Dipartimento di Ingegneria e Scienze
dell'Informazione e Matematica
Università degli Studi dell'Aquila
Via Vetoio, I-67100 L'Aquila, Italy

http://www.disim.univaq.it

Master Degree in Informatics, program Software Engineering for Adaptative Systems (SEAS)

Master's Thesis:

A Model-Driven Approach To Microservices Architecture

Student
Carlos Alberto Avendano Arango
Matriculation Number: 249572

Advisor
Prof. Alfonso Pierantonio

Co-Advisor
Dr. Juri Di Rocco

A.A. 2017/2018

ABSTRACT

Model-Driven Engineering (MDE) is a branch of software engineering aiming at making the development of complex software systems easier, reducing the time to market, and increasing software quality. Model-driven approaches shift development focus from code expressed in third generation programming languages to models expressed in domain-specific modeling languages. This thesis proposes a model-driven approach to construct a model that covers the main concepts, features and components of the Microservices Architecture (MSA) systems. Moreover, a business architecture modeling is carried out to guarantee that all those concepts have a suitable implementation along the phases. Using this model most of the problems or drawback presented in MSA systems will be reduced and the user will be able to take advantage of the MDE techniques.

ACKNOWLEDGMENTS

I would like to thank my God for allowing me to achieve this important goal in my professional life. It's wonderful to see that all sacrifices I made during this period have turned into happiness on this day. I never lost faith, even in difficult times in L'Aquila like earthquakes, a lot of snowing and rains, I always believe that I could do it. I'm so grateful to my God.

I would like to thank Prof. Alfonso Pierantonio and Dr. Juri Di Rocco of the Department of Information Engineering, Computer Science and Mathematics at University of L'Aquila. The doors of their office were always open every time I got confused along my thesis. I hope someday we'll work together again.

I would also like to thank my wife Martha Estefany Caro Avila, who decided to live with me this amazing but hard experience away from our home. During last year, she was doing a Master's Degree in Spain, I'm so proud of her since she got graduation few days before me. I want to continue living with her more experiences.

I would like to thank my family, especially my mom Mirian Arango, who surprised me by coming to see my graduation. I feel so lucky to have her as my mother. Also my brothers Cris, Ricardo, Giklis and Beli. They have supported me all the time. I wish to have time to contribute with them all the affection they have always given me.

It's time to thank all my friends in L'Aquila. Tala, Didi, Stephanie, Dima, Shankar, Lea, Salvador, Deyanira, Everson, Hilda, Ahmed, Yaseen, Kartik and Paquito. They have been as my family in these two years. In the end, each one will take different ways, but something sure is that our friendship will remain forever. Cheers guys!!!

Now, I need to mention a person who did as much as he could to make this possible. Kelwin Stevin Payares Ladeuth, thanks for guiding me to perform all the procedures of this Master's application. I wish all his dreams come true because he is always taking care about others, especially of his mom. All the best my friend.

Finally, I would like to thank the entire team of both the Universidad del Magdalena and the University of L'Aquila for being partners of this Erasmus Mundus. The best experience of my life.

TABLE OF CONTENTS

List of Figures

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Model-Driven Engineering (MDE) refers to a branch of software engineering aiming at making the development of complex software systems easier, reducing the time to market, and increasing software quality. Model-driven approaches shift development focus from code expressed in third generation programming languages to models expressed in domain-specific modeling languages [50].

A Domain Specific Language (DSL) consists of abstract and concrete syntax - whereas the abstract syntax defines the constructs of the language, the concrete syntax defines the representation of these constructs [8, 33]. For example, the abstract syntax of the Web Service Description Language (WSDL) defines a set of entities and their properties, such as *ServiceType* and *InterfaceType*, representing, respectively, a service exposed to a client, and its interfaces. To specify *Services* and *Interfaces*, one uses XML statements like <*service* /> and <*interface* />. These XML statements are the concrete syntax of the WSDL [55].

Microservices are a fast growing trend in cloud-based application development [38]. The traditional monolithic application is divided into small pieces that provide a single service: the full capabilities of the application emerge from the interaction of these small pieces. Microservices are independent from each other and organized around capabilities, e.g. user interface, front-end, etc. Their decoupling allows developers to use the best technology for their implementation according to the task they have to accomplish: the application becomes polyglot, involving different programming languages and technologies [19].

In the literature, there has been many work done to address the monolithic application problems by adopting the microservices architecture (MSA) approach [24, 17, 4, 59]. Which as mentioned in the next chapters, has many benefits such as technology heterogeneity, resilience, scaling, ease of deployment among many others. Once microservices architecture is adopted, it is important to select an inter-process communication mechanism to consider services interaction.

The MSA is not a silver bullet. It has both benefits and drawbacks. In fact, there are numerous issues and challenges that a developer must address when using this approach. Most of these refers to the complexity that arises from the fact that a microservices appli-

cation is a distributed system. Such problem has a strong impact during building, testing and deployment phases. Many applications may need to take the event-driven approach in which a microservice publishes an event when something notable happens, such as when it updates a business entity. Other microservices subscribe to those events. When a microservice receives an event it can update its own business entities, which might lead to more events being published [44].

Fortunately, there exist a collection of patterns and principles presented in the literature that resolves most of these issues and challenges related to MSA. They allow to make app development faster and easier to manage following some instructions. For instance, in [44], the author describes how to refactor an existing monolithic application into a MSA.

This thesis proposes a model-driven approach to construct a model that covers the main concepts, features and components of the MSA systems. Moreover, a business architecture modeling is carried out to guarantee that all those concepts have a suitable implementation along the phases. Using this model most of the problems or drawback presented in MSA systems will be reduced and the user will be able to take advantage of the MDE techniques.

Moreover, the use of microservices approach implies to build cloud-ready applications that have declarative formats for automation and setup [59]. This project is involved with technologies such as Docker that uses containers to run an application as an isolated process and its all dependencies (the kernel of the Operating System is shared among other containers) and providing to it only the resources it requires [19].

The thesis is structured as follows:

Chapter 2 - Model-Driven Engineering: it provides the reader with the basics concepts underpinning this work. In particular, it introduces Model-Driven Engineering (MDE) and some relevant related concepts i.e., models, metamodels, model transformation, model difference, among others.

Chapter 3 - Microservices Architecture: it gives notions about microservices-based architecture, Event-Driven Architecture (EDA) and their principal characteristics, as well as the shared database problem and the 12 factor methodology.

Chapter 4 - Patterns and Frameworks: it describes some patterns such as Database per Service, Publish-Subscribe, Saga's coordination, among others, used to build MSA systems. Moreover, different frameworks to perform Saga's coordination are mentioned.

Chapter 5 - A model-driven approach: it introduces a model that covers the main concepts, features and components of the MSA systems. Also, a code generator is mentioned in three different sections such as service components, target platform and generator flow. Finally, an use case is presented to better understand the importance of the work.

Chapter 6 - Conclusion and Future Work: it gives conclusions and discusses future directions.

CHAPTER 2

MODEL-DRIVEN ENGINEERING

Model-Driven Engineering (MDE) is a branch of software engineering aiming at making the development of complex software systems easier, reducing the time to market, and increasing software quality. To this end, prescriptive and descriptive models are produced and manipulated throughout the development process, by means of model transformation, validation, merge and comparison mechanisms. MDE is a general concept, which embodies more specific ones (see Figure 2.1) i.e., Model Driven Development (MDD), and Model Driven Architecture (MDA). For the sake of clarity, the essentials of such concepts are given below.

Model-Driven Architecture (MDA): the first white paper from OMG talking about MDA was published in 2000 and in 2003 the current version of MDA guide has been published [40]:

"MDA is an OMG initiative that proposes to define a set of non proprietary standards that will specify interoperable technologies with which to realize model-driven development with automated transformations. Not all of these technologies will directly concern the transformations involved in MDA. MDA does not necessarily rely on the UML, but, as a specialized kind of MDD (Model-Driven Development), MDA necessarily involves the use of model(s) in development, which entails that at least one modeling language must be used. Any modeling language used in MDA must be described in terms of the MOF language, to enable the metadata to be understood in a standard manner, which is a precondition for any ability to perform automated transformations."

Model-Driven Development (MDD): in [32] authors give the following definition: "Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing". MDD differs from MDA because MDA is coupled with OMG standards and MDD has the flexibility offered to define development processes.

Model-Driven Engineering (MDE): in [50] authors describe MDE as a technology that "offers a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively".

As discussed in [6] it is possible to compare Object Oriented principles with MDE notions. In particular, as shown on the left-hand side of Figure 2.2, an Instance is instanceOf

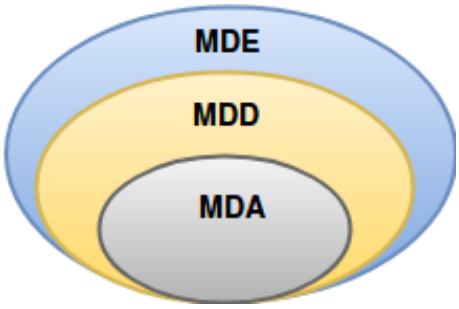


Figure 2.1: MDE vs MDD vs MDA.

a Class, which in turn inheritsFrom a Superclass. Similarly, in MDE a System is representedBy its Model, which is an abstraction of the real system. A Model conformsTo its Metamodel, which defines the concepts of the considered application domain (see the right-hand side of Figure 2.2).

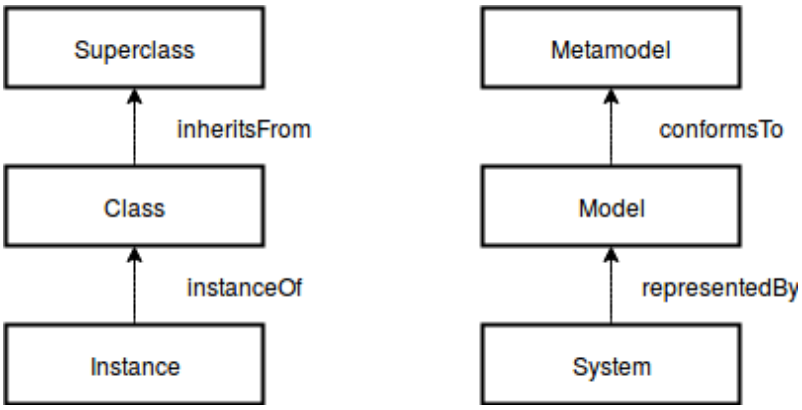
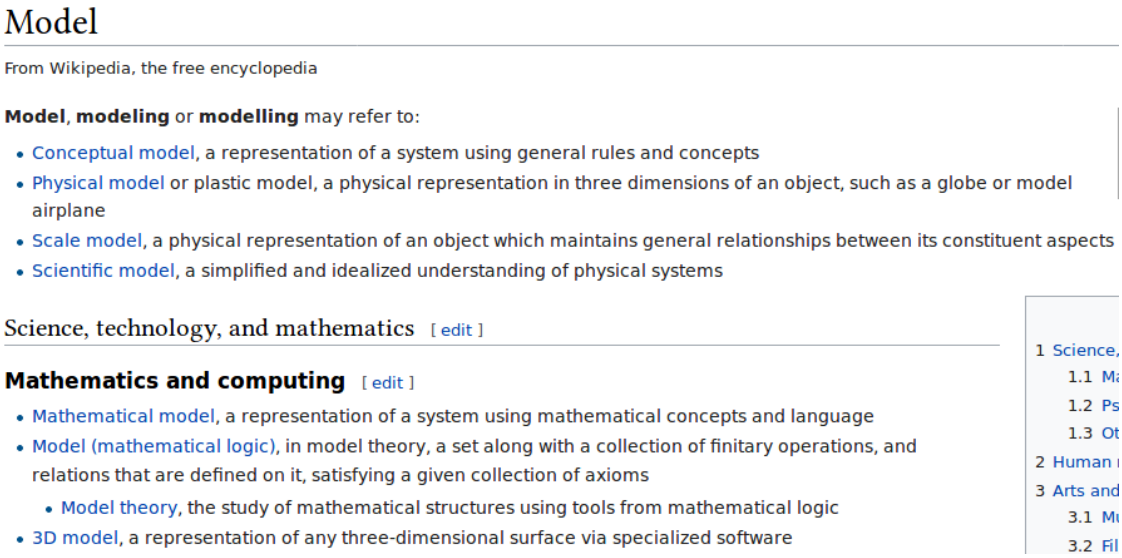


Figure 2.2: Relating models with objects oriented concepts.

2.1 Models and Metamodels

Models are employed in a multitude of fields and consequently there are many definitions for them. For instance, as shown in Figure. 2.3, there are statistical, meteorological, biological, ecological, and economical models. In [29], Kurtev collected several definitions starting from the Latin word modulus, which means a small measure through modern sense of "a representation of a concept" [54]. In [7] Boman et al. define a model as "a simple and familiar structure or mechanism that can be used to interpret some part of reality". MDA and MDE approaches introduce an additional characteristic to define models: "A model is a description of a system written in a well-defined language" [27]. An aspect, which is in common to all the definitions given above is that a model is not intended to capture all the aspects of a system, but mainly to abstract out only some of its characteristics. Thus, a system is usually represented by a set of different models, each capturing some specific aspects.

In MDE models are not considered as merely documentation but precise artifacts that can be automatically processed by computers. In this setting meta-modeling can be seen as the construction of a collection of concepts within a certain domain. A model is an abstraction of a phenomena in the real world, and a meta-model is another abstraction highlighting properties of the model itself. As previously stated, a model is said to conformsTo its meta-model like a program conforms to the grammar of the used programming language, or like an XML file conforms to its schema. Consistently, OMG (Object Management Group) has introduced the four-level metamodeling architecture shown in Figure 2.4. The bottom level is the M0 layer where the real system is placed; the model representing this system is at M1 and this model conforms to the metamodel at M2. The metamodel itself conforms to the metametamodel at M3.


