JOONAS JÄRNSTEDT

DESIGN OF AN ENTERPRISE RESOURCE PLANNING SYSTEM USING SERVICE-ORIENTED ARCHITECTURE

Master of Science thesis

Examiner: Prof. Petri Ihantola
Examiner and topic approved by the
Faculty Council of the Faculty of
Pervasive Computing
on 5th October 2016

ABSTRACT

JOONAS JÄRNSTEDT: Design of an Enterprise Resource Planning System Using Service-Oriented Architecture
Tampere University of Technology
Master of Science thesis, 52 pages
May 2017
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Prof. Petri Ihantola
Keywords: service-oriented architecture, enterprise resource planning, SOA, ERP, REST, API

A typical ERP (Enterprise Resource Planning) system integrates many business functionalities such as billing, accounting, order processing, manufacturing and customer relationship management. Generally they are implemented as separate modules that are changed and modified as needed. However, lack of proper design can cause the modules to become tightly coupled. Unnecessary dependencies decrease maintainability because change in one part of the system can affect other parts. Large enterprise applications are often build by multiple organizations using varying methods and technologies. Development teams are not always fully aware of the work of others. This makes reusing existing features difficult and causes developers to recreate the same logic multiple times.

This thesis describes how a monolithic architecture can be migrated into a service-oriented architecture (SOA). An ERP system designed for Apple service providers is used as an example. The large example system is converted into small services to improve its maintainability. The aim is to create an architecture that does not depend on a specific technology and supports reusing functionality. Although SOA has many benefits, effective use of it requires technological changes and adjustments to the whole development process. In the example system, lots of changes were made to the development, testing and deployment processes. The new architecture lead to eliminating lots of manual work. This was achieved by implementing a deployment pipeline that takes advantage of Docker virtual containers.

TIIVISTELMÄ

JOONAS JÄRNSTEDT: Toiminnanohjausjärjestelmän suunnittelu käyttäen palvelupohjaista arkkitehtuuria
Tampereen teknillinen yliopisto
Diplomityö, 52 sivua
Toukokuu 2017
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Petri Ihantola
Avainsanat: palvelupohjainen arkkitehtuuri, palvelupohjaiset järjestelmät, toiminnanohjausjärjestelmä, SOA, ERP REST, API

Toiminnanohjausjärjestelmä yhdistää tyypillisesti monia yrityksen toimintoja kuten laskutusta, kirjanpitoa, varastonhallintaa, tuotannonohjausta sekä asiakastietojen hallintaa. Nykyaikaisessa järjestelmässä nämä toiminnot voidaan toteuttaa erillisinä moduuleina, joita kehitetään ja lisätään tarpeen mukaan. Ilman suunnittelua osien välille voi kuitenkin syntyä tarpeettomia riippuvuuksia, jotka tekevät järjestelmän ylläpitämisestä haastavaa. Suurta järjestelmää on yleensä kehittämässä useita organisaatioita, jolloin käytetyt tekniikat ja toimintatavat voivat vaihdella. Tämä voi johtaa yhteensopimattomiin ja päällekkäisiin toteutuksiin, jotka ovat kalliita ylläpitää.

Työssä tutkitaan miten toiminnanohjausjärjestelmä voidaan jakaa itsenäisiksi palveluiksi, jotta ison järjestelmän laajentaminen olisi tehokkaampaa. Esimerkkinä käytetään Apple huoltoliikkeille suunniteltua toiminnanohjausjärjestelmää. Työ kuvaa lyhyesti järjestelmän toiminnalliset sekä tekniset vaatimukset ja havainnollistaa miten järjestelmä jaettiin itsenäisiksi palveluiksi. Palvelupohjainen arkkitehtuuri on teknologiariippumaton ratkaisu. Sen tarkoitus on parantaa ohjelmistojen uudelleenkäytetävyyttä sekä jakaa isot kokonaisuudet helpommin ymmärrettäviksi palveluiksi. Se eroaa kuitenkin monella tapaa perinteisestä monoliittisesta arkkitehtuurista. Esimerkkijärjestelmässä palvelupohjaisuus vaikutti erityisesti valittuihin testaus- ja kehitysmenetelmiin, jotka toteuttiin käyttäen Docker-virtualisointitekniikkaa.

PREFACE

This thesis started in the winter of 2015-16. I had just shifted to a new start-up company that had decided to create a new version of their software product. The new version was going to be a huge remake and the software needed an improved architecture design. My job was to help with the design. We studied countless technology options and always tried to choose the best for our use case. This created the basis for this thesis and motivated me to start researching service-oriented architecture.

I would like to thank everyone at Voltio who has discussed about the design with me. We had many great debates that lead me to learn more about software architecture. I would like to thank my examiner, Petri Ihantola for helping with the writing process and for all the valuable feedback. I would like to thank my family for the support they have given me. A special thank you to Milla for your love and care.

Tampere, 24.5.2017

Joonas Järnstedt

TABLE OF CONTENTS

1. Introduction 1
1.1 Need for the Research 2
1.2 Goals 3
1.3 Structure 4
2. Service-oriented architecture 5
2.1 History of SOA 6
2.2 Benefits of SOA 6
2.3 Reliability 7
2.4 Scalability 8
2.5 Web Services 9
2.6 Representational State Transfer 9
2.7 Stateless and Stateful Services 10
2.8 Virtualization 12
3. Management Software Requirements 14
3.1 Apple Authorized Service Providers 14
3.2 An Example Use Scenario 15
3.3 Functional Requirements 16
3.4 Non-functional Requirements 17
3.5 Defining the Required Services 18
4. Architecture, Testing and Deployment 20
4.1 Monolithic Architecture 20
4.2 Separating the Application to Services 21
4.3 Connecting Services 23
4.4 API Gateway 24
4.5 Multitenancy 26

4.6 Authentication 29
4.6.1 OAuth 2.0 30
4.6.2 Authentication as a Service 32
4.6.3 JSON Web Tokens 35
4.7 Deployment 36
4.7.1 Docker 37
4.7.2 Deployment Pipeline 38
5. Evaluation 42
6. Conclusions 45
Bibliography 47

LIST OF FIGURES

1.1 Voltio provides management software for multiple AASPs. 3

2.1 Example of service-oriented architecture. 5

2.2 A load-balanced server cluster. 11

2.3 Server virtualization. 12

3.1 Connections between different parties involved in a repair case. 16

4.1 Old monolithic architecture of SirWise. 21

4.2 SirWise services. 22

4.3 All services implement a REST API. 23

4.4 Overview of the new SirWise architecture. 25

4.5 Multiple client applications and tenants. 27

4.6 SirWise authentication service and database architecture. 33

4.7 Access token validation. 34

4.8 Comparison of virtual machines and Docker containers. 38

4.9 Deployment process. 39

4.10 Docker deployment. 40

4.11 Blue-green deployment. 41

LIST OF ABBREVIATIONS AND SYMBOLS

AASP	Apple Authorized Service Provider
API	Application Programming Interface
CI	Continuous Integration
ERP	Enterprise Resource Planning
GSX	Global Service Exchange
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
RAM	Random-Access Memory
REST	Representational State Transfer
SMS	Short Message Service
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
WWW	World Wide Web
XML	Extensible Markup Language

1. INTRODUCTION

A typical enterprise information system has multiple subsystems. The subsystems are designed by different companies using varying technologies. New systems are added when needed and the integrations are not always designed carefully. This easily causes unnecessary dependencies between the systems.

When the systems are not designed to work together, it is hard to reuse functionalities from other subsystems. This forces software developers to implement the same functionality multiple times. The same feature will have many implementations in different parts of the system and after a while this type of environment will become complicated and hard to maintain. Software developers have to understand multiple different subsystems to make changes to one part of the application and updating one part of the application can have undesirable effects to the performance of the whole system.

This is one of the problems service-oriented architecture (SOA) was designed to solve. SOA is a software architecture that guides how software should be structured. It is not a technology standard and does not depend on any specific protocols. It is an architecture blueprint that can be implemented by using many different technologies.

While SOA has become a widely used acronym in software design, the term still has lots of variation in its use and there is lots of confusion about the terms related to it. The World Wide Web Consortium (W3C) defines SOA as "A set of components which can be invoked, and whose interface descriptions can be published and discovered" [28]. This is a technical description and considers SOA as a technical architecture implementation. It does not describe a style or process of building the architecture. The Open Group's definition of SOA is a lot wider: "Service-Oriented Architecture (SOA) is an architectural style that supports service-orientation" [48]. Service-orientation means thinking in terms of services. Features are designed as services which are self-contained and represent a repeatable business activity.

SOA promotes re-usability of services and agile development. In SOA, each service is an autonomous piece of software that performs a certain business function. An important aspect of SOA is that it takes away the focus from technology oriented entities like database rows or Java objects and focuses on business-centric services [37]. "The focus is on defining cleanly cut service contracts with a clear business orientation" [37]. Abstract services which implement a clear business need, such as a weather service or restaurant reservation service, are easier to describe for somebody who does not work in the software industry. Because of this SOA can bring software engineering closer to the business point of view.

1.1 Need for the Research

Voltio Oy is a Finnish startup company that develops management software for Apple authorized service providers (AASPs). An AASP is a company that is authorized to provide repair services to all Apple customers [5]. World wide there exists thousands of AASPs. A typical AASP employs multiple technicians who make device repairs. Voltio's management software is designed to speed up the repair process, provide tools for better customer communication and help handling internal resources and workflows.

The provided management software can be classified as enterprise resource planning (ERP) software. ERP is a category of business management software that generally integrates multiple applications together [7, 16]. It can be used to manage data from many business activities. Service providers need ERP software especially for handling service orders, billing, shipping and inventory management.

Since 2015 Voltio has operated the first version of the management system called SirWise. SirWise has been used to fill over 100,000 service orders in Finland and Sweden. To scale the software for more customers, a new version was needed to speed up the installation process for new service providers. This project was started in the beginning of 2016 to create a second version of the SirWise management system. For the second version the goal was to create a service-oriented architecture that allows faster development speed of custom components and allows customers to select and install their own integrations.

As many AASPs have a unique work-flow and even repair technicians work differently based on their preferences, the software has to be extendable. This means

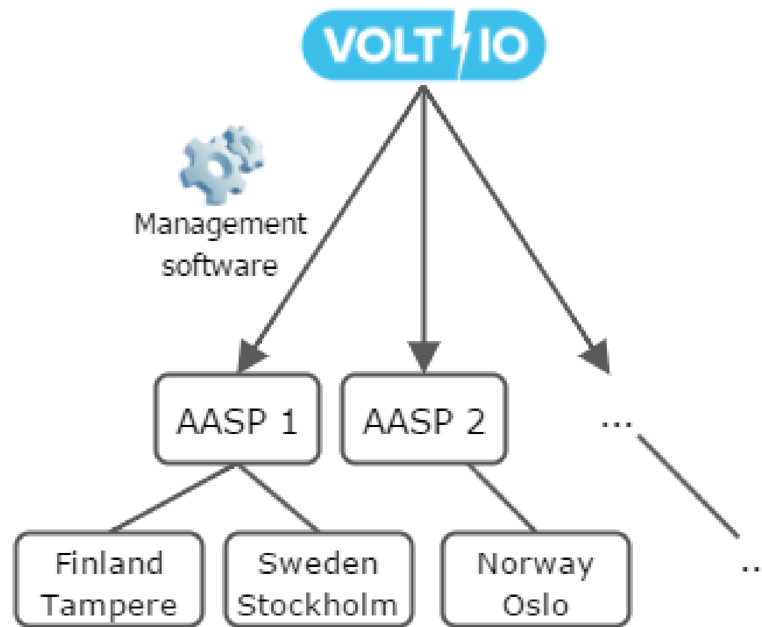

Figure 1.1 *Voltio provides management software for multiple AASPs. Each AASP can have one or more service locations.*

that the software has to support different organization structures and sizes, operating platforms, languages and user interfaces. Figure 1.1 displays Voltio related to its AASP customers. The management software serves AASPs in multiple countries and service locations. Scaling the software to thousands of service providers requires that the system can be installed without lots of manual work.

SOA suits well in situations where different parts of the system are developed by different companies. When a large group of people is responsible for an application, it is difficult to determine who is accountable for a specific function in the application [27]. For a SOA service there can be a specific group who is responsible for the functionality and can be held accountable. For SirWise it is important that new features can be implemented without re-writing the existing implementations and developers are not limited to certain implementation technologies. SOA can be used because it does not require any specific technology.

1.2 Goals

The goal of this thesis is to design a new version of SirWise that is easier to develop, deploy and scale. The new version of SirWise is designed using service-oriented

architecture. The management system for service providers is used as an example case to demonstrate why SOA is needed and what challenges implementing it creates. The old and new architecture are compared based on reusability, ease of testing and deployment, fault tolerance, scalability and development environment setup.

