UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

GUSTAVO DE MEDEIROS CORREA

A cloud, microservice-based Digital Twin for the Oil industry — A Flow Assurance case study

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering

Advisor: Prof. Dr. João Cesar Netto

Porto Alegre
September 2022

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor João Cesar Netto for supporting me throughout the development of this work, for his constant engagement and availability, and for introducing me to the concept of Digital Twins.

I would also like to thank my family (especially my mother Lenira) for the help and encouragement they have given me during all these years as a Computer Engineering student. Without that constant support, I'm sure I wouldn't have made it this far.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer o meu professor orientador João Cesar Netto por todo o apoio que meu durante o desenvolvimento deste trabalho, por seu constante engajamento e disponibilidade, e por me introduzir ao conceito de Digital Twins.

Gostaria, também, de agradecer minha família (especialmente minha mãe Lenira) pelo suporte e encorajamento que me deram durante todos esses anos como estudante de Engenharia da Computação. Sem esse constante apoio, tenho certeza que não chegaria tão longe.

ABSTRACT

Digital Twins are one of the top recent technology trends, considered a front-runner in enabling the next generation of intelligent, digitalized industries. A Digital Twin is basically a real enterprise asset mirrored into the virtual world, created entirely in software, and utilized to innovate business, generate new revenue, and create value-producing opportunities. Naturally, building such demanding systems is not an easy task. Many organizational and technological concerns should be addressed, like real-time processing, data management, cybersecurity, and low latency communication. Moreover, Digital Twins should be extensible and support data integration with its physical counterpart, allowing enterprises to streamline their operations more safely. Considering there are no standard methodologies or technologies to design a Digital Twin, software developers struggle to develop a robust, reliable Digital Twin architecture that meets customers' needs.

Therefore, in consideration of these technical challenges, we decided to explore solutions in the oil and gas industry. Oil plants are known for having notoriously complex processes and an oppressive ecosystem when it comes to the adoption of new technologies, especially due to its intricate and ever-changing landscape. Despite these difficulties, this industry would certainly benefit from a well-designed Digital Twin. Hence, this work proposes a solution for an Oil Well Digital Twin architecture. We decided to use the flow assurance process of an Oil Well as a base to build a Digital Twin, aiming to study and fulfill those requirements and technical limitations. We implemented and tested several APIs, each one representing the virtual version of a real Oil Well component, enforcing the importance of microservices and cloud-native patterns in its design. When functioning together, these microservices comprise an entire Oil Well Digital twin solution, covering basic storage, communication, and monitoring fundamentals. This proof of concept demonstrates how a complex asset — like an Oil Well — can be built and organized in a completely digital manner. Lastly, this design is extensible and opens up further development opportunities.

Keywords: Digital Twin. Microservices. Cloud computing.

Um gêmeo digital baseado em microsserviços em nuvem para a indústria de petróleo — Um estudo de caso de garantia de fluxo

RESUMO

Gêmeos Digitais são uma das principais tendências tecnológicas recentes, considerados pioneiros na habilitação da próxima geração de indústrias inteligentes e digitalizadas. Um Gêmeo Digital é basicamente um ativo corporativo espelhado no mundo virtual, criado inteiramente em software e destinado a inovar os modelos operacionais e de negócios, ao mesmo tempo em que fornece novas oportunidades de geração de receita e valor. Naturalmente, construir sistemas tão exigentes não é uma tarefa fácil. Muitas preocupações organizacionais e tecnológicas devem ser abordadas, como processamento em tempo real, gerenciamento de dados, segurança cibernética e comunicação de baixa latência. Além disso, os Gêmeos Digitais também devem ser extensíveis e oferecer suporte à integração de dados com sua contraparte física, permitindo que as empresas agilizem suas operações com mais segurança. Considerando que não existem metodologias ou tecnologias padrão para projetar um Gêmeo Digital, os desenvolvedores de software tem dificuldade em desenvolver uma arquitetura de Gêmeo Digital robusta e confiável que atenda às necessidades dos clientes.

Além desses desafios técnicos, a indústria de óleo e gás é conhecida por ter processos notoriamente complexos e um ecossistema opressor na adoção de novas tecnologias, principalmente devido ao seu cenário intrincado e em constante mudança. Apesar dessas dificuldades, essa indústria certamente se beneficiaria de um Gêmeo Digital bem projetado. Assim, este trabalho propõe uma solução de uma arquitetura para um Gêmeo Digital de um Poço de Petróleo. Decidimos usar o processo de garantia de vazão de um poço de petróleo como base para construir um Gêmeo Digital, visando estudar e atender a esses requisitos e limitações técnicas. Implementamos e testamos várias APIs, cada uma representando a versão virtual de um componente real de um poço de petróleo, reforçando a importância de microsserviços e padrões nativos de nuvem em seu design. Ao funcionarem juntos, esses microsserviços compreendem uma solução completa de gêmeos digitais de poços de petróleo, abrangendo os fundamentos básicos de armazenamento, comunicação e monitoramento. Esta prova de conceito demonstra como um ativo complexo – como um poço de petróleo – pode ser construído e organizado de forma totalmente digital. Por fim, esse design é extensível e abre mais oportunidades de desenvolvimento.

Palavras-chave: Gêmeo digital. Microserviços. Cloud computing.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ANM	Arvore de Natal Molhada - Wet Christmas Tree
API	Application Programming Interface
CRUD	Create, Read, Update and Delete
DT	Digital Twin
DPTT	Downhole Pressure and Temperature Transmitter
DB	Database
ELK	Elasticsearch, Logstash and Kibana
HTTP	Hypertext Transfer Protocol
HTML	HyperText Markup Language
IPC	Inter-process communication
MQTT	Message Queuing Telemetry Transport
PDG	Permanent Downhole Gauge
RPI	Remote Procedure Invocation
REST	Representational State Transfer
UUID	Universally unique identifier
QoS	Quality of service
POM	Project Object Model

CONTENTS

1 INTRODUCTION

The oil and gas industry's relevance to the energy market and global economy is paramount. With notoriously complex processes, oil and gas plants operate with multiple closely interrelated subsystems, usually managed by independent teams, and have long and laborious maintenance schedules. Additionally, oil plants are also highly regulated due to health, safety, and environmental risks (WANASINGHE et al., 2020). For these reasons, the need to keep pace with technological advancements is vital for business prosperity. Following the movement towards industry digitalization, most oil companies are looking for state-of-the-art technologies to enhance productivity, increase revenues, and minimize capital and operational costs. However, implementing such novel concepts can be difficult and costly when working with complicated, constantly changing environments like an Oil Well plant. One of the biggest challenges in the Oil industry is incorporating any new asset or operational change into the existing infrastructure, making the adoption of digital technologies harder and a high-risk investment. Given the long lifecycle of such systems, integration issues became prominent over the years (PERNO; HVAM; HAUG, 2021), and interest in cyber-physical integration and simulations increased drastically. In order to solve many of these problems and promote digital transformation, the concept called Digital twin (DT) was introduced and quickly became a recent trend across both academia and industry.

A Digital Twin could be defined, in a general way, as a virtual representation of a physical product. Data from this virtual entity can be collected, analyzed, and visualized to make more informed decisions and to serve as a basis for simulations to optimize operations. In theory, there are many potential and perceived benefits related to the Digital Twin concept (JONES et al., 2020), but successful implementation cases are rare, and design definitions usually vary in scope. The lack of solutions could be explained by the difficulty in meeting software requirements with older, obsolete architectural design and management systems. Digitalization is time-consuming and requires complex coordination. The data monolith needs to be slowly broken into a distributed data fabric, and the computing substrate must be significantly different from those of the past. Only with the emergence of Industry 4.0 technologies that the DT concept matured, and real prototypes in the automotive, manufacturing, healthcare, and aviation industry were built (BARRICELLI; CASIRAGHI; FOGLI, 2019). Hence, cutting-edge technologies like artificial intelligence, machine learning, cloud, IoT, and microservice architectural patterns

are considered enablers for Digital Twins design (PENG; ZHANG; WU, 2017).

The main **goal** of this work is to propose and implement a solution for an Oil Well Digital Twin. The system will serve as a proof of concept to test the feasibility of this technical challenge, revealing good and bad design approaches for future implementations. We will take into account the different types of equipment, components, and sensors from the Flow Assurance process and create a set of microservices that will comprise the virtual version of the DT. A collection of APIs modeling the Oil Well components are also implemented and described, following a set of modern, high-grade API design patterns. In addition, a robust stack of backing services and monitoring technologies will be described and incorporated into the final solution. This arrangement of microservices will work in tandem with other supporting applications, and the design will consider cloud-native patterns for storage and messaging.

The remainder of this text is organized as follows. Chapter 2 gives an overview of the main concepts and challenges regarding Digital Twins, cloud-native development, and microservices. Chapter 3 presents a proposal for the solution architecture, highlighting the major building blocks and providing a brief description of each service in the design. Chapter 4 lists the techology stack used during development, emphasizing the features that made them eligible to be used in our final DT solution. Chapter 5 describes the developed APIs in each service of the Digital Twin, going over implementation and usability details. Chapter 6 presents the monitoring stack configuration used by the Oil Well DT. Chapter 7 goes over the installation and usability of the final solution. Chapter 8 concludes this article with some final remarks.

2 BACKGROUND

This chapter approaches the main topics of this work on a theoretical standpoint, providing an overview of Digital Twins, Microservice architecture, and Cloud Native software.

2.1 Digital Twin

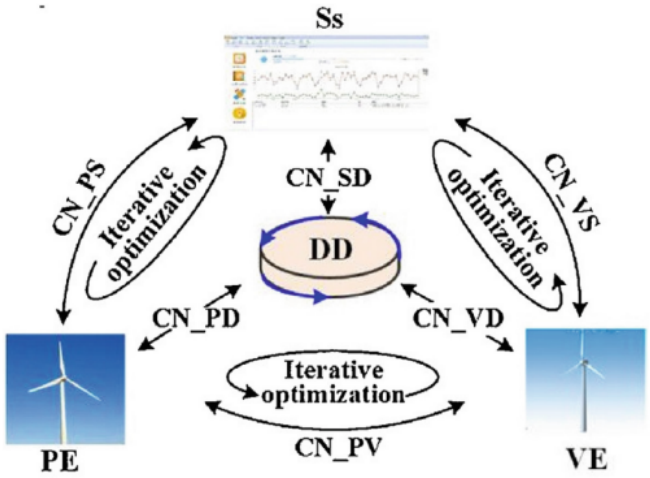
The term Digital Twin (DT) was first introduced in 2003 by (GRIEVES, 2014), presenting the idea that mirroring a physical asset in the digital space could improve operations throughout the life cycle of a product. At the time, most researchers had different understandings of what a DT entailed, and there was no clear definition of the concept. With the emergence of new technologies and Industry 4.0, however, the research by both industry and academia on Digital Twins grew exponentially, bringing clarity and a general consensus amongst the academic community on its meaning and importance.

According to (ADAMENKO; KUNNEN; NAGARAJAH, 2020), the core principle of a Digital Twin is that data should be exchanged, preferably in real-time, between the physical and virtual spaces through all phases of the product life cycle, bringing constant optimization to both models along with data analytics and visualization. (RASHEED; SAN; KVAMSDAL, 2020) presented the following definition of a Digital Twin: A digital twin is defined as a virtual representation of a physical asset enabled through data and simulators for real-time prediction, optimization, monitoring, controlling, and improved decision making.

From a modeling perspective, this definition leads to a five-dimension framework for DTs, consisting of physical, virtual, connection, data, and service parts, as presented in (Figure 2.1). PE represents the physical entity; VE represents the virtual entity; Ss represents the services for both PE and VE; DD stands for the DT data; and CN means the connection of different parts (TAO et al., 2019). The physical layer contains the real asset; a product like an aircraft, a supply chain, or an oil well. The virtual layer has the virtual representation of that asset, receiving information from the IoT sensors, processing them, and then feeding their simulations back to the physical layer. The service layer contains microservices that should add value to the product lifecycle. With all the information exposed by the DT, users are looking for ways to reduce cost, enhance operations, and improve decision-making. Enterprise software tools such as visualization services,

product quality services, diagnostic services, model calibration services, algorithm services, and various data services can be added to the service layer to fulfill those demands (WANASINGHE et al., 2020).

Figure 2.1 – Dimensions of a Digital Twin



Source: (ADAMENKO; KUNNEN; NAGARAJAH, 2020)

(MALAKUTI et al., 2020) also mentions that Digital Twins should have hierarchical and associational relationships, just like their real-world counterparts. A single entity that provides value without needing to be broken down is a discrete DT, whereas a combination of entities is a composite DT.

Despite the increasing amount of papers and growing interest from large companies on DTs, the design methodology of a Digital Twin is not standardized. Many challenges should be considered when building a DT, such as data management, security, real-time communication, interaction with the physical asset, large-scale computation, and continuous model updates (RASHEED; SAN; KVAMSDAL, 2020). In addition, with the large variations on infrastructure, software, and business cases, most Digital Twins are built with either data or system-based design approaches. The former focuses on building a data structure that organizes and links the sensor data and other information, where the latter creates a system model of the physical object, defining logical links and relationships between the individual components (ADAMENKO; KUNNEN; NAGARAJAH, 2020).

In this work, we are interested in both data and system-based Digital Twins designs in the Oil Industry context, resulting in a hybrid solution.

2.2 Cloud Native Software and Microservices

Recent advancements in digitalization and Industry 4.0 raised customers' expectations regarding software development. Large companies that run mission-critical businesses like finance require their software to have zero downtime, shortened feedback cycles, mobile and multi-device support, scalability, embedded intelligence, and many more requirements which are unattainable by monolithic software systems. Cloud-Native software, alongside Microservice design patterns, is a new architectural style devised to fulfill those demands.

One key necessity for modern applications is to be constantly available, which can be challenging when dealing with complex systems and unstable environments. (DAVIS, 2019) affirms that cloud-native software should be designed to anticipate failure and remain stable even when the infrastructure it is running on is experiencing outages or is otherwise changing. One way to achieve this is through redundancy. With multiple application instances deployed in a highly distributed manner across all infrastructure, the redundant copies can compensate for cases of inevitable failure. Moreover, software built as small, loosely coupled components, independently deployable and releasable, often prevents failures from cascading throughout the entire system. This aforementioned architectural style is often called Microservices (RICHARDSON, 2019), and gained popularity as developers began to decompose their monolithic systems into smaller units. A Microservice oriented system also facilitates frequent releases and scalability. Smaller and independent components have a more agile release model, enabling daily deployments and upgrades without compromising the entire solution or having downtime. In addition, with the ever-increasing volume of data flowing over the internet, especially with IoT devices, applications should scale dynamically as request traffic changes, which can be hard to achieve when working with monolithic solutions.

Of course, there are considerable challenges to consider when developing cloud-native software. When working with multiple instances of an application, software developers must reexamine how they deal with application state, scale-out/in models, configuration throughout environments, as well as shutdown and startup methods of applications. A highly distributed environment also brings challenges related to data management, synchronization of multiple databases, synchronous and asynchronous communication, automated retries, metrics, logging, and routing. And for that reason, software developers recurred to platforms such as AWS, Google Cloud Platform (GCP), and Microsoft Azure,

as they provide support for and even implementations of many of the patterns presented in this section. A cloud-native platform is absolutely essential for organizations building and operating cloud-native software (DAVIS, 2019).

In this work, we are interested in how Cloud-native and Microservice patterns can be applied to Digital Twins design.

3 PROPOSAL

This chapter will introduce the main building blocks of our Digital Twin architecture. Section 3.1 starts with a brief motivation. Section 3.2 will give an overview of the microservices comprising the physical and virtual models of the DT, as well as go over connection and data fundamentals used in our solution.

3.1 Motivation

A definitive design pattern for building a Digital Twin does not exist, regardless of the industrial context. Enterprises and academia have mostly chosen frameworks and architectural styles based on business needs, provided that the design fulfilled the functional and technological requirements of the stakeholders. The lack of standardization led to a plethora of different DT solutions, employing diverse technologies and architectures, and therefore became a challenge amongst software developers.