Model

From Wikipedia, the free encyclopedia

Model, **modeling** or **modelling** may refer to:

- Conceptual model, a representation of a system using general rules and concepts
- Physical model or plastic model, a physical representation in three dimensions of an object, such as a globe or model airplane
- Scale model, a physical representation of an object which maintains general relationships between its constituent aspects
- Scientific model, a simplified and idealized understanding of physical systems

Science, technology, and mathematics [edit]

Mathematics and computing [edit]

- Mathematical model, a representation of a system using mathematical concepts and language
- Model (mathematical logic), in model theory, a set along with a collection of finitary operations, and relations that are defined on it, satisfying a given collection of axioms
  - Model theory, the study of mathematical structures using tools from mathematical logic
- 3D model, a representation of any three-dimensional surface via specialized software

1 Science,
1.1 Ma
1.2 Ps
1.3 Ot
2 Human
3 Arts and
3.1 Mu
3.2 Fil

Figure 2.3: Model definitions from Wikipedia.

2.2 Model Management

When employing MDE approaches, models are typically subject of model management operations, which can have as main goals generating further artifacts (e.g., implementation code) or supporting specific analysis activities (e.g., checking if the modeled system is deadlock-free). An overview of the relevant model management techniques and tools underpinning this work is given in the following.

2.2.1 Model Transformation

The MDA guide [35] defines a model transformation as “the process of converting one model to another model of the same system”. Kleppe et al. [27] defines a transformation as the automatic generation of a target model from a source model, according to a

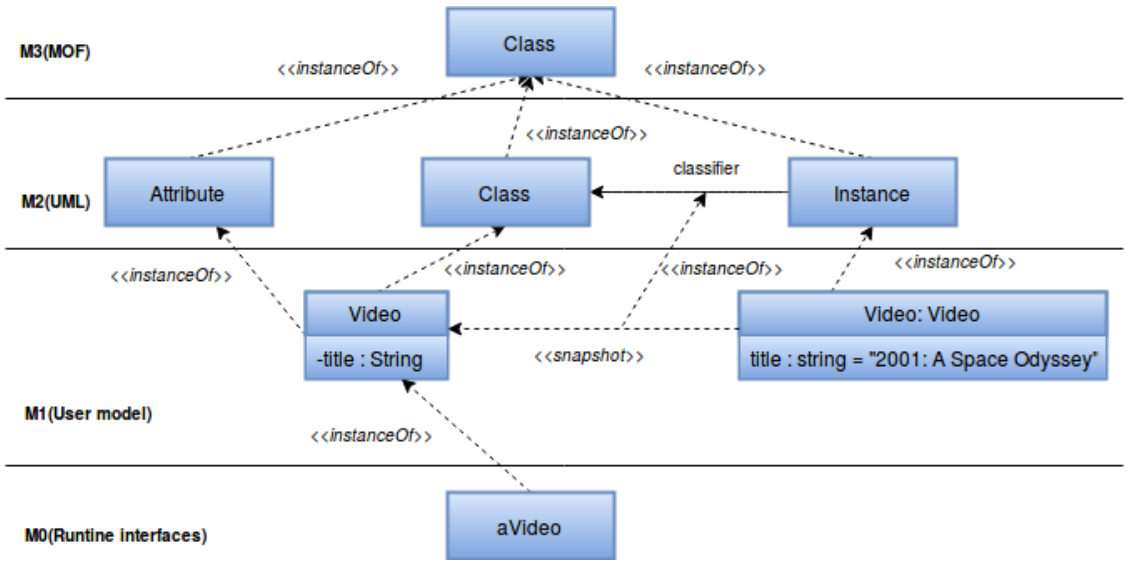


Figure 2.4: OMG four layers metamodeling architecture.

transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed to a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed to one or more constructs in the target language.

Rephrasing these definitions by considering Figure 2.5, a model transformation program takes as input one or more models conforming to a given source meta-model and produces as output models conforming to a target meta-model. The transformation program, composed of a set of rules, should itself considered as a model. As a consequence, it is based on a corresponding meta-model, that is an abstract definition of the used transformation language.

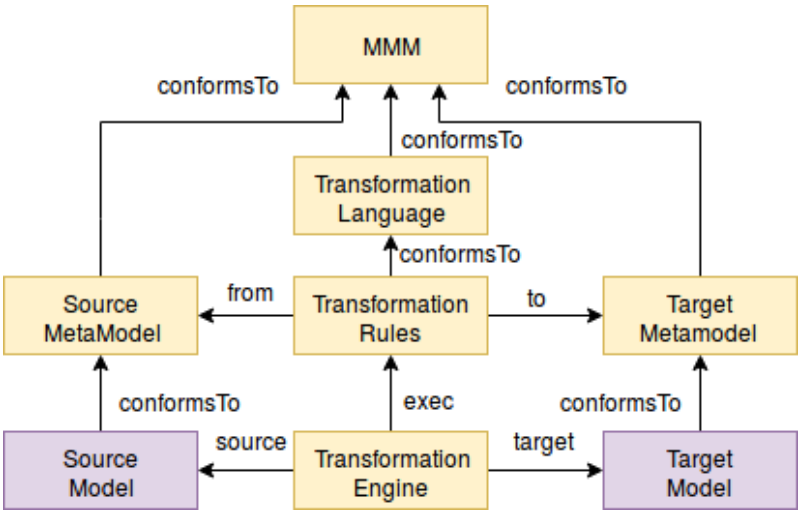


Figure 2.5: Basic Concepts of Model Transformation.

Many languages and tools have been proposed to specify and execute transformation programs. In 2002 OMG issued the Query/View/Transformation request for proposal [41] to define a standard transformation language. Even though a final specification has been adopted at the end of 2005, the area of model transformation continues to be a subject of intense research. Over the last years, in parallel to the OMG process a number of model transformation approaches have been proposed both from academia and industry. The paradigms, constructs, modeling approaches, tool support distinguish the proposals each of them with a certain suitability for a certain set of problems.

In the following, a classification of the today's model transformation approaches is briefly reported by relying on the work presented in [11]. At top level, model transformation approaches can be distinguished between model-to-code and model-to-model. The distinction is that, while a model-to-model transformation creates as target a model, which conforms to the target meta-model, the target of a model-to-text transformation is essentially consists of strings. In the following some classifications of model-to-model transformation languages discussed in [11] are described.

Direct manipulation approach. It offers an internal model representation and some APIs to manipulate it. It is usually implemented as an object oriented framework, which may also provide some minimal infrastructure. Users have to implement transformation rules, scheduling, tracing and other facilities, mostly from the beginning in a programming language.

Operational approach. It is similar to direct manipulation but offers more dedicated support for model transformation. A typical solution in this category is to extend the utilized meta-modeling formalism with facilities for expressing computations. An example would be to extend a query language such as OCL with imperative constructs. Examples of systems in this category are QVT Operational mappings [42], XMF [63], MTL [61] and Kermeta [37].

Relational approach. It groups declarative approaches in which the main concept is mathematical relations. In general, relational approaches can be seen as a form of constraint solving. The basic idea is to specify the relations among source and target element type using constraints that in general are nonexecutable. However, declarative constraints can be given executable semantics, such as in logic programming where predicates can be used to describe the relations. All of the relational approaches are side-effect free and, in contrast to the imperative direct manipulation approaches, create target elements implicitly. Relational approaches can naturally support multidirectional rules. They sometimes also provide backtracking. Most relational approaches require strict separation between source and target models, that is, they do not allow in-place update. Example of relational approaches are QVT Relations [42] and AMW [14]. Moreover, in [23] the application of logic programming has been explored for the purpose. Finally, in [10] we have investigated the application of the Answer Set Programming [22] for specifying relational and bidirectional transformations.

Hybrid approach. It combines different techniques from the previous categories, like

ATL [26] that wraps imperative bodies inside declarative statements.

Graph-transformation based approaches. It draws on the theoretical work on graph transformations. Describing a model transformation by graph transformation, the source and target models have to be given as graphs. Performing model transformation by graph transformation means to take the abstract syntax graph of a model, and to transform it according to certain transformation rules. The result is the syntax graph of the target model.

Being more precise, graph transformation rules have an LHS and an RHS graph pattern. The LHS pattern is matched in the model being transformed and replaced by the RHS pattern in place. In particular, LHR represents the pre-condition of the given rule, while RHS describes the post-conditions. LHR ∩ RHS defines a part which has to exist to apply the rule, but which is not changed. LHS - LHS ∩ RHS defines the part which shall be deleted, and RHS - LHS ∩ RHS defines the part to be created. The LHS often contains conditions in addition to the LHS pattern, for example, negative conditions. Some additional logic is needed to compute target attribute values such as element names. AGG [58] and AToM3 [12] are systems directly implementing the theoretical approach to attributed graphs and transformations on such graphs. They have built-in fixpoint scheduling with non-deterministic rule selection and concurrent application to all matching locations, and the rely on implicit scheduling by the user. The transformation rules are unidirectional and in-place. Systems such as VIATRA2 [60] and GReAT [2] extend the basic functionality of AGG and AToM3 by adding explicit scheduling. VIATRA2 users can build state machines to schedule transformation rules whereas GReAT relies on data-flow graph.

2.2.2 Model Difference

Calculating differences between models is an important task, which consists of a number of tasks starting with identifying matching model elements, calculating and representing their differences, and finally visualizing them in a suitable way. The problem of determining model differences is intrinsically complex. The overall problem can be separated into three steps [28]:

1. *Calculation*, a method or algorithm able to compare two distinct models: the task of model comparison consists of identifying the mappings and the differences between two models.

2. *Representation*, the output of the previous step must be represented in some form available to other manipulations: the information obtained from the previous step needs to be properly represented in a difference model, so that it can be used for subsequent analysis and manipulations. The visualization and the representation tend to overlap and the overall method is affected by the way the differences are computed.

3. *Visualization*, model differences are often required to be visualized in a human-readable notation: differences often need to be presented according to a specific need or scope, highlighting those pieces of information which is relevant only for the prescribed intent. In other words, a visualization is realized by specifying a concrete syntax which renders the abstract syntax and may vary from intuitive diagrammatic notations to textual catalogues. The same representation may include different visualizations, not necessarily diagrammatic ones, depending on the specific goal the designer has in mind.

EMFCompare[1] provides comparison and merge facility for any kind of EMF Model. It includes a generic comparison engine, the ability to export differences in a model patch, and it is integrated with the Eclipse Team API.

2.3 Model to Text Transformation Languages

Model transformation has been characterised as the heart and soul of Model-Driven Engineering (MDE) [51]. An important type of model transformation is model-to-text (M2T) transformation, which is used to implement code and documentation generation, model serialisation (enabling model interchange), and model visualisation and exploration. Today, developers wishing to implement a M2T transformation face a difficult decision, as there are numerous M2T languages with features that vary considerably. For example, some languages focus on providing tight integration with - and sophisticated developer tools for - a small number of target languages, such as Microsoft T4[2] which targets .NET languages. Other M2T languages seek not to constrain the form of generated text, but rather to be small and easy to learn, such as Java Emitter Templates[3] (JET) [49].

The difficulty in choosing a M2T language is compounded by a lack of interoperability between M2T languages, which means that it is generally very time-consuming to rewrite an existing transformation to work with a different M2T language [49]. Although the Object Management Group produced a M2T standard in 2008 (MOF Model-To-Text Transformation Language, MOFM2T), only Acceleo[4] seeks to conform to the standard, and other M2T languages deviate from it in both syntax and features. Promisingly however, most M2T languages are template-based [11] (rather than using a visitor pattern or explicit printing statements).

[1] http://www.eclipse.org/emf/compare/
[2] http://msdn.microsoft.com/en-us/library/bb126445.aspx
[3] http://www.eclipse.org/emft/projects/jet/
[4] http://www.eclipse.org/acceleo/

2.4 OCL CONSTRAINT

Model-based testing (MBT) has recently received increasing attention in both industry and academia. MBT promises systematic, automated, and thorough testing, which would likely not be possible without models. However, the full automation of MBT, which is a requirement for scaling up to large systems, requires solving many problems, including preparing models for testing (e.g., flattening state machines), defining appropriate test strategies and coverage criteria, and generating test data to execute test cases [3]. Furthermore, in order to increase chances of adoption, using MBT for industrial applications requires using well-established standards, such as the Unified Modeling Language (UML) and its associated language to write constraints: the Object Constraint Language (OCL) [39].

OCL [39] is a standard language that is widely accepted for writing constraints on UML models. OCL is based on first order logic and is a highly expressive language. The language allows modelers to write constraints at various levels of abstraction and for various types of models. It can be used to write class and state invariants, guards in state machines, constraints in sequence diagrams, and pre and post condition of operations. A basic subset of the language has been defined that can be used with meta-models defined in Meta Object Facility (MOF) [34] (which is a standard defined by Object Management Group (OMG) for defining meta-models).

This subset of OCL has been largely used in the definition of UML for constraining various elements of the language. Moreover, the language is also used in writing constraints while defining UML profiles, which is a standard way of extending UML for various domains using pre-defined extension mechanisms [3].

Due to the ability of OCL to specify constraints for various purposes during modeling, for example when defining guard conditions or state invariants in state machines, such constraints play a significant role when testing is driven by models. For example, in state-based testing, if the aim of a test case is to execute a guarded transition (where the guard is written in OCL based on input values of the trigger) to achieve full transition coverage, then it is essential to provide input values to the event that triggers the transition such that the values satisfy the guard. Another example can be to generate valid parameter values based on the precondition of an operation [3].

CHAPTER 3

MICROSERVICES ARCHITECTURE

Over the years, most enterprise software applications have been designed and developed as large, complex, monolithic applications. In a monolithic approach, applications are decomposed into layered architectures such as model view controllers. However, applications are still optimized with regard to the level of functions that each of the application are responsible for, therefore making them large and complex to manage. A monolithic application is an application where all of the logic runs in a single app server [4].