Deploying the old version of SirWise requires that the whole application is deployed at once. For example, making a simple change to the user interface requires updating the whole application. Deploying the application is done manually, which can cause downtime.

The old version of SirWise requires lots of time to setup a development environment that is identical to the production environment. Everything has to be installed manually by the developer. The old version has a monolithic architecture that can be hard to extend without breaking code modularity and introducing bugs to other parts of the system. The monolithic architecture also makes reusing functionality difficult for developers working outside of the core development team.

1.3 Structure

This document is structured as follows. Chapter 2 explains briefly service-oriented architecture and technologies that are commonly used to implement it. Chapter 3 specifies software requirements for the management system and defines the different services needed for the implementation. Chapter 4 describes the architecture of the example application and what SOA related challenges were solved. It also explains tools and methods that are used to deploy the application. Chapter 5 has a comparison between the old and new management system architecture and it describes the differences in testing, developing, deploying and maintaining the application. Chapter 6 summarizes the results and the whole thesis.

2. SERVICE-ORIENTED ARCHITECTURE

Service-oriented architecture (SOA) is an architectural style in software engineering [40]. Its main goal is to achieve loose coupling among interacting software agents [31]. This means building independent services that implement only a certain functionality, such as providing weather data, checking customer credit or sending an email.

Figure 2.1 shows an example how the architecture of an online store could look like if it was implemented using SOA. The architecture is separated into four independent services: a store front which displays the store's user interface, an inventory service that manages warehouse data, a customer service that saves customer information and a shipping service that handles delivering products to customers.

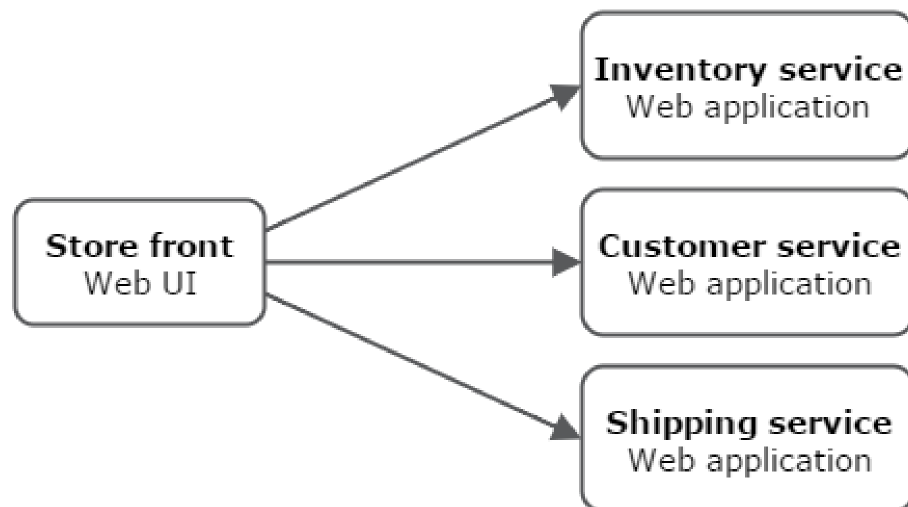

Figure 2.1 *An example of service-oriented architecture.*

Each service has a clear business function that can be used by other services. In SOA all communication between services is done through defined interfaces. A service from the consumers point of view is a "black box" [48] that has an interface. The interface abstracts the underlying complexity of a service and the consumer is able to

access it without knowledge of the service's implementation details. From a software developers perspective this makes development easier: A developer can concentrate on developing only one service without knowing the details about how other services work.

2.1 History of SOA

The first reports mentioning the term SOA were published in 1996 by Gartner Group [36, 60]. Although multiple companies in the '90s were already using similar concepts [58], they used different terms to describe their implementations. The concept itself has been used for a long time in distributed computing, especially in the finance and telecom industries [59]. The term SOA became established after there already were multiple implementations of the architecture, which makes it difficult to determine who first invented the idea.

SOA is also not the only architectural style for integrating distributed systems. For example, CORBA (Common Object Request Broker Architecture) is a standard developed by the Object Management Group for connecting distributed software systems [47]. CORBA enables systems that are implemented using different technologies to communicate with each other. The first version CORBA 1.0 was released in 1991 [47]. Unlike SOA that divides the architecture into services, CORBA applications are composed of objects. Each object has an interface that is defined using an interface definition language. Clients can then use this interface to invoke operations on the objects.

The big driver that brought SOA to the attention of mainstream users was web services [36]. Instead of requiring new communication protocols, SOA could take advantage of established protocols like HTTP and HTTPS [51]. Another reason for SOAs popularity is that large software companies like Microsoft, IBM and Oracle started promoting it [59]. Companies marketed lots of tools and platforms that support SOA.

2.2 Benefits of SOA

SOA does not rely on any specific vendor, product or technology [42]. It is a set of architectural principles. By being just a collection of best practices [11], it allows

developers to choose the best tools and technologies individually for each service being built. Although SOA does not promote of using different technologies for each service's implementation, the architecture allows it, as having many different technologies is often reality for large software projects.

SOA is suitable in scenarios where different parts of the system are developed by separate teams or companies. When services are designed to be loosely coupled, teams can develop them independently of each other. In a tightly coupled system teams would have to wait for one part to finish before continuing to develop the depending parts.

From the business perspective SOA can give more control to the organizations. Instead of creating reusable code structures like classes, the architecture promotes reuse of services. As each service implements a clear business functionality, the organization starts understanding existing systems and their interfaces. This can promote innovation and lower the time needed to get products to the market.

Some software architects argue that SOA can help businesses become more agile [22, 21]. The business can respond to change more quickly and more cost-effectively. This will result to an improved return on investment.

2.3 Reliability

Since it is extremely difficult to create failure-free systems under limited development resources and time, fault-tolerance is important for building reliable systems [65]. Fault-tolerance in software means that the system is able to operate properly even if some of its components fail to work [34]. In SOA this is particularly important when systems rely on services controller by third parties [43].

Nascimento et al. [43] have defined five groups of failure modes in SOA-based applications:

1. Delay failure, which is a failure to operate at the prescribed time
2. Output failure, when the output value provided by the service is incorrect
3. Specification failure, when a service performs a task that differs from what their name or description suggests

4. Server failure, a failure in the application server
5. Communication failure, a failure in the network

The designer of the application should determine how a failure mode should be detected, how it is handled and what is the best fault tolerance technique to be used. For example, when a service has performance problems and it cannot complete a request in a specified time, the fault is categorized as a delay failure. If the fault is caused by server overload, the failure handling could be to change the server. However if a fault is caused by an software error, running the same code on a different server will not fix the problem. In this case if there are no variants of the service, the application has to use some kind of fault tolerance technique.

In the service provider context some software faults can be tolerated but all faults should be detected. For example, if a third party integration fails, the system should still be able to operate and inform the users about degraded service level.

In some cases there are multiple variants of a service. Variants are software components, which "have the same or an equivalent specification but with different designs and implementations" [43, 39]. For example, when fetching device data for SirWise, there can be two or more services that have similar data. Different services might have different priorities and for certain operations one service can be preferred over another. In case one of the variants is not working properly, the system should be able to switch to the next variant.

2.4 Scalability

A simple approach to scaling a web application for more users and higher usage is to upgrade the server. This means adding more resources like a better processor, more memory, more storage and higher networking capacity. Cloud computing makes the process of hardware upgrades inexpensive, as it is possible to pay only for used resources [8]. This kind of scaling where more resources are added to an already running instance is called vertical scaling [66].

Vertical scaling only works to a certain extent as one single machine has its capacity limits. Adding more instances instead of adding resources to existing instances is called horizontal scaling [66]. In SOA where every service works independently, it

is possible to have each service running in a different instance. Each service can be scaled independently by using load-balancers to distribute server load between all service replicas [66]. In a monolithic architecture this is not possible and the only option is to replicate the whole application.

2.5 Web Services

Web services are a common way to implement a loosely coupled architecture and they can be used to implement SOA. Web services are not essentially service-oriented and are not a mandatory component of SOA — they merely have the capability for implementing SOA [11].

The W3C defines a Web service as a "software system designed to support interoperable machine-to-machine interaction over a network" [28]. To be more specific, Web services can be characterized as "self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces" [1]. This means that a Web service has a published interface that can be used over the Internet using standard protocols.

The two main styles to implement SOA using Web services are SOAP (Simple Object Access Protocol) and REST (Representational State Transfer). SOAP is a language and protocol that enables communication between a consumer and a provider service. It uses XML information set as a message format. SOAP can be used with any transport protocol but implementations commonly use HTTP.

2.6 Representational State Transfer

Representational state transfer (REST) is an architectural style [13, 57]. The term REST was introduced by Roy Fielding in his doctoral thesis [24], which demonstrates how REST can be used to guide the design and development of modern Web architecture.

Like World Wide Web (WWW) itself, REST promotes resource-orientedness and stateless communication. In REST each resource has their own universal resource identifier (URI). The identifier can be used to access the resource through a uniform interface such as HTTP. [57]

Web services implementing REST architecture can be called RESTful. RESTful web services typically use HTTP verbs such as GET, POST, DELETE and PUT to receive and send data to remote servers. For example, listing a collection of users could be implemented by calling:

GET http://api.example.com/users

Unlike for SOAP, there is no standard for building RESTful web services. This is because REST is an architectural style and not a protocol like SOAP.

2.7 Stateless and Stateful Services

Services can be stateless or stateful. A stateless service treats each request as an independent transaction. Requests are unrelated to each other and the service does not store information about sessions or other services for a duration of multiple requests [23]. For web services this means that each HTTP request happens in complete isolation [57]. A client has to include all information about the request for the server to process it and the information never relies on previous requests.

Stateless applications are easier to distribute across load-balanced servers [57]. Since requests do not depend on each other, they can be handled by different servers. As mentioned in section 2.4 application scaling can be achieved by adding more servers. In a stateful application adding or removing new servers requires sharing the state so that each server instance is able to access it.

Caching requests is also easier for stateless services. The application can decide if the request has to be cached by just looking at that particular request. Whereas a stateful application would have to worry about the cacheability of previous requests.

Stateless requests can be more easy to monitor because everything needed to understand a message is included in the request [31]. The monitoring software does not have to keep track of the current state.

Avoiding stateful services can be difficult. For example, a service might need authentication. Instead of sending a certificate with every request it is more efficient to store a shared authentication token between the consumer and provider [31]. For instance, Google Gmail has a RESTful API [26] that uses access tokens for authenticating the client application. In this case, the server cannot trust the client to report

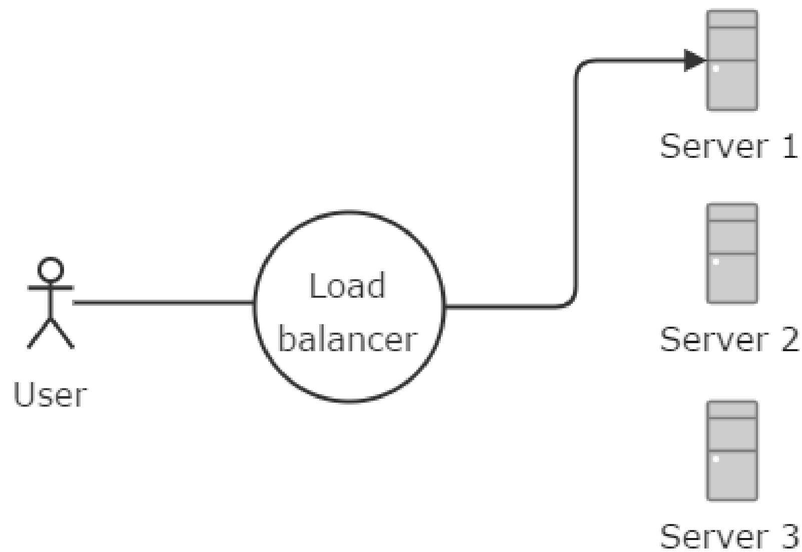

Figure 2.2 *A load balanced server cluster.*

its state. It has to validate the access token and check that it has not expired. This requires storing the application's state.

If the application is distributed across multiple machines, every machine in the cluster has to be aware of the authentication state. Otherwise the server application cannot verify the client. In Figure 2.2 the user connects to a server cluster. The load-balancer will route the user's request to one of the three servers. The server that the user connects to has to be able to access the current authentication state. The cluster is usually set up to exchange details between servers so that the state of one server is replicated at other servers.

An alternative solution is to add a routing rule to the load-balancer to make sure that every request made by a certain client has to go to the same server. This kind of method of controlling communication between a client and a server cluster is called session affinity [61].

Customized service for the consumer requires state, which means that the provider and consumer have to share the same specific context. Storing this context for multiple consumers can reduce scalability. This also makes the services more coupled together and makes switching providers more difficult.

2.8 Virtualization

The end result of building multiple small services is having many small software projects. Each project should only exist to provide a specific service which makes projects more simple and easier to understand. As developers switch projects and new developers are hired, installing projects and their dependencies becomes time consuming. Traditionally this has been solved by using unified systems and configurations for all development machines and servers, or by using virtual machines to replicate environments.

