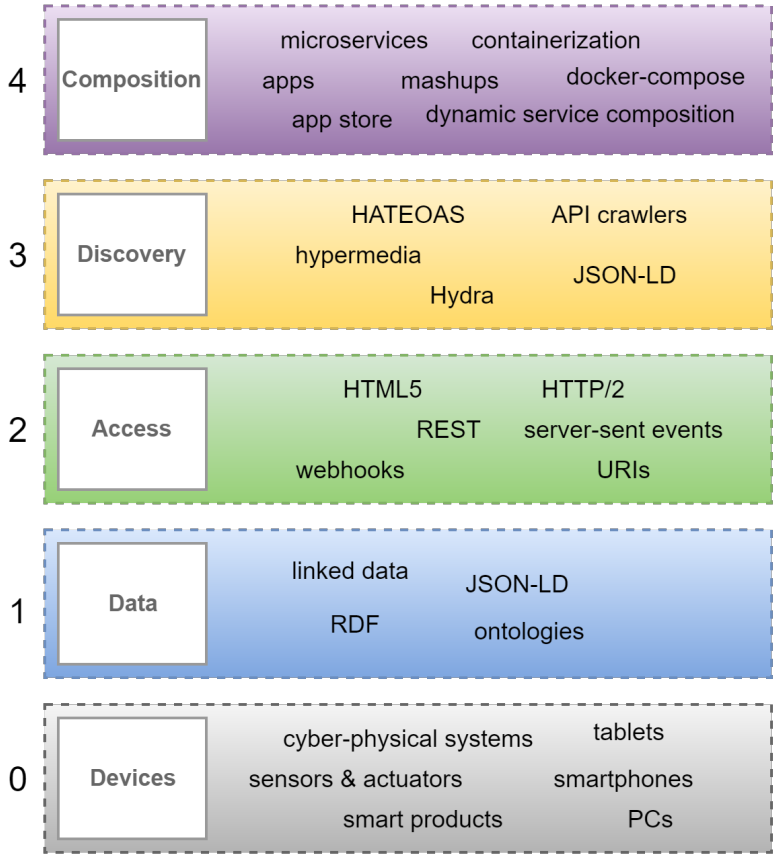

Figure 5.11: Layered architecture stack

6. Related Work

A lot of research had been conducted in the fields of distributed, Microservice-based software architectures, Linked Data, IoT and Industry 4.0 technology. However, to the best of our knowledge, there has only been a single attempt to elaborate a complete solution concept integrating most of these aspects (see "Web Things" in section 6.1).

6.1 Web Things

"Building the Web Of Things" [19] pursues very similar goals to ours, however not the same. Their so called "Web Things" basically propose a web-technology-based platform for integrating IoT devices and sensors, even though not focused on industrial but on private scenarios. They also touch on the topic of semantic-powered APIs - just as we are doing - but do not elaborate on this in greater detail. Moreover, they neither spend many thoughts on working out a higher-level, macro-like view nor do they employ Microservices or any architectural pattern at all. Instead they put their focus on more technical implementation details. Nevertheless, their work contains numerous parallels to ours without a doubt. While most parts of our architecture had already been developed at the time "Building the Web Of Things" got published, we still considered many of their ideas and guidelines a useful inspiration and reworked certain parts along these lines.

6.2 S-ToPSS

The "Semantic Toronto Publish/Subscribe System" [34] proposes a middleware architecture employing the publish / subscribe (pub/sub) communication pattern in combination with semantic-web concepts, which amounts to so called semantic events. The authors' aim was to outline a framework that is able to map large amounts of incoming, semantically annotated data to service calls or functions triggers in a loosely coupled way. Since we are obliged to deal with large amounts of highly heterogeneous data in our target scenarios and decided to use Microservices, which typically are event-driven, the insights from S-ToPSS served as an inspiration to us. The paper taught us how to apply semantic event matching and ratified pub/sub as a suitable pattern for our use case.

6.3 SemIoT

"Semantic Technologies for Internet of Things" (SemIoT) is a research project that "aims at providing an access to sensor networks using unified data models and interfaces that hide heterogeneity of the network and facilitate effective data access, interoperability, resource search and discovery."[1] Although not that comprehensive, their approach is very similar to ours to the effect that they also consider semantic annotations, e.g. using the sensor network ontology (SSN)[2], the preferred way for sensor data integration and also employ web technology. The researchers' focus in mainly on data integration and querying rather than software-architectural- and user experience-related aspects. However, we consider SemIoT's in-depth investigations on annotating sensor data, as well as the *wot-semdesc-helper*[3] essential when it comes to realizing our platform from a prototype to a real-world product.

6.4 Sense2Web

The authors of "A Service Oriented Middleware Architecture for Wireless Sensor Networks" [43] have developed "a Linked Data platform to publish sensor data and link them to existing resource on the semantic Web". Similar to our scenario they faced the problem of integrating and querying sensor data from various sources and considered Linked Data the most suitable foundation to do so. Their platform does also rely on web technology as well as on RDF data, but does not employ any publish / subscribe mechanisms. Furthermore, it only describes how to introduce semantics to sensor data but does not cover interfaces or application design. However, their platform demonstrates the suitability of Linked Data in sensor networks as a proof-of-concept.

6.5 Semantic Sensor Web

The semantic sensor web is a concept presented by Sheth, Hanson and Sahoo [44] and describes "Web-accessible sensor networks and archived sensor data that can be discovered and accessed using standard protocols and application program interfaces." Similar to Sense2Web, the authors attempt to realize a web of data among sensors. They have developed an abstract set of language- and interface specifications to publish, query and subscribe to sensor observations. Furthermore they provide a set of concrete properties to annotate sensors and introduce a sensor's dimensions of space, time and theme.

6.6 Fraunhofer Industrial Data Space

The Industrial Data Space (IDS) [17] is an initiative that was launched in Germany at the end of 2014 by representatives from business, politics, and research. It basically aims at developing a comprehensive reference architecture for a data platform that integrates smart services with Industry 4.0. The researchers' basic intention is similar to ours, however, their focus is rather limited to data management, while

[1] http://semiot.ru/en/
[2] https://www.w3.org/2005/Incubator/ssn/ssnx/ssn
[3] https://github.com/semiotproject/wot-semdesc-helper

we attempt to architect an overall, potentially production ready solution. While the IDS pays a lot attention to data-security, -authority and -sovereignty, we mostly neglect these aspects and concentrate on user- and developer experience, overall flexibility and performance instead. However, the IDS introduces promising ideas. Such include to use semantic annotations as a key enabler for data integration as well as decentralized inter-service communication. Furthermore they first mentioned the concept of apps and app stores to publish data providers, which is a central concept in our solution, too. We also pick up on the roles of data providers and -users, which explicitly might be humans as well as software application or things.

7. Evaluation

This chapter evaluates the architecture designed in chapter 4 and implemented in chapter 5 and separates into two logical parts. The first part includes a qualitative discussion on the degree to which our initial requirements were met. More precisely, we discuss insights, which we got from presenting our reference implementation to industry partners within the scope of a workshop.

The second part is more technical and quantitatively measures our platform's performance in a few simulated scenarios. In chapter 5.7.2, we found that just-in-time capability was the only requirement we could not consider fulfilled with certainty. Consequently, we focus on that requirement in greater detail in the following. We investigate, whether just-in-time communication between apps can still be maintained under high load. Furthermore, we investigate on the suitability of the currently used HTTP protocol in comparison to some alternatives, since we presumed plain HTTP/1.1 to potentially entail too high overhead for high performance scenarios.

7.1 Summative Assessment

Reussner states that "a system's quality is characterized through the degree of fulfillment of its quality requirements". [45] In chapter 5.7.2, we showed how all of our requirements stated initially can be considered fulfilled in the set scenario - except for performance, which requires further, quantified investigation as done in 7.2.

Besides focusing on these requirements, we wanted to get insights on actual usability. A goal of this work is to leverage businesses' productivity. Productivity is critically influenced by the system's usability. A shop floor employee, who can quickly understand and operate his system, is more productive. To get expert estimations, a summative assessment with industry experts had been performed.