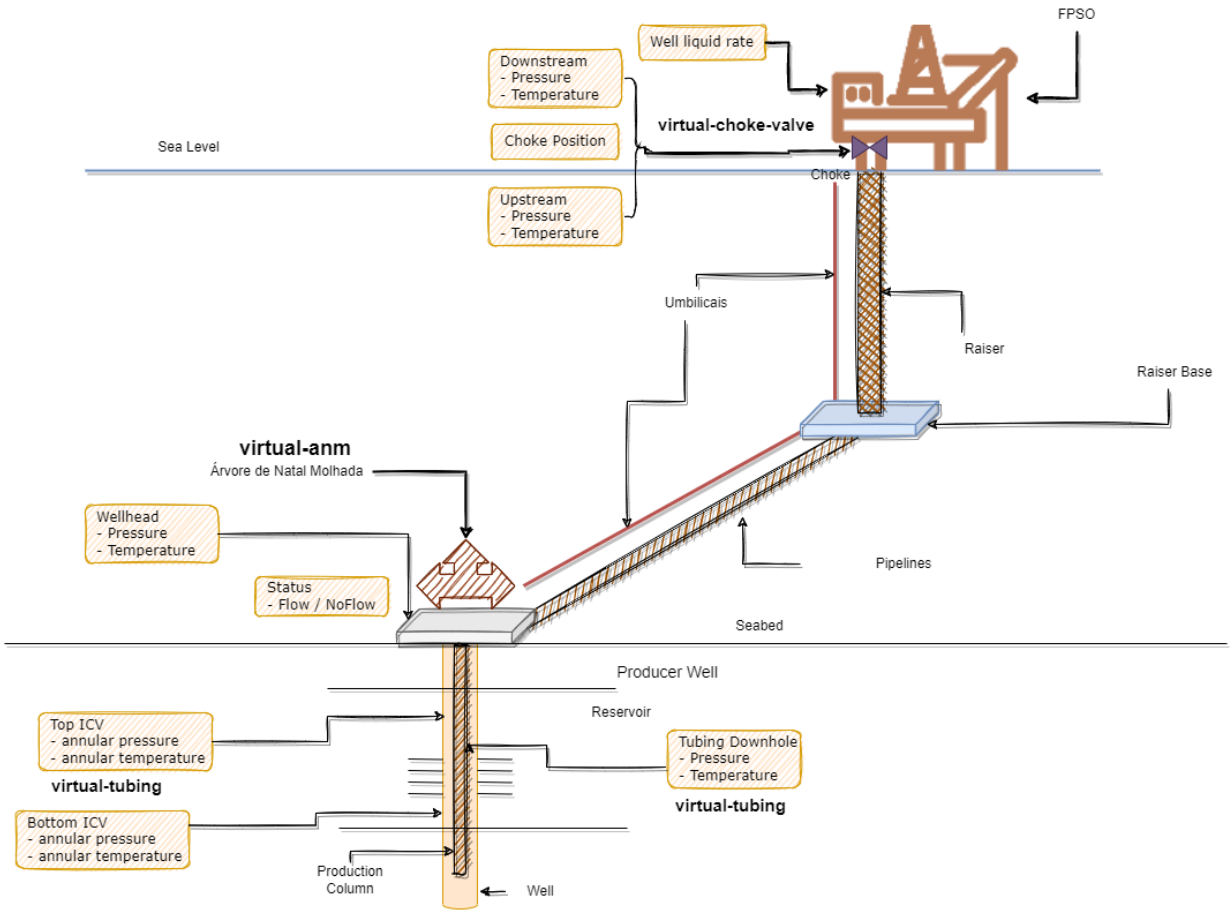
The well of an oil and gas plant is remarkably complex, possessing several types of components and machinery, some of which are regularly changing. Figure 3.1 illustrates a high-level overview of an oil well, highlighting some of its most important parts. In this graduation thesis, our primary motivation was to study and implement a potential architecture to model an oil well DT, describing the challenges in translating the physical components from Figure 3.1 to their virtual counterparts. Our study will also cover all the technical details regarding data, connectivity, and design decisions.

3.2 Architecture overview

To develop the Digital Twin of the Flow Assurance process of an Oil Well, the five-dimension framework presented in (Figure 2.1) was used as reference. Using this format, a series of microservices were created to address each dimension: physical, virtual, connection, data, and service.

A high-level overview of the solution is shown in (Figure 3.2). Additionally, a series of supporting services were arranged to provide a complete monitoring and analytics stack for the DT solution, offering log aggregation and several data visualization options. Due to its complexity, the integration of these tools is described only in Chapter 6.

Figure 3.1 – Simplified illustration of an Oil Well



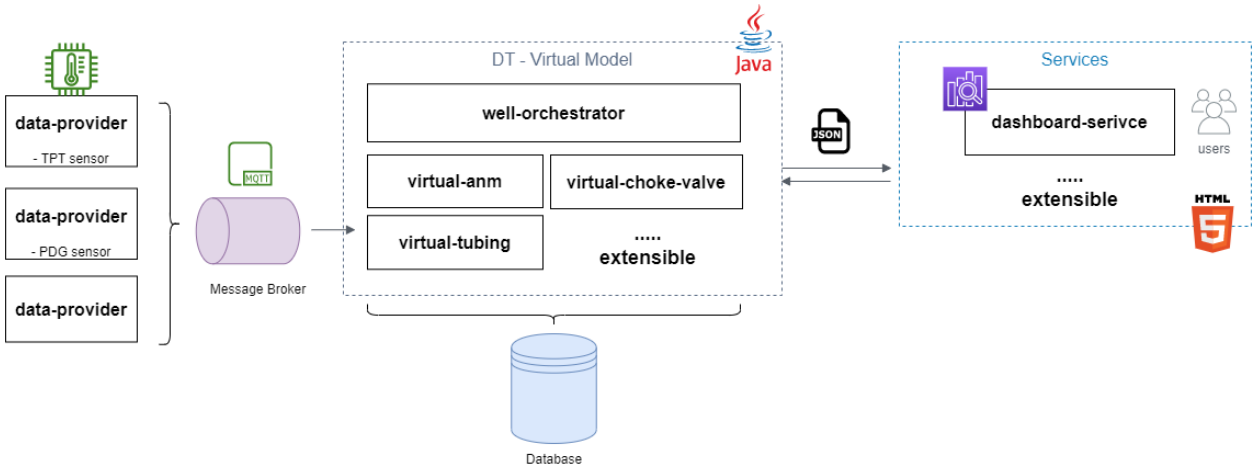
Source: Author

3.2.1 Physical

The physical instance of the DT — the Oil Well and assets to establish the flow control — has a vast number of sensors scattered throughout its components. For instance, we have the DPTT (Downhole Pressure and Temperature Transmitter) and the PDG (Permanent Downhole Gauge) sensors transmitting temperature and pressure measurements from different parts of the Well, like the ANM (Árvore de Natal Molhada - Wet Christmas Tree)[1] and many others. Considering we do not have a physical Well to perform any cyber-physical integration, the data exchange between physical and virtual spaces was simulated with the creation of the *data-provider* microservice. The *data-provider* is a small, easily deployable and scalable application that will emulate the real-world sensors, providing mock data to the virtual model of the DT. Distinct instances of the *data-provider* application can function as different data sources and support time-series data of different

[1]The mentioned acronyms were translated using dicionariodopetroleo.com.br

Figure 3.2 – High-level overview of the system architecture

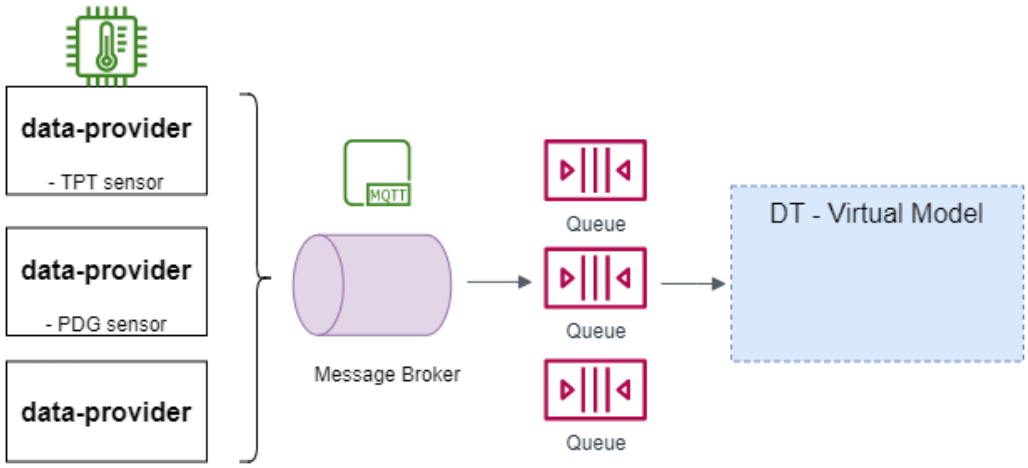


Source: Author

periodicities, just like their physical counterparts.

The *data-provider* can be seen in the leftmost part of (Figure 3.2) and highlighted in (Figure 3.3).

Figure 3.3 – *data-provider* instances communicating with the DT



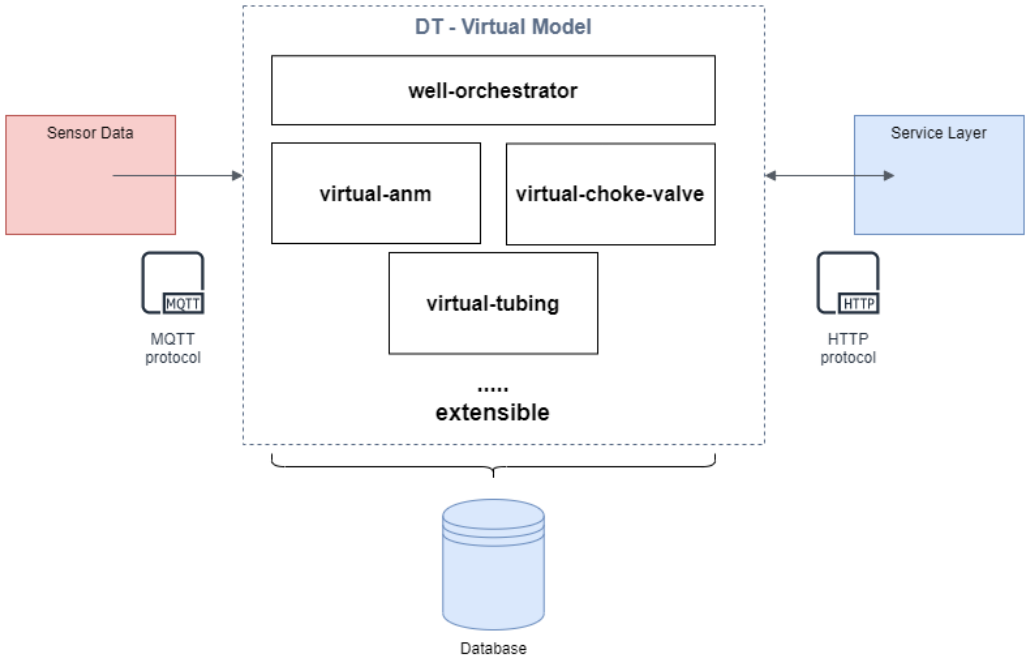
Source: Author

3.2.2 Virtual

For the virtual representation of the Oil Well, our proposal organizes a series of microservices in a composite and hierarchical way (MALAKUTI et al., 2020). Each application will define a component or equipment utilized by an Oil Well (or multiple Oil

Wells), acting as discrete Digital Twins. In this work, we decided to model three significant components within an Oil Well in a very simplistic manner, as creating a detailed and realistic representation of them goes beyond the scope of a POC. These components are the *virtual-anm*, *virtual-choke-valve*, and *virtual-tubing*. Each of these microservices is supposed to work independently, unbeknownst to one another. The microservices can emulate the physical behavior of their respective components, perform calculations and tasks, and work with data storage related to sensor readings and simulations. And finally, at the top of the service hierarchy, we have a microservice representing the Well proper, the *well-orchestrator*. The *well-orchestrator* is responsible for managing the logical links and relationships between components. It is worth mentioning that this design decision was made with extensibility and scalability in mind, trying to facilitate the addition and removal of new services as it would happen during the maintenance of an actual Oil Well. Thus, the combination of all these discrete Digital Twins, and the resources exposed by their APIs, will result in a complete representation of an Oil Well. The centermost part of (Figure 3.2) illustrates this arrangement and is highlighted in (Figure 3.4).

Figure 3.4 – Virtual model of the Oil Well DT



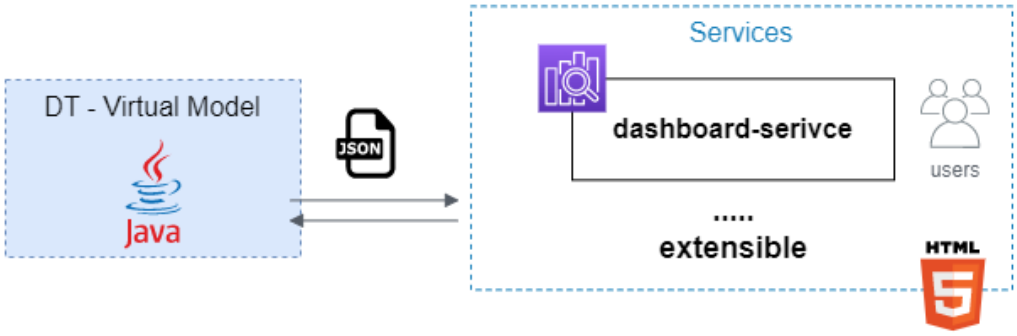
Source: Author

3.2.3 Service

Applications in the service layer are usually monitoring, alert, intelligence, or simulation services. In the Oil industry context, a monitoring service could offer a user interface so plant operators can check the health of the Oil Well in real-time. Similarly, pressure and temperature measurements of each component could be reviewed, manually, by the employees, or in an automated manner, by the DT microservices. In addition, if an anomaly is detected during operation, a warning message could be triggered by the alert service and inform the responsible parties using e-mail or phone. Lastly, a common use for applications in the service layer concern simulations and machine learning models to be applied on top of all the provided data, optimizing the virtual and physical DTs.

In this work, we opted for just one of the aforementioned alternatives, creating a simple dashboard microservice. Its interface is minimalistic, intended to be a direct and user-friendly way to interact with all of the developed APIs. With the *dashboard-service*, a user can perform all CRUD operations with the DT resources, add and remove components from previously created Wells, and observe or filter measurements coming from the *data-provider* sensor simulations. The *dashboard-service* can be seen in the rightmost part of (Figure 3.2) and is highlighted in (Figure 3.5).

Figure 3.5 – *dashboard-service* communicating with the DT



Source: Author

3.2.4 Connection

When working with multiple microservices in a constantly changing environment, inter-process communication (IPC) can become challenging. Applications can have different ways of communicating, such as *one-to-one*, where only one service pro-

cess the client's request, and *one-to-many*, where multiple services are involved. There are different methods to approach these interactions, and they can either employ synchronous/asynchronous request/response-based mechanisms or purely message broker-based communication (RICHARDSON, 2019).

In *one-to-one* interactions, most systems rely on request/response communication, also known as Remote Procedure Invocation (RPI). The most common RPI technology at present is definitely HTTP alongside REST, but there are alternatives like gRPC and Apache Thrift. When the communication is done synchronously, the client makes a request and remains blocked until it receives a response. This block ultimately reduces the entire system's availability, and is one of the many reasons as to why asynchronous IPC is more favored when working with microservices. In the asynchronous style, the client is never blocked, and the server may or may not respond immediately or even respond at all.

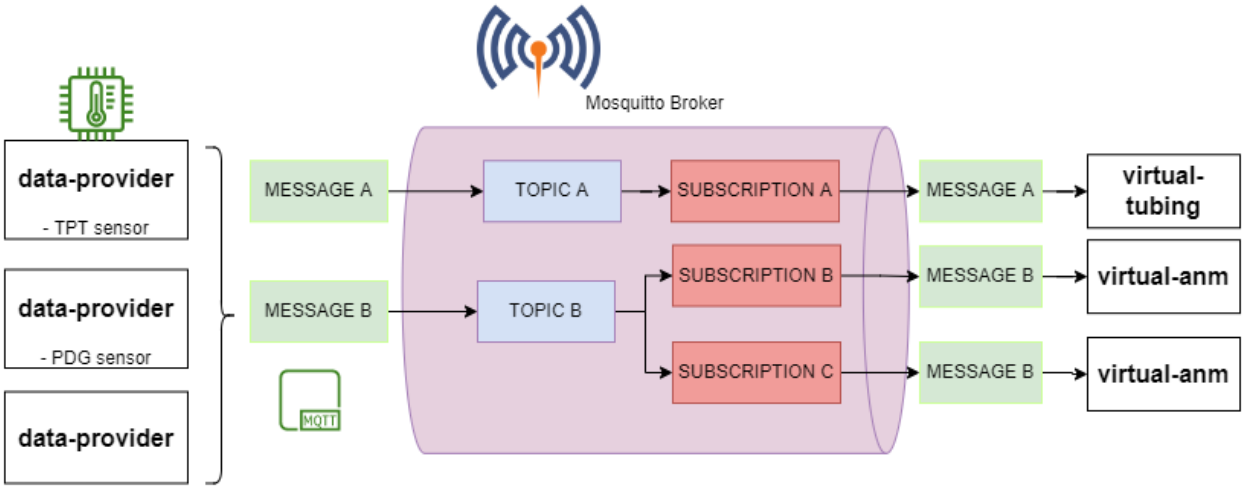
Synchronous IPC also raises concerns about routing and handling communication failures. In a landscape with hundreds of microservices, applications must not rely on physical IP addresses, as new service instances are always being added, scaled, and removed. A service discovery mechanism is necessary to abstract the physical location of large enterprises' microservices. The IP addresses can be stored in a service registry, or the deployment infrastructure itself can be responsible for service discovery. Moreover, communication timeouts and consistent failure calls should be handled by a circuit breaker pattern. Circuit breakers can prevent clients from repeatedly calling a failing service, thus increasing the service availability (CARNELL, 2017).