Modern computers for a while have been powerful enough to use virtualization. By using virtualization one computer can present an illusion of many smaller virtual machines (VMs) [10]. Each VM can run a separate operating system and work as if it was a real physical computer. This has enabled cloud computing providers such as Amazon, Google and Microsoft to create virtual machines per customer.

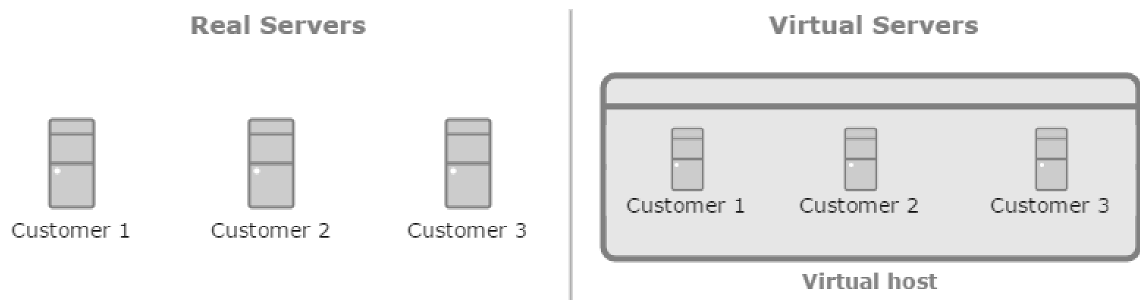

Figure 2.3 *One server can host multiple virtual machines. Each customer sees the virtual machine as a real physical server.*

The left side of Figure 2.3 illustrates three servers dedicated to three different customers. Each customer owns one physical server. On the right side of the Figure, there is only one physical shared server. The shared server is separated to three virtual machines that all share the host machine's resources. Although the three customers share the same physical server, each customer has their own virtual machine that acts like it was a real physical server.

Sharing computing resources between multiple customers has cost advantages because it allows customers to only pay for the used resources [38, 9]. Especially small companies can benefit from this if they are currently not able to make significant investments to server hardware but might have to scale up later.

Virtualization is cost efficient for running a service-oriented architecture. Instead of deploying each service to a physical server, services can be hosted in separate virtual machines. SOA services can all have different environment requirements, such as different operating systems, server configurations, programming languages or database engines. By using virtualization, it is possible to create multiple environments to a single physical server.

Service-oriented architecture can be difficult to maintain when multiple small services must be developed, tested and deployed as fast as possible. Software developers need to be able to setup their development environment for each service and be able to switch between projects. Manually installing each project and its environment can be slow and is not economically feasible [29]. By copying the same image for all development machines, each developer has a similar development environment. This is helpful for preventing bugs related to different run environments. The same environment or a slight modification of it, can also be deployed to testing, staging and production servers. This ensures that the system works almost identically in all parts of the deployment pipeline.

Virtualization can help avoiding vendor lock-in if the environment can be moved. Vendor lock-in means that the customer is dependent on a vendor for products and services and is unable to switch vendor without high costs [38]. Cloud based software is often dependent on underlying hosting cloud infrastructure [38]. Although virtualization can be used to avoid vendor lock-in, it can itself also cause vendor lock-in if the virtualization platform cannot be changed.

3. MANAGEMENT SOFTWARE REQUIREMENTS

This chapter describes the basic requirements for the service provider management system called SirWise. SirWise is used as a real example case of why to implement a service-oriented architecture, what problems SOA can solve and what SOA related challenges have to be addressed.

The first two sections of this chapter describe background information about the service provider industry and what problems SirWise is supposed to solve. Sections 3.3 and 3.4 specify the main functional and non-functional requirements of the software. The last section 3.5 represents how the system can be separated into independent services.

3.1 Apple Authorized Service Providers

An AASP (Apple authorized service provider) is a company that is authorized to provide repair services to all Apple customers [5]. An AASP is able to obtain parts directly from Apple and receive reimbursement for labor, parts and travel for fixing Apple devices. In exchange Apple has a list of requirements for AASP's. One of the requirements is to keep records of all service events for a period of 5 years and allow Apple to review these records [3]. Accurate reporting is therefore essential for AASP's. Apple also has different metrics for tracking performance and customer satisfaction [3]. The ratings given by Apple are important for AASPs, because retail companies use them to compare different service providers.

One AASP can have one or multiple service locations where customers can bring their devices for repair. Many AASPs also have partnerships with retail stores so that customers can return their devices to the same store where it was bought from.

3.2 An Example Use Scenario

An example of a common repair case for smart devices is changing a broken screen. A customer has a device with a cracked screen and she brings the device back to the retail store, where the device was bought. A retail store worker then checks if the device is under warranty and does a basic diagnosis of the problem. In case the store can not resolve the issue, the device has to be mailed to an authorized service provider or directly to the manufacturer.

Before mailing the device to an authorized service provider, information about the customer and the device has to be attached to the package. The service provider's technician, who repairs the device, needs to know the customer's contact details, required passwords, possible accessories and the problem description. After receiving the broken device, the technician makes a diagnosis. Depending on the results, the technician fixes the device, orders new parts if necessary or replaces the device with a new one.

The device manufacturer might require reporting of repairs. For example, Apple surveys customers for their repair experience and has four objective criteria [3] to measure the performance of service providers. This means that technicians have to report all repairs and end customer contact details to Apple's Global Service Exchange (GSX) [4].

Finally when the order is resolved, the customer and the retail store is notified about the completed order, an invoice is created and the repaired device is returned to the owner. If the customer had an insurance for the device, the insurance company is also informed about the repair.

Typical challenges related to the workflow are:

1. The retail store worker has to use multiple systems and type the same information many times.
2. All the systems have separate user credentials and varying user interfaces. All staff members have to be trained to use the systems.
3. Multiple people are working on the same order. They should be able to see the order statuses, customer messages and device locations.

4. Reporting to manufacturers should be mostly automated.

5. The end customer is not able to track the order.

The management system should improve the order process by removing duplicate work steps. Same information should only be needed once and the data should be sent automatically to other systems.

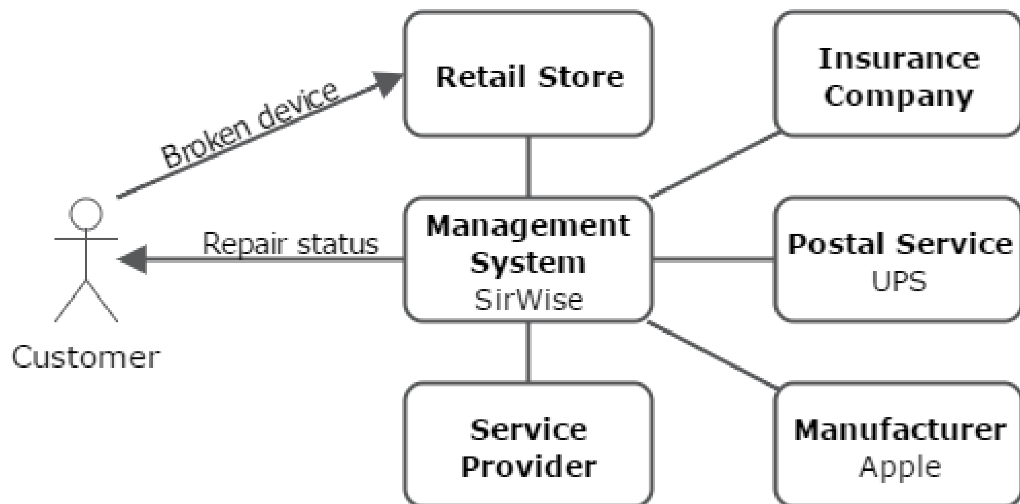

Figure 3.1 *Connections between different parties involved in a repair case.*

Figure 3.1 shows the different parties involved in a typical repair case. The management system integrates multiple systems from different companies into one. This enables sharing the repair status between retail store workers, service provider's technicians, insurance companies and manufacturers.

3.3 Functional Requirements

To present an example of a service-oriented architecture, we have to first list some functional requirements for the management system. The following list is not a complete list of requirements, but it includes most of the main features of the real system.

The system should be able to

- authenticate a user

- create an order
- manage customer information
- fetch device info from GSX using a device's serial number
- send reports to GSX
- send email and SMS notifications to users
- create an invoice.

The system should also have a separate user interface (UI) for customers, technicians and retail store workers. Customers should only be able to create new orders and view their order statuses. Technicians should be able to create new orders and be allowed to manage orders which are assigned to them. Retail store workers should only be able to see orders for their store location.

Each service provider should be able to create their own UI implementation if needed. For example, if a service provider uses a different workflow, they should be able to create a custom software implementation for technicians.

All data should be separated for each AASP. For example, one AASP should not be able to view other AASP's customers, users or orders. Data sharing between different AASPs should not be allowed by default.

3.4 Non-functional Requirements

Modern software development is often driven by agile processes [53]. When the software is constantly changing the whole development process has to be very nimble. The management system has many common non-functional software requirements such as fast development, continuous integration, application scalability and support for multiple operating platforms [53]. This section defines these requirements and explains why they are needed.

Fast development speed is required because of quickly changing business needs. A new developer should be able to set up the whole development environment as fast as possible and be able to get a working copy of the latest software version. The developer should be able to contribute to the project without learning all the details

of the entire application. An example of a changing business need for AASPs was in 2014 when Apple bought Beats headphones [6]. This meant that a new product category, Beats integration and repair codes had to be implemented. In this case a developer should be able to create a new software integration for Beats without breaking other product categories or repair codes.

A common problem in many software projects is that the latest version of the application is not always in a working state. This happens because nobody wants to install the whole application every time it has changed and then try to test it [33]. Continuous integration tries to solve this problem. Continuous integration means that after any code changes, the software is tested to be deployable. When a developer creates a new code commit the entire software is first build, then tested using automated tests and finally, if the tests pass, the software is deployed to a staging server. According to Humble and Farley (2010), "the goal of continuous integration is that the software is in a working state all the time" [33].

During business hours, the system must handle a large number of active users who need the system for their work. To prevent downgraded performance during peak hours, the system should be able to distribute user traffic to multiple servers. Balance loading between multiple servers is critical for scaling the application for a large user base because one server can only handle a limited amount of user traffic. Having multiple servers also enables the system to stay functional even if one of the servers must be updated, rebooted or if it for some other reason fails to respond.

The management system should support multiple software clients such as mobile browsers, desktop browsers and native mobile applications. Service providers should also be able to create their own custom client applications. Service providers need to have access to their own repair and customer data to be able to make integrations for existing systems.

3.5 Defining the Required Services

The management system requires multiple features which can be separated into services. Each service should be independent and have a clear business need. The service should also be small enough that it is easy to understand.

Defining how small a service should be, is a controversial subject. It is difficult to specify how many lines of code a service should have or how many hours it should

take to build. According to Newman (2015) a service should be "small enough and no smaller"[46]. Michael Feathers suggests that they should be "created by no more than a handful of people" [19] while Stefan Tilkov suggests that "it's not a goal to make your services as small as possible" [64]. Because of this services do not have an exact size limit, but for the management system we should try to keep each service small enough to be rewritten in a few weeks.

In the management system, we have at least five features that can be implemented as separate services:

1. Notification service that sends messages to customers and users using different integrations such as email, SMS and chat applications like Slack.
2. Order service that manages order information.
3. Customer service that manages customer details.
4. Device service that is connected to different manufacturer applications and fetches device data. In this example we have an integration for Apple's GSX.
5. Billing service for sending invoices and storing billing history.

Additionally, the system needs at least three separate user interfaces: one for service provider technicians, one for customers and one for mobile customers. The mobile application will be developed by a company specialized in mobile application development. The design of the architecture should support this.

4. ARCHITECTURE, TESTING AND DEPLOYMENT

This chapter describes the architecture for the management system specified in chapter 3. The architecture of the old version of SirWise is first described to highlight some of its main problems and to explain the reasons why a new version was needed. Then the old monolithic architecture is transformed into a service-oriented architecture that has a better support for multiple AASPs. This transformation will also require making changes to the development, testing and deployment process of the system. These changes are described at the end of this chapter.

4.1 Monolithic Architecture

The first version of SirWise consists of one core application that handles customer management, billing, device management, order management and sending notifications. This kind of monolithic architecture is extremely common [56]. Developers only have to open one project in their code editor and launch the application to see how the system works. Deployment is also simple because it can be done by copying just one project to a server. The application can also be scaled by creating multiple copies of the system and a load balancer can distribute traffic between these copies.

Figure 4.1 shows a simplified high level architecture diagram of the first SirWise implementation. The application has integrations to other systems such as Apple GSX, email servers and SMS services. The application has a web UI, that can be used by customers and technicians. It also exposes a simple REST API that can be accessed by other applications, such as mobile applications or other servers. Most parts of the application are logically separated into modules that can work independently, but the application is packaged, tested and deployed as a monolith.