In [44] author describes how a successful monolithic application like FTGO (Food To Go[1]) became in monolithic hell due to the growth of code among the years. In fact, this application, at the beginning had lots of benefits such as simple to develop, to deploy, to test and to scale since they ran multiple instances of the application behind a load balancer. Unfortunately, as the FTGO developers have discovered, the monolithic architecture has a huge limitation. Successful applications, like the FTGO application, have a habit of outgrowing the monolithic architecture which, of course, makes the code base larger. Moreover, as the company became more successful, the size of the development team steadily grew. Not only did this increase the growth rate of code base but it also increased the management overhead. So, the once small, simple FTGO application, which was developed by small team, grew over the past 10 years into a monstrous monolith developed by a large team. As a result of outgrowing its architecture, FTGO is in monolithic hell.

In the literature, there has been many work done to address the monithich hell problem by adopting the microservices architecture approach [24, 17, 4, 59]. Microservices - also known as the Microservice Architecture (MSA) - is an architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities. The MSA enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack [45].

The term *"microservices architecture"* has sprung up over the last few years to describe a particular way of designing software applications as suites of independently deployable services. While there is no precise definition of this architectural style, there are certain common characteristics around organization business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data [20].

[1] Food To Go: It was one of the leading online food delivery companies in the US.

Many explanations of microservices have emerged in recent years as communities attempt to propose a definition. As we work towards a definition, many inconsistencies and misnomers become apparent [52].

One such example is that of the meaning of 'micro'; some literature reports that a microservice is very small, and therefore such an architecture makes use of many, small services. As we have discovered, 'micro' is a relatively ambiguous term, that does not always describe the size of a service, as these can vary. Therefore, we have examined a range of literature that is relevant to MSA, in order to assist the development community who might be considering using microservices architectures for their applications [52].

Dragoni et al [15, 16] define microservices as:

> *"A microservice is a cohesive, independent process interacting via messages".*

This definition describes two key features of microservices. First, microservices should be highly cohesive units; they should do one thing well. Second, microservices must be able to execute their own processes which allows for independent deployment. However, the same definition might also be applied to an object oriented (OO) class [52].

Dragoni et al [15, 16] also offers a definition for a MSA:

> *"A MSA is a distributed application where all its modules are microservices".*

MSA is about a service addressing a single business capability, with a clearly defined interface. In a MSA a series of microservices are chained together to perform a bigger business function. To enable these characteristics, each microservice has its own data model and a class model. Figure 3.1 shows a sample application which utilises a MSA.

Adrian Cockcroft defines a microservice as:

> *"loosely coupled service in a bounded context."*

This definition refers to 'bounded context' which is derived from the Domain Driven Design (DDD) [18] literature. A bounded context captures the key properties of a MSA: the focus upon business capabilities, rather than program code decomposition and reuse. Such a perspective supports the capture and modelling of requirements in complex multi-agency domains such as the delivery of community healthcare, or applications for the Internet of Things (IoT) [52].

In general, microservices are small, autonomous services that work together [38]. The key here is small and autonomous. Microservices is an architecture style, in which large complex software applications are composed of one or more services. Microservices can

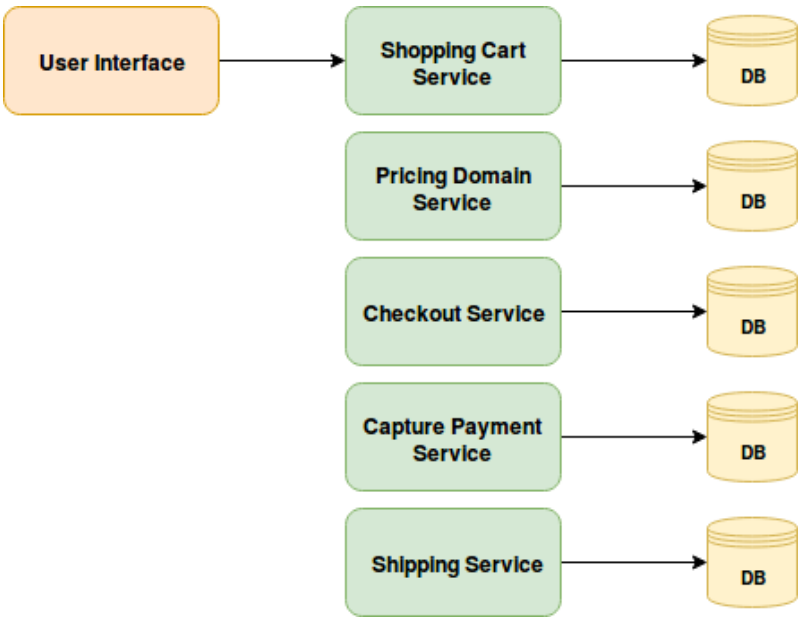


Figure 3.1: Example of application architecture using microservices principles.

be deployed independently of one another and are loosely coupled and each of these microservices focuses on completing one task only. Microservices support multiple clients in a typical architecture. Additionally, there are several factors to be considered in a typical Microservices based application, called 12-factor app which is explained in 3.5 [4].

3.1 Key Components

Microservices architecture consists of several components that work together to provide distributed services. The key components are mentioned as follows.

3.1.1 Containers

Containers have been in deployments for some time now in the form of Linux containers and several commercial implementations (like Dockers). Linux containers are self-contained execution environments with their own isolated CPU, memory, block I/O, and network resources that share the kernel of the host-operating system [36]. The experience is like a virtual machine, but without the weight and startup overhead of a guest operating system [4].

3.1.2 Services Communication

In a monolithic application, components are invoked via function calls. In contrast, microservices-based services interact using an Inter-Process Communication (IPC) mechanism. When selecting an IPC mechanism for a service, it is important to consider services interaction. One dimension is whether the interaction is synchronous or asynchronous. With synchronous interaction, the client expects a timely response from the service and might even block while it waits. With asynchronous interaction, the client does not block while waiting for a response. Additionally, there are other types of one-to-one interactions - namely, request/response where a client makes a request to a service and waits for a response. Notification occurs where the client sends a request to a service, but no reply is expected or sent. Request/async response is where a client sends a request to a service, which replies asynchronously. There are different types of one-to-many interactions: Publish/subscribe, where client publishes a notification message, which is consumed by zero or more interested services; Publish/async responses, where a client publishes a request message, and then waits a certain amount of time for responses from interested services. Each service typically uses a combination of these interaction styles. Below is a review of IPC mechanisms that a developer can implement in microservice architecture [4].

3.1.3 IPC Technologies

There are several IPC mechanisms that can be applied to microservices. Services can use synchronous request/response-based communication mechanisms such as HTTP-based REST or Thrift. Alternatively, they can use asynchronous, message-based communication mechanisms such as Advanced Message Queuing Protocol (AMQP) [36].

In an asynchronous system, client makes a request to a service by sending it a message. If the service replies, it sends a separate message back to the client. Since the communication is asynchronous, the client does not block waiting for a reply. Instead, the client is written assuming that the reply will not be received immediately. There are a large number of open source messaging systems to choose from, including RabbitMQ, Apache Kafka, Apache ActiveMQ [4].

When using a synchronous, request/response-based IPC mechanism, a client sends a request to a service. The service processes the request and sends back a response. In many clients, the thread that makes the request blocks while waiting for a response. There are numerous protocols to choose from. Two popular protocols are REST and Thrift [4].

3.1.4 Service Registry and Discovery

In a microservices application, the set of running service instances changes dynamically, including network locations. Consequently, in order for a client to make a request to a

service, it must use a service-discovery mechanism. A key part of service discovery is the service registry. The service registry is a database of available service instances. The service registry provides a management API and a query API. Service instances are registered with and deregistered from the service registry using the management API. The query API is used by system components to discover available service instances. There are two service-discovery patterns: client-side discovery and service-side discovery. In systems that leverage client-side service delivery, clients query the service registry, select an available instance, and make a request. In systems that use server-side discovery, clients make requests via a router, which queries the service registry and forwards the request to an available instance. There are two methods where service instances are registered with and deregistered from the service registry. One method is for service instances to register themselves with the service registry, the self-registration. The other method is a system component to handle the registration and deregistration on behalf of the service. In deployment environments one needs to setup a service-discovery infrastructure using a service registry such as etcd and Apache Zookeeper. In other deployment environments, service discovery is built in. For example, Kubernetes will service instance-registration and deregistration [4].

3.2 Event Driven Architecture

For many applications, the solution is to use an Event Driven Architecture (EDA). In this architecture, a microservices publishes an event when something notable happens, such as when it updates a business entity. Other microservices subscribe to those events. When a microservice receives an event it can update its own business entities, which might lead to more events being published [45].

EDA is an architecture design where services of independent software components communicate through event notifications (Woolf, 2006). Marechaux (2006) defines EDA as “a methodology for designing and implementing applications and systems in which events transmit between decoupled software components and services”. EDA uses publish-subscribe architecture to enable the communication among services and to end-users. It has three main components: event emitter, event broker and event subscriber. An event emitter detects events and posts an announcement to the event broker. The event broker collects all the triggered events and forwards them to interested subscribers. Finally, event subscribers receive event notifications and respond accordingly. Event subscribers are able to further trigger the event to other services [9].

3.3 Benefits

In [38] Sam Newman describes that the benefits of microservices are many and varied. Many of these benefits can be laid at the door of any distributed system. Microservices,

however, tend to achieve these benefits to a greater degree primarily due to how far they take the concepts behind distributed systems and service-oriented architecture.

3.3.1 Technology Heterogeneity

With a system composed of multiple, collaborating services, we can decide to use different technologies inside each one. This allows us to pick the right tool for each job, rather than having to select a more standardized, one-size-fits-all approach that often ends up being the lowest common denominator [38].

If one part of our system needs to improve its performance, we might decide to use a different technology stack that is better able to achieve the performance levels required. We may also decide that how we store our data needs to change for different parts of our system. For example, for a social network, we might store our users' interactions in a graph-oriented database to reflect the highly interconnected nature of a social graph, but perhaps the posts the users make could be stored in a document-oriented data store, giving rise to a heterogeneous architecture like the one shown in Figure 3.2 [38].

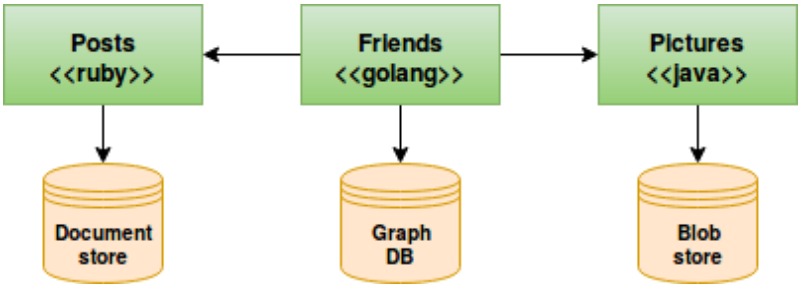


Figure 3.2: Microservices technologies example.

3.3.2 Resilience

A key concept in resilience engineering is the bulkhead. If one component of a system fails, but that failure does not cascade, you can isolate the problem and the rest of the system can carry on working. Service boundaries become your obvious bulkheads. In a monolithic service, if the service fails, everything stops working. With a monolithic system, we can run on multiple machines to reduce our chance of failure, but with microservices, we can build systems that handle the total failure of services and degrade functionality accordingly [38].

3.3.3 Scaling

With a large, monolithic service, we have to scale everything together. One small part of our overall system is constrained in performance, but if that behavior is locked up in

a giant monolithic application, we have to handle scaling everything as a piece. With smaller services, we can just scale those services that need scaling, allowing us to run other parts of the system on smaller, less powerful hardware [38].

3.3.4 Ease of Deployment

With microservices, we can make a change to a single service and deploy it independently of the rest of the system. This allows us to get our code deployed faster. If a problem does occur, it can be isolated quickly to an individual service, making fast rollback easy to achieve. It also means we can get our new functionality out to customers faster. This is one of the main reasons why organizations like Amazon and Netflix use these architectures to ensure they remove as many impediments as possible to getting software out the door [38].

3.3.5 Composability

One of the key promises of distributed systems and service-oriented architectures is that we open up opportunities for reuse of functionality. With microservices, we allow for our functionality to be consumed in different ways for different purposes. This can be especially important when we think about how our consumers use our software [38].

3.4 The Shared Database Problem

Figure 3.3 shows a registration user interface (UI), which creates customers by performing SQL operations directly on the database (DB). It also shows a call center application that views and edits customer data by running SQL on the database. And the warehouse updates information about customer orders by querying the database. This is a common enough pattern, but it is one fraught with difficulties [38].

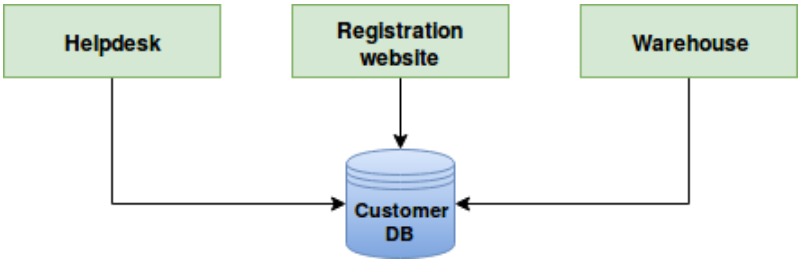


Figure 3.3: Database integration.

First, we are allowing external parties to view and bind to internal implementation details. The data structures I store in the DB are fair game to all; they are shared in their entirety

with all other parties with access to the database. If I decide to change my schema to better represent my data, or make my system easier to maintain, I can break my consumers. The DB is effectively a very large, shared API that is also quite brittle. If I want to change the logic associated with, say, how the helpdesk manages customers and this requires a change to the database, I have to be extremely careful that I do not break parts of the schema used by other services. This situation normally results in requiring a large amount of regression testing [38].

Second, my consumers are tied to a specific technology choice. Perhaps right now it makes sense to store customers in a relational database, so my consumers use an appropriate (potentially DB-specific) driver to talk to it. What if over time we realize we would be better off storing data in a nonrelational database? Can it make that decision? So consumers are intimately tied to the implementation of the customer service. As we discussed earlier, we really want to ensure that implementation detail is hidden from consumers to allow our service a level of autonomy in terms of how it changes its internals over time. Goodbye, loose coupling [38].