For *one-to-many* interactions, most systems use asynchronous message brokers with protocols like AMQP or STOMP. The data exchange between two applications using this method usually consists of sending data through a message channel created by the message broker. The sender and receiver application have no knowledge of each other, and their only concern is to listen or send messages to the proper channel. The message channels can be either *point-to-point*, where messages are delivered to exactly one consumer, or *publish-subscribe*, where messages are delivered to one or more consumers (HOHPE; WOOLF, 2022). There are also a few drawbacks to this alternative, as working with a message broker raises concerns regarding message delivery and handling duplicate events.

All of the Oil Well microservices mentioned above will have their own APIs, exposing data and resources to applications on both virtual and service layers. Conse-

quently, synchronous and asynchronous request/response communication patterns will be employed, as well as the use of an MQTT message broker with a *publish-subscribe* message channel, intended to work with the *data-provider* sensor readings. Figure 3.6 illustrates how the *publish-subscribe* channels work. More details on the implementation will be described in chapter 5.

Figure 3.6 – Publish-subscribe message channel



Source: Author

3.2.5 Data

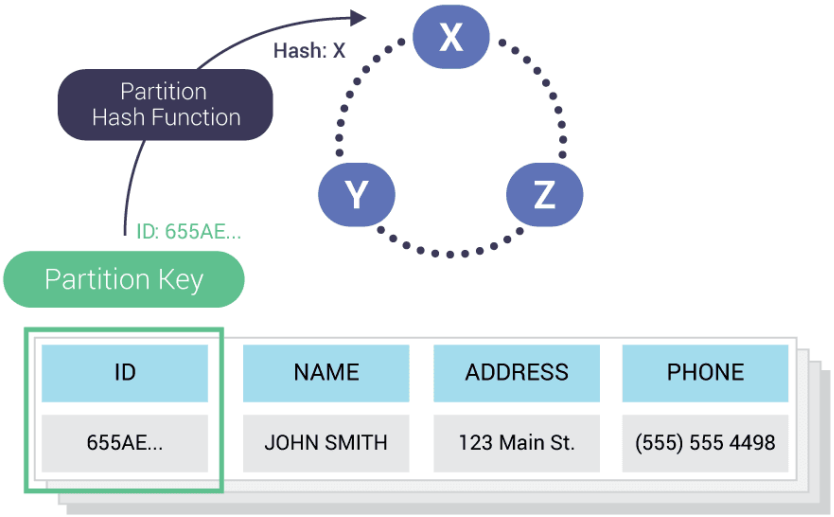
Digital Twins are known for generating massive amounts of data from different sources, possessing diverse types, formats, and sizes. Compared to monolithic applications, managing data in a highly distributed landscape is complex and requires additional resilience patterns and computing power. Consequently, Digital Twin databases should excel in working with big data while keeping a low latency and a high throughput performance. In order to keep data availability high, DT databases should also support replication with an acceptable storage cost.

Besides the mentioned performance constraints, a cloud environment is usually in a constant state of change, prompting DT databases to be viewed as stateful, attached resources. Treating backing services as a resource guarantees loose coupling, and allows DB administrators to swap, add, remove, and migrate databases from different vendors without any code changes (WIGGINS, 2017). There are many options in the market for time-series databases, like InfluxDB, and other performant alternatives like Redis and

ScyllaDB (KNEBEL; WICKBOLDT, 2020).

All "virtual" microservices of the Oil Well Digital Twin, with the exception of the *dashboard-service*, was bound to a shared, stateful backing service for storage during development and testing. For this POC, a three node ScyllaDB data center was set up using a Network Topology Strategy and replication factor of three. The replication factor decides to how many nodes the data will be replicated. The Network Topology Strategy is DC-aware, and allows the developer to set the replication factor independently for each data-center. In this work, we are only using one data-center with all nodes on the same rack, so updating the topology or replication strategy will not result in downtime or require a shutdown. Each microservice will also have its own dedicated keyspace, where all the tables are defined. ScyllaDB organizes the tables in a set of rows and columns identified by a primary key. Across the nodes, the data is replicated by partitions. The partition key (the unique identifier for a partition) is represented as a token, hashed from the primary key. Most resources in the developed APIs will use their universally unique identifiers as partition keys, and sub-resources will use their parent's. Figure 3.7 illustrates the ScyllaDB data model in a cluster with nodes X, Y, and Z.

Figure 3.7 – ScyllaDB Data model



Source: Scylla University

4 TECHNOLOGIES

This chapter will list the technology stack used during the development of the Oil Well Digital Twin, highlighting its most important features. The monitoring technologies will be described separately in chapter 6, as they are not part of the core DT implementation.

4.1 Java

Java is one of the most popular objected-oriented programming languages in the world, widely adopted by large and small enterprises. With millions of active developers, the Java community has kept the language up to date with modern standards for decades and continues to make it grow and evolve. Java is well-known for being platform-independent, running on any computer as long as it has the Java Runtime Environment (JRE) installed, justifying its longevity. By virtue of its maturity and ease of use, Java has gained more space in the mobile, IoT, and cloud landscape (ORACLE, 2022).

In this work, we use Java 8. Java 8 is arguably the version that brought the most profound changes in Java history, especially regarding parallel processing. With the advent of big-data applications, programmers need to lean on multi-core computers and clusters to process workloads efficiently. Java 8 brought several features to make parallelism easier, more concise, and maintainable. If the necessity to upgrade the Java version of our DT solution arises, the migration will be practically seamless (URMA; FUSCO; MYCROFT, 2019).

4.2 Spring Boot Framework

Spring has become one of the most popular Java frameworks, innovative and mature enough to be considered a default solution for large-scale, production-ready Java applications. It started as a dependency injection tool, helping Java developers manage object relationships in large enterprise software. Over the years, the framework evolved and expanded its ecosystem into several other projects, offering a comprehensive set of services like configuration, security, web, big data, and cloud. Following the recent architectural trends, the Spring team also moved away from monolithic architectures and

embraced small, highly distributed systems. Led by Pivotal Software (VMWARE, 2022) and backed by a large consortium of organizations and developers, Spring is guaranteed to remain relevant for years to come (CHANDRAKANT, 2022).

In this work, we rely heavily on the Spring Boot framework, which delivers features focused on Java-based, REST-oriented microservices.

4.3 Maven

Maven is a Java-based build and dependency management tool. It automates tasks like compiling source code and downloading Jar files, which are usually done manually by developers. When working with APIs, libraries are being constantly updated, added, and removed, which is error-prone when dealing with hundreds of microservices. Maven uses the Project Object Model (POM), an XML file containing configuration and versioning details to search and download the correct dependencies automatically. Looking at the POM file, a developer can quickly see the dependency list of a project and make the necessary adjustments. Ultimately, building and managing projects become easier and less time-consuming with Maven, and it's a must-have for any Java application (APACHE-FOUNDATION, 2022).

4.4 Mosquitto broker

Mosquitto is an open-source message broker that implements the MQTT protocol 5.0, 3.1.1, and 3.1. Its lightweight messaging makes it a perfect candidate for low-power IoT devices like sensors or microcontrollers, which are very prominent in Digital Twins. Mosquitto also uses a publish-subscribe model, ideal for applications that need to broadcast information around multiple consumers while ensuring efficiency and security, with minimal message loss and resource usage. If there's a need to distribute the workload, Mosquitto can use MQTT v5 shared subscription capabilities. Lastly, Mosquitto is backed by a large, active online community to aid in general consulting, custom developments, and commercial support (LIGHT, 2017).

4.5 Scylla DB

ScyllaDB (SCYLLADB, 2022) is a NoSql distributed database built for optimal speed, efficiency, and scale. It meets several requirements of modern IoT applications with its flexible data modeling and reliable performance. Scylla can process billions of time-series events in real-time while maintaining low and consistent latency, auto-tuning itself as data streams increase and reduce dynamically. Besides the high throughput ingests, Scylla provides high availability and resilience with cross-regional data replication, ideal for distributed IoT operations in the edge or clouds. Scylla is also highly scalable and mindful of hardware utilization, ensuring a low cost of ownership. For these reasons, Scylla is considered the definitive DynamoDB[1] and Apache Cassandra[2] alternative, easy to use with built-in Spark[3] and Kafka[4] integration, and backed by a large open source community.

4.6 Docker

Docker is an open-source platform for installing, shipping, and running software. It consists of a command-line program and a set of remote services that rely on container technology. With containers, system administrators can have better control over system resources, effortlessly inject environment-specific configuration, and have an overall simplified experience managing infrastructure and software deployments. Currently, containers are more favored than virtual machines, offering the same isolation and scalability features while being more lightweight and efficient. Many large enterprises are using Docker for real-world production applications, especially in the cloud-native landscape. For teams working with dynamic scaling and continuous integration/deployment pipelines, Docker has proven to be a robust tool that reduces customer impact and makes production deployments fast, repeatable, and trustworthy (NICKOLOFF, 2020).

[1] DynamoDB: https://aws.amazon.com/pt/dynamodb/
[2] Cassandra: https://cassandra.apache.org
[3] Spark: https://spark.apache.org
[4] Kafka: https://kafka.apache.org

5 API DEFINITION

This chapter will describe the DT microservices implementation in more detail. Section 5.1 will go over some general API design decisions, like layout, message formatting, and versioning strategies. Subsection 5.1.4 will explain how all MQTT channels in the DT were configured. Section 5.2 will describe all the API endpoints.

5.1 Design Principles

5.1.1 Resource-oriented API

Most of the developed APIs in this POC could be considered resource-oriented. The concept of a "resource" is usually defined as a business object that APIs manage with a set of standard actions: create, get, list, delete, and update. A famous synchronous IPC mechanism that relies heavily on resource layouts is REST. Most RESTfull APIs follow a set of architectural constraints, using HTTP and mostly JSON to manipulate resources. The rules established by REST guarantee that APIs remain consistent and predictable and, for the most part, that their methods remain idempotent and without undesirable side effects (GEEWAX, 2021).

The APIs developed in the subsequent sections of this work follow the "Level 2" REST maturity level defined by (RICHARDSON et al., 2022). A level 2 API possesses the REST standard methods implemented and uses HTTP verbs to perform actions, such as GET to retrieve, POST to create, and PUT to update. The request body and query parameters serve as inputs to define those actions. Furthermore, each resource is referenced in its respective endpoints URL. For identification, we opted to create IDs on the server side, as user-generated identifiers can lead to security vulnerabilities. When the creation methods are invoked, each resource is assigned with a UUID. UUIDs are common 128-bit identifiers with enough available IDs to make collisions negligible. Additionally, SycllaDB is already prepared to store and index UUIDs.

5.1.2 Message Format

Choosing the right message and serialization format is an important decision for API design. The APIs' performance, IPC, usability, and evolvability can depend on this choice. In the microservice landscape, it is also essential to use cross-language formats. The most common categories to choose from are either text-based like JSON and XML, or binary like Avro or Protocol Buffers. All of the microservices mentioned in this work will use **JSON**.

The JSON format became a standard in most modern web servers. It is a human-readable and dynamic data structure without a strict schema, making it remarkably flexible to use. JSONs' versatility is an important feature when building evolvable interfaces and making backward-compatible changes to the API. Consumers can merely use the fields of interest and ignore the rest, although that could accidentally create malformed data if misused. Like most String serialization formats, JSON uses UTF-8 encoding. A downside of JSON is its verbosity and the overhead of parsing text, which can be inefficient in some cases (FREEMAN, 2022).

5.1.3 Versioning and compatibility

Upgrading and changing a microservice-based API requires much more planning compared to a monolithic one. Clients can not upgrade their systems whenever a new API version is shipped, so interface changes should be done carefully and without side effects. In addition, cloud-native applications will frequently have both old and new versions running simultaneously, most commonly during rolling upgrades, to meet availability requirements without downtime. For that reason, our DT microservices use **semantic versioning** (PRESTON-WERNER, 2022).

Semantic versioning proposes a set of rules for numbering and controlling software releases. The version numbers are organized in the following format: *MAJOR.MINOR.PATCH*. Table 5.1 shows the rules for incrementing each version number.

In the developed DT microservices, we place the *MAJOR* version number at the start of the URL path. This method allows clients to call different versions of a single endpoint simply by changing the *"/v1/.."* path to *"/v2/..."*.

Table 5.1 – Semantic versioning increment rules

Version	*Increase when*
MAJOR	You make incompatible changes to the API.
MINOR	You add a new, backwards compatible feature.
PATCH	You make backwards compatible bug fixes.

Source: (PRESTON-WERNER, 2022)

5.1.4 MQTT outbound and inbound channel

To create the MQTT outbound and inbound message channels, consumers and producers use the Spring Integration libraries (FISHER et al., 2022), which in turn implement the Eclipse Paho MQTT Client library (ECLIPSE, 2022). There are several configuration options for the client connection and channels, and we leave most of them as default with a few exceptions. For instance, a user and password are set to the MQTT client to provide basic security to the Mosquitto broker. These password credentials are provided to the Mosquitto service running in docker via bind mount. Another important parameter is *clean session*, which sets whether the client and server should remember state across restarts and reconnects. This property is vital in the cloud-native environment, as applications are expected to stop and restart frequently. With the broker state maintained, message delivery will be reliable, and the subscriptions are treated as durable. Another configuration option that is parameterized is *max inflight*, which should be increased in a high-traffic environment. The MQTT Paho library stores messages on the client side (either in a file or in memory) and only removes it before the broker confirms its receipt. As a result, performance problems could occur if the limit of in-flight messages is reached.

As for the outbound and inbound channels, a few parameters are worth mentioning. To begin with, the MQTT level of service is configurable. In the developed microservices, we opted to use the highest level of service QoS 2. This level guarantees that each message is published and received only once, eliminating any chance of message loss and duplication. Although QoS 2 is the safest level of service, it is also the slowest due to the four-part handshake between sender and receiver. This additional request and response flow is made to ensure that each message is transmitted, despite the cost. Nevertheless, this property is easily changeable through a configuration file. Another mutual configuration between inbound and outbound channels is the message converter. The Paho library provides the option to parameterize the converter responsible for converting String pay-

loads to bytes and vice-versa. To this converter, we supply the QoS, retain settings (which keeps the last published message on the broker topic), and a UTF-8 charset. Uniquely to the outbound channel, we manually define asynchronous communication to avoid client blocks, and provide a default topic for cases when the MQTT header is empty.

5.2 Microservice endpoints

This section will describe all API endpoints exposed in the DT microservices. We start with services from the physical layer, consisting of the *data-provider*. Next, we will go over the virtual layer microservices, such as *well-orchestrator* and other component-related applications. For the service layer, we explain the purpose and general design details of the *dashboard-service*. We end the chapter by describing all endpoints related to the sensor readings, and a few utility endpoints like health and monitoring checks.

5.2.1 data-provider

The *data-provider* microservice only goal is to simulate the physical Oil Well sensors, generating streams of mock data. It will create an *N* number of messages specified by the client via endpoint call, as described in table 5.2. The generated messages will be published to the proper Mosquitto-broker topic, intended to be consumed by another service shortly after.

Table 5.2 – *data-provider* endpoints

HTTP method	*Path*	*Action*
POST	/v1/send-message	Creates and sends *N* messages to the specified MQTT broker topic.

Source: Author

In our DT solution, there are three microservices acting as consumers and many measurement types available, so the parameters shown in table 5.3 must be informed to post messages to the appropriate topics.

Every time a component resource is created (anm, tubing, etc.), a new MQTT topic with the following format is added to the subscription list:

Table 5.3 – */v1/send-message* query parameters

Name	*Description*	*Value example*
componentType	Specifies the type of component the message belongs to.	tubing
componentId	The UUID of the component entity.	7f8ba085-7834-4341-874b-e45e61c5cbb3
measurementType	The type of sensor reading the message will be.	pressure
number	The number of generated messages.	100
rate	The time between sending in milliseconds.	10
customPropertyName	The name of the property when using a custom measurement type..	speed

Source: Author

<componentType>.<componentId>.<measurementType>.

As described in chapter 3, each microservice in the virtual layer represents a component or equipment utilized by an Oil Well, so the options for the field **componentType** are the following:

1. choke
2. anm
3. tubing

The field **componentId** represents the component universally unique identifier (UUID). This field is required to handle cases in which we have multiple components of the same type, generating data related to the same kind of measurement.

Besides the virtual representation of DT components, physical sensors can also read data of different types and units of measurement. For this reason, a proper way to differentiate and generalize the sensor data type was required, and the field **measurementType** was created in the *data-provider* interface. One of the most prominent measurements in the Oil Well DT is temperature and pressure, which are estimated to generate hundreds of requests per second. All microservices mentioned in chapter 3 are prepared to handle temperature and pressure readings, possessing dedicated tables and repositories due to the high load of requests. The *choke-valve* microservice also possesses a unique measurement type besides temperature and pressure called *flow*, which serves to

demonstrate that each microservice can have its own properties and features independently. Admittedly, allowing the client to only send hard-coded measurement types is not ideal for a constantly changing environment like an Oil Well, so the possibility to generate custom measurements was also added. To generate messages with a custom property, the client should fill the **measurementType** field with the value *custom* and fill the field **customPropertyName** with the respective property name. Both **measurementType** and **componentType** fields are validated in the interface at each endpoint call, and **customPropertyName** is optional for any cases besides custom measurements. Ultimately, the options for the **measurementType** field are presented below:

1. temperature
2. pressure
3. flow
4. custom

The fields **number** and **rate** relate to the number of messages and the sent frequency, respectively. Both expect integers as input, and the rate is measured in milliseconds. Using the rate field, the data provider can simulate time-series data of different periodicities. The */v1/send-message* endpoint works **asynchronously**, so the client will not be blocked when setting a large number of messages or a very high rate value. The number of threads and concurrency configuration are auto-configured by Spring Boot, but the default settings can be fine-tuned in the application properties file (WEBB et al., 2022).