This monolithic architecture works well in the early stages of the project. However, after multiple years of development the application will have millions of lines of code

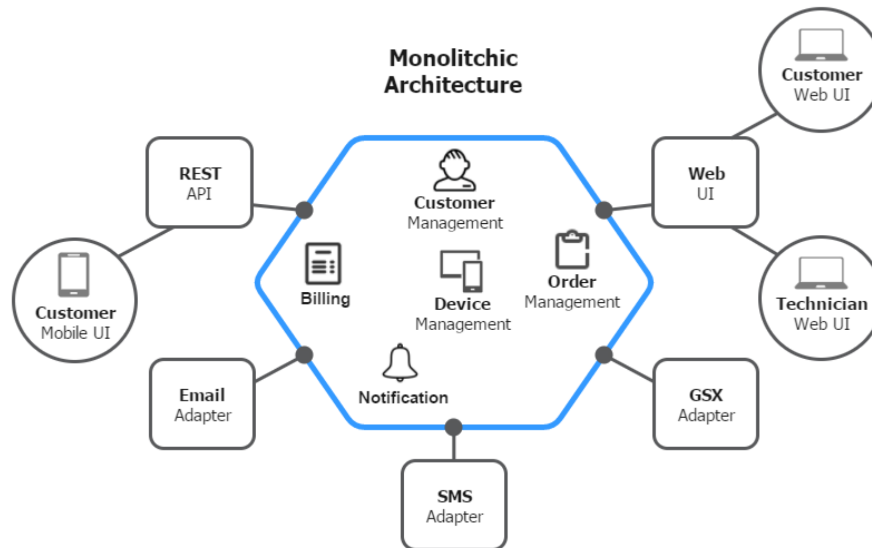

Figure 4.1 *Old monolithic architecture of SirWise.*

and becomes extremely complex. Deploying the application usually requires lots of manual testing because the impact of a change is not easy to predict.

Another problem in this kind of system is reliability. Because the whole application is running within the same process, a problem in one part of the system can potentially slow down or crash the entire process. As all application instances are identical, one bug can impact the whole monolithic system. Creating variants as described in section 2.3 is not possible.

A big problem in a monolithic architecture is that it makes adopting new software frameworks and languages very difficult. An application that has been developed for several years can have millions of lines of code. In that case rewriting the entire application to use a new framework or language is extremely expensive. This will limit how fast new technologies can be adopted and prevent from switching to a better framework. As a result, the whole development team will be stuck with the choices made at the start of the project.

4.2 Separating the Application to Services

The first version of SirWise has logically separate modules. These modules can be decomposed into independent services. In Figure 4.2 the original monolithic application has been transformed into services where each service works like a small

application.

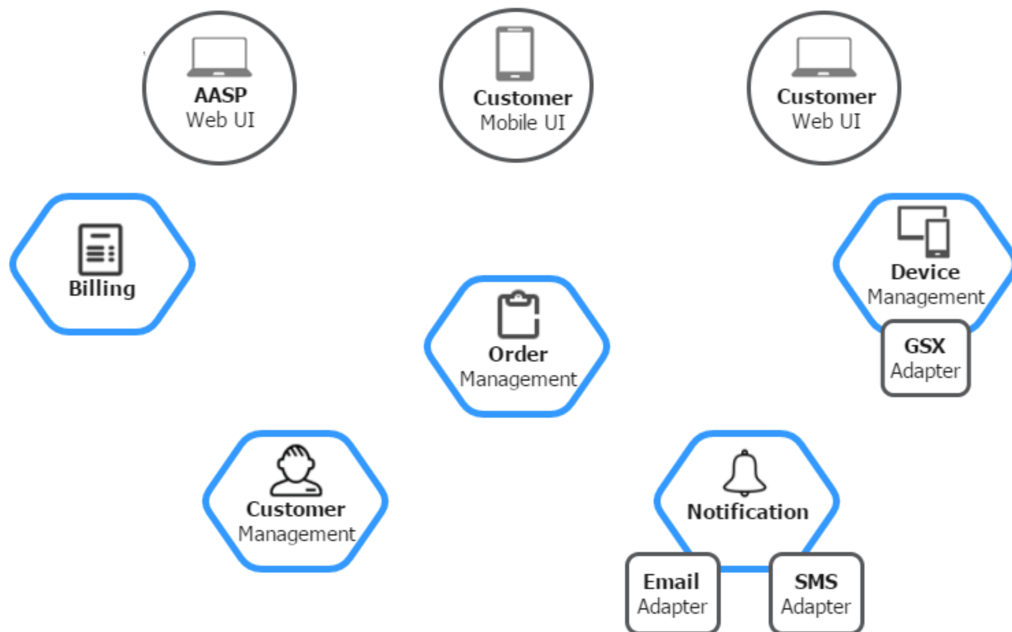

Figure 4.2 *The monolithic architecture broken down into independent applications.*

Services marked with blue hexagons, like the billing or customer management, are backend services. They generally have a database for storing persistent data and are responsible for implementing security related logic, such as input validation [49] and authentication. Backend services also expose an interface that can be consumed by other services.

Services marked with black circles, like the AASP and customer UIs, expose a user interface to the end users. These are called frontend services and their main responsibility is to present information to the user and send user input to the backend services. Frontend services can be installed to many different platforms such as web browsers, mobile devices or desktop computers.

Compared to the monolithic architecture, this separation of services also changes the relation between the application and the database. Instead of sharing one database between all services, each service has its own database. Having a separate database schema per service is needed to ensure loose coupling, otherwise the database would create an unnecessary dependency between all services. However, this allows choosing the best type of database for each service as database performance can vary depending on the use case [41].

4.3 Connecting Services

The services shown in Figure 4.2 are not yet connected together. To make this possible, each service has to expose an API that can be consumed by other services. As mentioned in section 2.5, both SOAP and REST are common techniques for implementing SOA. We chose to create a RESTful implementation using JSON (JavaScript Object Notation) because SOAP relies exclusively on XML [12]. SOAP requires that an XML structure has to be created every time a message is sent. When working with JavaScript clients, parsing and serializing the XML requires additional code and creates unnecessary overhead compared to JSON. Unlike SOAP, REST can be implemented using JSON, which does not require this kind of parsing.

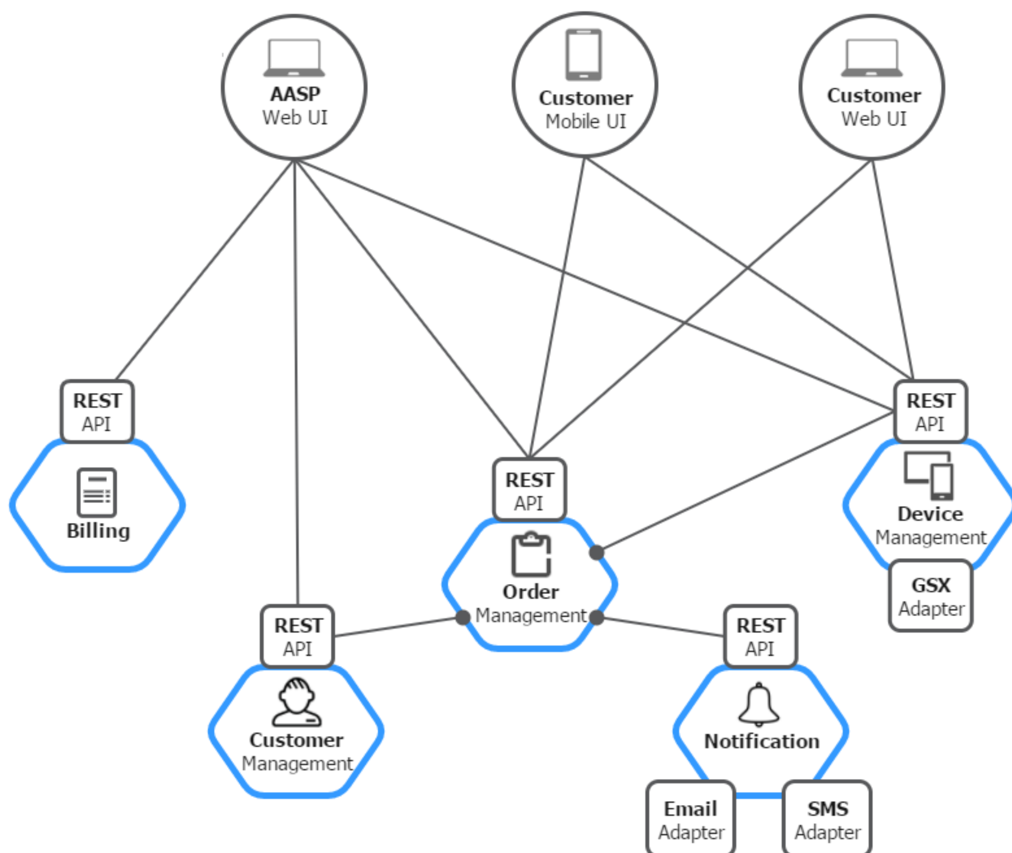

Figure 4.3 All services implement a REST API.

In Figure 4.3 the connections between services have been drawn. All backend services expose a REST API that can be consumed by other services. For example, the order service uses the notification service for notifying customers about repair

statuses. The order service also uses the device service for searching device information from manufacturers using serial numbers. Although this thesis only has Apple GSX as a device information source, the device search can support multiple brands. The main motivation for using a separate device service is that it can combine all sources into one stable API. This search API does not have to change when new sources are added or removed. It can also use alternative information sources or display cached results if the main information source is not available.

4.4 API Gateway

The Figure 4.3 already has a working architecture that could be implemented. However, the architecture can become problematic when developing frontend applications. For example, the mobile application for customers will require miscellaneous data from the backend services. The mobile application could display lots of repair information, such as the customer's order number, repair diagnosis, estimated cost of repair and order status. Additionally, the application could display the customer's name and show if her device is under warranty. In a monolithic architecture, the mobile client could make a single request to the backend, such as:

GET https://api.example.com/repairs/repairId

The response to this request would then contain everything needed to display the repair status. A monolithic application can fetch all the needed data from its database and return it to the mobile client.

In comparison, when using a service-oriented architecture the repair status data is owned by multiple services. The order service has the order number, repair diagnosis, cost estimate and the repair status. The customer service has the customer's name and the device service knows if the device's warranty has expired. In this example the data needed by the client is owned by three different services, but a more complex client could require even more services.

One option is that clients keep a list of services and make requests directly to each service. Sadly, this approach has many drawbacks. When fetching the repair status, the mobile client would have to make three different HTTP requests over the public Internet. Compared to requests made in a local network, requests over the public Internet are very inefficient. Especially on a slow mobile network extra requests have a significant performance impact.

Making multiple requests directly to each service also makes the client code more complex. Clients have to combine multiple responses and check that each response was successful. Another problem with the approach is that it makes refactoring difficult. Changing a backend service's address requires that all client applications using it are updated. Splitting one service to two services or merging multiple services to one is difficult if clients are directly connected to them.

A second option for connecting client applications to backend services is implementing a pattern called API gateway. The pattern was made popular by Netflix Zuul [45, 54]. Instead of connecting directly to each service, clients send requests to an API gateway. The API gateway then routes these requests to the appropriate services. The API gateway can also invoke multiple services and combine the results into one response [55]. This way a client application only has to make one request that can return data from multiple different services.