Finally, let's think about behavior for a moment. There is going to be logic associated with how a customer is changed. Where is that logic? If consumers are directly manipulating the DB, then they have to own the associated logic. The logic to perform the same sorts of manipulation to a customer may now be spread among multiple consumers. If the warehouse, registration UI, and call center UI all need to edit customer information, I need to fix a bug or change the behavior in three different places, and deploy those changes too. Goodbye, cohesion [38].

3.5 12-FACTOR APP

A common methodology, called the 12-Factor App methodology, has emerged with the purpose of providing an outline for building well-structured and scalable applications with the microservices approach. The application deployment process can be complicated and extensive. Virtualization, networking, and setting up runtime environments are just a few of the challenges involved. The 12-Factor App methodology helps to create a framework for organizing the process in order to maintain a healthy and scalable application [4].

1. Codebase

   A 12-Factor App has a version control system such as git. A copy of the revision-tracking database is known as a code repository, see in Figure 3.4. A codebase is any single repository or in any set of repository that shares a root commit [25]. A 12-Factor App requires one-to-one correlation between the codebase and the application. If there are multiple codebases, it is a distributed system with many applications. Each component in a distributed system is an application, and each can individually comply with 12-Factor. Multiple applications sharing the same code is

a violation of 12-Factor. There is only one codebase per application, but there will be many deploys of the app. A deploy is a running instance of the application. This is typically a production site and one or more staging sites [4].

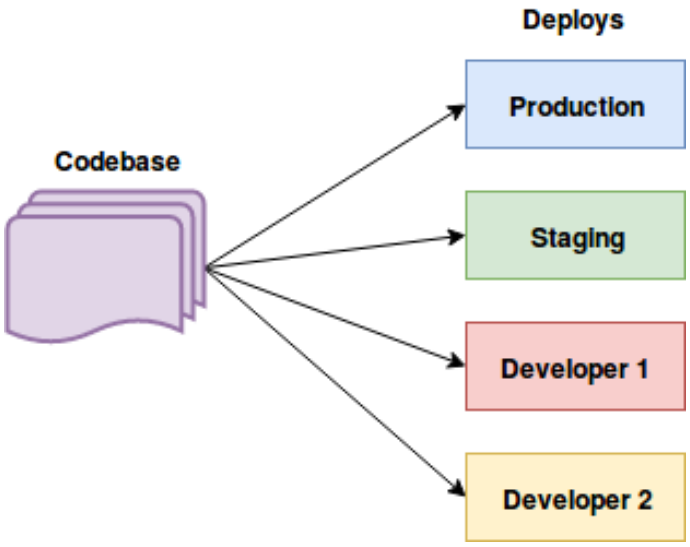


Figure 3.4: Codebase repository.

2. Dependencies

   Explicitly declare and isolate dependencies. Most programming languages offer a packaging system for distributing support libraries, such as CPAN for Perl or Rubygems for Ruby. Libraries installed through a packaging system can be installed system-wide (known as 'site packages') or scoped into the directory containing the app (known as 'vendoring' or 'bundling').

   A twelve-factor app never relies on implicit existence of system-wide packages. It declares all dependencies, completely and exactly, via a dependency declaration manifest [25].

3. Config

   An app's config typically changes from different deployment environments like staging, production, and development. The config could include resource handles to databases, credentials to external services, host names, etc. 12-Factor requires strict separation of config from code, as config varies across deployments, while code does not [25]. One approach to config is the use of config files, which are not checked into revision control.

   However, there is a tendency for config files to be scattered in different places and different formats, making it hard to see and manage all in one place. Another approach is to store config in environment variables. Env vars are easy to change between deploys without changing any code. Yet, another approach, to config management is grouping. Sometimes apps batch configs into named groups (often called "environments") named after specific deploys such as the development,

test, and production environments. In a 12-Factor application, env vars are granular controls, each disjointed to other env vars. This is a model that scales up smoothly as the app naturally expands into more deploys over its lifetime [4].

4. Backing Services

   Treat backing services as attached resources, see in Figure 3.5 [25]. A backing service is any service the app consumes over the network as part of its normal operation.

   Backing services like the database are traditionally managed by the same systems administrators as the app's runtime deploy. In addition to these locally-managed services, the app may also have services provided and managed by third parties. The code for a twelve-factor app makes no distinction between local and third party services [25].

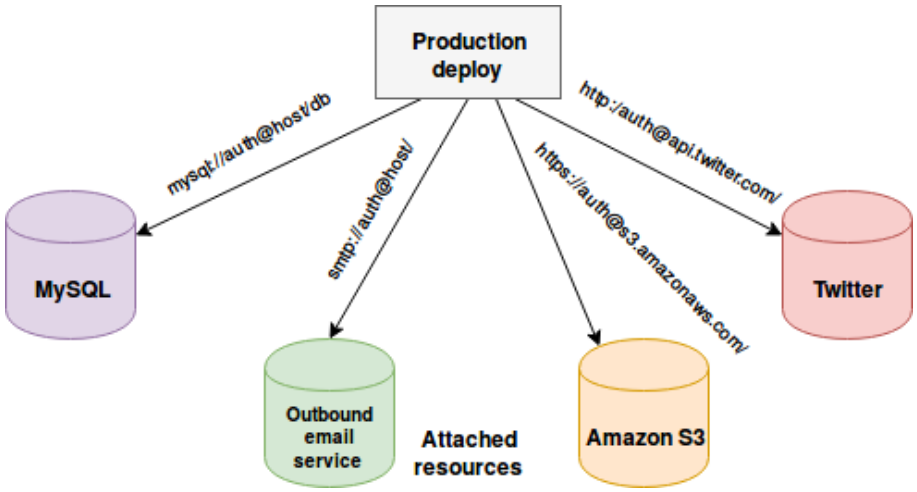


Figure 3.5: Backing Services.

5. Build, release, run

   Strictly separate build and run stages, see in Figure 3.6. A codebase is transformed into a (non-development) [25] deploy through three stages:

   - The build stage is a transform which converts a code repo into an executable bundle known as a build.
   - The release stage takes the build produced by the build stage and combines it with the deploy's current config.
   - The run stage (also known as "runtime") runs the app in the execution environment.

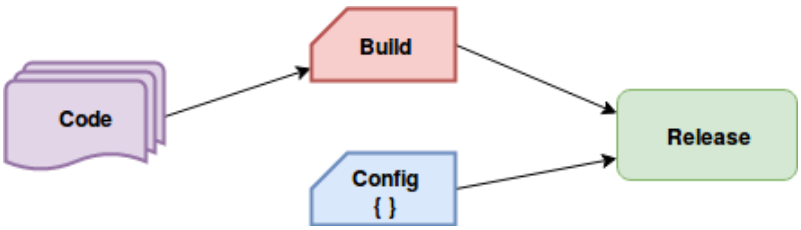


Figure 3.6: Backing Services.

6. Process

   The application is executed in an environment as one or more processes. On one end of the continuum, the code is a stand-alone, and the execution environment is a developer's local system with an installed language runtime [25].

   On the other end of the spectrum, a production deployment of a sophisticated app may use many processes. 12-Factor processes are stateless and share-nothing, hence any persistent data needs to be in a backing service, typically a database. The memory space or file system of the process can be used for a single-transaction cache; for example, downloading a large file, operations, and storing the results of the operation in the database. The 12-Factor App assumes that any data cached in memory or on disk will be unavailable for job request. Some web-based systems rely on caching user session data in memory of the process and expect future requests from the same visitor to be routed to the same process. Sticky sessions are a violation of 12-Factor and should never be used or relied upon. Session state data is a good candidate for a datastore that offers time-expiration [4].

7. Port Binding

   Export services via port binding. Web apps are sometimes executed inside a web-server container. For example, Java apps might run inside Tomcat.

   The twelve-factor app is completely self-contained and does not rely on runtime injection of a webserver into the execution environment to create a web-facing service. The web app exports HTTP as a service by binding to a port, and listening to requests coming in on that port [25].

8. Concurrency

   In the 12-Factor App, processes are first-class citizens [25]. Processes in the 12-Factor App are built on a unix process model. Using this model, the developer can architect the application to handle diverse workloads by assigning each type of work to a process type. For example, HTTP requests may be handled by a web process, and background tasks handled by a worker process. This process model is a scale-out approach. The share-nothing, partitioned process architecture provides a simple approach to concurrency. 12-Factor App processes should leverage the operating system's process manager to manage task output streams and handle user-initiated restarts and shutdowns [4].

9. Disposability

   Maximize robustness with fast startup and graceful shutdown, The twelve-factor app's processes are disposable, meaning they can be started or stopped at a moment's notice. This facilitates fast elastic scaling, rapid deployment of code or config changes, and robustness of production deploys.

   Processes should strive to minimize startup time. Ideally, a process takes a few seconds from the time the launch command is executed until the process is up and ready to receive requests or jobs [25].

10. Development and Production Parity

    The gaps between development and production environments are typically due to either time gap, as development releases code and operations deploy it, or a tool gap between developers and operations team as they may use different tools. A 12-Factor App is designed for continuous development with minimal gap in environments [25]. Additionally, the 12-Factor App should use the same backing services between development and production, even when adapters abstract away any differences in backing services. Differences between backing services could mean incompatibilities, defeating the purpose of continuous deployment. Adapters to different backing services are still useful because they make porting to new backing services relatively simple. But all deploys of the app (developer environments, staging, production) should be using the same type and version of each of the backing services [4].

11. Logs

    Treat logs as event streams. Logs provide visibility into the behaviour of a running app. In server-based environments they are commonly written to a file on disk (a 'logfile'); but this is only an output format.

    A twelve-factor app never concerns itself with routing or storage of its output stream. It should not attempt to write to or manage logfiles [40].

12. Administration Process

    In a 12-Factor App, admin tasks should be run as a one-off process. Developers often utilize one-off administrative or maintenance tasks for the app such as, database migrations, console or to inspect the app's models against the live database [25]. One-off admin processes should be run in an identical environment as the regular long process of the app, and against a release, using the same codebase and config. The admin code must ship with the application code to avoid synchronization issues [4].

3.6 Drawbacks

Of course, no technology is a silver bullet, and the microservice architecture has a number of significant drawbacks and issues. In [44] author identifies some of them presented as

follows.

"Finding the right set of services is challenging"

One challenge with using the microservice architecture is that there isn't a concrete, well-defined algorithm for decomposing a system into services. Like much of software development, it is somewhat of an art. To make matter's worse, if you decompose a system incorrectly you will build a distributed monolith, a system consisting of coupled services that must be deployed together. It has the drawbacks of both the monolithic architecture and the microservice architecture.

"Distributed systems are complex"

Another challenge with using the microservice architecture is that developers must deal with the additional complexity of creating a distributed system. Developers must use an inter-process communication mechanism. Implementing use cases that span multiple services requires the use of unfamiliar techniques. IDEs and other development tools are focused on building monolithic applications and don't provide explicit support for developing distributed applications. Writing automated tests that involve multiple services is challenging. These are all issues that are specific to the microservice architecture. Consequently, your organization's developers must have sophisticated software development and delivery skills in order to successfully use microservices.

"Deploying features that span multiple services requires careful coordination"

Another challenge with using the microservice architecture is that deploying features that span multiple services requires careful coordination between the various development teams. You have to create a rollout plan that orders service deployments based on the dependencies between services. That's quite different than when using a monolithic architecture where you can easily deploy updates to multiple components atomically.

"Deciding when to adopt the microservice architecture"

This can be a major dilemma for startups whose biggest challenge is usually how to rapidly evolve the business model and accompanying application. Using the microservice architecture makes it much more difficult to iterate rapidly. A startup should almost certainly begin with a monolithic application.

Later on, however, when the challenge is how to handle complexity then it makes sense to functionally decompose the application a set of microservices. However, you might find refactoring difficult because of tangled dependencies.

CHAPTER 4

PATTERNS AND FRAMEWORKS

The MSA is not a silver bullet. It has both benefits and drawbacks. In fact, there are numerous issues and challenges that a developer must address when using this approach as shown in the previous chapter. Most of these refers to the complexity that arises from the fact that a microservices application is a distributed system. Such problem has a strong impact during building, testing and deployment phases.

Fortunately, there exist a collection of patterns and principles presented in the literature that resolves most of these issues and challenges related to MSA. They allow to make app development faster and easier to manage following some instructions. For instance, in [44], the author describes how to refactor an existing monolithic application into a MSA.

In addition, this chapter describes a framework named Eventuate which is based on EDA (explained previously in chapter Microservices Architecture) and adopts one of the main patterns to build MSA systems. Finally, a problem section is also considered since the use of EDA is potentially error-prone.

4.1 CATALOGUE OF PATTERNS

This section presents a catalogue of patterns. The catalogue is not exhaustive, but limited to the most relevant patterns for the realization of a system using MSA characteristics.

4.1.1 DECOMPOSITION PATTERNS

The decomposition in small services makes the application less coupled by following the Single Responsibility Principle (SRP). This principle states that each module, subsystem, class or method should not have more than one reason to change [31]. The most common design pattern which is called domain-driven design is presented as follows.

Domain Driven Design

Domain-driven design (DDD) is a method used to analyze complex systems. This method allows extracting services, from monolithic software applications [18]. To foster domain understanding and correctness of an emerging design, DDD emphasizes agile, collaborative modeling of domain experts and software engineers [43].

Currently, MSA is maturing as an architectural style for distributed software systems with high requirements for scalability and adaptability [38].

In [30], authors propose domain-driven design patterns (DDDPs) to address with a set of core concrete design patterns. A DDDPs is a design pattern that addresses a domain modelling problem, is described in a structured format, and whose form is a template model that is expressed in a well-defined modelling language.

In [45] the author refers DDD as subdomains and presents a classification of them as follows:

- Core - key differentiator for the business and the most valuable part of the application.
- Supporting - related to what the business does but not a differentiator. These can be implemented in-house or outsourced.
- Generic - not specific to the business and are ideally implemented using off the shelf software.