7.1.1 Workshop Demonstration Scenario

We demonstrated how to realize the scenario described in 5.1, using our platform. The objective was, given two (simulated) OQA machines, to deploy functionality to visualize OQA observations in order to find anomalies. We demonstrated the following steps.

1. Deploy a DataApp to one machine using the AppHub.

2. Open the newly deployed DataApp's website to view device information and latest observation value.

3. Open the newly deployed DataApp's API in HydraConsole to understand the underlying, semantic meta data.

4. Deploy a SensorhubApp, as well as a second DataApp and link them together using the AppHub.

5. View the newly deployed apps' websites.

6. Deploy a DashboardApp and link it to all other apps using the AppHub.

7. View the newly deployed DashboardApp's website, subscribe to events concerning "http://scale-it.org/oqa/#feature2" and view the visualization of incoming data.

8. Explain technical backgrounds, especially how containers are spawned by the AppHub and how the dashboard's subscriptions work completely dynamic through browsing linked apps' APIs.

7.1.2 Feedback

The participants found it positive that the solution abstracts from the underlying technical details via the app concept. They have considered the platform to hold practical benefits for actual Industry 4.0 scenarios on the shop floor, if developed ready for market. Questions and doubts that came up during the workshop include the following.

1. The experts were unsure about the suitability and performance of HTTP and compatibility to real-time protocols, such as MQTT.

2. How to introduce apps to other apps (discovery) as a best practice?

3. What are hardware requirements?

4. How to declare data compatibility? For instance, how to declare that an app, which can only process temperature data, only is allowed to subscribe to sources emitting temparature data. Suggestions included to introduce a graphical language like Blockly[1].

5. Request for both API and payload annotation to be generated automatically.

[1] https://developers.google.com/blockly/

7.1.3 Discussion

This section discusses and clarifies the issues raised by the industry experts as described in section 7.1.2.

1. We explained that HTTP overhead can be minimized using server-sent events, encryption and HTTP/2 techniques to reach just-in-time capability. The actual performance is measured in chapter 7.2. We agreed on the point that adapters need to be built to integrate MQTT- or CoAP-based legacy systems.

2. We explained both methods, traversal-based- as well as self-discovery and agreed on that only the latter is actually production-ready.

3. An average app, such as the DataApp, could run on a Raspberry Pi or comparable device, but not less. Our DataApp has a memory footprint of around 40 MB. The source code including all dependencies is approximately 20 MB large. The NodeJS 6.4.0 environment takes approximately 90 MB of disk space when installed natively and about 100 MB when run in a minimal Docker container based on Alpine Linux - plus additional 130 MB for the Docker installation.

4. Due to the introduction of semantics, such declarations are not necessarily needed. Following the TolerantReader (see [41]) pattern, an app could potentially subscribe everywhere and just ignore data it can not process.

5. We agreed on that auto-generated annotations are inevitably necessary for production. However, due to the current lack of community support for Hydra, there are no such libraries available, yet.

7.2 Performance Evaluation

This section is two-fold. On the one hand, it concerns with evaluating the performance of our architecture in order to verify, whether our just-in-time capability requirement is actually met. On the other hand, this section discusses benchmarks performed on essential parts of our architectures implemented with various other communication mechanisms and protocols. The goal of these benchmarks is to get a comparison between the communication techniques used in our architecture and alternative approaches to eventually reason about their respective suitability. An important thing to note at this point is that the benchmarks do not aim to be universally valid, but apply specifically to our use case. For this reason, we create conditions similar to our platform's target scenarios, instead of setting up a general-purpose environment.

7.2.1 Methodology and Setup

We intend to investigate just-in-time capability, so the focus lies on measuring at what rate data can be sent to a certain number of actors. More precisely, we want to evaluate our publish / subscribe mechanisms and measure the maximum frequency at which new data fragments can be pushed reliably from a producer to many subscribers. We pick up on a common use case for our platform, which is a DataApp multicasting OQA measurements. For the benchmarks we isolate the

communication-related parts of our apps and implement them in two separate, small applications: a server- and a subscriber application. The server corresponds to a DataApp and is responsible for publishing data fragments to its subscribers. The subscriber application in turn corresponds to any arbitrary other app within an industrial scenario and is responsible for subscribing to the DataApp and receiving events from it. The reason for isolating the communication-related code from our apps into two separate scripts was that we only want to focus on pure communication performance, while ignoring additional computation effort for parsing, printing or persisting data, which varies between the different apps. Since both server and subscriber reuse code from our apps, they are both implemented in NodeJS 6.4.0. For HTTP and SSE communication, the same libraries as in our apps are used. For all other technologies, the most popular respective NodeJS implementations are used.

For all subsequent benchmarks our test setup was to have one server instance and a hundred subscribers. The number of 100 was chosen to simulate a large, but realistic real-world shop floor scenario, as stated by the experts. Server and subscribers run on two physically separated machines, which are connected by a 100 Mbit/s local area network via a network switch. Using NodeJS' cluster module[2], all subscribers are spawned as separate processes and have a separate TCP or UDP connection each.

The subscribers initially subscribe to the server. When all 100 subscribers have stated their subscription at the server, it starts to fan out a specific, sample data fragment, which is depicted in listing 7.1. The data fragment corresponds to those emitted by the actual DataApp. Because all OQA data fragments follow the exact same structure and their values do not vary in size, we can use the same data fragment for each emission and do not need to randomly re-create a new one in each iteration. The server's strategy is to emit a data fragment to every subscriber, wait for the respective subscriber's acknowledgement and then emit another one. With HTTP and CoAP, the arrival of a response to a request is considered an acknowledgement. With all other techniques, the subscribers explicitly emit an acknowledgment event to the server at application level. The process from emitting a fragment to all subscribers throughout to receiving all acknowledgments is called an iteration. The number of iterations to perform for each test scenario is set to 1000. Effectively, every server performs 10,0000 emissions (1000 iterations * 100 subscribers), while each subscriber receives 1000 events. The time from sending the very first data to receiving the very last acknowledgment is measured and used in combination with the total number of emissions to calculate the iteration rate. The iteration rate is a key figure and indicates how many data emissions can be performed within one second for a number of 100 subscribers and therefore is an indicator for the platform's just-in-time capability.

We employ four different test scenarios for each benchmark series, where we vary network bandwidth and switch between uncompressed and compressed data. To have equal conditions for all scenarios, compression is handled directly on application level using zlib[3] with gzip compression strategy and default parameters. Any of the investigated technologies' built-in compression- and caching mechanisms were turned

[2]https://nodejs.org/api/cluster.html
[3]https://nodejs.org/api/zlib.html

off. While compression had only been shifted from protocol- to application level, caching was completely eliminated. However, this is still reconcilable with a real-world scenario, because caching will most likely not be relevant in our type of event collaboration. Further discussion on this can be found in 7.2.2.2. Please note that when talking about server-sent events, we always refer to server-sent events based on HTTP/1.1. Since SSEs are basically just a chunked stream within a kept-alive HTTP response, they could also be used on top of HTTP/2.0 in theory. However, we have skipped this scenario, since we did not expect an increase in performance with HTTP/2.0. The only HTTP/2.0 feature that would apply in a stream is binary framing, which is not expected to reduce packet size noticeably. HTTP/2.0 Server push may also benefit Linked Data payloads. However, this has not been investigated in the evaluation and is left for future work.

We expect data compression to be especially effective in our scenarios, since we send JSON-serialized data, which has a large overhead for structure.