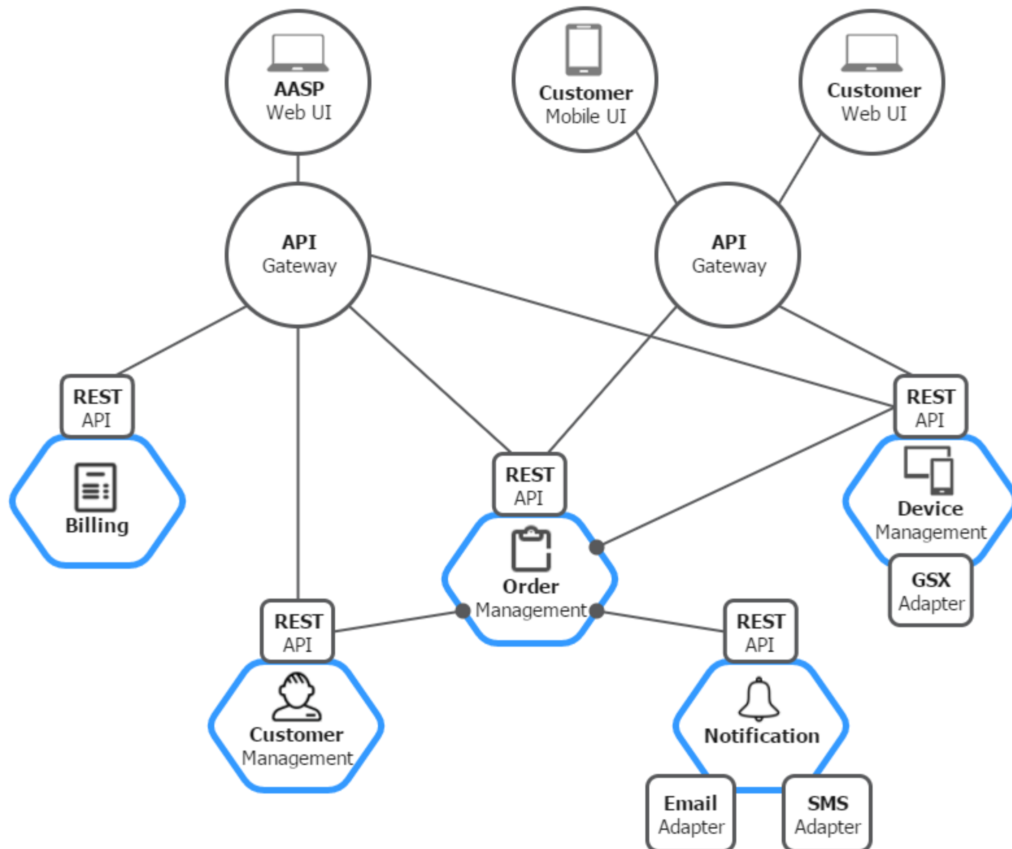

Figure 4.4 *Overview of the new SirWise architecture.*

In Figure 4.3 the API gateway is used to create a simple unified interface for

the client side applications. In most cases, frontend services need to fetch data from multiple backend services. For example, the customer UI needs information about the current order and has to know if the customer's device is under warranty. Without the API gateway, the customer UI would have to know addresses of all needed backend services and make requests to multiple servers. By using a gateway, client side applications only have to connect to one end-point.

The API gateway can also be used to simplify the exposed API. In Figure 4.4 there are two API gateways: the API for service providers and the customer API that exposes less features and has stricter permissions. The service provider API is used by trusted clients who are allowed to view and modify sensitive data. They are able to access customer management and billing services. The customer API is more simplified and can be used by third party developers who do not need access to all services.

The API Gateway provides many benefits but it has some drawbacks. It is a central part of the architecture and must always be updated when service end-points change. It is important that the update process is fast. Otherwise, developers need to wait and the API gateway becomes a development bottleneck. A second drawback is an increase in response times because the API has to make an extra network round-trip. However, for our use case the time is insignificant because the round-trip is made inside a fast local network.

4.5 Multitenancy

Multitenancy is a software architecture where one software instance serves multiple tenants [67]. The term "tenant" refers to a group of users who share access to the same software instance. The software appears to each tenant as if they were the only tenant in the system. But in reality the software serves multiple tenants. This allows software vendors to provide service to many customers on the same infrastructure.

Multitenancy is a common architecture for software as a service (SaaS) applications [14]. SaaS is a delivery model where customers buy a subscription to use a centrally hosted application [18]. SaaS applications typically store data from multiple organizations. To take advantage of the SaaS model, organizations have to trust the SaaS vendor to keep their data safe. Because of this, creating a secure architecture should be a high priority for all SaaS vendors.

The SirWise ERP also uses the SaaS model. In SirWise each AASP is a separate tenant and has its own data and configurations. Each tenant can edit customers, create orders and update device data without effecting other tenants. Data is only shared between tenants if explicitly requested.

Figure 4.5 illustrates how multiple client applications and different service providers (tenants) use the same interface. All three service providers have implemented two custom client applications: one client for customers and one for technicians. Users can not access data stored to another tenant. For example, a user of company A is not able to use company B's application, unless the user has accounts in both companies.

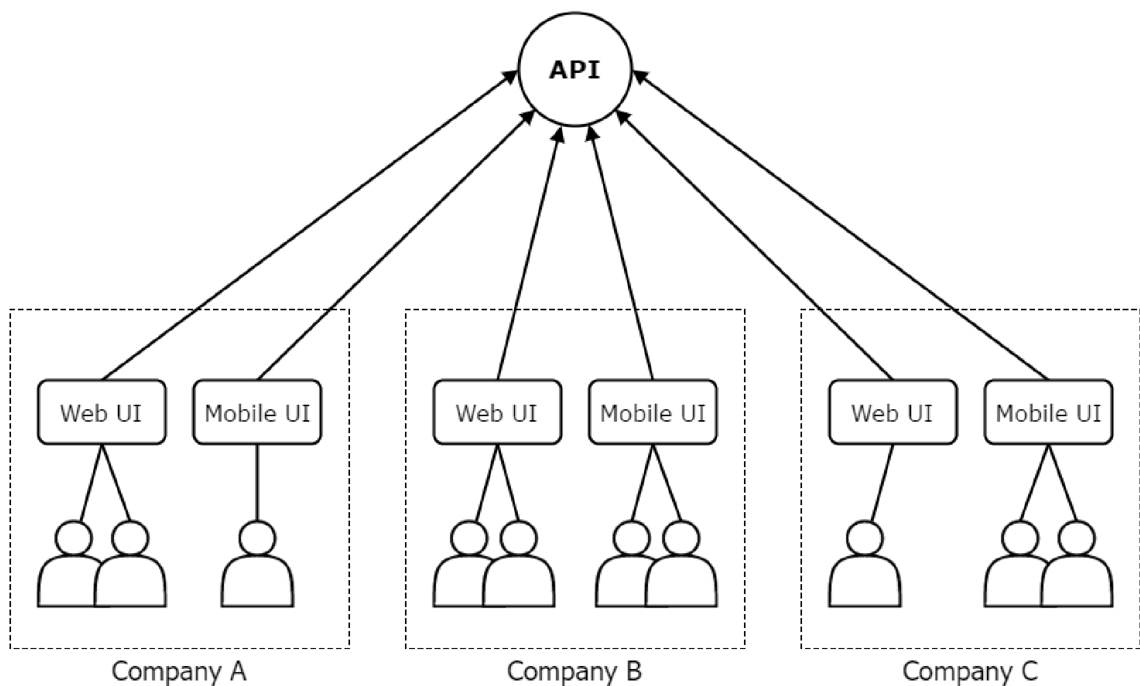

Figure 4.5 *Example of multiple client applications and tenants using the same application interface.*

There are many ways how the architecture in Figure 4.5 can be achieved. However, RESTful services are usually stateless and store the application state to the database. Because of this, the architecture design should define how the data is stored.

There are three different approaches how multitenant data can be stored to the database:

1. Each record in a database includes a field that has a tenant ID.
2. Database has a separate schema for each tenant.
3. Different database for each tenant. [14]

The first approach of having a shared schema has the lowest cost. One database server can serve a large number of tenants and only one database has to be backed up. This approach does not offer good isolation and requires more effort to prevent tenants from accessing each other's data. The software must be designed carefully to only load tenant specific data from the database so that each database query has the right tenant identifier.

In the second approach, each tenant has its own separate set of tables in a shared database. In this approach data is moderately isolated. The software cannot load data from a wrong tenant, unless it queries a wrong database table. This approach might not be suitable if each tenant has a high number of tables or the number of tenants is high.

The last approach of having a different database for each tenant offers the best isolation between tenants. In this case the database security prevents any tenant reading data from another tenant's database. Each database can be individually configured to meet the tenant's needs and restoring the database from backups is relatively simple. This is usually the most expensive approach as each database has to be maintained separately and it requires more database servers.

For SirWise, the first approach is not possible because many customers require that sensitive data is always stored separately. There is a high risk that a programming error could leak sensitive information to other tenants. The second approach where each tenant has its own tables, is possible to implement, but the high number of tables can make it impractical to manage.

In SirWise all tenants have their own database. All databases do not have a separate physical server but they require the software to create a new connection when accessed. This ensures that the software cannot accidentally query data from a wrong tenant. Services that do not store sensitive data can still take advantage of the other approaches if needed.

4.6 Authentication

When frontend applications make requests to the SirWise API, they have to be authenticated. As mentioned in section 2.7, frontend applications cannot be trusted to report their state and because of this, every request must be authenticated. The API has to keep track of the current state. For example, it should know if the user has logged in to the application or not.

A SOA that uses HTTP can apply many different authentication techniques. Common options are basic authentication, digest authentication and bearer tokens [50].

Basic authentication is a simple user authentication method. However, it has some potential risks because it requires sending a username and password on every request as a base64 encoded string. An attacker will be able to read the credentials, if requests are not encrypted. Even when using HTTPS, basic authentication is vulnerable to replay attacks [50]. Another problem in using basic authentication is that it has no token management. It is almost impossible to limit access to secured resources without disabling the user's account.

Digest authentication is essentially a more complex version of basic authentication. Instead of sending a readable base64 encoded string, it uses a checksum. The checksum is created from the username, the password, a given nonce value, the used HTTP method, and the requested URI using an algorithm like MD5. This makes digest authentication more secure compared to basic authentication. However, it has some limitations that prevent us from using it. [25]

Besides authenticating users, the SirWise API has to support authenticating other applications. In some cases, it is useful to make automated requests that do not involve the user. For example, we send repair status notifications to customers. In this case the request to send a notification is made by another application and there is no user to authenticate. Simple authentication mechanisms like basic authentication and digest authentication are mainly designed for user authentication and cannot identify the client application that made the request.

4.6.1 OAuth 2.0

OAuth 2.0 is an authorization framework that allows third-party applications to request access to HTTP services. By using OAuth[1], it is possible to set separate access limits for each client application [30]. For example, a trusted server side application could have access to all customer data, whereas a non-trusted mobile application might only see data of a specific customer.

In the OAuth standard, a client application starts the authorization process by requesting an access token. The token is obtained using one of the OAuth grant types: authorization code, implicit, resource owner password credentials or client credentials [30]. The different grant types and their usage is explained in the official standard[2].

The following examples only use the resource owner password credentials grant, but actual implementations should choose the grant type depending on the client application and the use case. Using user credentials is only suitable if the client application and resource owner, in this case the user, have a high degree of trust between them [30].

Listing 4.1 *Example OAuth token request using resource owner password credentials.*

```
POST /token HTTP/1.1
Accept: application/json
{
  "username": "john",
  "password": "johns_password",
  "client_id": "client123",
  "client_secret": "client123secret",
  "grant_type": "password"
}
```

Listing 4.1 shows an example OAuth token request. The HTTP request contains user credentials and client credentials in JSON format. The request also contains the grant type "password" which equals to the "resource owner password credentials" grant. The client identifier and secret are used for authenticating the client

[1] Currently OAuth standard has three different versions: 1.0, 1.0a and 2.0. In this thesis "OAuth" refers to version 2.0.

[2] https://tools.ietf.org/html/rfc6749

application and allows the SirWise API to implement application specific access control.

All fields in the request body are specified in the OAuth standard [30]. However, based on this information, the SirWise API would not be able to know the right tenant and would have to search the user "john" from multiple tenant databases. To fix this problem, we can add a tenant identifier to the request.

Listing 4.2 *Example OAuth token request with an added tenant identifier.*

```
POST /token HTTP/1.1
Accept: application/json
{
  "username": "john",
  "password": "johns_password",
  "client_id": "client123",
  "client_secret": "client123secret",
  "grant_type": "password",
  "tenant": "tenant123"
}
```

Listing 4.2 extends the earlier request with a "tenant" field that allows the SirWise API to quickly identify the intended tenant. Based on this field, the API can select the correct database and query it for a given username. Another popular method for identifying the tenant is including it in the request URL[3]. The downside of adding the tenant in the URL is that implementing authorization across multiple tenants is more difficult. For example, a "super user" could have access to multiple tenants.

After the API has successfully authenticated the client and user, it creates an access token response. Besides the access token the response contains information about how the token should be utilized (token_type), expiration time of the token in seconds (expires_in) and a refresh token (refresh_token) that can be used to request a new access token after expiration.