Figure 4.1 shows a subdomains example of an online store presented in [45] which includes product catalog, inventory management, order management and delivery management. The corresponding microservice architecture would have services corresponding to each of these subdomains.

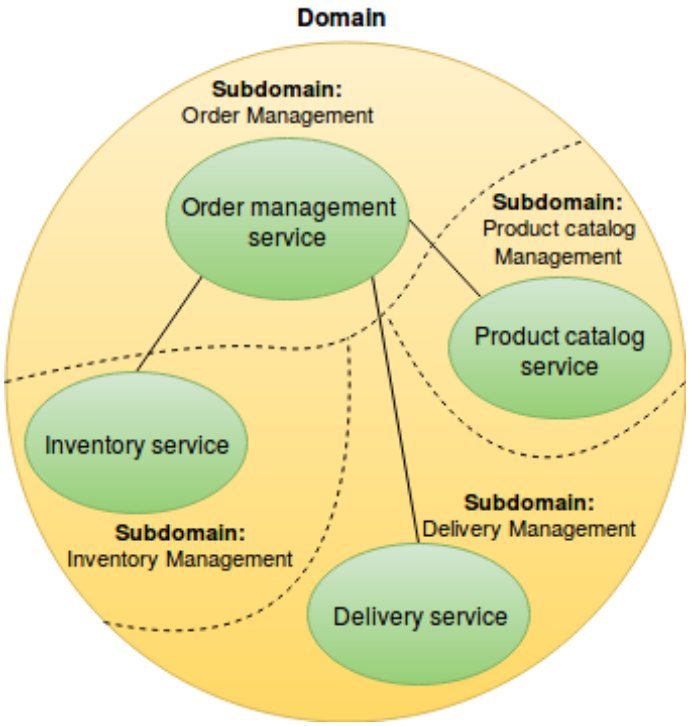


Figure 4.1: Domain-Driven Design (DDD) example.

4.1.2 Database per service pattern

Microservice keeps data persistent and private to the independent small service of an application, accessible via API. Every small and scalable service has its own database, in order to reduce the architectural complexity and the risk. The database is designed in such away that accessed by a key value store which in turn embedded with independent service. This database-per-service used with the future business logic and implemented for desired service. So the database service is strictly coupled with the data, hence this pattern is even stronger than the Database-Server per Service Pattern, because the database itself acts as a business service [53].

Figure 4.2, describes services divided into micro which is connected to independent database as this pattern explains. Each microservice is accessible to their independent database with duplex connection and each microservice connected to API gateway which can be accessible by users.

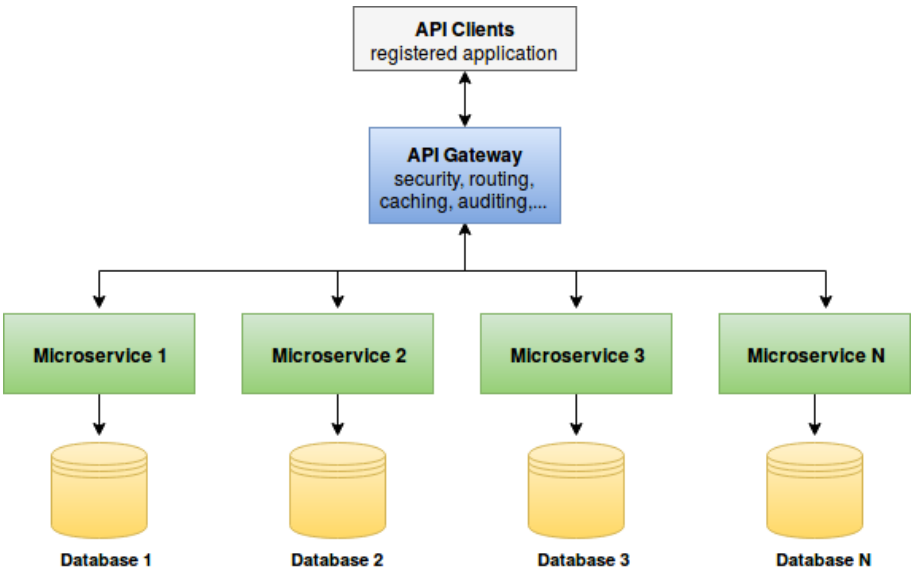


Figure 4.2: Database per service pattern.

API

A carefully-designed RESTFul web API defines the resources, relationships, and navigation schemes that are accessible to client applications. When you implement and deploy a web API, you should consider the physical requirements of the environment hosting the web API and the way in which the web API is constructed rather than the logical structure of the data [53].

4.1.3 Publish-Subscribe Pattern

The Publish/Subscribe (pub/sub) paradigm is a queuing, message-centric approach, which is often referred to as Message Oriented Middleware (MOM) [5]. Individual face-to-face or point-to-point communication is attached to rigid and cumbersome bindings between communicating parties (Figure 4.3, left side). For the development of dynamic large-scale applications, a dedicated Message Oriented Middleware infrastructure is required, where the communication of different entities can be asynchronously mediated (Figure 4.3, right side).

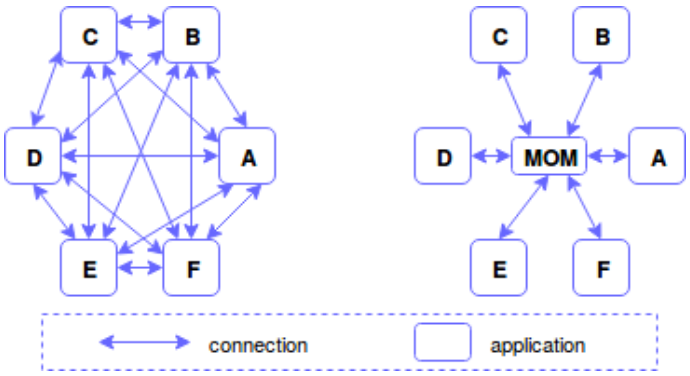


Figure 4.3: Scale issue (left side) and solution (right side).

4.1.4 Saga Pattern

Long lived transactions (LLTs) hold on to database resources for relatively long periods of time, significantly delaying the termination of shorter and more common transactions. A LLT is a saga if it can be written as a sequence of transactions that can be interleaved with other transactions. For instance, when a university student graduates, several actions must be performed before his or her diploma can be issued, the library must check that no books are out, the controller must check that all housing bills and tuition bills are checked, the student's new address must be recorded, and so on. Clearly, each of these real world actions can be modeled by a transaction [21].

The database management system guarantees that either all the transactions in a saga are successfully completed or compensating transactions are run to amend a partial execution. Both the concept of saga and its implementation are relatively simple, but they have the potential to improve performance significantly [21].

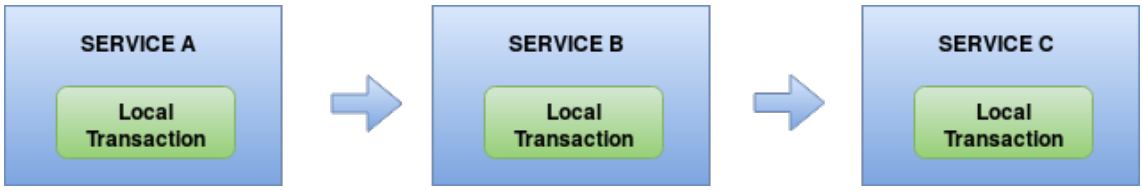


Figure 4.4: Saga pattern example.

Let us use the term saga to refer to a LLT that can be broken up into a collection of subtransactions that can be interleaved in any way with other transactions as shown in Figure 4.4. Each subtransaction in this case is a real transaction in the sense that it, preserves database consistency. However, unlike other transactions, the transactions in a saga are related to each other and should be executed as a (non-atomic) unit any partial executions of the saga are undesirable, and if they occur, must be compensated for [21].

To amend partial executions, each saga transaction T_i, should be provided with a compensating transaction C_i. The compensating transaction undoes, from a semantic point of

view, any of the actions performed by T_i, but does not necessarily return the database to the state that existed when the execution of T_i began [21].

There are two ways of coordinations of sagas which are Choreography and Orchestration [45, 47]. Next, a description for each one is presented.

1. **Choreography**

   When there is not central coordination, each local transaction publishes domain events that trigger local transactions in other services. To exemplify this coordination, consider an e-commerce implementation presented in [47] (see in Figure 4.5).

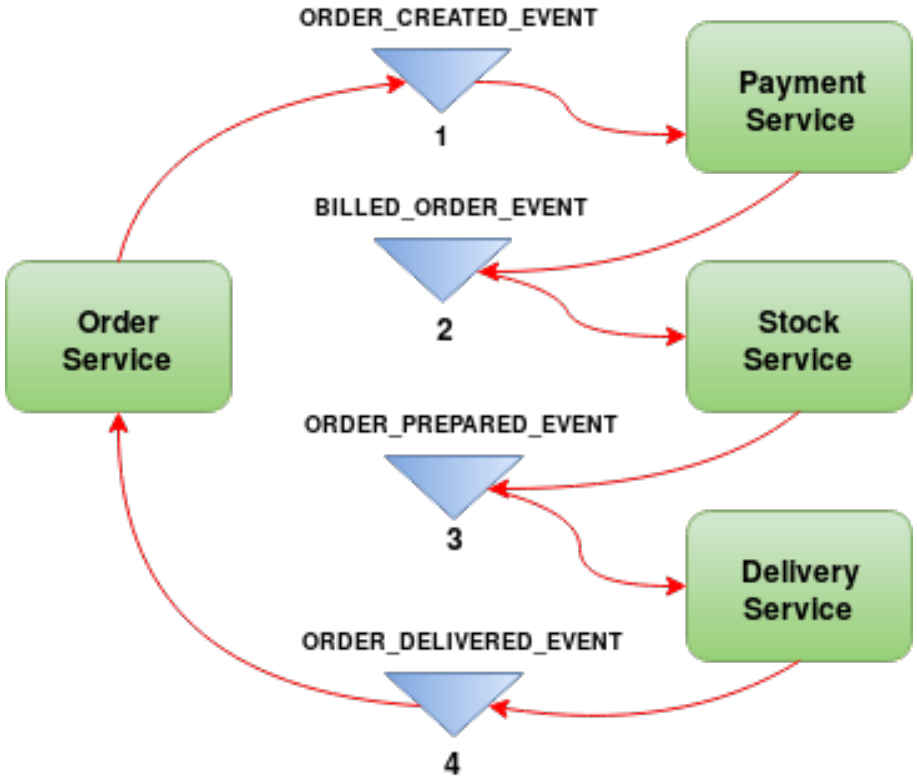


Figure 4.5: A choreography example.

   (a) *Order Service* saves a new order, set the state as pending and publish an event called ***ORDER_CREATED_EVENT***.

   (b) The *Payment Service* listens to ***ORDER_CREATED_EVENT***, charge the client and publish the event ***BILLED_ORDER_EVENT***.

   (c) The *Stock Service* listens to ***BILLED_ORDER_EVENT***, update the stock, prepare the products bought in the order and publish ***ORDER_PREPARED_EVENT***.

   (d) *Delivery Service* listens to ***ORDER_PREPARED_EVENT*** and then pick up and deliver the product. At the end, it publishes an ***ORDER_DELIVERED_EVENT***.

(e) Finally, *Order Service* listens to ***ORDER_DELIVERED_EVENT*** and set the state of the order as concluded.

In the case above, if the state of the order needs to be tracked, Order Service could simply listen to all events and update its state.

Rollbacks in distributed transactions

Rolling back a distributed transaction does not come for free. Normally you have to implement another operation/transaction to compensate for what has been done before. Suppose that Stock Service has failed during a transaction. Let's see what the rollback would look like (see in Figure 4.6):

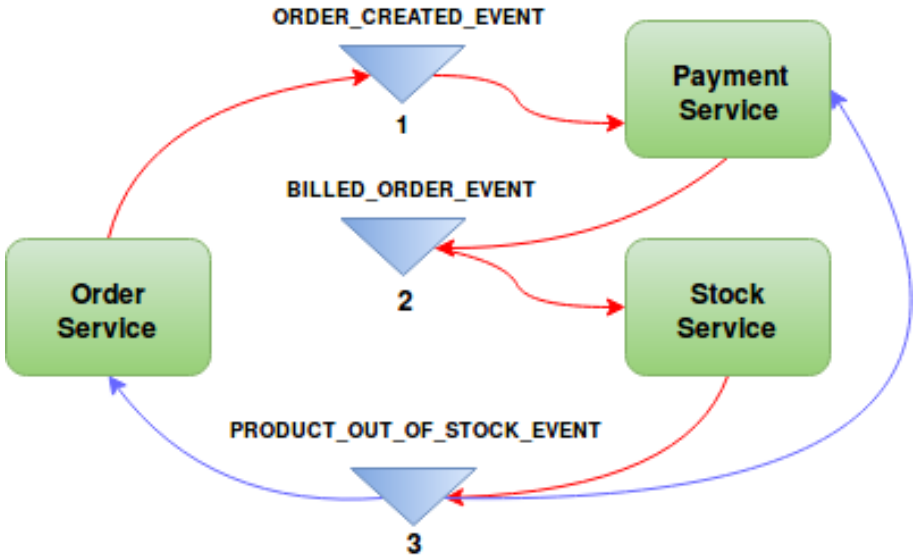


Figure 4.6: Rollback in distributed transactions.

(a) *Stock Service* produces ***PRODUCT_OUT_OF_STOCK_EVENT***;

(b) Both *Order Service* and *Payment Service* listen to the previous message:

- *Payment Service* refund the client.
- *Order Service* set the order state as failed.

Note that it is crucial to define a common shared ID for each transaction, so whenever you throw an event, all listeners can know right away which transaction it refers to.

2. **Orchestration**

When a coordinator service is responsible for centralizing the saga's decision making and sequencing business logic. To exemplify this coordination, consider an e-commerce implementation presented in [48] (see in Figure 4.7).