The message payload consists of a timestamp and a randomly generated number mocking the actual measurement, alongside some specific MQTT headers containing topic and QoS information. When using a custom **measurementType**, an additional **propertyType** field is also added to the final payload. Considering the diverse amount of sensors in an Oil Well, we should not expect all readings to be real numbers or use the same unit of measure. Thus, the generation of the value field uses a factory class named *DataGeneratorFactory*. The Factory pattern is a famous creational design pattern in Java, where the client can create objects without any knowledge of the creation logic. Based on the component type, the *DataGeneratorFactory* can create different data generator objects and refer to them using a common interface. Currently, the data provider has three implementations of the *DataGeneratorFactory*, one for each type of component (choke-valve, tubing, anm). Thereby, we guarantee more flexibility in data generation when

future microservices are added to the ensemble. If a developer is working with a new microservice or a more complex data type, he can simply create a new implementation of the *DataGeneratorFactory*.

5.2.2 well-orchestrator

The *well-orchestrator* microservice is mostly responsible for managing a "Well" resource and its relationships with other components. A physical representation of a Well resource would be equivalent to the entire Oil Well itself, containing general data about the plant and the discrete DTs associated with it. Table 5.4 shows the implemented endpoints for the *well-orchestrator* microservice.

Table 5.4 – *well-orchestrator* endpoints

HTTP method	*Path*	*Action*
GET	/v1/well	Return all Well resources.
GET	/v1/well/{id}	Returns a Well resource with given *id*.
POST	/v1/well	Creates a Well resource.
PUT	/v1/well/{id}	Updates a Well resource with given *id*.
DEL	/v1/well/{id}	Deletes a Well resource with a given *id*.
POST	/v1/add-component/{id}	Adds a component to a Well resource with given *id*.
DEL	/v1/remove-component/well/{id}/component/{comId}	Removes a component with *comId* from a Well resource with given *id*.

Source: Author

The REST standard methods are the first ones described, containing basic CRUD operations. The *id* field is the generated UUID returned when a new resource is created. The PUT method will only update general information related to the Well and will leave relationship changes to the custom methods.

A distinct characteristic of the Well resource is its many-to-many relationships with other discrete DTs. To map these associations, we opted to create an add and remove

custom method. The *add-component* method will create an association with a component, while the *remove-component* will delete it. Considering that the *well-orchestrator* is on top of the service hierarchy, we elected it to be the managing resource — or parent — of this relationship between well and components. The Well resource UUID must be informed as an endpoint path variable, and the associate resource information (in this case, UUID and the componentTypes described in section 3) should be passed in the JSON body. A list of associated components of a Well can be seen when using the standard GET methods.

For cases where we need extra relationship-oriented metadata, the creation of a dedicated association resource could be an alternative. In this POC, we considered that a complete association resource would be too excessive and opted to abstract this complexity entirely with the custom methods.

5.2.3 virtual-choke-valve

The *virtual-choke-valve* microservice represents the digital version of a device known as Choke Valve, commonly used in oil and gas production facilities. This type of control valve controls the flow of fluids as they are being produced by the well and can act as pressure regulators in the reservoir and flowlines. The standard CRUD actions for the "choke-valve" resource can be seen in table 5.5. The *virtual-choke-valve* microservice also has a dedicated measurement type called *flow*, intended to store sensor readings with the opening percentage of the control valve.

Table 5.5 – *virtual-choke-valve* endpoints

HTTP method	*Path*	*Action*
GET	/v1/choke-valve	Return all Choke Valve resources.
GET	/v1/choke-valve{id}	Returns a Choke Valve resource with given *id*.
POST	/v1/choke-valve	Creates a Choke Valve resource.
PUT	/v1/choke-valve{id}	Updates a Choke Valve resource with given *id*.
DEL	/v1/choke-valve{id}	Deletes a Choke Valve resource with a given *id*.

Source: Author

5.2.4 virtual-anm

The *virtual-anm* microservice represents the Wet Christmas Tree (Árvore de Natal Molhada in Portuguese), which is an assembly of valves that regulate the flow stream of pipes in an oil well. The ANM is also connected to the wellhead and controls access to the tubing as the well begins pumping oil. In the developed API, the "anm" resource also models a series of valves present in the flow path, such as master valve M1 and M2, wing valve W1 and W2, and cross-over valve XO and PXO. When the creation method is invoked, all master and wing valves are set with the value *open* as default, and all the cross-over valves are set with the value *closed*. These values, however, can be changed using the update PUT method. Table 5.6 lists the available actions with the ANM resource.

Table 5.6 – *virtual-anm* endpoints

HTTP method	*Path*	*Action*
GET	/v1/anm	Return all ANM resources.
GET	/v1/anm{id}	Returns a ANM resource with given *id*.
POST	/v1/anm	Creates a ANM resource.
PUT	/v1/anm{id}	Updates a ANM resource with given *id*.
DEL	/v1/anm{id}	Deletes a ANM resource with a given *id*.

Source: Author

5.2.5 virtual-tubing

The *virtual-tubing* microservice models the inner and external tubes of the oil well production column. The tubing is responsible for transporting oil and gas from deep in the well to the surface. Table 5.7 lists the standard CRUD methods for the tubing resource.

It is a standard procedure for engineers to install specific gauges inside the tubing, named permanent downhole gauges (PDG). The PDGs primarily measure pressure and temperature at multiple points of the well, but they can also function as other types of sensors and, therefore, should support other measurement types. Hence, the virtual-tubing microservice also supports the concept of creating and managing PDGs. Table 5.8 shows the standard endpoints for the PDG resource.

A key difference with the PDG resource is that it is associated with a tubing resource. This relationship works almost identically to how the *well-orchestrator* manages

Table 5.7 – *virtual-tubing* endpoints

HTTP method	*Path*	*Action*
GET	/v1/tubing	Return all tubing resources.
GET	/v1/tubing{id}	Returns a tubing resource with given *id*.
POST	/v1/tubing	Creates a tubing resource.
PUT	/v1/tubing{id}	Updates a tubing resource with given *id*.
DEL	/v1/tubing{id}	Deletes a tubing resource with a given *id*.

Source: Author

Table 5.8 – *PDG* endpoints

HTTP method	*Path*	*Action*
GET	/v1/pdg	Return all PDG resources.
GET	/v1/pdg{id}	Returns a PDG resource with given *id*.
POST	/v1/pdg{id}	Creates a PDG resource for tubing with given *id*.
PUT	/v1/pdg{id}	Updates a PDG resource with given *id*.
DEL	/v1/pdg{id}	Deletes a PDG resource with a given *id*.

Source: Author

his associations with other discrete DTs, except on a lower level of the service hierarchy. A tubing UUID must be informed when creating a PDG, and this identifier will be saved. When listing the tubing resources, the API will also return a list of which PDGs are installed inside this tubing.

5.2.6 dashboard-service

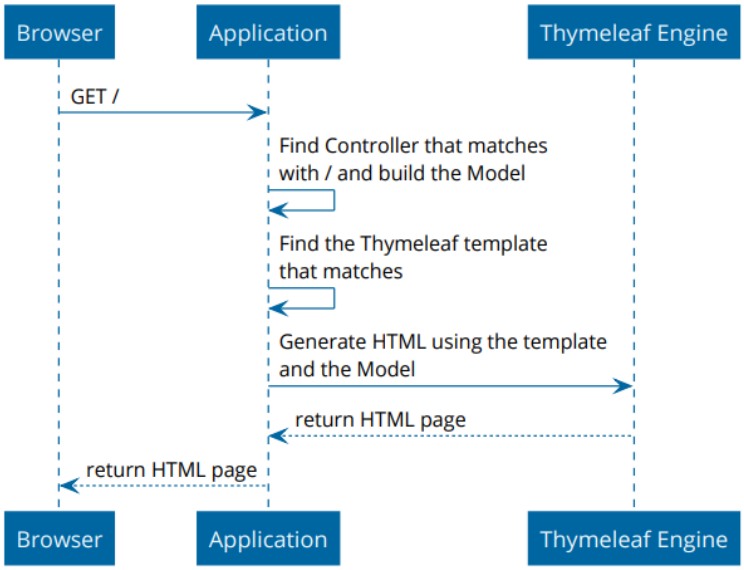
The *dashboard-service* is the application that comprises the service layer in our DT five-dimension framework. It acts as a client for all the APIs described so far, displaying the information in a simple interface to simulate the user experience. By using the *dashboard-service*, a user will be able to:

1. Create, delete, update, and visualize all Digital Twins components.
2. Associate or disassociate a component to a Well.
3. Create and manage PDGs.
4. Visualize temperature, pressure, and other types of sensor readings.
5. Filter and search data in the appropriate dashboards.

For rendering, the dashboard-service uses the Thymeleaf (THYMELEAF, 2022) template engine and Spring Web MVC. For some additional features like buttons, filters, and search bars, the application uses Bootstrap (BOOTSTRAP, 2022) and DataTables plugin for jQuery (SPRYMEDIA, 2022).

Spring Web MVC (Model View Controller) is a well-known design pattern, and Thymeleaf is a server-side Java template engine for web environments. Combined, these two technologies provide the developer with a quick way to prototype web applications. Essentially, the *dashboard-service* will match browser requests to a controller. Controllers will then communicate with the virtual layer microservices, fetch the required data, and build a collection of Java objects called Models. The Models will be integrated with the Thymeleaf templates and generate HTML pages. Finally, the Controller will return the created views, and the web browser will render them to the user. Figure 5.1 illustrates the Thymeleaf MVC interaction.

Figure 5.1 – Thymeleaf server-side rendering



Source: (DEBLAUWE, 2020)

More details on the *dashboard-service* usage will be described in chapter 7.

5.2.7 Sensor data endpoints

Excluding the *dashboard-service*, all microservices described in this chapter can process the sensor data generated by the *data-provider*. As new resources are created and

removed, new MQTT topics are also added and removed from the broker subscription lists, and a simple API to fetch this information was necessary. Thus, all virtual microservices possess a list of sensor data endpoints with the same signature path, as shown in Table 5.9.

Table 5.9 – Sensor data endpoints

HTTP method	*Path*	*Action*
GET	/v1/temperature	Returns all temperature sub-resources.
GET	/v1/temperature{id}	Returns all temperature sub-resources with given parent *id*.
GET	/v1/pressure	Return all pressure sub-resources.
GET	/v1/pressure{id}	Returns all pressure sub-resources with given parent *id*.
GET	/v1/flow	Return all flow sub-resources.
GET	/v1/flow{id}	Returns all flow sub-resources with given parent *id*.
GET	/v1/measure	Return all custom measure sub-resources.
GET	/v1/measure{id}	Returns all custom measure sub-resources with given parent *id*.
GET	/v1/measure{id}/property/{propertyName}	Returns all custom measure sub-resources with given parent *id* and *propertyName*.

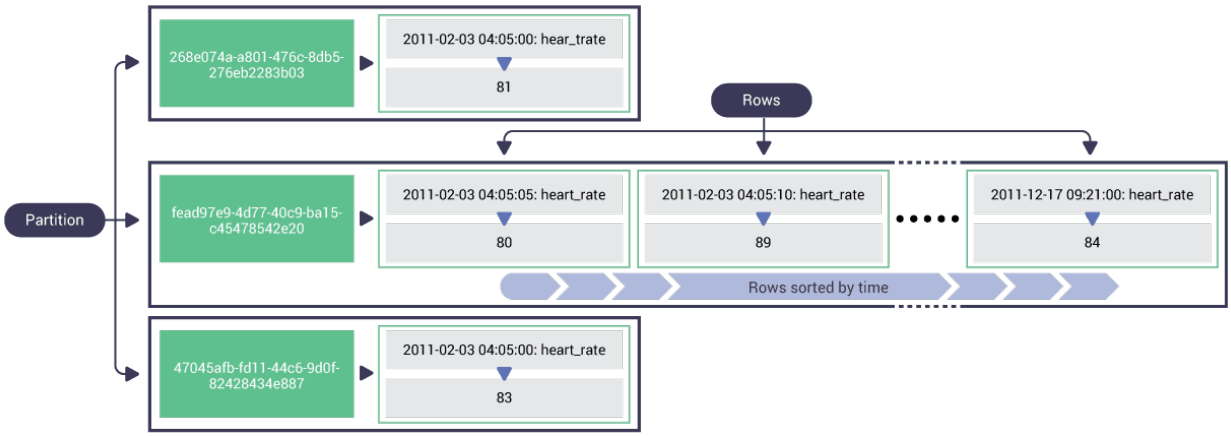
Source: Author

To design a resource relationship that supports these scenarios, we decided to treat the measurements (temperature, pressure, flow, etc.) as sub-resources. More specifically, the readings are considered subcollections — many sub-resources of the same type — of a parent resource such as *anm*, *choke-valve*, or *tubing*. A sub-resource is basically a hybrid between an attribute and what we deem a full-fledged resource. We opted for this design decision because, in theory, a pressure or temperature measurement could be considered a property inherent to one of these component resources. However, treating these fields as an attribute would mean storing them with the parent, and that would be discouraged. Considering the number of requests the temperature and pressure sensors generate, listing all this data with the parent would lead to size and access control problems. For instance, we could not expect to return all pressure measurements related to a *tubing* resource every time a client makes a GET request. The GET operation would have to filter hundreds of values before simply returning the *tubing* resource fields. Moreover, the parent resource would have to be updated multiple times per second as the MQTT messages are con-

sumed, which would require write contention mechanisms and, ultimately, increase the chances of write conflicts or data loss.

As a result, the endpoints in Table 5.9 use the parent resource UUID for data retrieval, and the subcollections are stored in complete separate tables. For performance reasons, the parent UUID is also the partition key for Scylla tables related to measurements, allowing even data distribution across nodes. To differentiate the sub-resources, we use the timestamp field as a clustering key. The clustering key with timestamp values will sort the rows within the partition by time, optimizing filtering. With a partition key and a clustering key defined, each partition can have multiple rows associated with measurements of a single resource. An example of this arrangement can be seen in Figure 5.2, with a timestamp clustering column and a field called *heart_rate*. Additionally, the GET endpoints also support two date-time query parameters for filtering, as shown in Table 5.10.

Figure 5.2 – Scylla partitions and rows example



Source: Scylla University

Table 5.10 – Sensor data endpoints query parameters

Name	*Description*	*Value example*
startDateTime	Specifies the start date for filtering.	2022-07-02T17:26:17.342
endDateTime	Specifies the end date for filtering.	2022-07-02T17:26:19.357

Source: Author

5.2.8 Health check endpoint

With the amount of network traffic and external services involved in a microservice-based solution, sometimes applications can malfunction. Therefore, it is important for the deployment infrastructure to know the health status of its applications periodically. If a service is unable to handle requests, the load should be routed to another service. In our DT solution, all microservices use Spring Boot Actuator (WALLS, 2019).

Spring Boot Actuator is a sub-project of the Spring Boot Framework. It offers health, monitoring, and metrics services to an application, all exposed via HTTP endpoints. In addition, the Actuator can easily return information about the internal state of a microservice, such as configuration properties, logging levels, memory consumption, HTTP traces, and much more. Table 5.11 shows a few of the exposed actuator endpoints. The Actuator works using health indicators, gathering data from all external systems such as databases and message brokers, and condenses the information into a single report. In our DT microservices, most Actuator endpoints are disabled by default for security reasons, but can be enabled by changing the application properties file. To demonstrate, the *virtual-choke-valve* microservice has all Actuator endpoints made available.

Table 5.11 – Actuator endpoints

Path	*Description*
/actuator/health	Returns the aggregate health of the application and (possibly) the health of external dependent applications.
/actuator/env	Produces a report of all property sources and their properties available to the Spring application.
/actuator/metrics	Returns a list of all metrics categories.
/actuator/httptrace	Produces a trace of the most recent 100 requests.

Source: Author. Descriptions by (WALLS, 2019)

6 MONITORING

A Digital Twin solution will always consist of dozens or even hundreds of microservices, and software developers and operations teams should observe the system's health periodically. Unfortunately, tracing and troubleshooting distributed systems is notoriously hard, requiring more effort than the former monolithic solutions. If an application is misbehaving or degrading, developers must be alerted and take action preemptively before any customers are impacted. In the microservice architecture, software malfunctions can also easily create a big chain of events, propagating issues over multiple services and hiding the real root cause of a failure. With such an ever-changing environment, monitoring data should be persisted, or else they will not be available after service restarts or scaling. Metrics like throughput, latency, resource utilization, and log files should all be aggregated in a single place for easier access (BRUCE; PEREIRA, 2020).