Listing 4.3 *Example OAuth token response.*

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
```

[3] For example, https://tenant123.example.com/token or https://api.example.com/tenant123/token

```
Pragma: no-cache
{
  "access_token": "3ZomlGZAEdr3sCfecEApFA",
  "token_type": "bearer",
  "expires_in": 3600,
  "refresh_token": "sAGH1KIkA2XV2Ns6LgCGAF"
}
```

In Listing 4.3 the API has given an OAuth token response. The HTTP headers specify that the request was successful (200 OK), the content is JSON, as requested by the client and the two cache headers tell clients to not cache the response. The token type "bearer" tells the client to add the access token to each protected resource request. For example, the client could send the following request to access customer data:

Listing 4.4 *Example of an authenticated request using the received access token.*

```
GET /customers/123 HTTP/1.1
Authorization: Bearer 3ZomlGZAEdr3sCfecEApFA
```

By reading the "Authorization" header, the API is able to identify the user and client who made the request. In case an attacker is able to read the access token, the API can simply revoke the token instead of disabling a user's account. This can also be implemented so that users are able to revoke their own token if they suspect that their account is compromised.

It is important to note that the OAuth standard also specifies many other authentication methods, such as refreshing the access token, and it documents common attack vectors. Most of these are not specific to a service-oriented architecture or were not required for the implementation of SirWise. Therefore, they are left outside the scope of this thesis.

4.6.2 Authentication as a Service

The previous section described how the SirWise API can authenticate requests using the resource owner password credentials grant. However, authentication is not different from other features. It is a clear business functionality that is likely to be

reused in many parts of the system. This means that it should be implemented as a service like other features.

In Figure 4.6 authentication is shown as a separate service that has connections to multiple databases. One database named as "master" contains shared data that does not directly belong to any specific tenant. SirWise uses it for storing all client applications and tenants. When an application makes an access token request, this database is queried to validate the tenant and client credentials. If the request is valid and contains user credentials, the authentication service selects the requested tenant database and tries to authenticate the user. Finally, if the user is authenticated successfully, the authentication service returns an OAuth access token response as described in section 4.6.1.

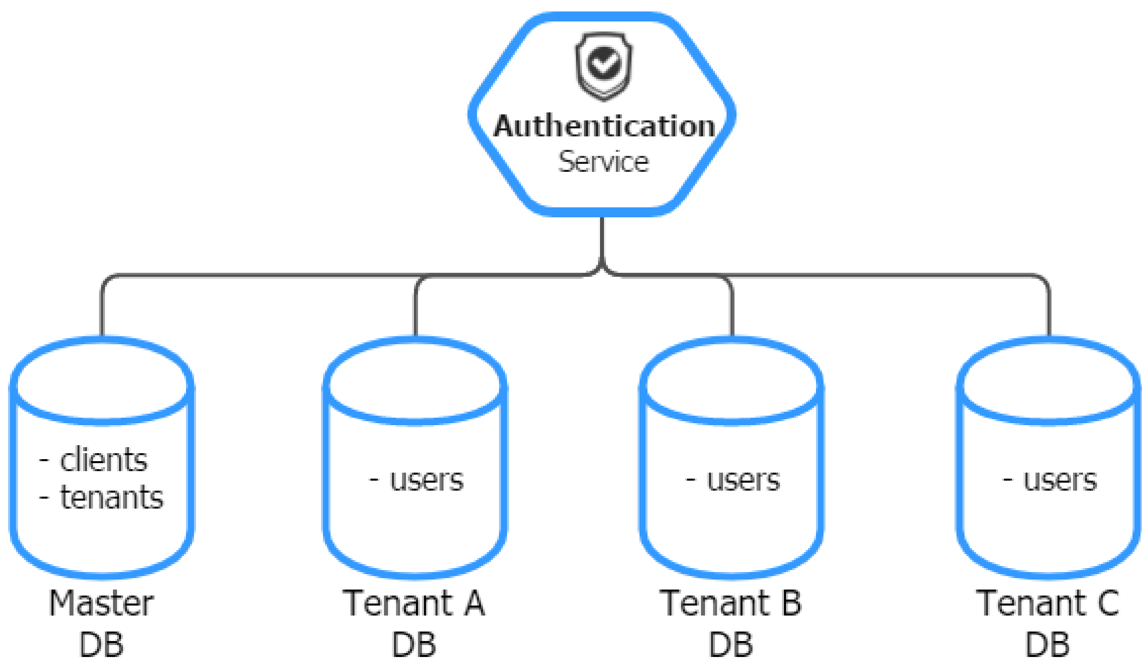

Figure 4.6 *SirWise authentication service and database architecture.*

The client application can then use the received access token to make requests to protected resources. Figure 4.7 illustrates how protected SirWise services are related to the authentication service and how the client has to first obtain an access token before it can access any protected services. The figure also demonstrates how the authentication service is separated from the API gateway and other services.

When implementing the architecture in Figure 4.7, one of the first problems will be, how does the API gateway validate the access token given by the client? Unless

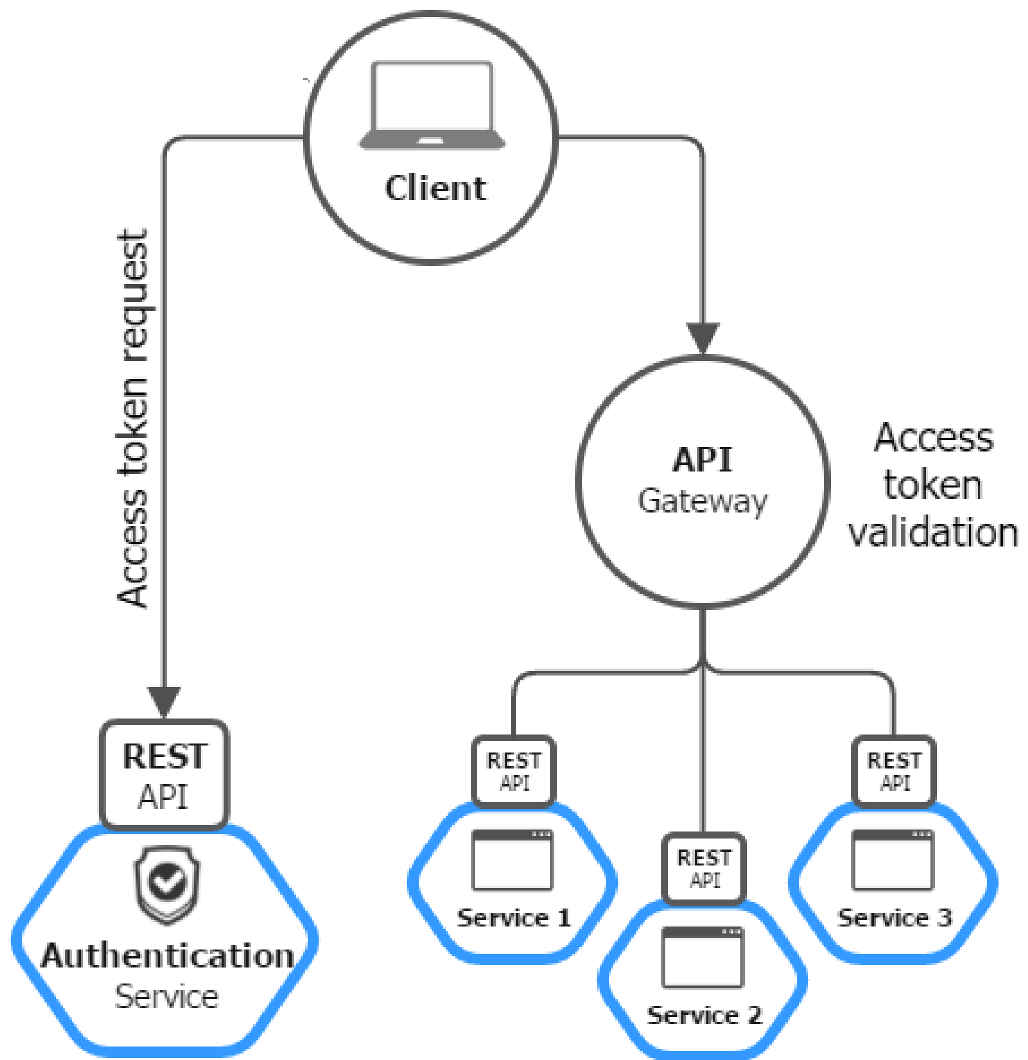

Figure 4.7 *The client application sends an access token request to the authentication service. After obtaining a token, the client uses it to send an authenticated request to the API. The API gateway validates the request before allowing access to any protected services.*

the authentication service and the API gateway share a database, the gateway will not be able to validate tokens.

A simple solution to this problem is to make the API gateway send a HTTP request to the authentication service every time a client tries to access a protected service. The authentication service would then tell the API gateway if the access token is valid. The downside of this approach is the additional overhead of each request.

In many cases validating the access token is not enough because many services

also need user data to function properly. For example, a service might require user roles, permissions or an identifier to decide if the user is allowed to access a protected resource. One option is that each service fetches the user data from the authentication service separately. This, however, has a few disadvantages:

- The authentication service becomes a single point of failure. Few seconds of downtime makes the whole system unusable because requests cannot be authenticated.
- It is inefficient if many services have to make the same request.
- Each service has to be able to find the authentication service.
- Services need to implement user data fetching, which adds some complexity and extra code.
- The authentication service has to be mocked for development and testing.

The cause of these issues is that the access token is a reference and is only useful for locating the data in the authentication service. This can be solved by replacing the so-called "token by reference" with "token by value" [32, 63]. This means that the authentication and authorization data is included with the token.

The reason why the authentication and authorization data should not be included in plain text format is that an attacker could easily read or modify the token. To make transferring the token secure in untrusted environments, the data should be encrypted and signed by the authentication service. By using symmetric or public key cryptography, the authentication service can be used for creating encrypted and signed tokens. The services can then independently validate these tokens without communicating with the authentication service.

4.6.3 JSON Web Tokens

A popular solution for transferring signatures and encrypted data is a JSON Web Token (JWT). JWT is a standard for representing claims to be transferred between two parties [35]. Claims are statements about an entity like a user's name or permissions. Claims can also contain metadata such as the expiration time, the subject of the claims or the name of the issuer. A JWT token consists of three parts:

1. Header that defines the used algorithm and token type.
2. Payload, which contains the claims.
3. Signature that is used to verify the sender and that the message was not changed.

The three encoded parts are separated by dots. An example of a JSON Web Token looks like this:

Listing 4.5 *Example of a JWT. Line spaces are added for better readability.*