1. 100 subscribers, 1000 iterations, unlimited network bandwidth, uncompressed payload data
2. 100 subscribers, 1000 iterations, unlimited network bandwidth, gzip compressed payload data
3. 100 subscribers, 1000 iterations, limited bandwidth (5 Mbit/s up- and downstream), uncompressed payload data
4. 100 subscribers, 1000 iterations, limited bandwidth (5 Mbit/s up- and downstream), gzip compressed payload data

Listing 7.1: Sample OQA data fragment used for benchmarks (1244 bytes)

```
[
  {
    "@context": {
      "ssn": "http://purl.oclc.org/NET/ssnx/ssn\#",
      "qudt": "http://qudt.org/1.1/schema/qudt",
      "hasValue": "ssn:hasValue"
    },
    "@id": "http://scaleit/sensor_oqa1/observations/23/result",
    "@type": "qudt:QuantityValue",
    "hasValue": {
      "@type": "xsd:integer",
      "@value": "-83"
    }
  },
  {
    "@context": {
      "ssn": "http://purl.oclc.org/NET/ssnx/ssn\\#"
    },
    "@id": "http://scaleit/sensor_oqa1/observations/23/sensor-output",
    "@type": "ssn:SensorOutput",
    "ssn:isProducedBy": "http://scaleit/sensor_oqa1/device",
    "ssn:hasValue": "http://scaleit/sensor_oqa1/observations/23/result"
  },
  {
    "@context": "http://scaleit/sensor_oqa1/contexts/Observation",
    "@id": "http://scaleit/sensor_oqa1/observations/23",
    "@type": "ssn:Observation",
    "featureOfInterest": "oqa:shift-feature",
    "observedProperty": "oqa:shift",
    "observationResult": "http://scaleit/sensor_oqa1/observations/23/sensor
       -output",
    "observationResultTime": "Fri Aug 26 2016 14:38:27 GMT+0200 (CEST)",
    "observedBy": "http://scaleit/sensor_oqa1/device",
```

```
33      "analysisMode": "oqa:OLAM",
34      "observedPart": "http://scale-it.org/parts/C5034",
35      "observedStep": "UZT_AS_1"
36    }
37  ]
```

7.2.2 Measuring Just-in-Time Capability

In this first part of our performance evaluation, we investigate, whether the current platform architecture supports just-in-time communication. To do so, we deploy a server and 100 subscribers for both of our implemented subscription mechanisms, which are HTTP/1.1 webhooks and server-sent events. To simulate a real-world scenario as close as possible, we use a RaspberryPi 2 Model B with Raspbian Jessie as a constrained device for the server (or publisher). All 100 subscribers are deployed as separate processes - as described in 7.2.1 - on an Apple MacBook Pro Retina with a 2.9 GHz hyper-threaded dual-core Intel Core i5 processor and 16 GB of DDR3 RAM, running Mac OS X 10.11 and NodeJS 6.4.0.

7.2.2.1 Results

As described in section 7.2.1, we performed four different scenarios for both publish / subscribe mechanisms with unlimited- and limited bandwidth (5 Mbit/s up- and downstream) and uncompressed and gzipped payload data. Please note that by referring to ”unlimited” we actually mean 100 Mbit/s, which appears to be enough to not form a bottleneck for our scenarios. In each scenario we measured the execution time. Dividing the total number of iterations (1000 in our case) by the total consumed time, we get a measure for iterations per second, where an iteration is the process of emitting one data fragment to every subscriber. Using Wireshark network analysis tool[4] we found the following packet sizes:

- **HTTP/1.1 uncompressed:** 1429 bytes
- **HTTP/1.1 compressed:** 634 bytes
- **SSE uncompressed:** 1323 bytes
- **SSE compressed:** 591 bytes

These numbers represent the size of the data fragment, which contains the actual content. They include the respective application protocol's overhead, but intentionally exclude Ethernet-, IP- and TCP overhead.

7.2.2.2 Discussion

A key insight to get from the results is, that server-sent events are noticeably faster than webhooks in every scenario. This is evident, because with webhooks, a new request is spawned for every data emission, which may also include to establish a completely new TCP connection. With SSE, a connection is held open all the time and new data is simply appended to a stream. Therefore, SSE has no overhead for establishing a new connection every request.

[4] https://www.wireshark.org/

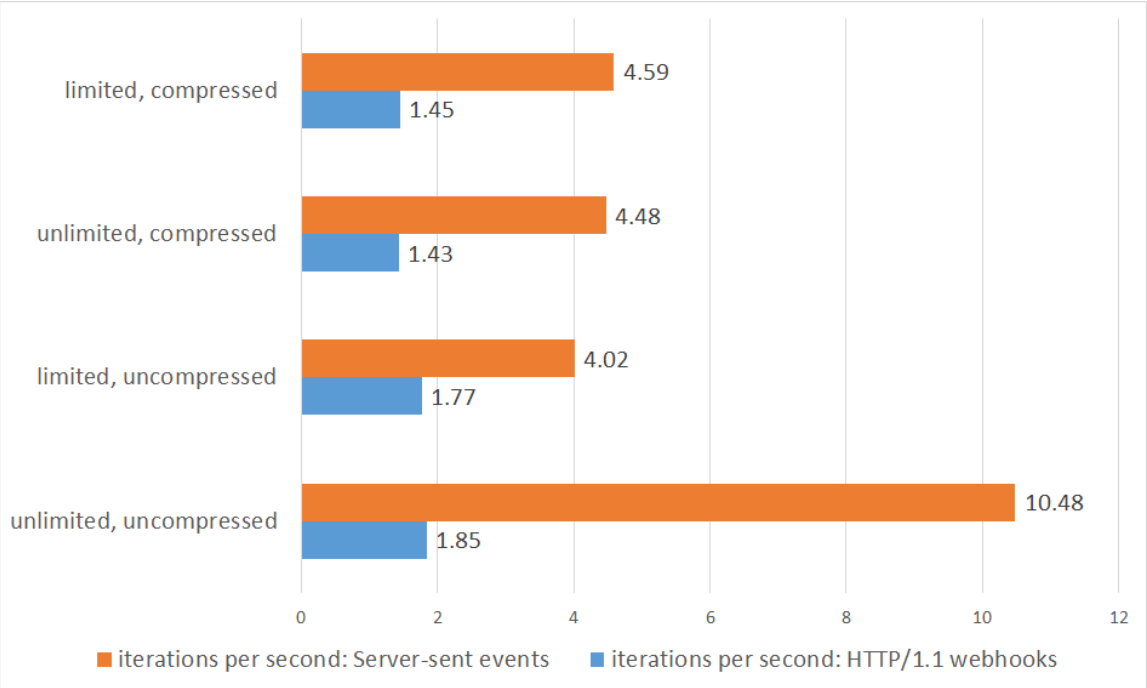

Figure 7.1: Benchmark 1: Server-sent events vs. HTTP/1.1 webhooks on constrained device

The fact that iterations per second are similar on every of the four scenarios with webhooks gives the implication that webhooks' major bottleneck is the connection establishment, rather than bandwidth or computational effort.
The fact that sending uncompressed data with unlimited bandwidth is faster than sending compressed data with unlimited bandwidth is attributable to the RaspberryPi's highly constrained CPU capability needed for compression. Therefore, having unlimited bandwidth, the CPU seems to be the major bottleneck, while with limited bandwidth, it is the network.

Remembering the goal of this benchmark to test the capability of delivering information just in time with webhooks and server-sent events, we conclude that at least webhooks can not be considered just-in-time capable for the most scenarios. With a number of 100 subscribers, a RaspberryPi is able to do a maximum of 1.8 iterations per second. While this is sufficient for OQA, one can think of other industrial use cases where a data rate much higher than 1.8 Hz is required.
Server-sent events can, according to our benchmarks, do a maximum iteration rate of 10.5 Hz. However, this is the best case and we can not assume a constant bandwidth of 100 Mbit/s in real-world scenarios, since one data producer is usually not isolated but in a network with many others. Therefore, we assume an average rate of around 4.5 Hz with 100 subscribers and a RaspberryPi producer. It depends on the use case whether this rate suffices just-in-time requirements, but generally we consider the rate still too low. As a consequence, we recommend to not use the DataApp - or any other app to be run on a constrained device - with a large number of direct subscribers without some kind of broker (e.g. our SensorhubApp) between the constrained device and its subscribers, which runs on a more powerful computing device.

7.2.3 Comparison of Publish / Subscribe Technologies

In chapter 5.3 we presented various technologies to realize publish / subscribe based communication in web scenarios and decided to employ HTTP/1.1 webhooks alongside server-sent events. However, we want to scrutinize our decision and perform a comparison between all presented technologies to get insight on their respective suitability in terms of performance - especially with regard to support for just-in-time capability.