For these reasons, we built a robust monitoring stack for the Oil Well Digital Twin. We make use of the Elastic Stack (ELK), which is a collection of open-source products to aid us with log aggregation, distributed tracing, exception tracking, and metrics collection. The following section describes the responsibilities of each component of the ELK. Then, section 6.2 goes over how the ELK is implemented and used by the Oil Well DT.

6.1 Elastic Stack

The ELK is a famous logging infrastructure, allowing users to search, analyze, and visualize log data in real-time. It consists of Elasticsearch, Logstash, and Kibana. Subsequently, a family of data shippers called Beats was also added to the stack.

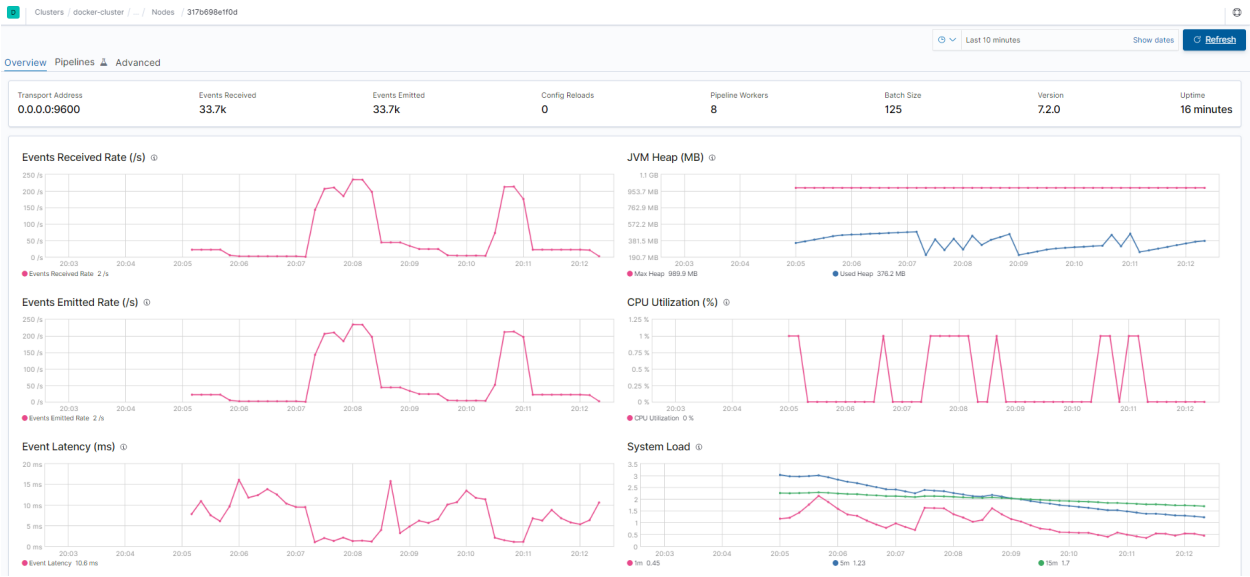
6.1.1 Elasticsearch

Elasticsearch (ELASTICSEARCH, 2022) is a search and analytics engine that stores data in a centralized, NoSQL database. Elasticsearch has many different use-cases, but its most common is of a scalable, easy-to-use logging server. It excels in indexing text and semi-structured data, performing efficient JSON-based search and aggregation operations.

6.1.2 Logstash

Logstash (LOGSTASH, 2022) is a log pipeline that aggregates data from multiple sources, parses and enriches it, and then writes them to Elasticsearch. It has a vast number of plugins to configure data collection and transformation. Logstash also allows the developer to route data to different outputs, opening up a vast possibility of use-cases. Lastly, Logstash provides pipeline monitoring features to observe the active node's performance and availability, as seen in Figure 6.1.

Figure 6.1 – Logstash node monitoring in Kibana

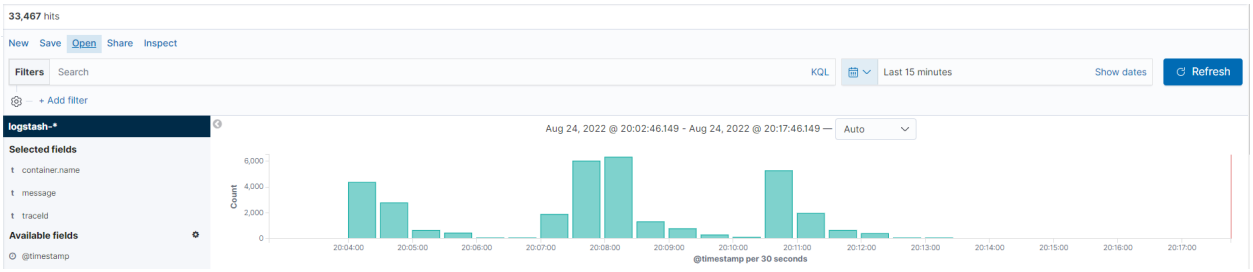


Source: Author

6.1.3 Kibana

Kibana (KIBANA, 2022) is a highly customizable user interface for visualizing Elasticsearch data. It can also work as a search and analytics tool, allowing users to create a variety of graphs, tables, and dashboards using the Elasticsearch indices. Kibana is essential to understanding how requests flow through the deployed microservices, helping developers detect anomalies, troubleshoot issues, and query huge loads of log data. Figure 6.2 shows a small dashboard created from the Oil Well DT logs.

Figure 6.2 – Simple Kibana dashboard in discover mode

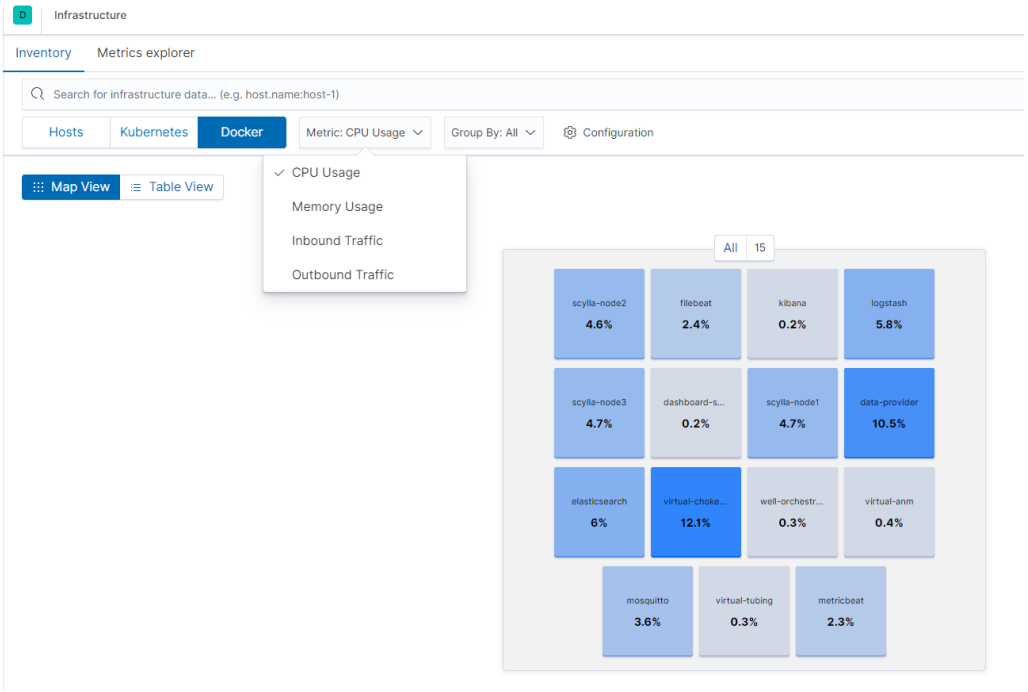


Source: Author

6.1.4 Beats

Beats (BEATS, 2022) is a family of single-purpose, open-source data shippers. They are responsible for sending operational data to Logstash or Elasticsearch, where that data can be further enhanced. In our DT solution, we use Filebeat (FILEBEAT, 2022) and Metricbeat (METRICBEAT, 2022). Filebeat is a lightweight shipper for forwarding and centralizing log data, whereas Metricbeat ships system and service statistics. Figures 6.3 illustrates some of the data collected periodically from the Oil Well DT applications and backing services.

Figure 6.3 – Metrics explorer in Kibana

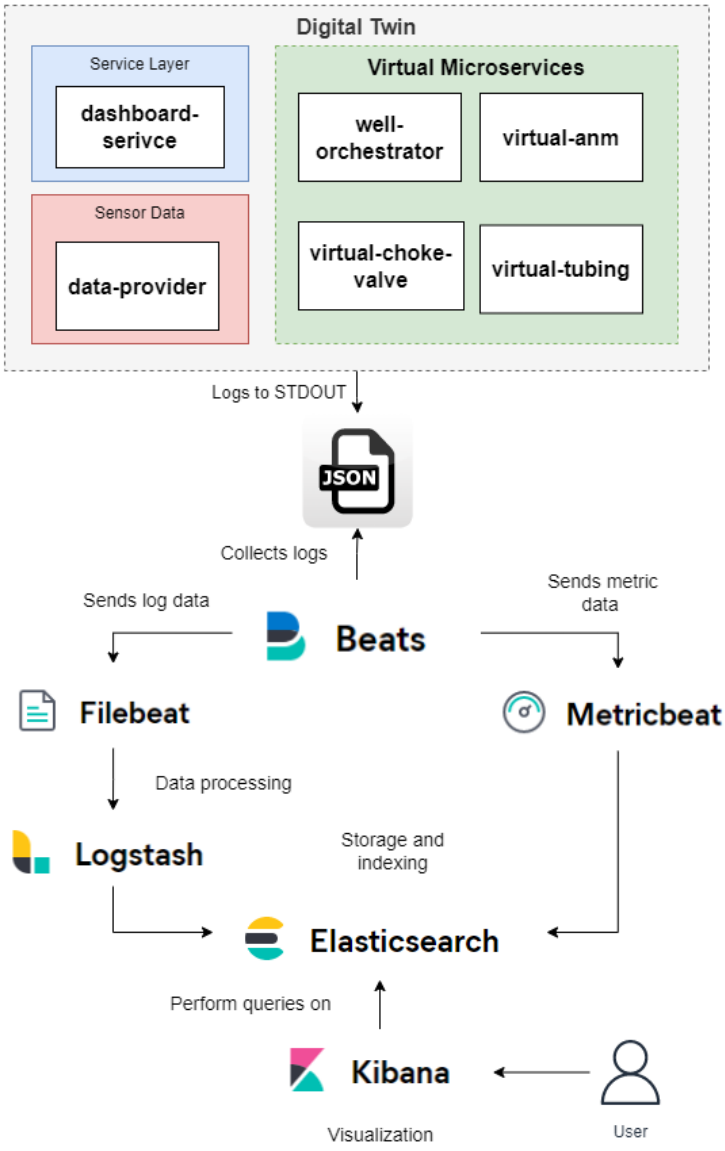


Source: Author

6.2 Monitoring stack implementation

Figure 6 illustrates how the ELK tools interact with each other and the Digital Twin.

Figure 6.4 – Interaction between the Oil Well DT monitoring services

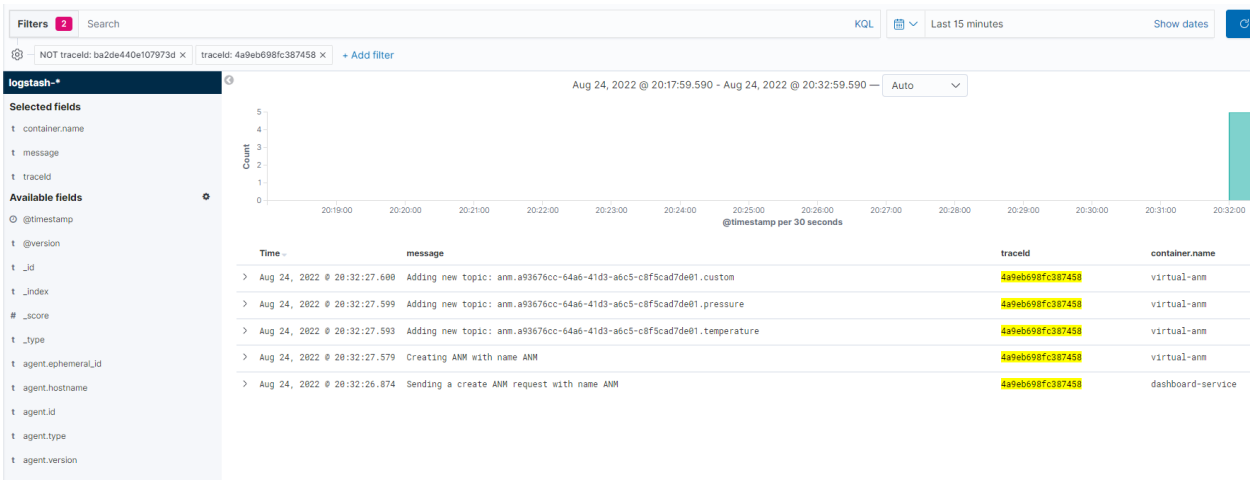


Source: Author

To begin with, all DT microservices implement the same logging library, which in this case, is Logback (LOGBACK, 2022) and SLF4J (SLF4J, 2022). As advised in the Twelve-Factor methodology (WIGGINS, 2017), all application logs are written to *stdout*, and the logging infrastructure is the one responsible for handling the output. Before that, however, logs are enhanced with tracing details. With distributed tracing, every request is

assigned with a unique ID (often called *correlationId* or *traceId*), which flows from one microservice to the other until the task is completed. Combined with log aggregation, this technique allows the developers and operations team to debug and understand interactions between applications much quicker. To automatically add tracing information to our logs, all DT microservices use the Spring Cloud Sleuth framework (SLEUTH, 2022). Spring Cloud Sleuth autoconfigures tracing information, injecting a *traceId* whenever a new service call is made. The tool also manages ingress and egress points from Spring applications, ensuring that the *traceId* is properly propagated. Figure 6.5 shows a set of logs in Kibana filtered by the *traceId*.

Figure 6.5 – DT requests filtered by traceId



Source: Author

When any of the DT applications start running in their respective containers, they begin to write our enhanced logs in the Docker file-system. After that, we can use the Filebeat auto-discovery feature to track running containers and collect data from the log files. Considering that we want our log entries to be human-readable but also easily parsable by a machine, the log messages are decoded in a JSON format. Afterward, once Filebeat starts collecting log events, they are sent to Logstash for processing. Logstash has three types of plugins: input plugins, which are used to consume data from a specified source; optional filter plugins, which can modify the incoming data; and output plugins which write data to a specified destination. In this POC, we do not do any heavy transformations with the filter plugin and basically just route the data to an output. All this log data is ultimately stored in Elasticsearch, queried by Kibana, and then visualized by the end user.

At the same time, as the log aggregation process is happening, Metricbeat is gathering a collection of system metrics from the containers. Metricbeat works identically

to Filebeat, though it re-routes the data directly to Elasticsearch. Those metrics are extremely important for operations teams to monitor the health of the applications and backing services. Using Kibana, developers can visualize CPU, memory, disk usage, request number, and latency in real-time. Kibana and Metricbeat also enable thresholds to be set dynamically as the metrics are collected, sending alerts to the developers when any of the configured limits are surpassed. Additionally, this monitoring stack could also be used as a performance test tool, highlighting system bottlenecks and vulnerabilities. Chapter 7 describes how the monitoring stack is configured in the Docker containers and how to set Kibana index patterns.

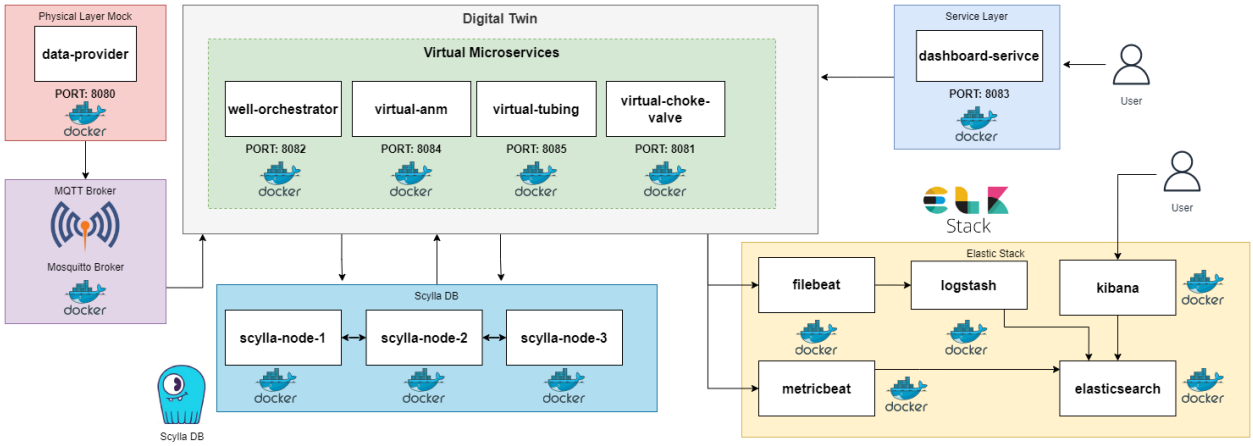
7 SYSTEM SETUP AND USAGE

Our Oil Well Digital Twin solution uses Docker as a deployment tool. Working with microservices and multi-container applications can become hard to manage with only scripts and docker commands, so we use Docker Compose to simplify this process. With Docker Compose, we can describe the entire DT solution in a single declarative configuration file, deploy it, and then manage its entire lifecycle. Furthermore, the onboarding of new developers is significantly reduced, as all backing services, dependencies, and configurations are centralized in a Docker Compose YAML file (POULTON, 2020).