```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IlNPQSJ9.
EvghfGTVIxt7TPrdK_SiIaQ4h1YwZlUup6SN4oTK9A8
```


The JWT in listing 4.5 is not readable without a valid key. The authentication service can create a similar token that contains the needed user data and sign it using a private key. All other services in the system can have a corresponding public key that can verify and decrypt the token. As long as the authentication service is kept secure and the used cryptographic algorithm is not broken, services can trust all messages signed by the private key. If one service is compromised, it will not compromise the whole system because each service validates the token independently.

4.7 Deployment

In many organizations application deployment is associated with a high degree of risk [33]. The deployment process usually involves many steps which have to be executed in a specific order. For example, upgrading the operating system, installing new libraries, updating the database schema, changing configuration and copying the latest stable version to the server. After completing all the steps, the deployment team might have to make some manual testing to verify that the application still runs properly.

If anything goes wrong, the users will have a broken system and the development team receives bug reports stating that the application is not working. The developers then have hurry for a quick fix, which is deployed to production. In worst case, there is no time for testing and the fix itself will cause new problems.

In this kind of projects, deployment requires lots of manual work and there are many things that can go wrong. This often leads to infrequent deployments, because the process is risky and time consuming [33]. The deployment process of a service-oriented architecture can be even more time consuming and error prone. Instead of deploying one large application, many small applications have to be deployed. Because of this reason, automating the deployment process is highly recommended [33].

4.7.1 Docker

Nane Kratzke (2014) describes Docker as "a lightweight virtualization tool that can package an application and its dependencies in a virtual container that can run on any Linux server" [38]. Docker can also be run on Windows 10 and OS X Yosemite operating systems by using a native Docker application. Despite being a fairly new project, Docker has gained lots of interest[4] and large technology companies, such as Ebay, Spotify and Uber, have started using it[5].

Docker containers can be deployed on a VM (section 2.8) or a "bare metal" non-virtualized server [20, 52]. Deploying Docker containers directly on a non-virtualized Linux server removes the overhead of a VM and offers the same isolation [20]. In general, performance of Docker is equal or better compared to a Linux virtual machine [20].

Figure 4.8 compares a virtual machine and a Docker container. The virtual machine requires a hypervisor, or virtual machine monitor, that creates and runs virtual machines. Each virtual machine has a unique operating system and contains its own binaries, libraries and applications. Docker containers do not need a separate operating system. Only binaries, libraries and application are added to each container, which leads to a smaller overhead.

Virtual machines take up a lot of system resources. Each VM runs a copy of the operating system and all the hardware that the operating system needs has to be virtualized. Compared to Docker, VMs require more RAM and have higher CPU usage. This limits the number of VMs developer machines and servers can run simultaneously.

[4] https://blog.docker.com/2016/02/docker-hub-two-billion-pulls/
[5] https://www.docker.com/customers

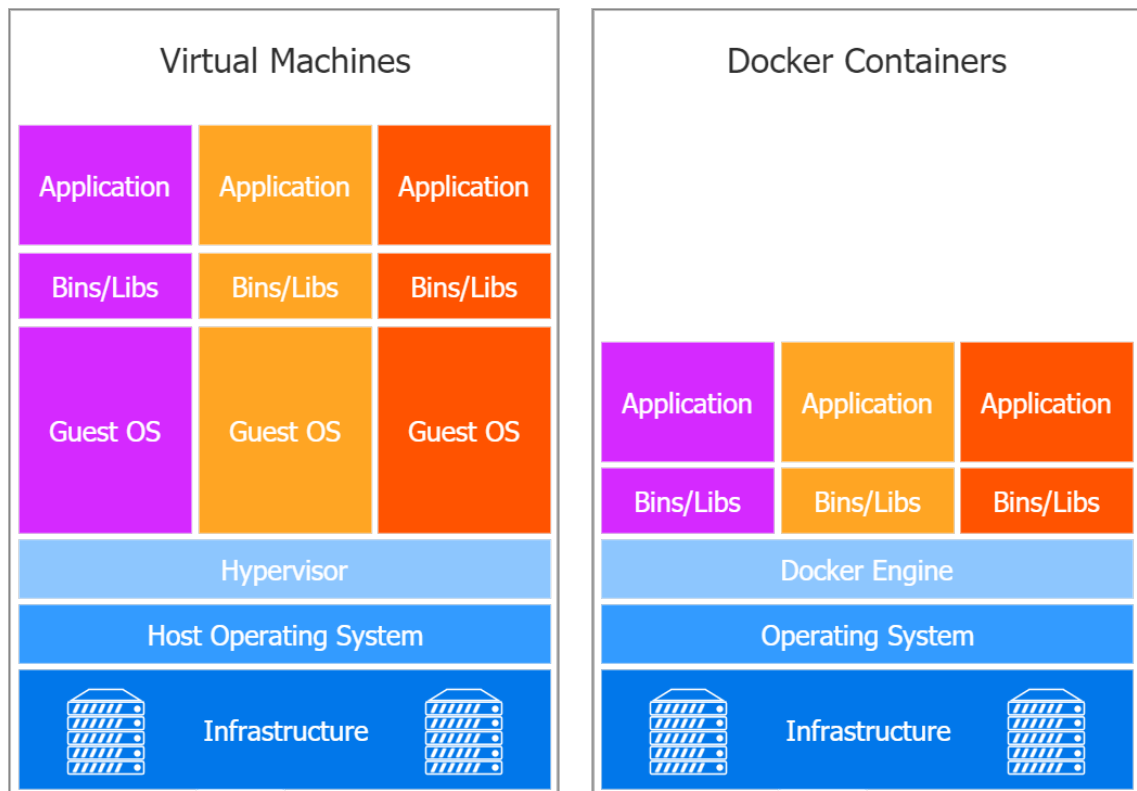

Figure 4.8 *Comparison of virtual machines and Docker containers [62].*

A big difference in Docker, compared to other container tools, is layered filesystem images. One operating system image can be used as a base for multiple containers. All extended images can have their own files, configuration and libraries. When images are shared over a network, this usually saves time and disk storage because only the changed image layers have to be copied. Another big benefit of a layered filesystem is caching. If we have a operating system image like Linux Ubuntu and want to install a PHP interpreter to it. We only have to build the layer that contains PHP and everything else can be read from cache [2].

4.7.2 Deployment Pipeline

Figure 4.9 shows an overview of the application's deployment process. The process starts when a developer commits new changes to the version control system. At this point, the continuous integration system (CI) is prompted to create a new pipeline. The pipeline will start by loading the changes from version control and creating a new Docker container. All code and project dependencies will be then added to

the container and the code is compiled, analyzed and unit tested. If the build is successful and all unit and code quality tests pass, the functional testing stage is started.

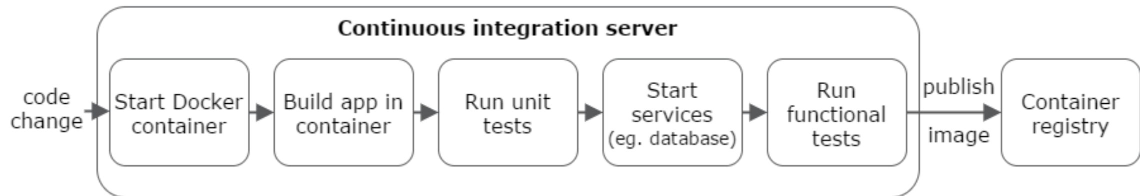

Figure 4.9 *Overview of the deployment pipeline.*

Functional tests are run separately from unit tests because they typically take longer to execute. Before starting the functional tests, the CI will launch additional Docker containers. All services such as databases, caches and worker queues are run in separate containers. Separating the services to different containers minimizes the need to build the whole environment when one of the services is updated or changed. When all needed services are running the CI prepares test databases, warms up application caches and finally starts the functional test suite.

While running the functional test suite the CI stores the test results in standard XUnit XML format[6]. This allows viewing statistics about passed, failed and skipped tests. It helps spotting out slow running tests and saving test history data.

If all tests have passed, the CI stores the executable application in an artifact repository. The artifact repository makes it easy for developers and testers to access the release candidate and allows other servers to load the final build. All software versions can be downloaded from the artifact repository. If a bug is found, the artifact repository can be used to find the first occurrence. The artifact repository is also used for storing test results and code statistics that can be used to track code quality.

The passing build is deployed to staging servers. The staging environment is ideally a copy of the production environment [33]. It allows testing that all external services, databases and configuration settings work as a whole before moving to production. The staging environment is also used for end-user acceptance testing. Acceptance testing allows the end-users to determine whether or not to accept the software [17].

[6]https://xunit.github.io/docs/format-xml-v2.html

It is important that the testing environment matches the actual production environment as closely as possible [44]. Otherwise there is a risk that the production version works differently from the tested version. By using Docker containers it is possible to deploy the exact same environment to production that was used in testing.

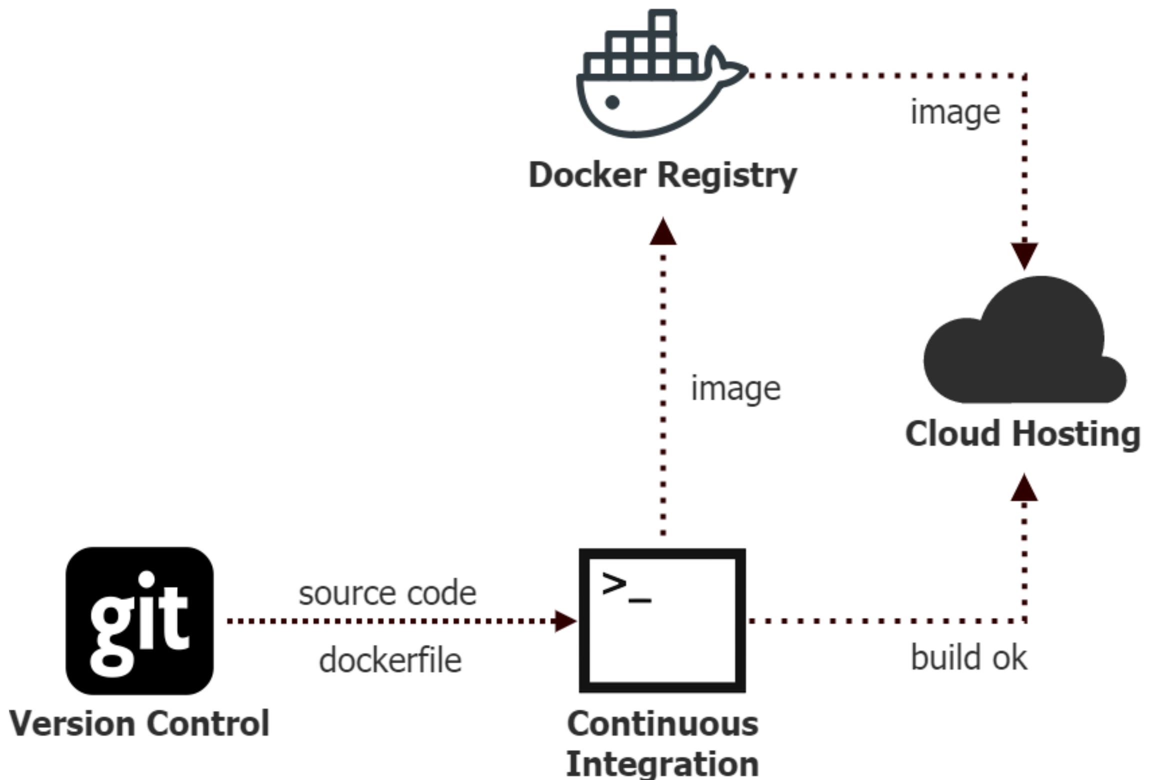

Figure 4.10 *Building, testing and deploying a Docker container.*

In Figure 4.10 the CI server fetches new code changes from version control. The source code also contains a Dockerfile[7] that is used to build a Docker image. The CI builds the image and adds the source code into the image. The image is then used to start a Docker container that is used to run code analytics and test suites. If all tests pass successfully the CI uploads the image to a Docker registry. Docker registry is an application that can store and distribute Docker images. Prebuilt Finally the CI will notify the hosting server to start a deployment.

After deploying and testing the software in the staging environment, a production release can be made. Even after testing, the release might have defects. Therefore it

[7]Dockerfile is a text document that contains command line instructions to assemble an image https://docs.docker.com/engine/reference/builder/

is important to be able to quickly roll back any changes [15]. A deployment method called blue-green deployment[8] is designed to reduce downtime by making it fast to roll back an update. In blue-green deployment two production environments, blue and green, are set up. Only one of them is live serving all user traffic.

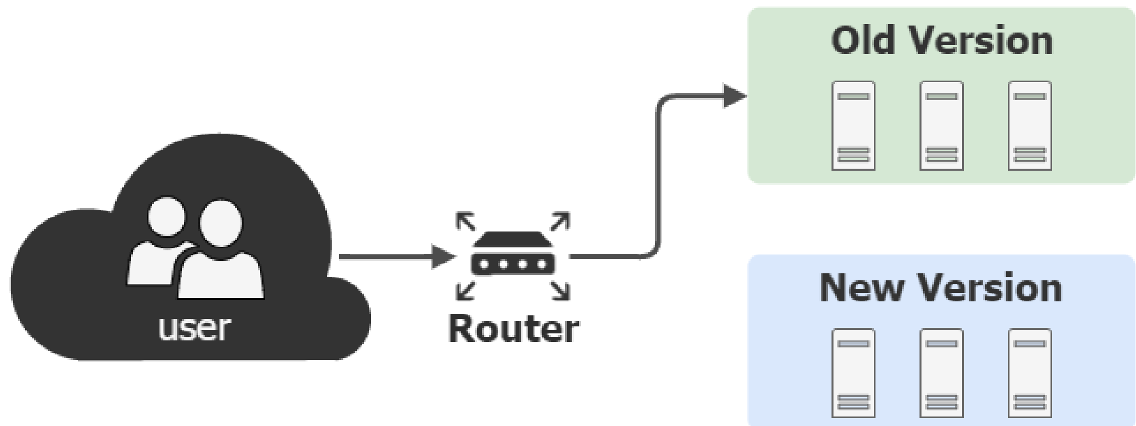

Figure 4.11 *Blue-green deployment.*

In Figure 4.11 the old version (green) is live and old version (blue) is idle. Once the new version (blue) is ready and tested, the router is switched so that all incoming traffic goes to the new version. If something unexpected happens with the new version, the old version can immediately be switched back.

[8]https://martinfowler.com/bliki/BlueGreenDeployment.html

5. EVALUATION

This chapter describes the differences between the old and new management system's architecture. The old version of the application has a monolithic architecture and was originally designed to support only one company. The new version has multiple independent services and it follows a service-oriented architecture. The new version is designed to support AASPs and it has an open interface for third-party applications.

The differences of the old and new versions are presented in Table 5.1. The main requirement was to support reusing functionality. The system is used on a variety of different operating platforms. Reusability is important to prevent implementing the same features multiple times. Although the old version had support for desktop and mobile browsers, support for third-party applications was very limited. It only had a simple API for fetching customer and order data. In the new version a separate REST API for customers and AASPs was implemented. Third-party applications can have different access scopes by using OAuth 2.0 authentication. For example, a retailer's system can place new orders for a certain AASP if they have allowed it.

The old version lacked proper testing and it was done mostly manually. The new version has a deployment pipeline that builds a new version of the software after every code change. It also runs automated tests and deploys the software to a staging server. Using continuous integration ensures that the software is always tested to be deployable.

The new version consists of multiple independent services. The system is able to work in a degraded mode if some service stops functioning. Business critical services can also be duplicated to multiple servers. If one of the servers does not respond traffic can be redirected to other servers.

The same method can also be used for distributing traffic between multiple servers. In the old version the server had to be upgraded manually if it was performing

Table 5.1 *Comparison of the old and new architecture.*

	Old version	**New version**
Reusability	Reusing functionality requires changes to the current system.	Data is accessible to any system capable of consuming REST APIs. New features are possible to implement without modifying current code.
Testing	Manual testing.	Automated tests are automatically run after each code change.
Fault tolerance	One server. Hardware errors can cause downtime.	Multiple servers. If one server fails traffic is redirected to other servers. Application is able to function even if some components stop working.
Scalability	Server resources (RAM, disk memory, CPU) are added manually as needed.	Load balanced servers. More servers are added and removed automatically when resource usage changes.
Deployment	Manual updates using version control or FTP. Restarting the server or a failing deployment causes downtime.	Automatic rolling updates. A high error count causes an automatic revert.
Development environment	Requires installing a web server, compiler, dependency management system and database. Manual server, database and application configuration. Software versions have to be checked manually.	Requires installing Docker and running the wanted services. Software versions are managed automatically.

poorly. In the new version more service instances are automatically launched if resource usage is high. During low usage periods the number of instances is scaled down.

The old version was updated by pulling code from version control to the production server. The web server would then be configured to use the new code. Any problems in deployment could cause downtime or make the system unusable. In the new version deployment is done by running two Docker containers. After a developer marks a new deployment the process is fully automatic. A new Docker container

that has the updated code is run simultaneously with the old version. While both containers are running, web traffic is slowly redirected to the updated version. The new container is monitored and if its error rate goes over a threshold, web traffic is redirected back to the old version. The automated deployment process also mitigates human errors.