Alongside HTTP/1.1 webhooks and SSE we investigate HTTP/2.0 webhooks, which work analogously to HTTP/1.1 webhooks, except for that they make use of the HTTP/2.0 enhancements. In a short summary, these include header compression, binary framing in transfer and server-push mechanisms, although the latter is only relevant in a Linked Data scenario and beyond the scope of this benchmark. Due to header compression and binary transmission, we expect data packets to be smaller and consequently a higher transmission rate. However, HTTP/2.0 is not widely spread, yet, and therefore only few good implementations exist for NodeJS. Furthermore, we benchmark Websocket-based communication, as well as the MQTT- and CoAP protocols. The two latter ones are regarded as the major IoT protocols due to their lightweight design. However, even if MQTT would turn out to be most efficient in our subsequent benchmarks, it was not an option for us anyway, because it would not fit into the we committed to in chapter 5.3.

We implemented a separate pair of server and subscriber for every of these technologies. For HTTP/1.1 webhooks we used *ExpressJS*[5] for the subscribers and *request*[6] for the server, just as in the actual app implementations. For the SSE version, the *eventsource*[7] and *server-event*[8] libraries were used - just like in the actual apps. To implement the HTTP/2.0 subscriber and server we employed the *spdy*[9] library and for both the CoAP subscriber and -server we used the *coap*[10] library. MQTT is a special case in the way that besides server and subscriber, which we implemented with the *mqtt*[11] library, it further necessitates a message broker. We deployed the de-facto standard, standalone C++ implementation *Mosquitto*[12].

During our benchmarks, the server application was executed on a Dell Inspiron 14z 5423 computer with an Intel i5-3317U hyper-threaded dual-core processor with 1.7 Ghz and 8 GB of DDR3 memory, running Debian 8.0 Jessie. All 100 subscribers were executed on an Apple MacBook Pro Retina with a 2.9 GHz hyper-threaded dual-core Intel Core i5 processor and 16 GB of DDR3 RAM, running Mac OS X 10.11. Both machines ran NodeJS 6.4.0.

Server-sent events are only investigated with HTTP/1.1, because of the reasons mentioned in 7.2.1. MQTT is used with a QoS level of zero to get a preferably high comparability with the other technologies, which all lack of such a strict reliability mechanism as MQTT's QoS levels. As a consequence, all technologies - expect for

[5] http://expressjs.com/
[6] https://www.npmjs.com/package/request
[7] https://www.npmjs.com/package/eventsource
[8] https://github.com/kevinmehall/server-event
[9] https://www.npmjs.com/package/spdy
[10] https://www.npmjs.com/package/coap
[11] https://www.npmjs.com/package/mqtt
[12] https://mosquitto.org/

CoAP, which sits on top of UDP - only make use of TCP's reliability mechanisms, which we consider sufficient for our platform's targeted scenarios. Additionally, we have implemented an acknowledgment mechanism at application level. For instance, in both webhooks scenarios, as well as in the CoAP scenario the client acknowledges the received data fragment by sending a unique identifier in the response. With SSE a separate SSE connection from subscriber to publisher is established to do the same kind of acknowledgement. Similarly, with MQTT and Websockets, a subscriber acknowledges a received message by emitting another event containing a unique ID to a separate channel, which the original publisher listens on. This is used to determine, how many emissions have failed and to re-perform the failed ones.

7.2.3.1 Results

For each of the six benchmark suites, we performed four tests, while varying network bandwidth and data compression, as described in section 7.2.1. The results are depicted in figure 7.2 and 7.3. These charts show the numbers of iterations per second that were measured with the respective technology. The number of iterations per second is calculated as the quotient of the total number of iterations performed (1000 in this case) divided by the total execution time.

$$iterations\ per\ second = \frac{total\ iterations\ performed}{total\ execution\ time}$$

The total number of actual data emissions can be derived from the number of iterations per second through multiplying it with the total number of subscribers (100 in this case), since 100 emissions are performed in each iteration.
Additionally, figure 7.4 compares the respective packet sizes transferred over the network as we measured them using Wireshark. The size of the plain payload data (see listing 7.1) is 1244 bytes, the size of the gzipped payload data is 435 bytes. The respective technology's overhead is the difference between its total packet size and the actual payload size, while only considering application-protocol-specific overhead and neglecting Ethernet-, IP- and TCP-implied overhead.

7.2.3.2 Discussion

The first thing to note when analyzing the above results is that both webhook-based approaches are the slowest in every scenario. This is evident and can be ascribed to the same reasons as described in 7.2.2.2, namely that webhooks are the only technology, where a new connection gets established for every single request. This implies a large network latency and slows the entire process down. With unlimited bandwidth, HTTP/1.1 is slightly faster than HTTP/2.0, although the request size is larger, as to be seen in figure 7.4. This can be attributed to the fact that header compression implies computational effort, which is not needed with HTTP/1.1. In scenarios with limited bandwidth, one can clearly see that HTTP/2.0 reveals its potential, because it utilizes the network bandwidth more efficiently due to compressed and cached headers. A more detailed comparison between these two HTTP protocol standards takes place in section 7.2.4. Another interesting insight when comparing these two HTTP versions is, that headers can be neglected almost completely with HTTP/2.0. We found that, due to header caching, the total request

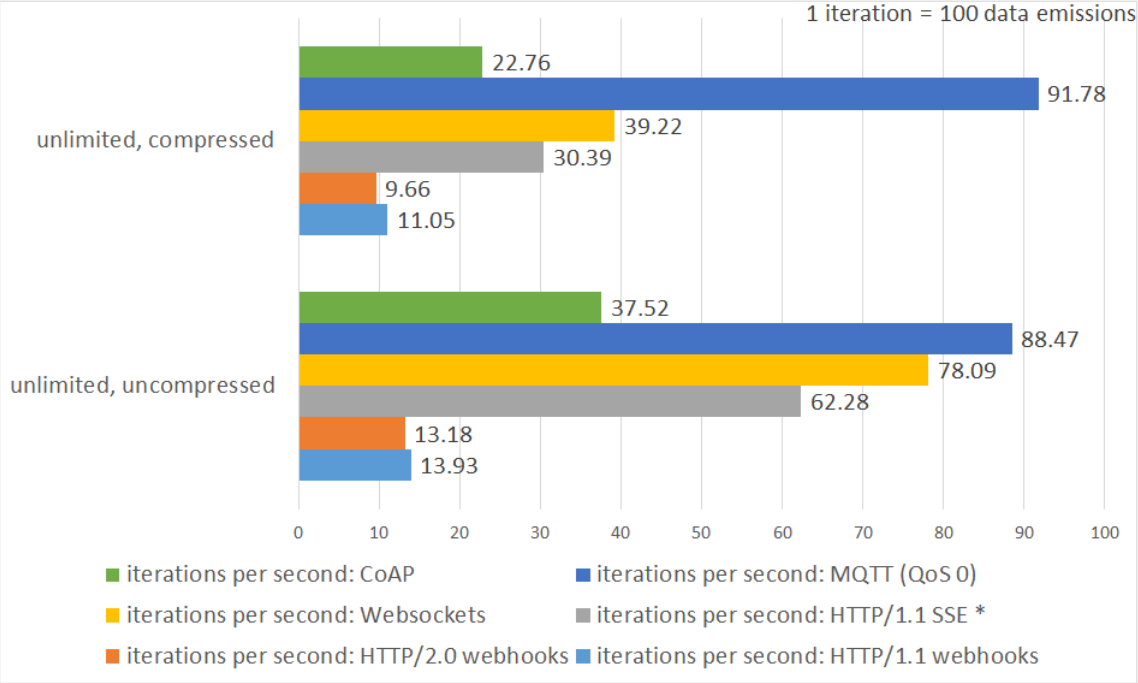

Figure 7.2: Benchmark 2: Comparison of pub / sub technologies with unlimited network bandwidth