7.1 Docker set up

The *docker-compose* file for the DT services is located in the root folder of the *well-orchestrator* microservice. In the YAML file, the top-level *service* key defines the ScyllaDB nodes, the Mosquitto instance, the ELK services, and the virtual DT applications, generating containers with the same names. The *ports* key maps the network traffic from a container port to the declared host's port, and all the DT services use the same single-host Docker network to connect containers with each other. Figure 7.1 shows all the Docker containers of the Oil Well DT and their respective ports. Lastly, the *volumes* key is used to mount custom configurations directly into the DT containers, which are also declared in the *well-orchestrator* root folder. For service discovery, the containers' IPs are injected in the containers as environment variables.

Figure 7.1 – Oil Well Digital Twin containers overview



Source: Author

7.1.1 Repositories

All the mentioned microservices are versioned in the following Github repository:

- https://github.com/guscorrea

To run the solution, run the *docker-compose up -d –build* command in the *well-orchestrator* root folder. If the Docker images are not found locally in the host machine, they are also uploaded to the following DockerHub repository:

- https://hub.docker.com/u/guscorrea

Make sure the backing services are up, and that the ScyllaDB datacenter is working as shown in Figure 7.2.

Figure 7.2 – ScyllaDB data-center

```
Datacenter: DC1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns    Host ID                               Rack
UN  172.20.0.4  101.29 KB  256          ?       b4ac1af7-32b1-4932-b1c0-f23720582a9b  Rack1
UN  172.20.0.5  215.8 KB   256          ?       e67918c0-bb8e-44b3-b1bf-4adfac71460b  Rack1
UN  172.20.0.6  193.77 KB  256          ?       e73ea367-194c-4411-b8fc-29658b2ae189  Rack1
```

Source: Author

7.2 Making API requests with the dashboard-service

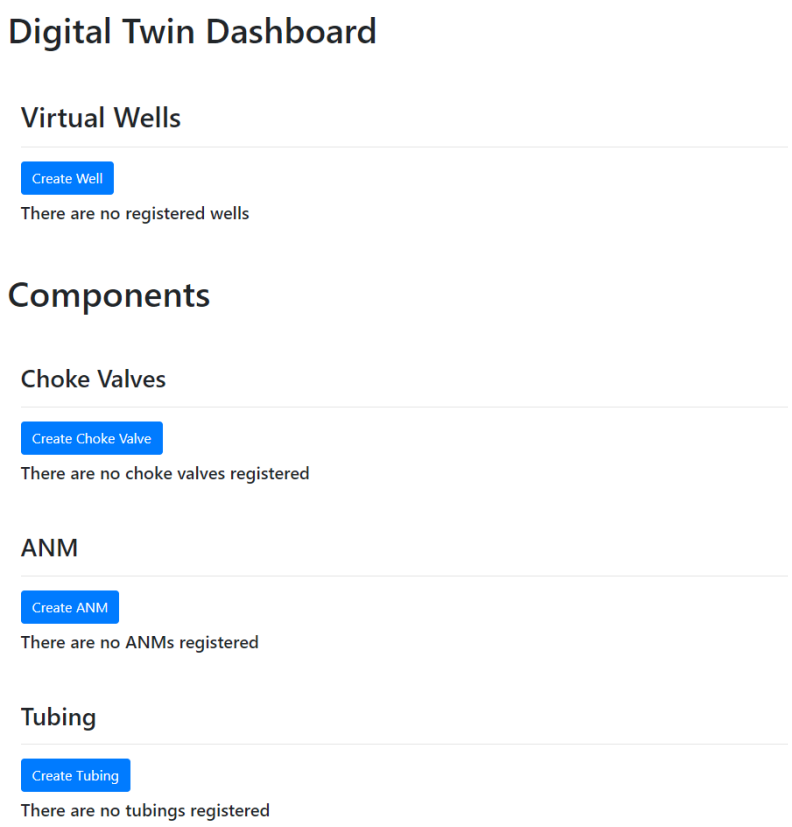
The easiest way to consume and test the Digital Twin APIs is through the *dashboard-service*, as it provides the user with a visual interface and doesn't require any external tools to perform HTTP requests. After starting up, the *dashboard-service* container exposes the port number 8083 and is accessible through the context path */dashboard-service/*.

7.2.1 Managing resources and creating associations

On the index page, the *dashboard-service* will perform GET requests to retrieve all resources from the DT. At the first start-up, wells or components will not be found, as shown in Figure 7.3. The users can then proceed to create Well resources with the *Create Well* button, which will enable the *Update*, *Delete*, and *Details* actions, as shown in Figure

7.4. When performing those actions, the dashboard-service will make API requests to the endpoints described in Table 5.4.

Figure 7.3 – *dashboard-service* index page without any resources



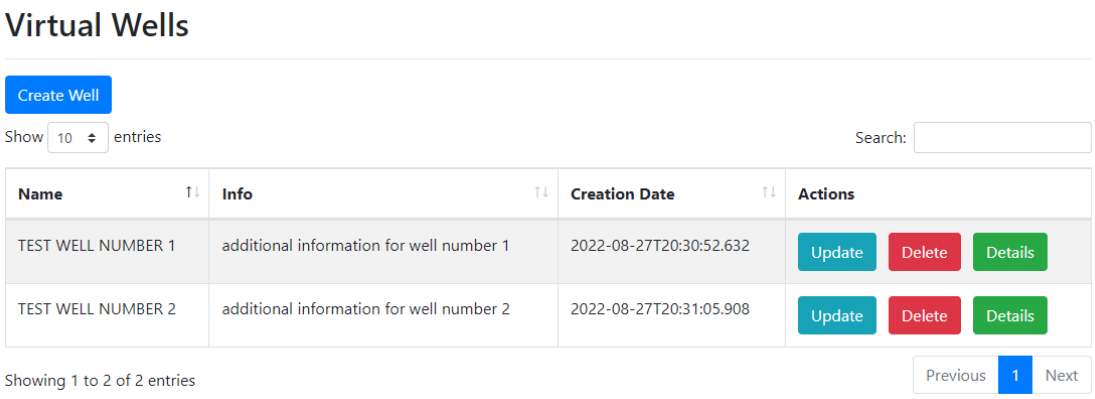
Source: Author

Below the Virtual Wells tables on the index page, users can also use the other *Create* buttons to create choke-valve, anm, and tubing resources, making service calls to the endpoints described in Tables 5.5, 5.6, and 5.7, respectively. The *Update* and *Delete* actions will be available at resource creation (as shown in Figure 7.5), and the ANM flow valves (M1, M2, W1, W2, XO, and PXO) and Tubing ICV status can be either opened or closed using the *Update* button. Figure 7.6 shows an anm and tubing resource with their respective flow valves opened and closed.

To associate a component to a specific Well, the user should use the *Details* button, illustrated in Figure 7.4, and then click the *Add component* button, as shown in Figure 7.7. These interactions will trigger requests to the well-orchestrator *add-component* endpoint described in Table 5.4, and after the association is done, it can also be undone with the *Remove* button, creating a service call to the *remove-component* endpoint. An association requires a virtual well and at least a single component to be created beforehand.

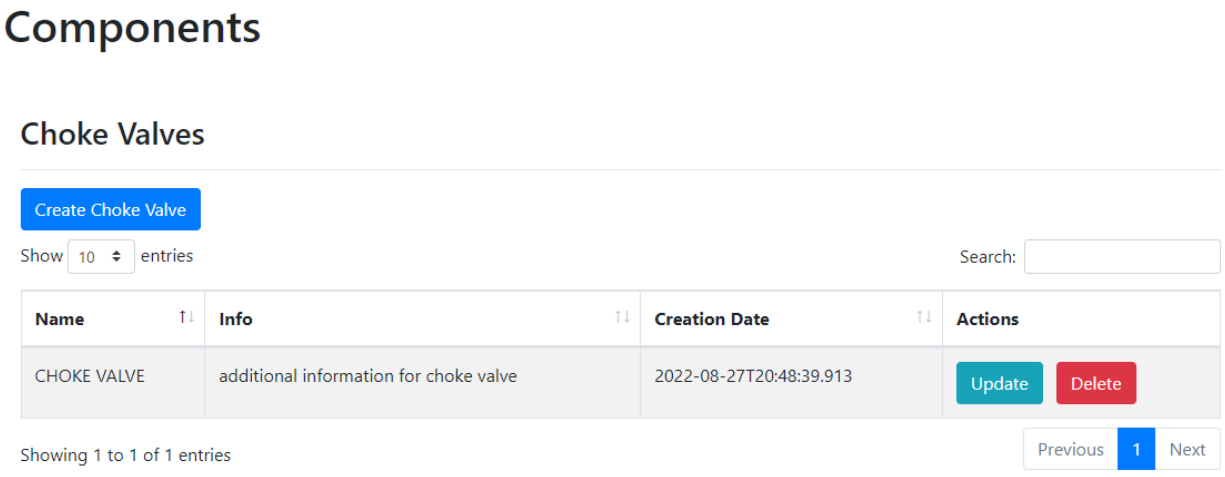
The tubing microservice also has a dedicated feature mentioned in chapter 5,

Figure 7.4 – *dashboard-service* Virtual Wells table with two resources



Source: Author

Figure 7.5 – *dashboard-service* page with two choke-valve resources



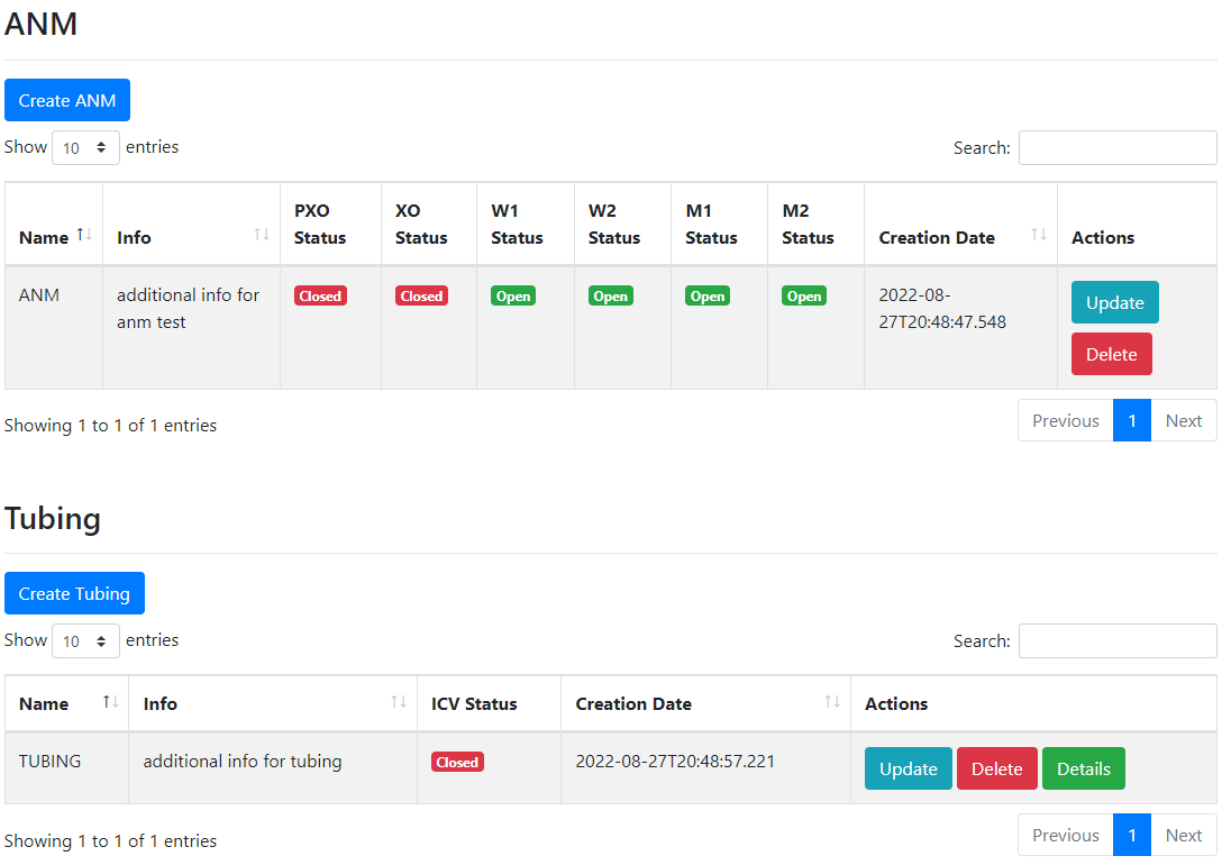
Source: Author

which is the possibility to manage PDGs gauges. Clicking on the *Details* button shown in Figure 7.6 and then clicking on the *Add PDG* button illustrated in Figure 7.8, a user will be able to perform all CRUD operations with PDG resources. In short, these actions will make service calls to the endpoints listed in Table 5.8.

7.3 Generating sensor data with the data-provider

A significant part of the DT solution is the generation and handling of mock IoT sensor data. The *dashboard-service* allows the user to visualize and filter this information using the *Check Component Readings* button, which can be seen in the component list of Figure 7.7. In the case of sensor data related to a PDG, the user can use the

Figure 7.6 – *dashboard-service* page with anm and tubing resources



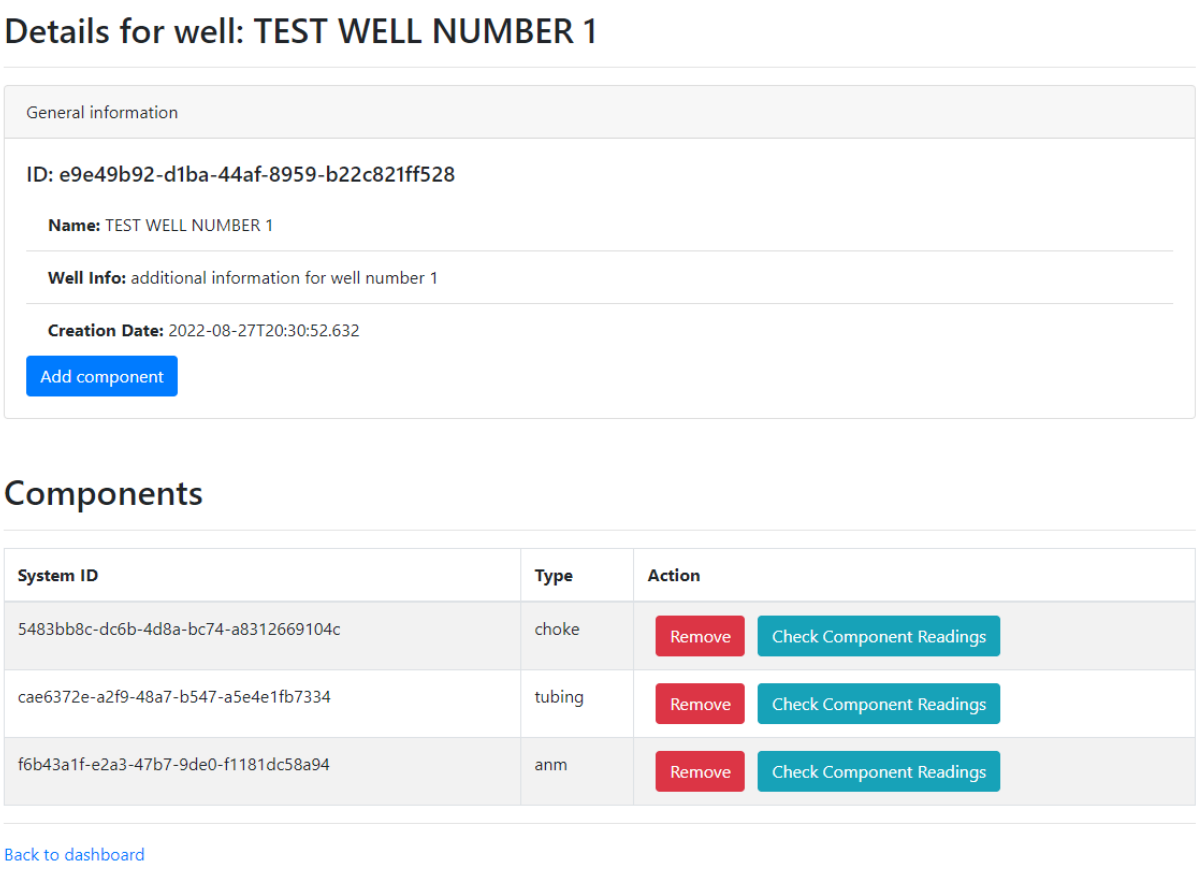
Source: Author

PDG Readings button seen in Figure 7.8. To generate the mock data, however, the user should make HTTP requests to the *data-provider* endpoints described in Table 5.2. The *dashboard-service* does not provide any type of visual interface with the *data-provider*, so it's recommended to use an API tool like Postman [1] or Insomnia [2] to perform the requests. The *data-provider* port is exposed on number 8080, and the request body should use the parameters described in Table 5.3. A *data-provider* payload example is shown in Figure 7.9. As soon as the payloads are generated, sent to the Mosquitto broker, and consumed by the virtual microservices, the user can visualize the sensor data using the aforementioned buttons in the details page. An example of visualizing mock sensor data with the *dashboard-service* is illustrated in Figure 7.10 and 7.11.