Creating an identical development environment for all developers in the old system was challenging. The problem was first solved by using virtual machines. Developers had a copy of a virtual machine image that was very similar to the production environment. This eliminated most errors like having a wrong compiler version, different configuration or a file system that worked differently. However, virtual machines were too resource intensive for running multiple services. Each virtual machine runs its own operating system which requires lots of system resources, especially RAM. Using Docker instead of a VM, allows running multiple containers without having the overhead of many operating systems. Each container can run an independent service that has custom libraries and dependencies.

6. CONCLUSIONS

The first version of SirWise management system was designed for a single company. After new customers got interested in the software, Voltio started to look for options to scale the system. In the beginning of 2016 the development of a new version was started. The new version would use the SaaS model which meant that the same software should be able to fill the needs of multiple customers. We decided to create an open REST API to allow customers to integrate existing systems and implement custom work-flows.

The new version has slowly started to move to a more service-oriented architecture. New integrations can be easily created as separate applications that connect to the REST API. These can be either trusted applications that work inside the private network or third-party applications that connect from the public Internet. As the development team has grown, more members have specialized to certain parts of the system. So far this has happened very naturally without too much designing.

All SaaS software designs have to consider, how the customer data is stored. Especially the level of isolation between different customers has to be decided. Our decision was to have a database per customer, which gives a high level of isolation but is more expensive than having shared databases. In a SaaS application that has lots of customers, this might be too expensive. However, the customers of Voltio are mainly companies that have many employees. This keeps the number of databases per customer low.

Design of the API authentication flow was challenging. Although we used OAuth 2.0 for authentication, implementing it for all different application types and use cases was non-trivial. To simplify the authentication process in a service-oriented architecture, authentication can be dedicated to single service. We used the JWT standard for signing requests. Each service can verify a request independently without connecting to the authentication service. This approach requires only a small amount of authentication logic in each service while keeping services independent.

At Voltio we have used the described deployment pipeline for a year now. We use the continuous integration process to make sure our software always builds and passes tests successfully. Developers will quickly receive a notification of a failing code change. We have also found it valuable to store each build for later inspection. In cases where identifying a problem is difficult, comparing different versions of the software has been helpful. If we can find the first version that started the problem we can also find the exact code change that caused it.

Docker has quickly become a "must have" tool in our company's projects. In the beginning of 2016, when the new SirWise project was started, we started testing if it was suitable for our use. At that time Docker was still a quite new technology. Since then it has become mature and a lot easier to use as new native Docker applications were released and they became stable. Also many cloud hosting providers have added support and services for managing Docker containers.

BIBLIOGRAPHY

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, "Web services," in *Web Services*. Springer, 2004, pp. 123–149.

[2] C. Anderson, "Docker [software engineering]," *IEEE Software*, no. 3, pp. 102–c3, 2015.

[3] "Apple authorized service provider program," Apple, Available: https://www.apple.com/support/programs/resources/en/AASP_requirements_summary.pdf.

[4] "Apple GSX," Apple, Available: https://gsx.apple.com.

[5] "Apple service programs," Apple, Available: https://www.apple.com/in/support/programs/aasp/.

[6] "Apple to acquire beats music and beats electronics," Apple, Available: http://www.apple.com/pr/library/2014/05/28Apple-to-Acquire-Beats-Music-Beats-Electronics.html.

[7] T. M. S. Arik Ragowsky, "Enterprise resource planning," *Journal of Management Information Systems*, vol. 19, no. 1, pp. 11–15, 2002.

[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672

[9] A. Azeez, S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle, "Multi-tenant soa middleware for cloud computing," in *2010 IEEE 3rd International Conference on Cloud Computing*. IEEE, 2010, pp. 458–465.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 164–177.

[11] J. Bloomberg, *The agile architecture revolution: how cloud computing, rest-based SOA, and mobile computing are changing enterprise IT*. John Wiley & Sons, 2013.

[12] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple object access protocol (soap) 1.1," 2000.

[13] S. K. Chakrabarti and P. Kumar, "Test-the-rest: An approach to testing restful web-services," *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD'09. Computation World:*, pp. 302–308, 2009.

[14] F. Chong, G. Carraro, and R. Wolter, "Multi-tenant data architecture," *MSDN Library, Microsoft Corporation*, pp. 14–30, 2006.

[15] G. G. Claps, R. B. Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," *Information and Software technology*, vol. 57, pp. 21–31, 2015.

[16] T. H. Davenport, "Putting the enterprise into the enterprise system," *Harvard business review*, vol. 76, no. 4, 1998.

[17] F. D. Davis Jr, "A technology acceptance model for empirically testing new end-user information systems: Theory and results," Ph.D. dissertation, Massachusetts Institute of Technology, 1986.

[18] A. Dubey and D. Wagle, "Delivering software as a service," *The McKinsey Quarterly*, vol. 6, no. 2007, p. 2007, 2007.

[19] M. Feathers, "Microservices Until Macro Complexity," Available: https://michaelfeathers.silvrback.com/microservices-until-macro-complexity.

[20] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE, 2015, pp. 171–172.

[21] G. Feuerlicht, "Enterprise soa: What are the benefits and challenges," *Systems Integration*, pp. 36–43, 2006.

[22] G. Feuerlicht and J. Voříšek, "Utility computing: Asp by another name, or a new trend," *Proceedings of "Systems Integration*, pp. 269–280, 2004.

[23] R. Fielding and J. Reschke, "Hypertext transfer protocol (http/1.1): Message syntax and routing," 2014, Available: https://tools.ietf.org/html/rfc7230.

[24] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[25] J. Franks, P. Hallam-Baker, J. Hostetler, P. Leach, A. Luotonen, E. Sink, and L. Stewart, "An extension to http: digest access authentication," Tech. Rep., 1996.

[26] Google, "Gmail API," 2016. [Online]. Available: https://developers.google.com/gmail/api/

[27] A. Grigoriu, "Soa, bpm, ea, and service oriented enterprise architecture," *BP-Trends, www. bptrends. com*, 2007.

[28] H. Haas and A. Brown, "Web services glossary," W3C," W3C Note, Feb. 2004, Available: https://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/.

[29] A. N. Habermann and D. Notkin, "Gandalf: Software development environments," *IEEE transactions on software engineering*, no. 12, pp. 1117–1127, 1986.

[30] D. Hardt, "The oauth 2.0 authorization framework," 2012.

[31] H. He, "What is service-oriented architecture," *Publicação eletrônica em*, vol. 30, p. 50, 2003, Available: http://uic.edu.hk/~spjeong/ete/xml_what_is_service_oriented_architecture_sep2003.pdf.

[32] A. Holbreich, "JSON Web Tokens are made for Microservices," 2016. [Online]. Available: http://alexander.holbreich.org/jwt/

[33] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

[34] B. W. Johnson, "Fault-tolerant microprocessor-based systems," *IEEE Micro*, vol. 4, no. 6, pp. 6–21, Dec 1984.

[35] M. Jones, J. Bradley, and N. Sakimura, "Json web token (jwt)," Internet Requests for Comments, RFC Editor, RFC 7519, May 2015, http://www.rfc-editor.org/rfc/rfc7519.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7519.txt

[36] N. M. Josuttis, *SOA in practice: the art of distributed system design.* " O'Reilly Media, Inc.", 2007.

[37] D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA: service-oriented architecture best practices.* Prentice Hall Professional, 2005.

[38] N. Kratzke, "Lightweight virtualization cluster how to overcome cloud vendor lock-in," *Journal of Computer and Communications*, vol. 2, no. 12, p. 1, 2014.

[39] J. C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and analysis of hardware- and software-fault-tolerant architectures," *Computer*, vol. 23, no. 7, pp. 39–51, July 1990.

[40] K. B. Laskey and K. Laskey, "Service oriented architecture," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 1, no. 1, pp. 101–105, 2009.

[41] Y. Li and S. Manoharan, "A performance comparison of sql and nosql databases," in *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on.* IEEE, 2013, pp. 15–19.

[42] "Chapter 1: Service Oriented Architecture (SOA)," Microsoft, Available: https://msdn.microsoft.com/en-us/library/bb833022.aspx.

[43] A. S. Nascimento, C. M. Rubira, R. Burrows, F. Castor, and P. H. Brito, "Designing fault-tolerant soa based on design diversity," *Journal of Software Engineering Research and Development*, vol. 2, no. 1, pp. 1–36, 2014. [Online]. Available: http://dx.doi.org/10.1186/s40411-014-0013-7

[44] S. Neely and S. Stolt, "Continuous delivery? easy! just change everything (well, maybe it is not that easy)," in *Agile Conference (AGILE), 2013.* IEEE, 2013, pp. 121–128.

[45] "Announcing Zuul: Edge Service in the Cloud," Netflix, Available: http://techblog.netflix.com/2013/06/announcing-zuul-edge-service-in-cloud.html.

[46] S. Newman, "Building microservices," 2015.

[47] Object Management Group, "Corba," 2012. [Online]. Available: http://www.omg.org/spec/CORBA/

[48] "What Is SOA?" The Open Group, Available: http://www.opengroup.org/soa/source-book/soa/soa.htm.

[49] OWASP, “Input Validation Cheat Sheet,” 2016. [Online]. Available: https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet

[50] D. Peng, C. Li, and H. Huo, “An extended usernametoken-based approach for rest-style web service security authentication,” in *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*. IEEE, 2009, pp. 582–586.

[51] H. Petritsch, “Service-oriented architecture (soa) vs. component based architecture,” *Vienna University of Technology, Vienna*, 2006.

[52] “OnMetal: The Right Way To Scale,” Rackspace, Available: https://blog.rackspace.com/onmetal-the-right-way-to-scale.

[53] M. Rahman and J. Gao, “A reusable automated acceptance testing architecture for microservices in behavior-driven development,” in *Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium on*. IEEE, 2015, pp. 321–325.

[54] C. Richardson, “API gateway pattern,” Available: http://microservices.io/patterns/apigateway.html.

[55] ——, “Building Microservices: Using an API Gateway,” Available: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/.

[56] ——, “Introduction to Microservices,” Available: https://www.nginx.com/blog/introduction-to-microservices/.

[57] L. Richardson and S. Ruby, *RESTful web services*. " O’Reilly Media, Inc.", 2008.

[58] M. Ronayne and E. Townsend, “Case study: Distributed object technology at wells fargo bank,” *Cushing Group white paper) US: The Cushing Group, Inc*, 1996.

[59] M. Rosen, B. Lublinsky, K. T. Smith, and M. J. Balcer, *Applied SOA: service-oriented architecture and design strategies*. John Wiley & Sons, 2012.

[60] R. W. Schulte and Y. V. Natis, “Service oriented architectures, part 1,” *Gartner, SSA Research Note SPA-401-068*, 1996.

[61] G. Shachor, “Maintaining http session affinity in a cluster environment,” Sept. 20 2005, uS Patent 6,947,992.

[62] "Docker containers vs. virtual machines: What's the difference?" SolidFire, Available: https://www.solidfire.com/blog/containers-vs-vms/.

[63] T. Spencer, "API Security: Deep Dive into OAuth and OpenID Connect," 2014. [Online]. Available: http://nordicapis.com/api-security-oauth-openid-connect-depth/

[64] S. Tilkov, "How small should your microservice be?" Available: https://www.innoq.com/blog/st/2014/11/how-small-should-your-microservice-be/.

[65] K. S. Trivedi, M. Grottke, and E. Andrade, "Software fault mitigation and availability assurance techniques," *International Journal of System Assurance Engineering and Management*, vol. 1, no. 4, pp. 340–350, 2010. [Online]. Available: http://dx.doi.org/10.1007/s13198-011-0038-9

[66] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 45–52, 2011.

[67] C. D. Weissman and S. Bobrowski, "The design of the force.com multitenant internet application development platform," in *SIGMOD Conference*, 2009, pp. 889–896.