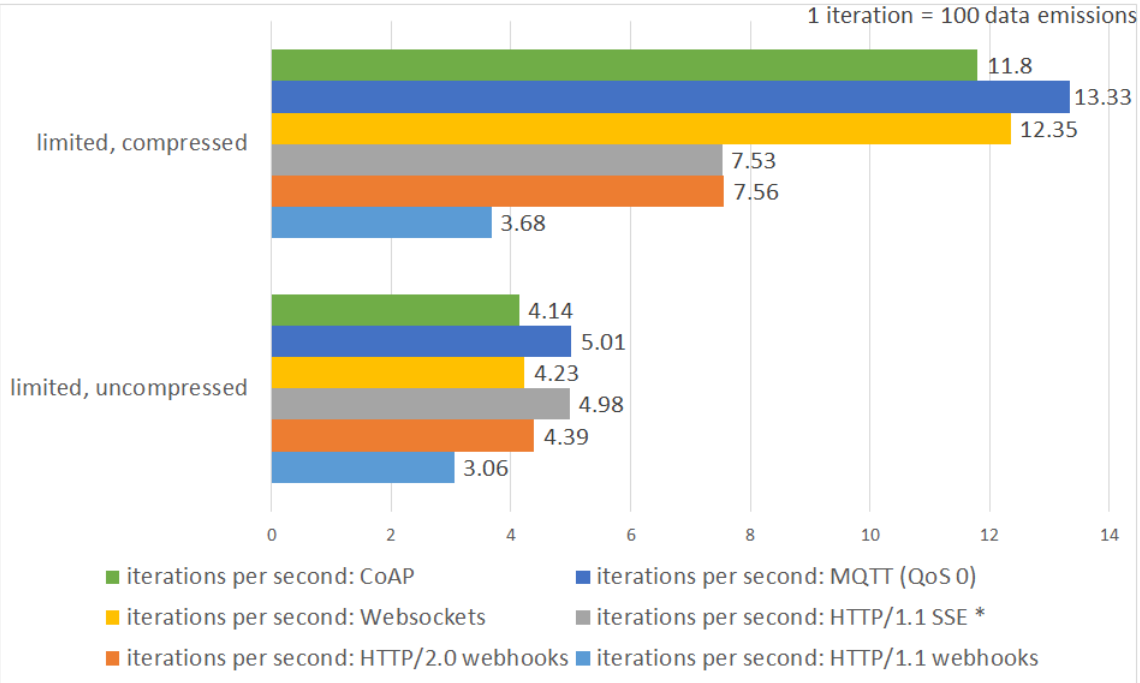

Figure 7.3: Benchmark 3: Comparison of pub / sub technologies with limited network bandwidth (5 Mbit/s up and down)

size of HTTP/2.0 headers only increases linearly by one byte per additional header field from the second request on - regardless of the header value's length.

Another thing to note is that MQTT is by far the fastest technology in three of four scenarios. It also has the second-smallest packet size when payload data is gzipped, which coincides with its claim to be very lightweight. MQTT's dramatic

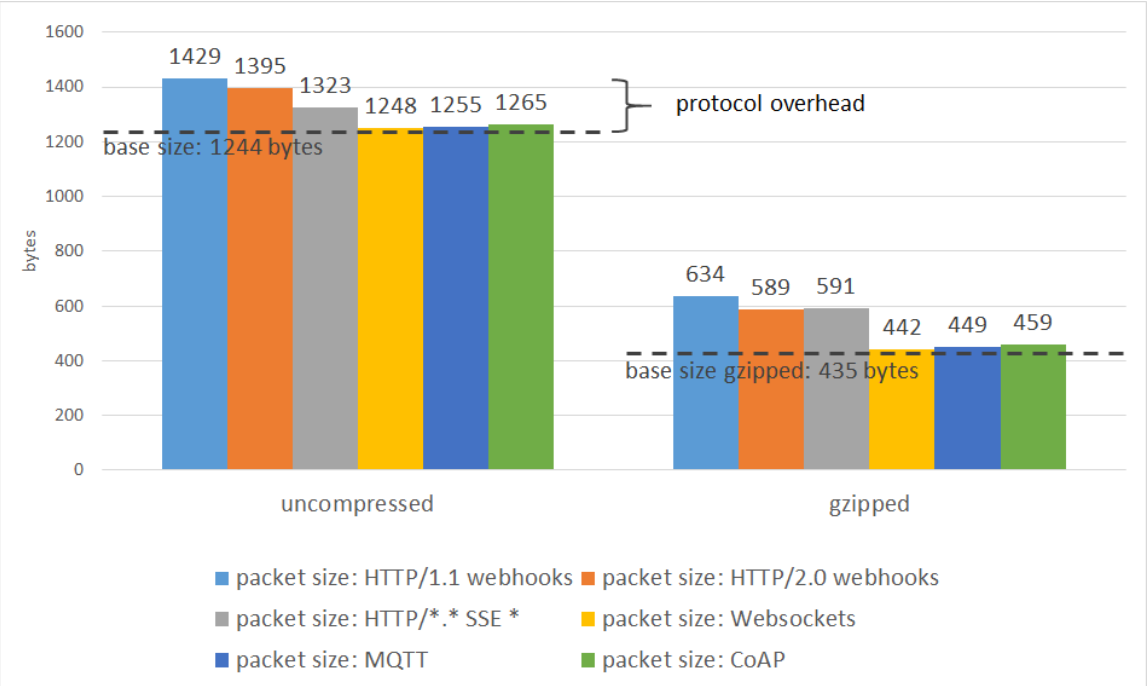

Figure 7.4: Comparison of pub / sub technologies' network packet size

lead most likely also relates to the fact that it follows a different pattern concerning message delivery. Sending the messages is not handled by the application itself but by another, standalone application - the broker - which is optimized for exactly that purpose. In our benchmarks, the server application just fans out all data to the broker, which is then responsible for delivery. The reason why we used MQTT with a QoS level of zero is explained in 7.2.3. MQTT appears to be the best choice regarding performance for the most cases. However, it is not suitable for our architecture, because we explicitly want to avoid dependency on a central message bus.

CoAP also performs good, especially with limited bandwidth, since it was designed for exactly the purpose of constrained, unreliable networks in the IoT. However, its performance does not increase a lot when dropping the bandwidth limit. While it is among the fastest with limited bandwidth, it performs worst right after webhooks with unlimited bandwidth. It appears to be not as steady as SSE or Websockets in terms of performance. Furthermore, it is not as universal as pure web technologies like SSE are. Its additional gain in performance in limited-bandwidth scenarios is not high enough for us to be negligent of our uniformity requirement.

Websockets and server-sent events perform similar in average. Before discussing their results, the user should get familiar with the reason, why we used SSE only with HTTP/1.1 and not with HTTP/2.0, which is explained in 7.2.1. The noticeably higher overhead of SSE when employing compression as depicted in 7.4 is due to the fact that SSEs do not support transmitting binary data. Instead it needs to get encoded with base64, which implies about 30 % overhead. Our test setup, where we first compress the payload data at application level before sending it to the protocol handler, does not align well with SSEs, since they do not allow for binary encoded payloads (such as gzip output). Although the payload itself must not be binary encoded, in theory the entire chunk could. In that case, the chunk might be

compressed at a lower level, namely either within the SSE- or within the HTTP-server implementation. However, implementing this would clash with the HTTP principle that chunking should always be the very last performed encoding step. Besides that, compressing single chunks would not be that efficient. Consequently, we simply view SSE not capable to transport binary data and consider base64-encoding mandatory to introduce at least any compression. Nevertheless, SSEs are more lightweight in terms of implementation, because they do not necessitate a separate server integrated to the webserver, but are built right into HTTP. Therefore we favor SSE towards Websockets.

Generally, it can be said that with unlimited bandwidth, technologies perform better with uncompressed data, while with strongly limited bandwidth it is faster to use compression. This is evident, because with unlimited bandwidth, the maximum data throughput is not a bottleneck, but the CPU time needed for compression is. With limited bandwidth, the bottleneck is shifted to the network and the time needed for compressing data has less impact than sending larger packets through a limited connection. However, since compression requires a lot of CPU time, it is not recommended on constrained devices, as 7.2.3.1 has shown. Generally, there is a trade-off between CPU time and network throughput. However, we recommend to use compression on non-constrained devices to save network usage and get higher data rates.