To conclude, all of the operations mentioned in this chapter will generate application logs, which can be seen in the Kibana tool exposed on port 5601. Yet, to use Kibana properly for the first time, the user should create a Kibana index pattern so data can be

[1] Postman: https://www.postman.com
[2] Insomnia: https://insomnia.rest

Figure 7.7 – Virtual Well details page



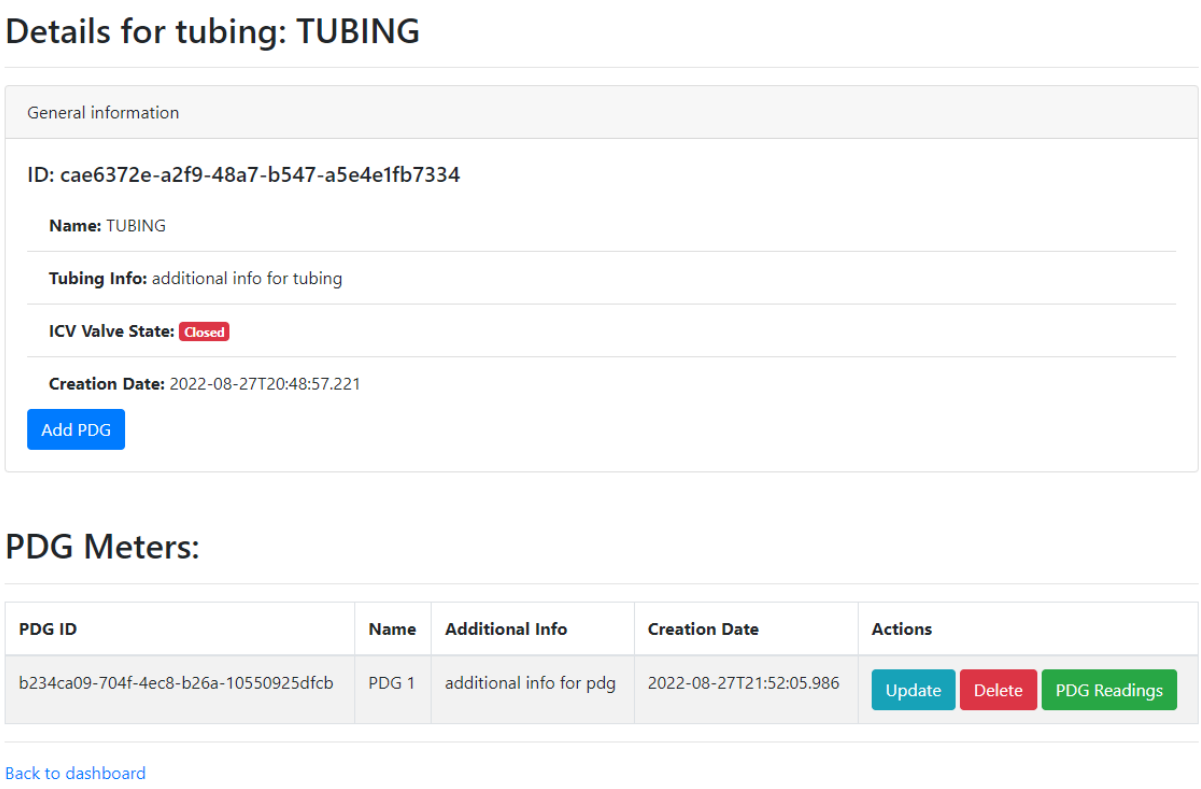
Source: Author

retrieved from Elasticsearch. Figure 7.12 shows an index pattern created by Logstash. To filter log events, the index pattern should use the timestamp field. Then, all of the generated DT logs will be available when clicking the Discover icon. It is also worth mentioning that all operations described in this chapter can be performed without the *dashboard-service*. The only requirement is a tool like Postman or Insomnia to make requests, using the Tables listed in chapter 5 as reference.

7.4 API Documentation

A more detailed description of all the APIs defined in this work is also available in the Swagger documentation (SMARTBEAR, 2022). Swagger is a set of open-source tools built around the OpenAPI Specification that can help you design, build, document, and consume REST APIs. As every microservice uses the SpringDoc Open API library, a user can automatically generate documentation for all the mentioned endpoints, listing

Figure 7.8 – Virtual Tubing details page



Source: Author

Figure 7.9 – *data-provider* payload example

```
{
    "componentType": "choke",
    "componentId": "f2947b1d-0c7c-4363-8a11-288ab35e3d45",
    "measurementType": "pressure",
    "number": 1,
    "rate": 10
}
```

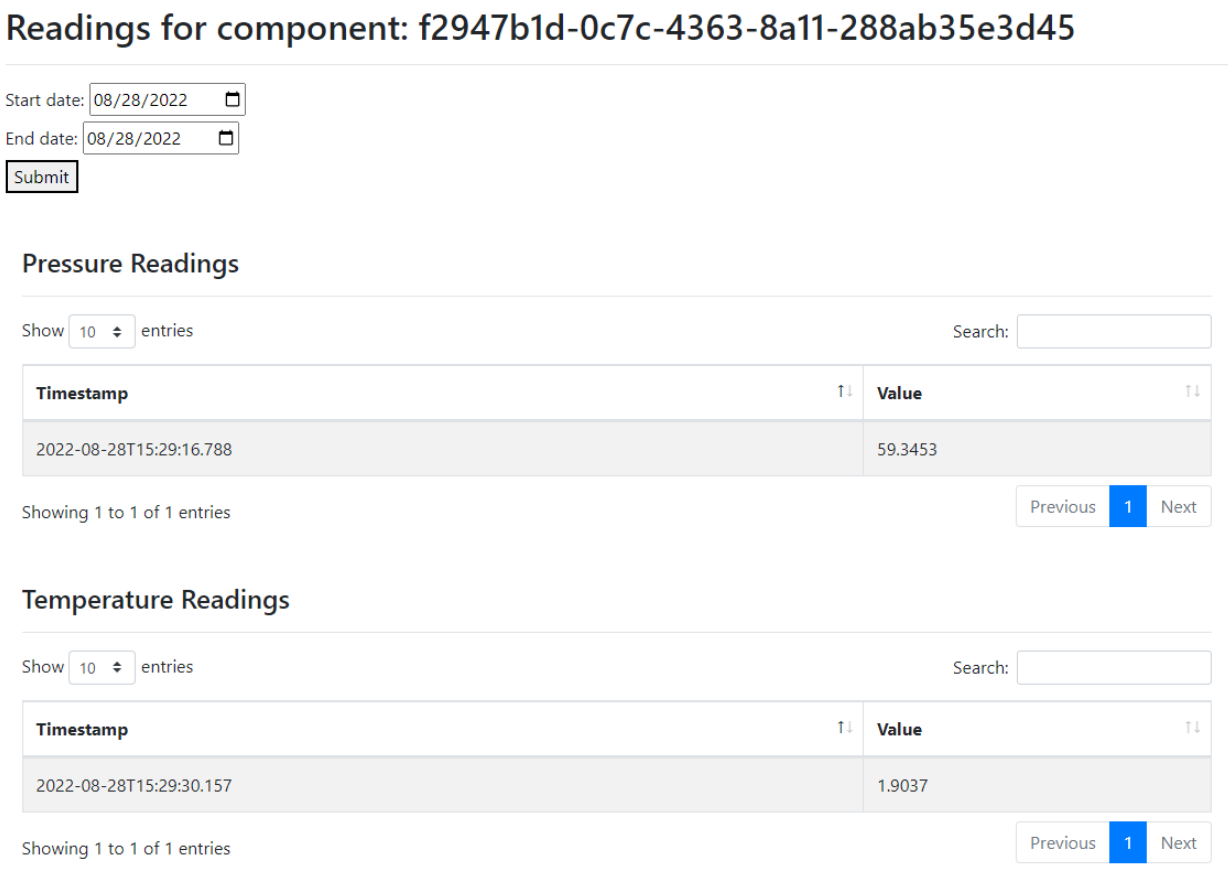
Source: Author

HTTP operations and all request and response payloads with examples. By making a GET request to the */v3/api-docs* endpoint in any microservice, a YAML documentation of the service will be returned as a response. This can then be posted in a Swagger editor and visualized, as shown in Figures 7.13, 7.14, and 7.15.

Links for the Swagger editor:

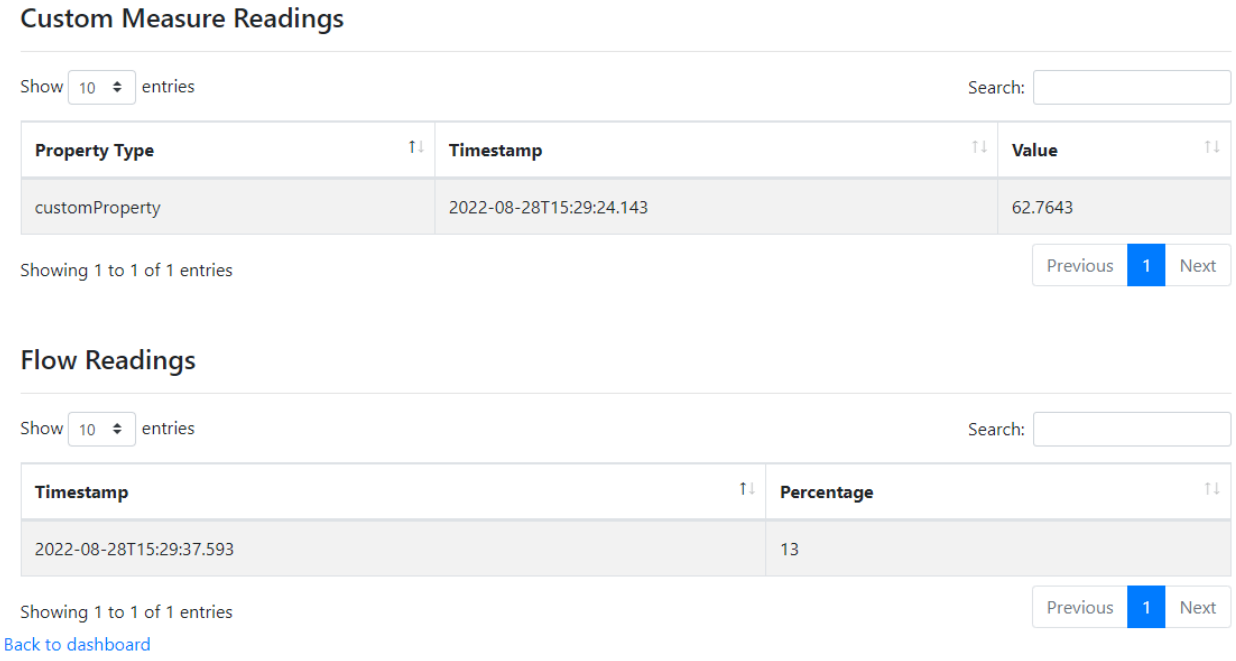
- https://editor.swagger.io/
- https://editor-next.swagger.io/

Figure 7.10 – *dashboard-service* temperature and pressure readings visualization