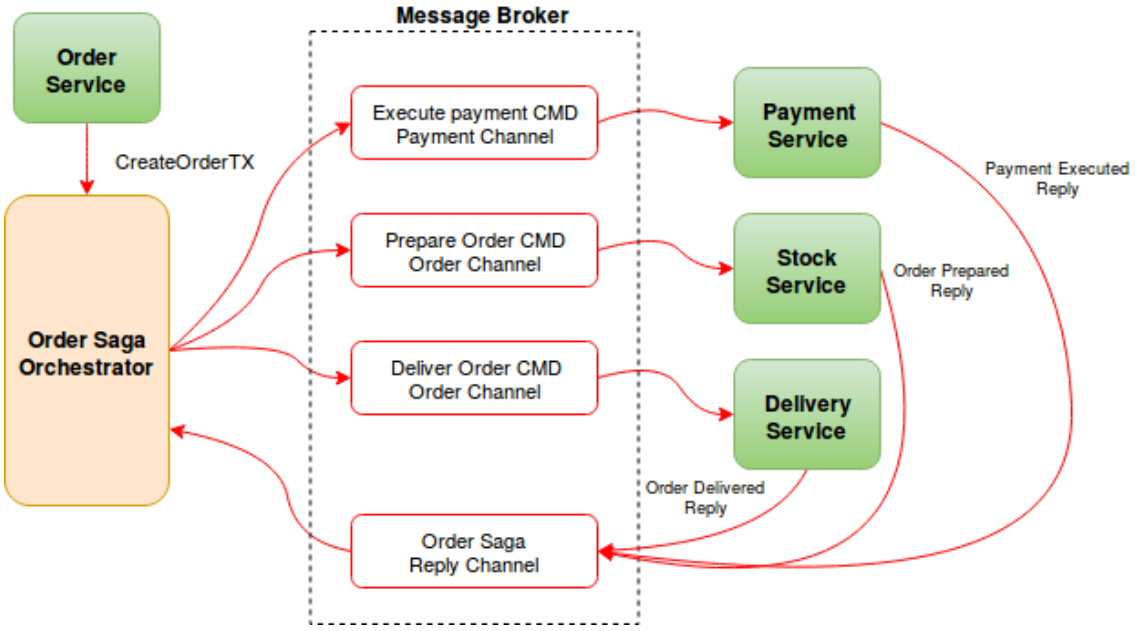


Figure 4.7: An orchestration example.

(a) *Order Service* saves a pending order and asks Order Saga Orchestrator (OSO) to start a create order transaction.

(b) OSO sends an ***Execute Payment*** command to *Payment Service*, and it replies with a ***Payment Executed*** message.

(c) OSO sends a ***Prepare Order*** command to *Stock Service*, and it replies with an ***Order Prepared*** message.

(d) OSO sends a ***Deliver Order*** command to *Delivery Service*, and it replies with an ***Order Delivered*** message.

In the case above, Order Saga Orchestrator knows what is the flow needed to execute a "create order" transaction. If anything fails, it is also responsible for coordinating the rollback by sending commands to each participant to undo the previous operation.

Rolling back in Saga's Orchestration

Rollbacks are a lot easier when you have an orchestrator to coordinate everything (see in Figure 4.8):

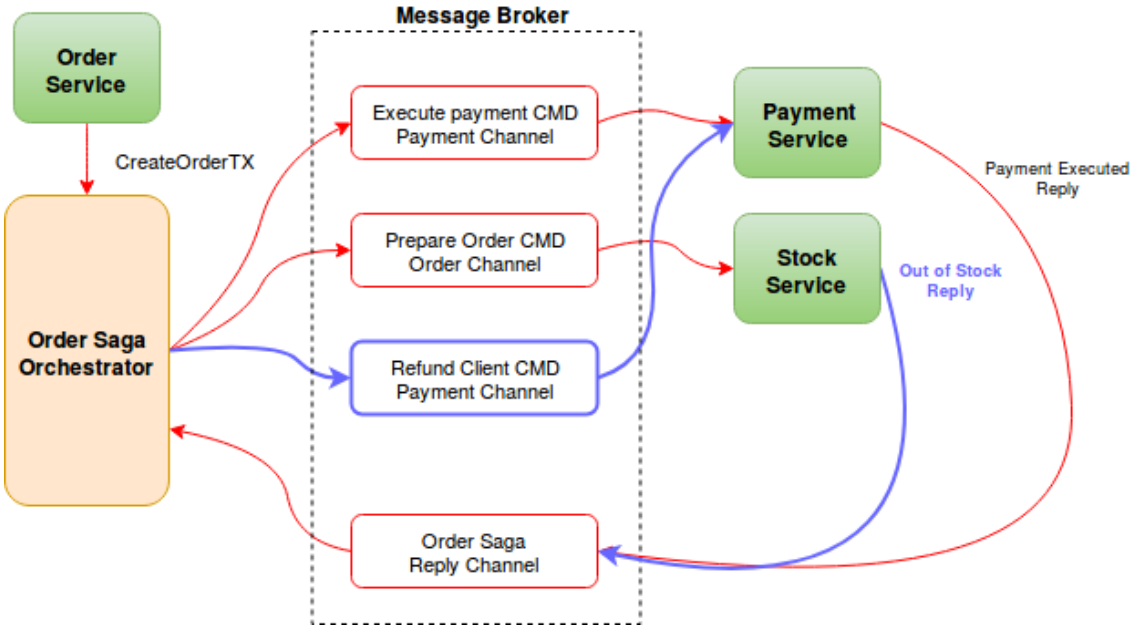


Figure 4.8: Rolling back an orchestration example.

(a) *Stock Service* replies to OSO with an ***Out-Of-Stock*** message;

(b) OSO recognizes that the transaction has failed and starts the rollback

- In this case, only a single operation was executed successfully before the failure, so OSO sends a ***Refund Client*** command to Payment Service and set the order state as failed.

4.1.5 Composition Strategies

A microservice architecture is composed by independent small services, that work together in which each service has a single responsibility [31]. In addition, each service exposes an API by means of events, commands and information in order to establish a communication between them using the coordination of sagas as shown above. However, It is necessary to implement some composition strategies to perform queries that involve one or more services. In [44], the author describes two different patterns for implementing query operations in a microservice architecture.

- The API Composition pattern, which is the simplest and should be used whenever possible. It works by making clients of the services that own the data responsible for invoking the services and combining the results.
- The Command Query Responsibility Segregation (CQRS) pattern, which is more powerful than the API composition pattern but is also more complex. It maintains one or more view databases whose sole purpose is to support queries.

1. **API Composition**

   One way to implement query operations that retrieve data owned by multiple services is to use the API composition pattern. This pattern implements a query operation by simply invoking the services that own the data and combining the results. Figure 4.9 shows the structure of this pattern. It has two types of participants:

   - An API composer, which implements the query operation by querying the provider services.
   - A Provider service, which is a service that owns some of the data that the query returns.

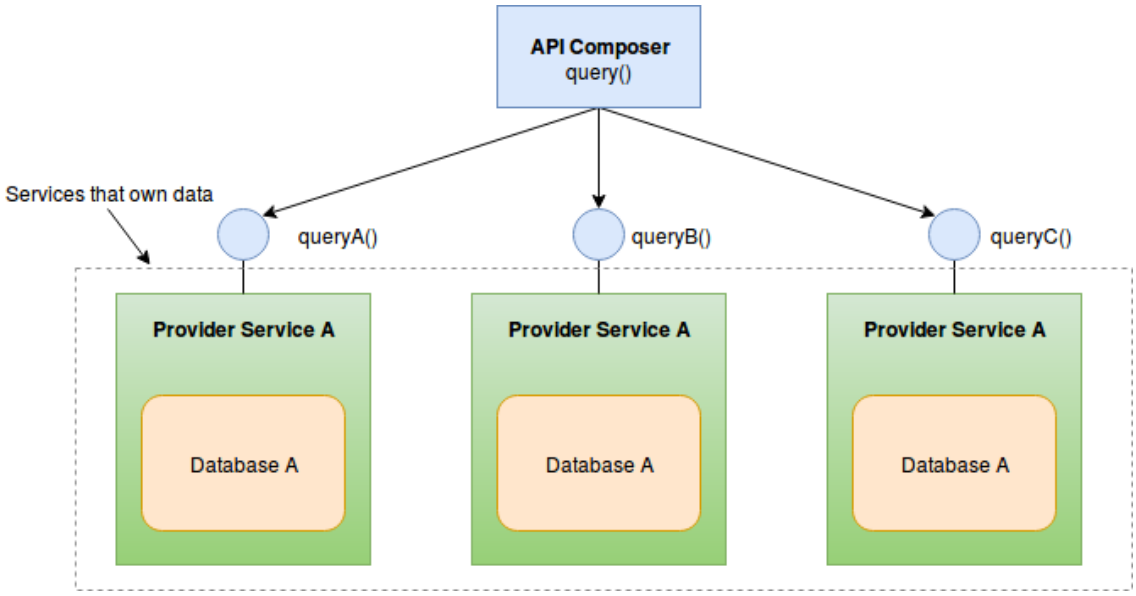


Figure 4.9: API composition.

The API composition pattern is a good way to implement many queries that must retrieve data from multiple services. Unfortunately, it is only a partial solution to the problem of querying in a microservice architecture. That is because, there are multi-service queries the API composition pattern can't implement efficiently. In fact, this pattern has some drawbacks such as increased overhead, risk of reduced availability and lack of transactional data consistency.

2. **Command Query Responsibility Segregation (CQRS) pattern**

   The Command-Query Responsibility Segregation (CQRS) pattern refers to an alternate model for storing and querying information. With normal databases, we use one system for performing modifications to data and querying the data. With CQRS, part of the system deals with commands, which capture requests to modify state, while another part of the system deals with queries [38].

   The change that CQRS introduces is to split that conceptual model into separate models for update and display, which it refers to as Command and Query respectively following the vocabulary of *CommandQuerySeparation*. The rationale is that

for many problems, particularly in more complicated domains, having the same conceptual model for commands and queries leads to a more complex model that does neither well [20].

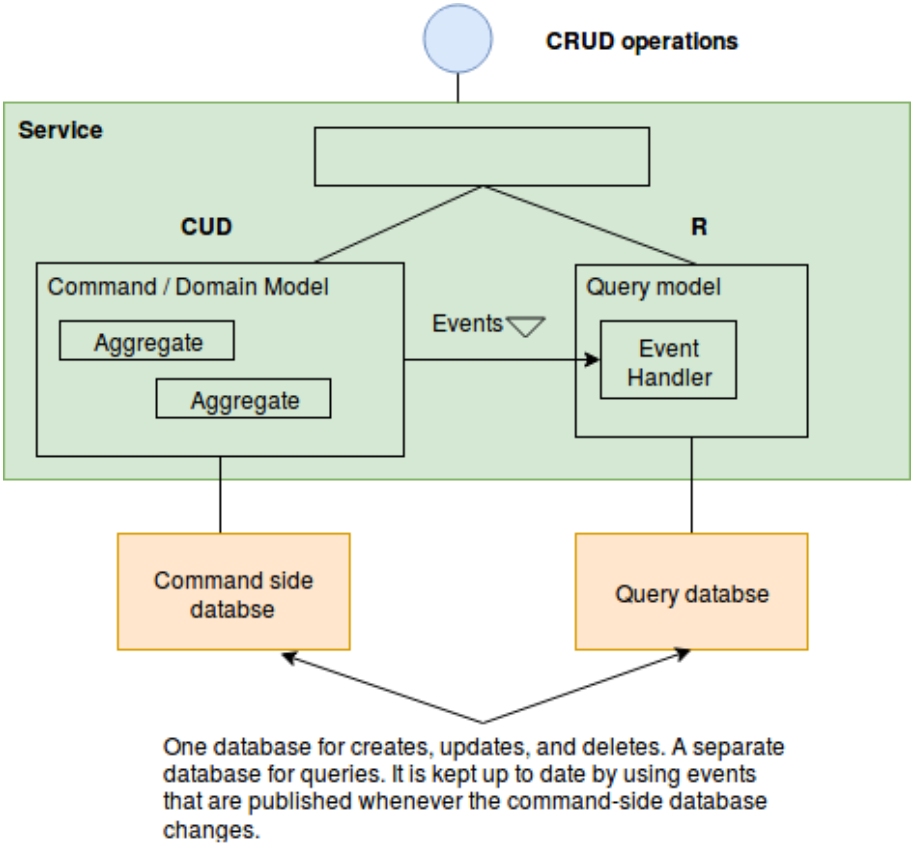


Figure 4.10: Command Query Responsibility Segregation.

CQRS, as the name suggests, is all about segregation, a.k.a. separation of concerns. As Figure 4.10 shows, it splits a persistent data model and the modules that use it into two parts: the command side and the query side. The command side modules and data model implement create, update and delete operations (e.g. HTTP POSTs, PUTs, and DELETEs). The query side modules and data model implement queries (e.g. HTTPGETs). The query side keeps its data model synchronized with the command side data model by subscribing to the events published by the command side [44].

CQRS has both benefits and drawbacks. The benefits of using CQRS are multiple. It enables the efficient implementation of queries in a microservice architecture, it enables the efficient implementation of a diverse queries and it makes querying possible in an event sourcing-based application and improves separation of concerns. On the other hand, CQRS also presents some drawbacks, such as the more complex architecture and the delay of replication [44].

4.2 Deployment of Services

After cutting the business functions into services applying decomposition patterns presented before, comes the next phase, namely deploying the services. Deployment a big monolithic application (says an application with more than 30 services) with all its dependencies and prerequisites could be a matter of hours. Thereby, in [24], authors refer the use of container-based technologies, and in particular its most-known implementation Docker, made deployment of applications possible through several characteristics:

- A Docker image contains all its dependencies, which means a given service can be treated as a black box, only exposing its API in exchange for resources.
- The containers are by default sealed from one to another, which results in guaranteed low coupling, without the high cost associated with virtual machines.
- Docker Compose made it possible to easily deploy any number of services, by composing in a text file an application made of several services.
- Docker Swarm mode allows for complete decoupling of the containers and the machines supporting them. In its recent version 3, Docker Compose allows for Distributed Application Bundles, which define applications made of several services without any dependence other than the presence of a Docker host IP address and access credentials.