We conclude that SSEs have an average performance, which is very similar to the one of Websockets, below MQTT's and CoAP's, but above webhooks'. They appear to be a good compromise between performance and simplicity or uniformity. With a data rate between 7.53 Hz in worst case and 62.28 Hz in best case, depending on network capability and compression, they can be considered just-in-time capable in our specific test scenarios for many use cases.

In summary we got the following insights.

- Due to header caching, the total request size of HTTP/2.0 headers only increases linearly by one byte per additional header field from the second request on - regardless of the header value's length.
- MQTT appeared to be the fastest technology, however not dramatically faster than web technology.
- SSEs entail a 30 % larger overhead when compressing data, since it does not support binary streams, but requires base64 encoding. However, workarounds are conceivable (see section 7.2.3.3), that should be taken into account when designing a production-ready platform.
- Webhooks are only suitable as an option beside one of the other push technologies.
- In environments with constrained network bandwidth, compression pays off, while relinquishing compression appeared to be faster in high-capable networks.
- The performance gap between general purpose web technology and specialized protocols is reasonable, especially when regarding the additional gain in uniformity and simplicity.

7.2.3.3 Future Work

In addition to our investigations, there are even a few more scenarios that seem interesting to be examined. One idea is to utilize HTTP/2.0 server push to implement pub/sub. Although it is not intended to be used that way, it might be possible to trigger *PUSH_PROMISE*s at application level. Influencing the protocol's built-in mechanism this way could enable to send events within an HTTP/2.0 stream from server to client or publisher to subscriber. However, this would probably only be experimental and not a best-practice, since it is a purpose of HTTP/2.0 to be transparent to the client, which would be violated, if the application explicitly relied on protocol-level mechanisms. A more consistent approach would be to use server push in combination with server-sent events. Doing so, a server could push a data fragment to the client and subsequently send an SSE event to notify the client about the availability of new data. Afterwards the client could initiate an ordinary HTTP request to fetch the data from the server, but actually the request never gets executed over the network, since the data is already found in cache, where it was previously pushed to by server push. Although this approach might be quite error prone, it still seems to be worth to be investigated in future work.

Another potential future enhancement also utilizes server push. It builds on the fact that our platform is consequently based on Linked Data. Usually, a subscriber would not only want to receive data fragments based on semantic events, but also further process them. This usually might include to resolve some of the included URLs. To save time in such a subsequent request, the server could push, for instance, all first-level linked resources in advance. This basic idea might be pursued in future work with our platform.

7.2.4 Comparison between HTTP/1.1 and HTTP/2.0

In section 7.2.3 we compared, among others, HTTP/1.1 and HTTP/2.0 webhooks and found that the latter are especially more efficient in terms of network usage and consequently of greater performance in constrained networks. To get even better insights to these differences, we set up another, isolated benchmark suite. It aims to be more general and less coupled to our specific scenarios. Consequently, we did not implement server and subscriber with third-party NodeJS library, but built the server using Go language's built-in HTTP API and the *h2load*[13] benchmark tool, which is implemented in C, as client. Furthermore, we relinquished to have a number of separate subscribers to a server, but instead only employed one - namely the Go HTTP server. The h2load client corresponds to the server application in our previous scenarios and uses four threads to make a predefined number of 100,000 POST requests. We attached the same 1244 bytes request body and 145 bytes headers as in 7.2.3 to the requests, so we still simulate kind of a webhook approach, where h2load is the publisher. Additionally, we attached another, relatively long header of 337 bytes length. For both technologies, we performed two runs: one with unlimited (100 Mbit/s) and another with limited (5 Mbit/s) network bandwidth. h2load was executed on the Dell Inspiron-, the Go server on the Apple MacBook machine mentioned in section 7.2.3. The machines were connected via a 100 Mbit/s local area network. The results of these benchmarks are shown as the output of h2load in listing 7.2, 7.3, 7.4 and 7.5.

[13] https://github.com/nghttp2/nghttp2#benchmarking-tool

7.2.4.1 Results

The following benchmark results are the respective outputs of the h2load command-line program. The referred start scripts containing all used h2load parameters can be viewed in the appendix of this work.

Listing 7.2: h2load HTTP/1.1 benchmark with unlimited network bandwidth

```
ferdinand@ferdinand-notebook:~$ bash ./h2load_http1_100000-requests.sh

finished in 32.97s, 3033.05 req/s, 343.59KB/s
requests: 100000 total, 100000 started, 100000 done, 100000 succeeded, 0 failed, 0
    errored, 0 timeout
status codes: 100000 2xx, 0 3xx, 0 4xx, 0 5xx
traffic: 11.06MB (11600000) total, 8.11MB (8500000) headers (space savings 0.00%),
    0B (0) data
                     min         max         mean         sd        +/- sd
time for request:      588us      13.06ms      1.31ms       305us    75.40%
time for connect:    39.76ms      45.06ms     42.35ms      2.04ms    50.00%
time to 1st byte:        0us          0us         0us         0us     0.00%
req/s           :     758.27       760.24      759.33        0.81    50.00%
```

Listing 7.3: h2load HTTP/2.0 benchmark with unlimited network bandwidth

```
ferdinand@ferdinand-notebook:~$ bash ./h2load_http2_100000-requests.sh

finished in 37.00s, 2631.60 req/s, 67.01KB/s
requests: 100000 total, 100000 started, 100000 done, 100000 succeeded, 0 failed, 0
    errored, 0 timeout
status codes: 100000 2xx, 0 3xx, 0 4xx, 0 5xx
traffic: 2.49MB (2607384) total, 394.17KB (403626) headers (space savings 95.75%),
    0B (0) data
                     min         max         mean         sd        +/- sd
time for request:      598us       8.26ms      1.48ms       629us    88.78%
time for connect:    27.55ms      35.46ms     31.46ms      2.82ms    50.00%
time to 1st byte:    33.47ms      36.99ms     34.76ms      1.33ms    75.00%
req/s           :     657.90       659.79      658.83        0.83    50.00%
```

Listing 7.4: h2load HTTP/1.1 benchmark with limited network bandwidth

```
ferdinand@ferdinand-notebook:~$ bash ./h2load_http1_100000-requests.sh

finished in 290.27s, 344.51 req/s, 39.03KB/s
requests: 100000 total, 100000 started, 100000 done, 100000 succeeded, 0 failed, 0
    errored, 0 timeout
status codes: 100000 2xx, 0 3xx, 0 4xx, 0 5xx
traffic: 11.06MB (11600000) total, 8.11MB (8500000) headers (space savings 0.00%),
    0B (0) data
                     min         max         mean         sd        +/- sd
time for request:      542us     156.05ms     11.59ms     30.71ms    90.39%
time for connect:    28.97ms      36.37ms     32.91ms      2.73ms    50.00%
time to 1st byte:        0us          0us         0us         0us     0.00%
req/s           :      86.13        86.27       86.20        0.05    50.00%
```

Listing 7.5: h2load HTTP/2.0 benchmark with limited network bandwidth

```
ferdinand@ferdinand-notebook:~$ bash ./h2load_http2_100000-requests.sh

finished in 251.34s, 397.86 req/s, 10.22KB/s
requests: 100000 total, 100000 started, 100000 done, 100000 succeeded, 0 failed, 0
    errored, 0 timeout
status codes: 100000 2xx, 0 3xx, 0 4xx, 0 5xx
traffic: 2.51MB (2630661) total, 413.83KB (423757) headers (space savings 95.54%),
    0B (0) data
                     min         max         mean         sd        +/- sd
time for request:      596us     153.47ms      9.99ms     27.74ms    92.34%
time for connect:    40.56ms      46.09ms     43.30ms      2.13ms    50.00%
time to 1st byte:    49.02ms      50.90ms     50.16ms       721us    50.00%
req/s           :      99.47        99.75       99.56        0.11    75.00%
```