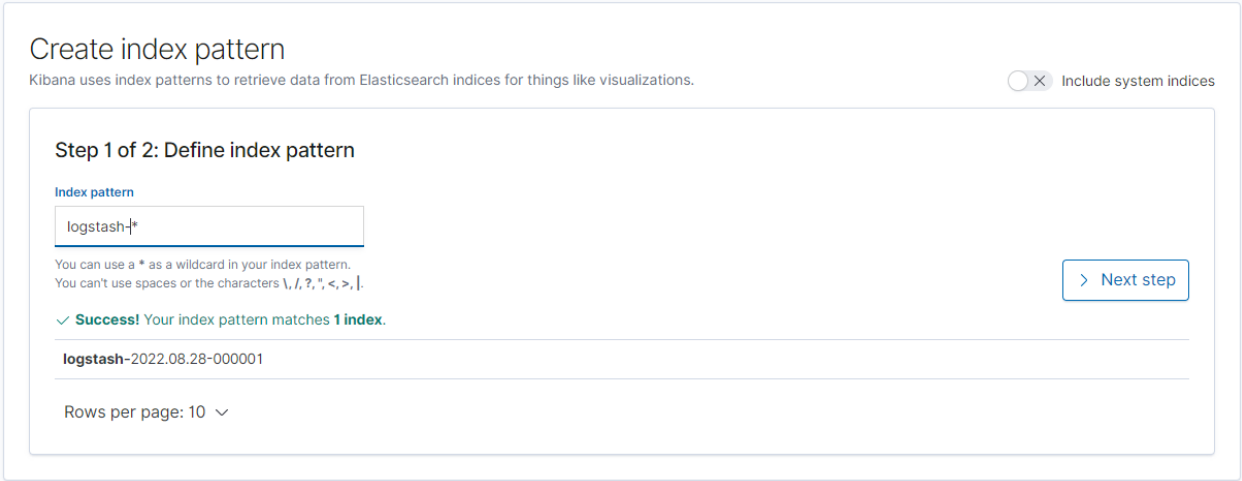
Source: Author

Figure 7.11 – *dashboard-service* custom and flow readings visualization



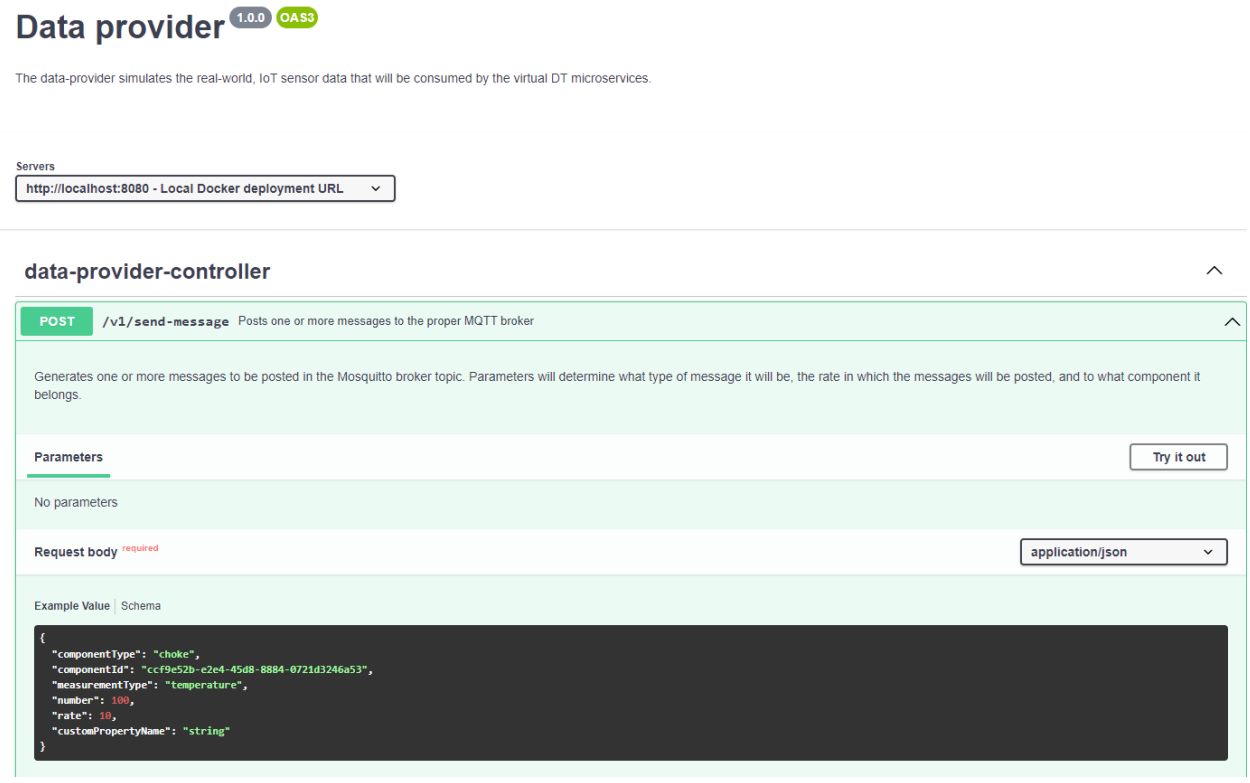
Source: Author

Figure 7.12 – Kibana index pattern using Logstash



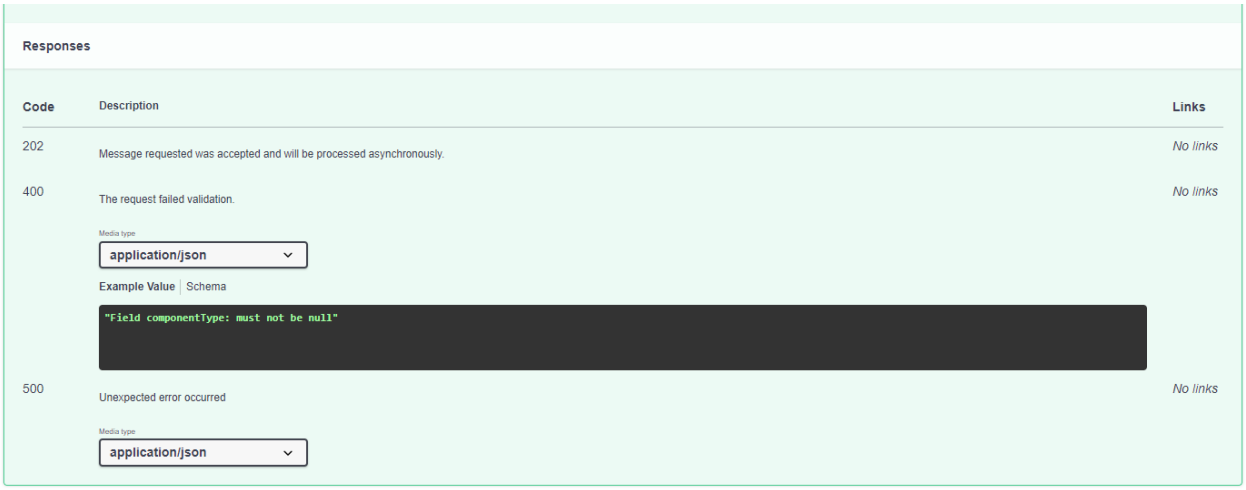
Source: Author

Figure 7.13 – *data-provider* endpoint documentation in Swagger



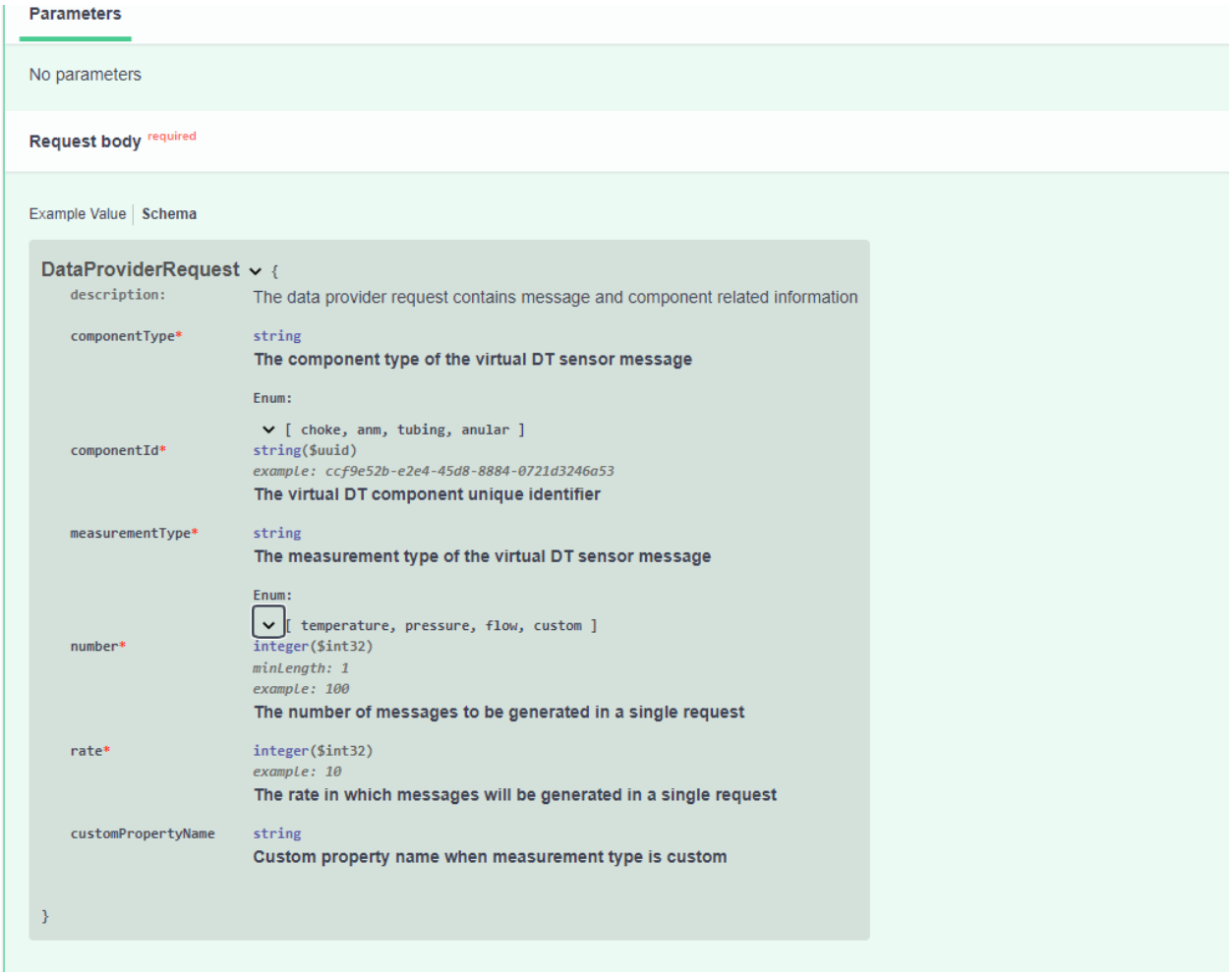
Source: Author

Figure 7.14 – *data-provider* endpoint HTTP responses documentation in Swagger



Source: Author

Figure 7.15 – *data-provider* request payload in Swagger documentation



Source: Author

8 CONCLUSION

Digital Twin (DT) is one of the most promising enabling technologies for realizing Industry 4.0. And yet, different interpretations of a DT exist without any architectural template, leading to implementations driven by specific use cases (STEINDL et al., 2020). Nevertheless, academic researchers concur that cloud solutions and microservice patterns are vital to achieving the non-functional requirements of a Digital Twin, such as reliability, scalability, and maintainability. (KASTNER; STEINDL, 2021).

By working on the DT solution for the flow assurance process of an oil well, we can demonstrate the importance of breaking down the functional requirements of the Oil Well into small and scalable microservices. The oil industry context is too complex, and the landscape is constantly changing to be approached in a monolithic manner. The microservice architectural pattern can be compared to developing discrete Digital Twins, and hierarchical associations in the microservice and API level can be used to model the relationships between components.

Following that principle, we presented and developed a high-level Digital Twin architecture for the Flow Assurance process of an Oil Well, working alongside a complete monitoring stack. We modeled three main components used in the Well plant to the virtual world (*anm*, *choke-valve*, and *tubing*) in the form of microservices, possessing several resource-management APIs. We simulated the IoT devices of an Oil Well with the *data-provider* microservice, publishing mock messages to an MQTT broker, which are then consumed by the virtual layer of the DT. We also covered basic communication and data fundamentals and implemented a simple UI in the service layer to consume the APIs. Naturally, we identified several pitfalls and improvements that could be made to the overall architecture along the way, especially regarding security, scalability, and continuous deployments.

A crucial missing piece in this current DT solution is security. Digital Twins should possess high levels of safety and security, ensuring integrity, confidentiality, and traceability. Data privacy is of extreme importance in DTs, even more so if a multitenancy architecture is employed. Several frameworks can provide security abstractions, but the Spring security libraries should be easily integrated into the current code.

Another key area that could be improved is the deployment process. The DT solution should be moved to a cloud platform such as AWS, Google Cloud Platform (GCP), or Microsoft Azure. A cloud-native platform could help with many of these tasks: secu-

rity and compliance, multitenancy, change control assurance, and control of the deployment process. This change would also allow the developer to perform scalability tests, as Docker itself is not sufficient to achieve good results with dynamic scaling. To encompass these changes, modifications related to service discovery will be required, or the deployment infrastructure itself should take care of routing.

Lastly, the current architecture doesn't demonstrate any calculations or operations related to the DT components, as most APIs are more focused on handling the sensor data. Functionalities related to calculus, alert services, and simulations could be added to test how well this arrangement does. Operations that involve more than one component could also have a dedicated microservice, and the PDG resource could have its own application as well, further testing how the hierarchy would behave. In short, the current DT architecture was designed with extensibility in mind, so many of these changes should be considered low-effort. On a functional level, the current design is not the only possible arrangement. Oil Wells can have very similar sets of equipment configurations, so another alternative would be to organize the microservices as a family of components instead of a type-based approach. And yet, regardless of our design choices, the key requirements of a DT solution remain the same.

Digital Twin APIs should be extensible and flexible, possessing a dynamic messaging format that can keep pace with changes and evolve over time. Event-driven architectures using message brokers and asynchronous communication are favored, improving service availability and ensuring loose coupling. Applications should be stateless, and backing services should be treated as attached resources and avoid being technology-dependent. Deployment technologies and cloud-native platforms are vital: they should offer resilience and redundancy capabilities in case of unexpected failures and handle abstractions such as environment configurations and service discovery. To make troubleshooting and analytics quicker and easier, a robust monitoring stack that offers services like log aggregation, visualization, and performance metrics is also essential.

It is clear that a successful Digital Twin comes from a team effort between developers and operations. As the oil industry begins to shift toward digitalization, we believe more DT success cases will arise in the market, and architectural design standards will become more defined. Ultimately, our proof of concept reveals how dependent a Digital Twin can be on cutting-edge cloud technologies and microservice patterns to succeed.

REFERENCES

ADAMENKO, D.; KUNNEN, S.; NAGARAJAH, N. Comparative analysis of platforms for designing a digital twin. John Wiley & Sons ltd., 2020.

APACHE-FOUNDATION. **Apache Maven**. 2022. Last accessed 02 August 2022. Disponível em: <https://maven.apache.org/index.html>.

BARRICELLI, B.; CASIRAGHI, E.; FOGLI, D. A survey on digital twin: Definitions, characteristics, applications, and design implications. IEEE Access, vol. 7, pp. 167653-167671, 2019.

BEATS. **Beats**. 2022. Last accessed 21 August 2022. Disponível em: <https://www.elastic.co/pt/beats>.

BOOTSTRAP. **Bootstrap**. 2022. Last accessed 29 July 2022. Disponível em: <https://getbootstrap.com/>.

BRUCE, M.; PEREIRA, P. A. **Microservices in Action**. 1. ed. [S.l.]: MANNING, 2020.

CARNELL, J. **Spring Microservices in Action**. 1. ed. [S.l.]: MANNING, 2017.

CHANDRAKANT, K. **Why Choose Spring as Your Java Framework?** 2022. Last accessed 02 August 2022. Disponível em: <https://www.baeldung.com/spring-why-to-choose>.

DAVIS, C. **Cloud Native Patterns**. 1. ed. [S.l.]: MANNING, 2019.

DEBLAUWE, W. **Taming Thymeleaf**. 1. ed. [S.l.]: Lulu, 2020.

ECLIPSE. **Eclipse Paho Java Client**. 2022. Last accessed 26 July 2022. Disponível em: <https://www.eclipse.org/paho/index.php?page=clients/java/index.php>.

ELASTICSEARCH. **Elasticsearch**. 2022. Last accessed 21 August 2022. Disponível em: <www.elastic.co/products/elasticsearch>.

FILEBEAT. **Filebeat**. 2022. Last accessed 21 August 2022. Disponível em: <https://www.elastic.co/beats/filebeat>.

FISHER, M. et al. **Spring Integration Reference Guide**. 2022. Last accessed 26 July 2022. Disponível em: <https://docs.spring.io/spring-integration/reference/html/index.html>.

FREEMAN, J. **What is JSON? A better format for data exchange**. 2022. Last accessed 27 July 2022. Disponível em: <https://www.infoworld.com/article/3222851/what-is-json-a-better-format-for-data-exchange.html>.

GEEWAX, J. **API Design Patterns**. 1. ed. [S.l.]: MANNING, 2021.

GRIEVES, M. Digital twin: Manufacturing excellence through virtual factory replication. White paper, 2014.

HOHPE, G.; WOOLF, B. **Enterprise Integration Patterns**. 2022. Last accessed 24 July 2022. Disponível em: <https://www.enterpriseintegrationpatterns.com/PublishSubscribeChannel.html>.

JONES, D. et al. Characterising the digital twin: A systematic literature review. CIRP Journal of Manufacturing Science and Technology 29 (2020) 36–52, 2020.

KASTNER, W.; STEINDL, G. Semantic microservice framework for digital twins. Appl. Sci. 2021, 11, 5633, 2021.

KIBANA. **Kibana**. 2022. Last accessed 21 August 2022. Disponível em: <www.elastic.co/products/kibana>.

KNEBEL, F.; WICKBOLDT, J. An open digital twin framework based on microservices in the cloud. UFRGS, 2020.

LIGHT, R. A. Mosquitto: server and client implementation of the mqtt protocol. The Journal of Open Source Software, vol. 2, no. 13, DOI: 10.21105/joss.00265, 2017.

LOGBACK. **Logback**. 2022. Last accessed 21 August 2022. Disponível em: <https://logback.qos.ch>.

LOGSTASH. **Logstash**. 2022. Last accessed 21 August 2022. Disponível em: <www.elastic.co/products/logstash>.

MALAKUTI, S. et al. Digital twins for industrial applications. definition, business values, design aspects, standards and use cases. An Industrial Internet Consortium White Paper, 2020.

METRICBEAT. **Metricbeat**. 2022. Last accessed 21 August 2022. Disponível em: <https://www.elastic.co/beats/metricbeat>.

NICKOLOFF, J. **Docker in Action**. 1. ed. [S.l.]: MANNING, 2020.

ORACLE. **Java**. 2022. Last accessed 04 August 2022. Disponível em: <https://www.java.com/pt-BR/download/help/whatis_java.html>.

PENG, S.; ZHANG, Z.; WU, C. Geological cloud platform based on micro service architecture. Atlantis Press, 2017.

PERNO, M.; HVAM, L.; HAUG, A. Implementation of digital twins in the process industry: A systematic literature review of enablers and barriers. Elsevier B.V. $\mathrm{CC}_B Y_4.0, 2021$.

POULTON, N. **Docker Deep Dive**. 1. ed. [S.l.]: Independently published, 2020.

PRESTON-WERNER, T. **Semantic Versioning**. 2022. Last accessed 27 July 2022. Disponível em: <https://semver.org/#semantic-versioning-200>.

RASHEED, A.; SAN, O.; KVAMSDAL, T. Digital twin: Values, challenges and enablers from a modeling perspective. IEEE Access, 2020.

RICHARDSON, C. **Microservices Patterns**. 1. ed. [S.l.]: MANNING, 2019.

RICHARDSON, L. et al. **Richardson Maturity Model**. 2022. Last accessed 27 July 2022. Disponível em: <https://martinfowler.com/articles/richardsonMaturityModel.html>.

SCYLLADB. **ScyllaDB**. 2022. Last accessed 20 August 2022. Disponível em: <https://www.scylladb.com>.

SLEUTH, S. C. **Spring Cloud Sleuth**. 2022. Last accessed 21 August 2022. Disponível em: <https://spring.io/projects/spring-cloud-sleuth>.

SLF4J. **Slf4j**. 2022. Last accessed 21 August 2022. Disponível em: <https://www.slf4j.org>.

SMARTBEAR. **Swagger**. 2022. Last accessed 21 September 2022. Disponível em: <https://swagger.io/>.

SPRYMEDIA. **DataTables | Table plug-in for jQuery**. 2022. Last accessed 29 July 2022. Disponível em: <https://datatables.net/>.

STEINDL, G. et al. Generic digital twin architecture for industrial energy systems. Appl. Sci. 2020, 10, 8903, 2020.

TAO, F. et al. Digital twin in industry: State-of-the-art. IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, VOL. 15, NO. 4, APRIL 2019, 2019.

THYMELEAF. **Thymeleaf**. 2022. Last accessed 29 July 2022. Disponível em: <https://www.thymeleaf.org/>.

URMA, R.-G.; FUSCO, M.; MYCROFT, A. **Modern Java in Action**. 1. ed. [S.l.]: MANNING, 2019.

VMWARE. 2022. Last accessed 02 August 2022. Disponível em: <https://tanzu.vmware.com/pivotal>.

WALLS, C. **Spring in Action**. 5. ed. [S.l.]: MANNING, 2019.

WANASINGHE, T. et al. Digital twin for the oil and gas industry. overview, research trends, opportunities, and challenges. Digital Object Identifier 10.1109/ACCESS.2020.2998723, 2020.

WEBB, P. et al. **Spring Boot Reference Documentation**. 2022. Last accessed 26 July 2022. Disponível em: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#legal>.

WIGGINS, A. **The Twelve Factor App**. 2017. Last accessed 25 July 2022. Disponível em: <https://12factor.net/>.