4.3 Sagas' Frameworks

In [57] authors compare three frameworks that presently support saga processing, namely Narayana LRA, Axon framework and Eventuate.io. Next, a short description for each one is presented.

1. **Narayana LRA**

   Narayana Long Running Actions is a specification developed by the Narayana team in the collaboration with the Eclipse MicroProfile initiative. The main focus is to introduce an API for coordinating long running activities with the assurance of the globally consistent outcome and without any locking mechanisms.

2. **Axon framework**

   Axon framework is Java based framework for building scalable and highly performant applications. Axon is based on the Command Query Responsibility Segregation (CQRS) pattern. The main motion is the event processing which includes the separated Command bus for updates and the Event bus for queries.

3. **Eventuate.io**

   Eventuate is a platform that provides an event-driven programming model that focus on solving distributed data management in MSAs. Similarly to the Axon, it is based upon CQRS principles. The framework stores events in the MySQL database and it distributes them through the Apache Kafka platform.

   This thesis aims to use a framework that involves most of the MSA patterns that were described above. Clearly, these frameworks need to refer to CQRS pattern as well as Sagas. That is why, Axon and Eventuate frameworks were considered at the beginning. Even though, they have some similarities, this project chose Eventuate for being the clearest and because it involves Event Sourcing pattern, one of the favorites for Netflix as shown in this article[1].

A description of Eventuate Framework is presented as follow.

4.4 Eventuate Framework

In [46] Chris Richardson provides a section with an overview of important Eventuate concepts and how they relate to each other. Figure 4.11 shows the main components of Eventuate such as Entity, EntityType, Event, EventType, Subscriber and Subscription. Each attribute would be described as follow:

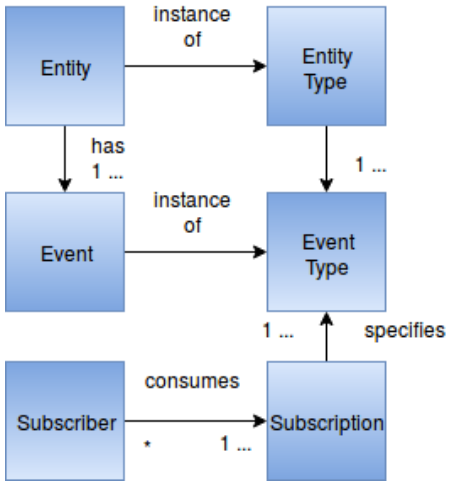


Figure 4.11: The main components of Eventuate.

[1] https://medium.com/netflix-techblog/scaling-event-sourcing-for-netflix-downloads-episode-2-ce1b54d46eec

Entity

A an entity is a business object such as an Order. An entity is an instance of an EntityType. In Domain-Driven Design terms, an entity is an aggregate. Each entity has an ID, unique among instances of the same entity type, which you can choose or allow Eventuate to assign to you. An entity has a sequence of one or more events.

You can perform the following operations on an entity

- Create - Create a new instance of the specified entity type with an initial sequence of events. Or, you can specify a unique ID.
- Update - Append one or more events to the specified entity. Concurrent updates are handled using optimistic locking.
- Find - Retrieve the events for the specified entity.

An Entity type is the type of an entity. It is a String and, typically, is a fully qualified class name.

Event

An event is an object representing the occurrence of something important. Usually an event represents the creation or update of a business object such as an *OrderCreated* event or an *OrderCancelled* event. An event can also represent the attempted violation of a business rule such a *CreditCheckFailed* event.

The key attributes of an event are:

- ID - A unique indicator assigned by the system to every event. The string representation of an event ID is “x-y” where x and y are long values. Event IDs increase monotonically for a given entity.
- Type - An event is an instance of an EventType.
- Sender - Together, the ID and type of the business object that emitted the event.
- *Data** - The serialization of the event’s data. The data is currently JavaScript Object Notation (JSON).

An EventType is the type of event objects. It is a String that is typically the fully qualified event class name.

Subscription

A Subscription is a named, durable subscription to a set of event types. The key attributes of a Subscription include:

- ID - A unique identifier chosen by the application.
- Events - A non-empty map from EntityType to a non-empty set of EventTypes.

A Subscriber is the client of the Eventuate Server that consumes events. The subscriber uses the server's STOMP API to consume events that match the specified subscription. A subscriber can consume multiple subscriptions simultaneously and multiple subscribers can consume the same subscription simultaneously.

The event delivery semantics are as follows:

- Subscriptions are durable, so events are queued until they can be delivered to a subscriber.
- Events published by each entity are delivered to subscribers in order.
- Entities are partitioned across multiple concurrent subscribers for the same subscription.
- Events are guaranteed to be delivered at least once.
- The delivery order of events published by different aggregates is not guaranteed, that is, an *OrderCreated* event (for customer) might appear before a *CustomerCreated* event.

4.5 Problems

The use of the concepts, components, technologies and patterns presented so far gives to the developer the faculty to create, deploy and run MSA systems. Indeed, they make it a lot easier to migrate from monolithic application to a MSA. However, those patterns must be well-used since a poorly decomposed solution will inevitably lead to higher maintenance costs and long-term problems.

One drawback of EDA is that it is potentially error-prone since the developer must remember to publish events. For example, suppose an e-commerce system and consider services such as *Order Service*, *Customer Service* and *Invoice Service*. In case of create an order, *Order Service* needs to validate a *CustomerID* and request for an invoice, so the interaction of all these services is needed. This interaction can be performed by means of events as shown along this chapter. If a service does not publish an event the system will no be consistent enough. This can generate huge problems in the system.

Another drawback or limitation of this approach is that it is challenging to implement when using some NoSQL databases because of their limited transaction and query capabilities. Moreover, it has a different and unfamiliar style of programming and so there is a learning curve [45].

CHAPTER 5

A MODEL-DRIVEN APPROACH

MDE is a branch of software engineering aiming at making the development of complex software systems easier, reducing the time to market, and increasing software quality. To this end, prescriptive and descriptive models are produced and manipulated throughout the development process, by means of model transformation, validation, merge and comparison mechanisms.

MDD is one approach of MDE which is used for enabling rapid, collaborative application development. Since MDD uses visual modeling techniques to define data relationships, process logic, and build user interfaces, model driven software development empowers both developers and business users to rapidly deliver applications without the need for code.

MDD has two important concepts: abstraction and automation. The software application model is defined on a higher abstraction level and then converted into a working application using automated transformation or interpretations. The right model driven development approach leverages model execution at run time, where the model is automatically transformed into a working software application by interpreting and executing the model (removing the need to generate or write code).

This chapter proposes a model-driven approach to construct a model that covers the main concepts, features and components of the MSA systems seen so far. Moreover, a business architecture modeling is carried out to guarantee that all those concepts have a suitable implementation along the phases. Using this model most of the problems mentioned in previous chapters will be reduced and the user will be able to take advantage of the MDD techniques.

5.1 A METAMODEL FOR MICROSERVICES

The meta-model, which describes the domain covered by the language, is the cornerstone of a Domain Specific Language (DSL). A DSL consists of abstract and concrete syntax - whereas the abstract syntax defines the constructs of the language, the concrete syntax

defines the representation of these constructs [8, 33]. The concepts and features about microservices that have been mentioned so far are covered during this section.

Microservices meta-model is conformed by a set of class entities that are related to each other, which allows to split an application into small services such as the Y axis of Scale Cube definition [44]. This is essential to work and scale independently, to offer different functionalities regarding its own models and to provide an API that allows the communication between services.

The main class entity in this meta-model is called Service. It contains some attributes such as *name, fullname, description, shortname* and *port*. The first 4 attributes will be taken to create independently project files during the code generation phase while *port* attribute refers to the port's number in which the service will be deployed. There are two kinds of services presented by this meta-model as shown in Figure 5.1, one is called *AggregateService* which contains the necessary elements to create a service and the other is called *ViewService* which is used to duplicate aggregate services by means of events.

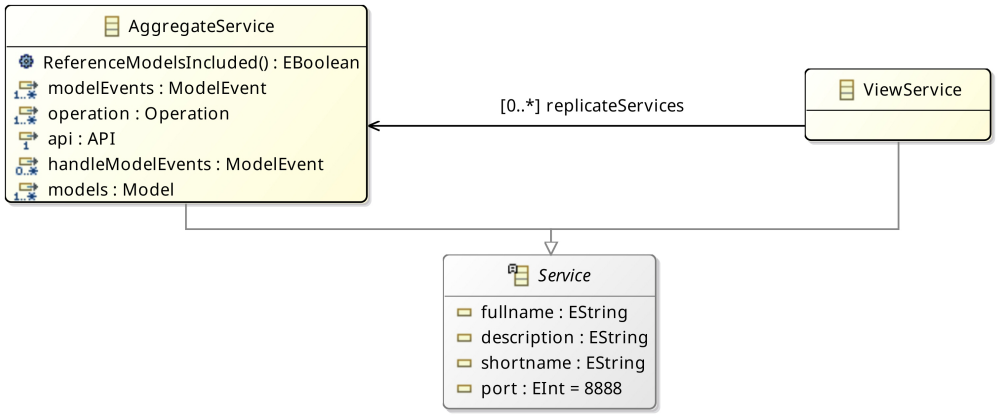


Figure 5.1: Service class entities.

AggregateService and *ViewService* were thought to refers an alternate model for storing and querying information such as Command Query Responsibility Segregation (CQRS). With CQRS, part of the system deals with commands, which capture requests to modify state, while another part of the system deals with queries [38].

AggregateService is composed by an *API*, *Models*, *ModelEvent*, *HandleModelEvents* and *Operation*.

Model class entity has a set of attributes as shown in Figure 5.2. Each attribute is composed by *name, isMany, isId,* and *isGenerated*. The *isMany, isId* and *isGenerated* attributes are boolean variables to reference if it is a list, an identificator or a generated value respectively. An attribute can be either a *PrimitiveTypeAttribute* such as int, boolean, char, etc, or a *ReferenceAttribute* that refers to other attribute of the Model created previously. *PrimitiveTypeAttribute* also has a relationship with Model that might be activated indicating that a primitive value will be converted to a reference attribute in a *ViewService*. For

instance, a String attribute as *customerId* in Order Service will be converted to a reference attribute in a *ViewService*, in this example the attribute will do reference to Customer class.

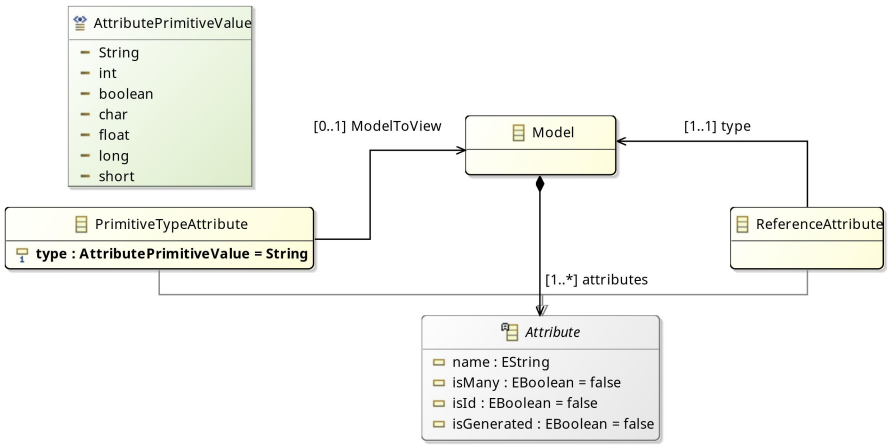


Figure 5.2: Model class entity.

The API exposes a set of events, commands and information as shown in Figure 5.3. The Event class entity is used to perform coordinations of sagas using choreography while Command class entity is used to perform orchestration. A Command has 3 different types that are compensate, invoke and reply. Each service uses the Information class entity to share the data about its models.

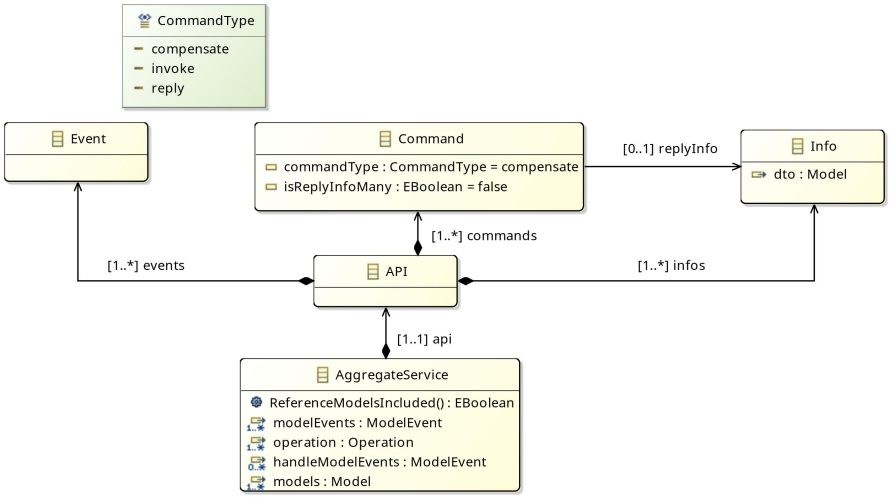


Figure 5.3: API class entity.

Each aggregate service may have a set of models referenced in the *models* attribute. They represent the knowledge and activities of a particular application domain. That domain is taken from some of the decomposition strategies mentioned previously in chapter Patterns and Frameworks.

ModelEvent refers to one or more models of a service that will be allowed to publish events to the system in order to inform to its subscribers that a business object of the domain model has been modified.

The *HandleModelEvents* attribute allows services handle a set of events exposed by other services. This is essential to complete choreography workflow between services.