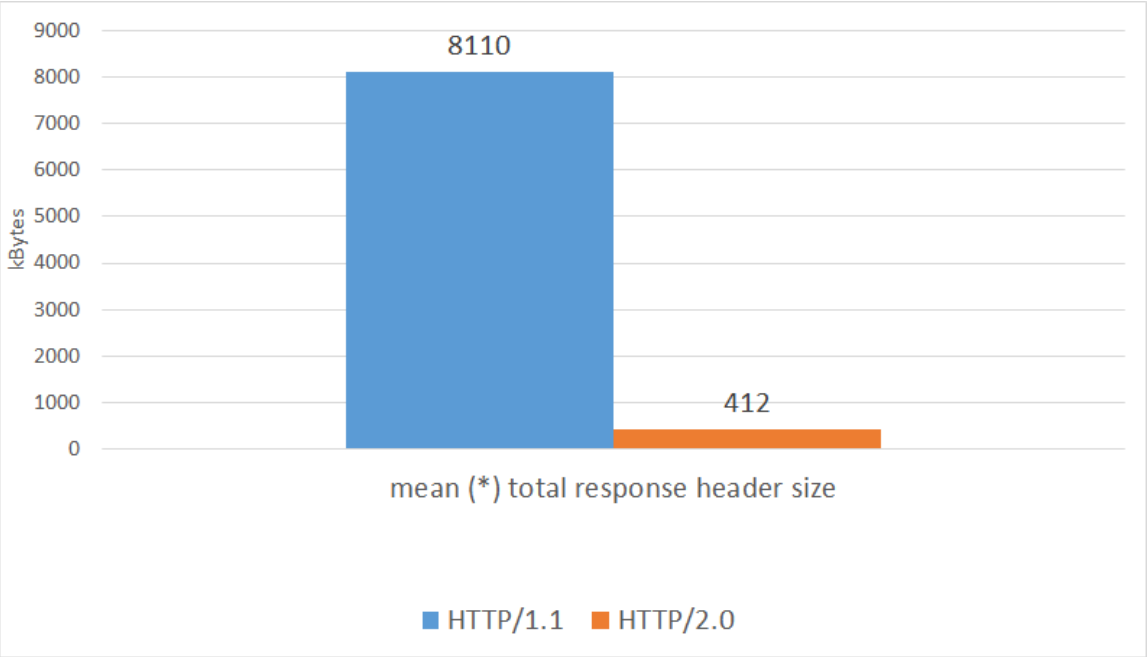

Figure 7.5: Comparison of HTTP/1.1 and HTTP/2.0 total transmitted response header sizes (mean over 12 runs) for benchmark 7.2.4

7.2.4.2 Discussion

From the results in section 7.2.4.1, we can conclude that HTTP/1.1 was around 23 % faster in scenarios with completely unlimited bandwidth, whereas HTTP/2.0 was 24 % faster with limited bandwidth. Header compression and -caching could reduce the overall received amount of header data by 95 %. This difference is depicted in figure 7.5, where the mean total header sizes for both versions of HTTP over a number of twelve h2load benchmarks are plotted.

However, an important thing to note at this point is, that h2load only measures inbound traffic, but not outgoing one. This means that, for instance, in listing 7.2 8.11 MB of headers and 11.06 MB of body payload were received from the server by h2load. The bodies and header sent by h2load are not included in the measurement and would be much higher (e.g. for HTTP/1.1 around 1.244 kB * 100,000 requests = 124.4 MB of body data). The effect, that HTTP/2.0 is faster in limited bandwidth scenarios would be even more dramatic, if the ratio between headers and body was larger or, in other words, if the request bodies were smaller, because HTTP/2.0's major performance gains are caused by header compression and -caching.

All in all, we can conclude that, given a proper implementation for the respective technology stack, the impact of additional computation effort for header compression with HTTP/2.0 is negligible and by far overcome by the additional performance gain in networks where the traffic reaches the physical limits of the medium. Consequently, we recommend to employ HTTP/2.0 to any web application as soon as it is widely spread enough.

8. Conclusion & Future Work

8.1 Summary

This final chapter provides a summary of the previous chapters and outlines the key considerations made while designing a dynamic software platform for the Industrial Internet of Things. We point out to which degree the initial intention of this work was met as well as related key problems. Furthermore, we clarify the relevance of this work together future research to be conducted on our topic.

After a brief introduction to the topic, related research questions and the intention of this work in chapter 1, basic concepts and background knowledge were introduced in chapter 2. Meaning, characteristics, principles and problems of Industry 4.0 were described. Technical concepts to be used during design were explained, including semantic web and Linked Data techniques as well as common design patterns concerning software architecture and -communication.

Afterwards a real-world use case, based on the needs of real-world industry partners, was presented in chapter 3 to have a guideline during design. The use case was about dynamically publishing and accessing sensor data, which are emitted by optical inspector machines, among different machine- and human participants within a company.

In chapter 4, we abstractly designed a platform architecture. First, a set of requirements were stated in order to ensure our platform to meet our targeted goals. Afterwards, we agreed on implementing a highly modular architecture on the basis of Microservices, which should be represented as apps. To enhance user experience and ease of use, we decided to add an app store as a central platform component. Technical thoughts on how to dynamically integrate heterogeneous data, as well as on how to autonomize machine-clients in terms of composition and communication were made. As an outcome, we decided to employ Linked Data techniques and hypermedia APIs within a uniform web technology landscape.

In chapter 5 we discussed various technologies to be used for event-based communication and agreed on a technology stack based on NodeJS, HTTP webhooks, server-sent events and the Hydra vocabulary to self-describe APIs. We presented the design

of a universal interface to be implemented by every app and demonstrated suitable deployment approaches using containers. Finally, we implemented a set of concrete apps according to the use case from chapter 3 and an app store as a management console for app deployment and showed how these apps meet the requirements from chapter 4.

Having finished the key part of architecture design and -implementation, we examined related research projects and industry alternatives with their analogies and differences to our approach. We could conclude, that, to the best of our knowledge, no truly alternative platform approach as comprehensive as ours does exist. Although the work from Guinard and Trifa [19] created a broad platform, they do not elaborate on architecture as detailed as we do and put their focus on neither industrial scenarios nor machine autonomy or dynamic service compositions.

In the final chapter 7 we evaluated our platform architecture with regard to different aspects. We explained to which extent we can consider our initial requirements to be met and presented and discussed criticism of industry experts derived from out workshop with them. As a second part, we conducted performance benchmarks in order to investigate the platform's just-in-time capability and to ratify our technology choices from chapters 4 and 5 in comparison with alternatives.

8.2 Conclusion and Relevance

To classify the relevance and impact of this work we can conclude the following. We presented an entirely new approach to build an Industrial IoT platform based on web technology, while integrating Linked Data techniques and enabling autonomous machine-to-machine interaction. Besides designing for technical performance, scalability and uniformity, we also heavily lied focus on user experience and ease of use for both end-users and developers. The latter should help companies to easily tap into the potential of digitization without requiring comprehensive knowledge and high upfront planning and -costs for adoption. While our platform is not yet production ready, our results can act as both a guideline for professional-level implementations and a proof-of-concept for the capabilities of web technology in combination with state-of-the-art development practices in this field of research.

8.3 Future Work

Future work might elaborate on the ideas and concepts presented in this thesis in even more detail and implement an actual production-ready solution to be tested in a real-world shop floor scenario. The service discovery mechanisms designed in this work might be refined to be more smart and efficient. In addition, frameworks to allow for auto-generating semantic data- and API descriptions will be required in the future. All in all, this work is to be seen as a guideline and basic idea for manufacturing companies willing to tap into the potential of increasing their performance flexibility and productivity by adopting Industry 4.0. It should demonstrate how existing technology can be combined to facilitate the IoT and teach best practices on not only architecture design and technical implementation, but also on development techniques and culture.

Bibliography

[1] Armin Roth, ed. *Einführung und Umsetzung von Industrie 4.0*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. ISBN: 978-3-662-48504-0.

[2] Mario Hermann, Tobias Pentek, and Boris Otto. "Design Principles for Industrie 4.0 Scenarios". In: *2016 49th Hawaii International Conference on System Sciences (HICSS)*. IEEE, 2016, pp. 3928–3937. ISBN: 978-0-7695-5670-3.

[3] Daniel Giusto et al. *The Internet of Things: 20th Tyrrhenian Workshop on Digital Communications*. 2010, p. 452. ISBN: 9781441916730.

[4] Edward A. Lee. "Cyber Physical Systems: Design Challenges". In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 2008, pp. 363–369. ISBN: 978-0-7695-3132-8.

[5] Thomas Bauernhansl. "Die vierte industrielle Revolution - Der Weg in ein wertschaffendes Produktionsparadigma". In: *Industrie 4.0 in Produktion, Automatisierung und Logistik*. 2014, pp. 5–35. ISBN: 978-3-658-04681-1. eprint: 9809069v1.

[6] Dominik Lucke, Carmen Constantinescu, and Engelbert Westkämper. "Smart Factory - A Step towards the Next Generation of Manufacturing". In: *The 41st CIRP Conference on Manufacturing Systems*. 2008, pp. 115–118. ISBN: 978-1-84800-267-8.

[7] Henning Kagermann. "Change through digitization - value creation in the age of industry 4.0". In: *Management of Permanent Change*. 2015, pp. 23–45. ISBN: 9783658050146.

[8] Andreas Harth. *Introduction to Linked Data*. Preview. 2016. ISBN: 978-1-4973-6478-3.

[9] Tim Berners-Lee. *Linked Data*. https://www.w3.org/DesignIssues/LinkedData.html. Last accessed 22 April 2016. 2014.

[10] Martin Fowler. *Microservices*. http://www.martinfowler.com/articles/microservices.html. Last accessed 18 April 2016. 2014.

[11] Eric Evans. *Domain-driven design : tackling complexity in the heart of software*. 1. print. Boston: Addison-Wesley, 2004. ISBN: 0-321-12521-5.

[12] Yury Izrailevsky and Ariel Tseitlin. *The Netflix Simian Army*. http://techblog.netflix.com/2011/07/netflix-simian-army.html. Last accessed 13 August 2016. 2011.

[13] Michael T. Nygard. *Release it! : design and deploy production-ready software*. Pragmatic Bookshelf, 2007, p. 350. ISBN: 9780978739218.

[14] Mark Richards. *Microservices vs. Service-Oriented Architecture*. Ed. by Nan Barber and Rachel Roumeliotis. Sebastopol: O'Reilly Media, 2016. ISBN: 978-1-491-95242-9.

[15] Tomasz Janczuk. *What is serverless*. https://auth0.com/blog/what-is-serverless/. Last accessed 13 August 2016. 2016.

[16] *Defining "Serverless" and Why It Matters to Developers | blog.serverless.com*. http://blog.serverless.com/defining-serverless/. Last accessed 05 September 2016. 2016.

[17] Boris Otto et al. "INDUSTRIAL DATA SPACE - White Paper". https://www.fraunhofer.de/content/dam/zv/en/fields-of-research/industrial-data-space/whitepaper-industrial-data-space-eng.pdf. Last accessed 30 October 2016. München, 2016.

[18] Neal Ford and Rebecca Parsons. *Microservices as an Evolutionary Architecture*. https://www.thoughtworks.com/insights/blog/microservices-evolutionary-architecture. Last accessed 30 October 2016. 2016.

[19] Dominique D. Guinard and Vlad M.Trifa. *Building the Web of Things*. Vol. 2. Zurich: Manning, 2015.

[20] Martin Abbott. *The art of scalability : scalable web architecture, processes, and organizations for the modern enterprise*. New York: Addison-Wesley, 2015. ISBN: 0134032802.

[21] Christian Posta. *Why Microservices Should Be Event Driven: Autonomy vs Authority*. http://blog.christianposta.com/microservices/why-microservices-should-be-event-driven-autonomy-vs-authority/. Last accessed 15 August 2016. 2016.

[22] Sam Newman. *Building Microservices*. 1. ed. Beijing: OReilly, 2015. ISBN: 978-1-491-95035-7.

[23] Martin Fowler. *Event Sourcing*. http://martinfowler.com/eaaDev/EventSourcing.html. Last accessed 15 August 2016. 2005.

[24] Martin Fowler. *MicroservicePrerequisites*. http://martinfowler.com/bliki/MicroservicePrerequisites.html. Last accessed 22 April 2016. 2014.

[25] Christian Posta. *The Real Success Story of Microservices Architectures*. http://blog.christianposta.com/microservices/the-real-success-story-of-microservices-architectures/. Last accessed 15 August 2016. 2015.

[26] Rouan Wilsenach. *DevOpsCulture*. http://martinfowler.com/bliki/DevOpsCulture.html. Last accessed 30 October 2016. 2015.

[27] J. Richard Hackman and Diane Coutu. *Why teams don't work*. 2009.

[28] Martin Fowler. *ContinuousDelivery*. http://martinfowler.com/bliki/ContinuousDelivery.html. Last accessed 15 August 2016. 2013.

[29] Martin Fowler. *InfrastructureAsCode*. http://martinfowler.com/bliki/InfrastructureAsCode.html. Last accessed 15 August 2016. 2016.

[30] Guido Steinacker. *On Monoliths and Microservices*. https://dev.otto.de/2015/09/30/on-monoliths-and-microservices/. Last accessed 24 August 2016. 2015.

[31] Phil Calçado. *Building Products at SoundCloud —Part I: Dealing with the Monolith*. https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith. Last accessed 22 April 2016.

[32] Martin Fowler. *Event Collaboration*. http://martinfowler.com/eaaDev/EventCollaboration.html. Last accessed 15 August 2016. 2006.

[33] Jon Udell. *Jon Udell: The Event-Driven Internet*. http://jonudell.net/bytecols/2001-11-26.html. Last accessed 16 August 2016. 2001.

[34] Milenko Petrovic, Ioana Burcea, and Hans-Arno Jacobsen. "S-ToPSS: Semantic Toronto Publish/Subscribe System". In: *Proceedings of the 29th international conference on Very large data basesVolume 29* (2003), p. 4. eprint: 0311041 (cs).

[35] Martin Fowler. *Richardson Maturity Model*. http://martinfowler.com/articles/richardsonMaturityModel.html. Last accessed 23 June 2016. 2010.

[36] Marcus Sporny et al. *JSON-LD 1.0*. https://www.w3.org/TR/json-ld/{#}basic-concepts. Last accessed 30 October 2016. 2014.

[37] Markus Lanthaler and Christian Gütl. "Hydra: A vocabulary for hypermedia-driven web APIs". In: *CEUR Workshop Proceedings*. Vol. 996. CEUR-WS, 2013.

[38] Matthias Thoma, Klaus Sperner, and Torsten Braun. "Service descriptions and linked data for integrating WSNs into enterprise IT". In: *2012 3rd International Workshop on Software Engineering for Sensor Network Applications, SESENA 2012 - Proceedings*. 2012, pp. 43–48. ISBN: 9781467317931.

[39] Chris Richardson. *Client-side service discovery pattern*. http://microservices.io/patterns/client-side-discovery.html. Last accessed 24 August 2016. 2014.

[40] Adam Wiggins. *The Twelve-Factor App*. http://12factor.net/. Last accessed 18 April 2016. 2012.

[41] Martin Fowler. *TolerantReader*. http://martinfowler.com/bliki/TolerantReader.html. Last accessed 18 August 2016. 2011.

[42] Wikipedia. *App Store*. https://en.wikipedia.org/wiki/App_store. Last accessed 18 August 2016. 2016.

[43] Payam Barnaghi and Mirko Presser. "Publishing linked sensor data". In: *CEUR Workshop Proceedings*. Vol. 668. 2010. ISBN: 9781424466221.

[44] Amit Sheth, Cory Henson, and Satya S. Sahoo. "Semantic sensor web". In: *IEEE Internet Computing* 12.4 (2008), pp. 78–83. ISSN: 10897801.

[45] Ralf Reussner. *Handbuch der Software-Architektur*. Heidelberg: Dpunkt-Verl, 2009. ISBN: 978-3-89864-559-1.