Operation has a set of attributes that describe operations of a service such as *operationType, publish, model, isMethodController* and *Saga* as shown in Figure 5.4. The *operationType* attribute refers to a CRUD operations (i.e. Create, Retrieve, Update and Delete), *publish* attribute is related to a class event exposed by an API of a service which is published once an aggregate operation (i.e. Create, Update and Delete) is performed, model attribute refers to a model of the service that will be modified through operations, *isMethodController* is a boolean attribute indicating whether the operation will exposed through the controller of a service or not, and *Saga* is used to describe the steps of a transaction in an operation that involves two or more services. Each step of a saga is associated with a command which is a class exposed by an API.

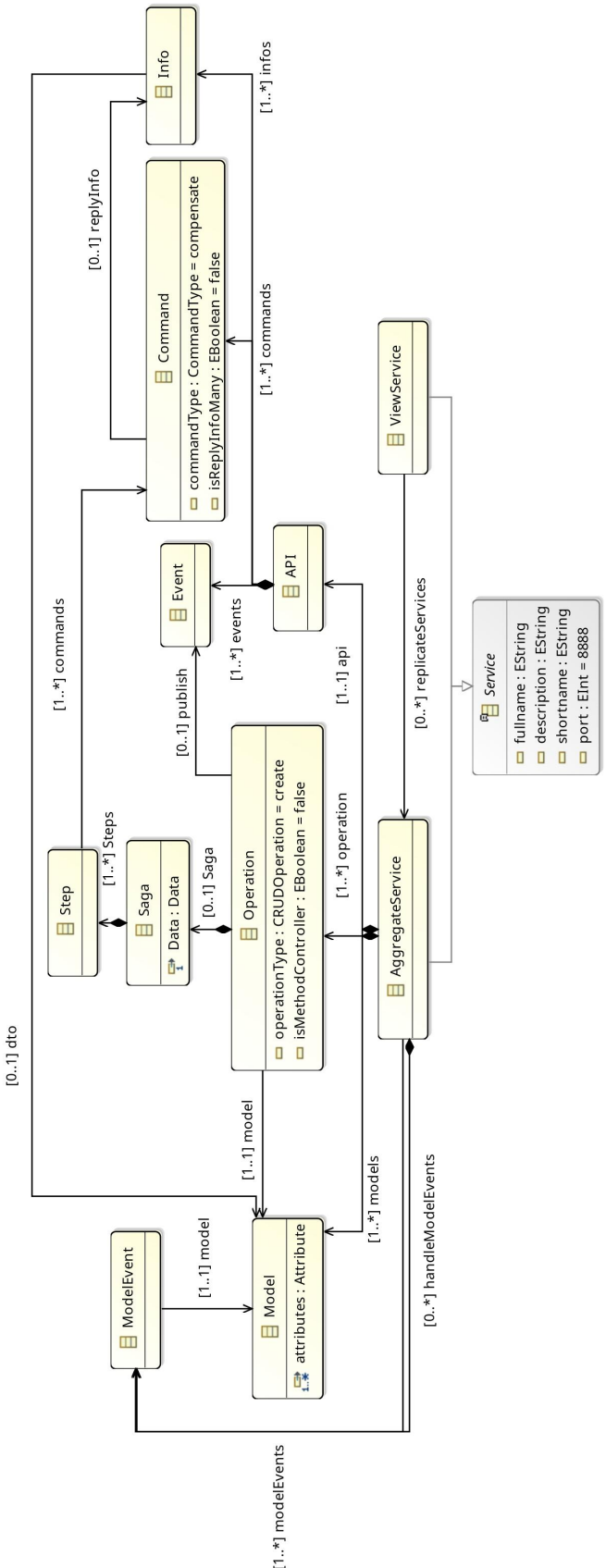


Figure 5.4: Abstract syntax of the MSA metamodel.

The concrete syntax of this metamodel shows that a service can be either an *Aggregate-Service* or a *ViewService*. A *ViewService* has a relationship with *AggregateService* in order to perform multiple replicas of it. An *AggregateService* has a set of *Models*, *ModelEvents* and an *API* in which *Events*, *Commands* and *Information* will be exposed to other services. Each operation of an *AggregateService* is in relationship with *Saga* in case of performing transactions which involve two or more services. A *Saga* has a set of steps related with commands.

The validation of this metamodel has been done using an Object Constraints Language (OCL) which is a declarative language for describing rules (see Listing 5.1). Those rules are implemented through OCLinEcore[1] tool (i.e. an editor of OCL constraints). The rules ranging from reserved port numbers for *AggregateService* and *ViewService* to validate that an event is published correctly for aggregate operations are presented as follows.

1. The next information of a *Service* must be unique and mandatory.
   - Name
   - Port (numeric field, can only accept 4 numbers i.e. 8888)
   - Fullname
   - Shortname
   - Reserved port numbers by Eventuate Framework components: 27017, 8099, 3306, 9092, 2181, 2888, 3888, 5020.
2. The name of a *Model* must be unique and it can be composed only of characters (i.e. not by numbers).
3. A *Model* must not contain attributes with the same name.
4. Each *Model* must contain only one *Id*. This *Id* is the unique attribute that can activate "Is Generated" property. Moreover, *Id* can't activate "Is Many" attribute.
5. *Event* names, *Commands* names and *Info* names that belong to the same API must be unique and not empty.
6. *Operation* names that belong to the same *Aggregate Service* must be unique.
7. If a *Command* is typed "reply", the field "Reply Info" must refer to the "Info" that belongs to the same API.
8. *ModelEvents* names must be unique for each *Aggregate Service*.
9. Each *Operation* that is typed as "create", "update" or "delete" must publish an *Event*. This *Event* must belong to the same operation API.
10. The name of the *Saga* must be unique for each *Aggregate Service*.

[1] https://wiki.eclipse.org/OCL/OCLinEcore

```
1  ServicePortMustBeUnique = MicroserviceArchitecture.allInstances().services->select(s : Service |s.port = self.port)->
       size() = 1
2
3  ServiceFullnameMustBeUnique = MicroserviceArchitecture.allInstances().services->select(s :Service | s.fullname = self.
       fullname)->size() = 1
4
5  ServiceShortNameMustBeUnique = MicroserviceArchitecture.allInstances().services->select(s :Service | s.shortname =
       self.shortname)->size() = 1
6
7  ServicePortMustBeUnique = MicroserviceArchitecture.allInstances().services->select(s : Service | s.port = self.port)->
       size() = 1
8
9  ServiceReservedPortNumbers = MicroserviceArchitecture.allInstances().services->select(s : Service
10 | (s = self and (s.port.toString() = '27017' or s.port.toString() = '8099' or s.port.toString() = '3306' or
11 s.port.toString() = '9092' or s.port.toString() = '2888' or s.port.toString() = '3888' or s.port.toString()
12 = '5020')))->size() = 0
13
14 MustHaveFullname = fullname <> ''
15
16 FullnameMustNotContainWhiteSpace = fullname.toString().characters()->select(c : String | c = ' ')->size() = 0
17
18 MustHaveShortname = shortname <> ''
19
20 ShortnameMustNotContainWhiteSpace = shortname.toString().characters()->select(c : String | c = '')->size() = 0
21
22 ModelNameMustBeUnique = MicroserviceArchitecture.allInstances().models->select(s : Model |s.name = self.name)->size()
       = 1
23
24 ModelMustBeContainedInOneAggService = MicroserviceArchitecture.allInstances().services->select(s : Service | s.
       oclIsTypeOf(AggregateService)).oclAsType(AggregateService).models->select(m : Model | m = self)->size() = 1
25
26 ...
```

Listing 5.1: OCL Constraints

Besides this approach, there are other tools which involve the development of MSAs. For instance, in [62] authors propose a tool to create microservices based on Maven archetypes while in [56] authors propose a graphical modeling language for the development of an MSA. As well as this thesis, they present a metamodel that aims to cover most of the concepts, features and technologies of an MSA. However, they do not consider yet an important component of an MSA such as container, which makes easier aspects related to infrastructure, network, security, among others. Likewise, CQRS pattern is omitted. This pattern is essential for MSA systems which adopt EDA approach. Thereby, the code generation won't be enough in some cases since these features are missed.

5.2 Code Generator

The automation of code will be explained in three sections. The first describes the components of a service that are needed to generate an MSA. The second presents some features of the target platform. Finally, a description of the organization of the generation scripts is shown.

5.2.1 Service Components

Each service requires a set of components (see Figure 5.5) to conform an independent application. Indeed, most of these components are used in traditional Model-View Controller (MVC) developments, changing only the use of events, commands, sagas and handlers which are features of EDA approach. Table 5.1 shows a description for each component of a service.

Service's Components	Description
Application Program Interface (API)	Each service must define an API in which a set of events, commands and information will be exposed to allow the interaction with other services. The API component is quite used to carry out the coordination of sagas.
Model	Each service is composed by a set of models that represent the business logic. The model component is used to define both domain event publisher and a repository for every model.
Data Access Object (DAO)	DAO is a traditional pattern used to separate low level data accessing or operations from high level business services. Each service has a DAO interface which defines the standard operations to be performed on a model object.
Implementation (Impl)	Every operation defined in the DAO component, which is an interface, will be implemented through this component. In other words, each implementation extends from its DAO interface.
Controller	This component is used to expose a Rest API where all service operations can be carried out. In addition, external queries will be allowed through this component.
Command Handlers	This component allows the services to manage the operations or requests provided by saga's orchestrator.
Messaging	This component is quite similar to Command Handlers but it aims to manage events through operations or requests provided by saga's choreography. This is only used for services that require to handle events of other services.
Saga	This component allows the implementation of transactions that involve two or more services. Each transaction will be carried out by means of commands as shown in previous sections. In case that a service does not contain any operation this component is omitted.

Table 5.1: Description of the Service's components.

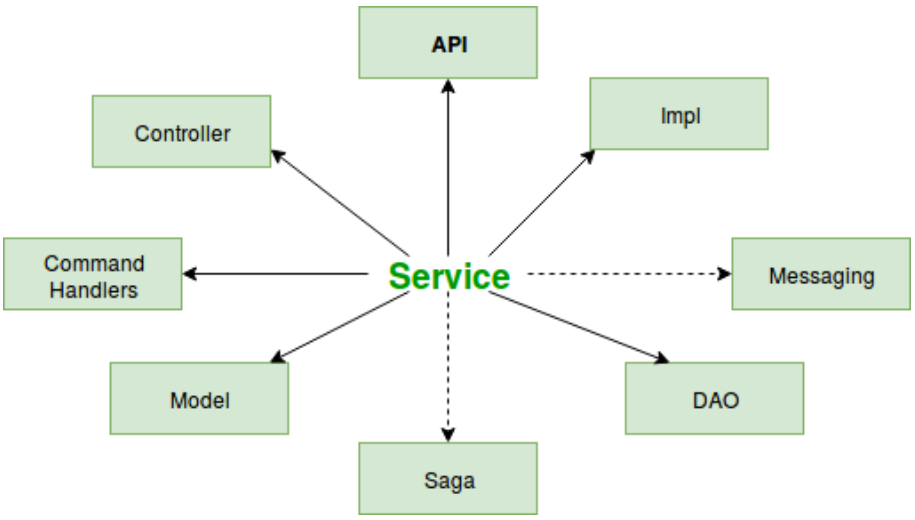


Figure 5.5: Components of a service.

5.2.2 Target Platform

This thesis adopted Acceleo, as a code generation alternative. It is an implementation of the MOF [1] model-to-text language (MTL) standard that provides a flexible and simple environment to design and develop a variety of code generators, using simple and standard templates. Moreover, it is compatible with the Eclipse environment. Eclipse [13] is a cross-platform, mature, extensible, and plug-in based development platform that supports multiple languages and tools. It allows different tools to be unified for seamless development and maintenance of software systems.

As a result, an MSA system based on Eventuate Framework as well as the patterns and strategies shown in previous chapters is generated. This system is represented by microservices that expose multiple operations through APIs. Each microservice corresponds a stand-alone application based on Spring Boot[2] which requires a very little configuration to run. In addition, a container-based technology as Docker is used to build, ship, deploy and run the generated microservices.

In particular, there are multiple open source components which have been developed to enrich the use of MSA systems. Components such as Swagger UI, Spring Boot Admin and Jenkins are generated in this phase. Next, a brief description for each one is mentioned.

Swagger UI

Swagger UI[3] allows anyone - be it your development team or your end consumers - to visualize and interact with the API's resources without having any of the implementation logic in place. It's automatically generated from your Swagger specification, with the

[2]https://spring.io/projects/spring-boot
[3]https://swagger.io/swagger-ui/

visual documentation making it easy for back end implementation and client side consumption.

Spring Boot Admin

Spring Boot Admin[4] is a simple application to manage and monitor your Spring Boot Applications. The applications register with our Spring Boot Admin Client (via http) or are discovered using Spring Cloud (e.g. Eureka). The UI is just an Angular.js application on top of the Spring Boot Actuator endpoints. In case you want to use the more advanced features (e.g. jmx-, log level-management), Jolokia must be included in the client application.

Jenkins

Jenkins[5] is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software. It can be installed through native system packages, Docker, or even run standalone by any machine with a Java Runtime Environment (JRE) installed.

A continuous delivery (CD) pipeline is an automated expression of your process for getting software from version control right through to your users and customers. Every change to your software (committed in source control) goes through a complex process on its way to being released. This process involves building the software in a reliable and repeatable manner, as well as progressing the built software (called a "build") through multiple stages of testing and deployment.

5.2.3 Generation Flow

As microservices, the implementation was divided into several parts in which each addresses a specific concern. In particular, there is a file code generator for each component of a service as shown in Figure 5.6 and the execution of these files is done by a Java class named *Runner.java*. It creates an MSA system ready to run. Table 5.2 shows a brief description for each Acceleo template.

[4]https://codecentric.github.io/spring-boot-admin/1.4.3/
[5]https://jenkins.io/doc/book/pipeline/

Template (.mtl)	Description
servicemodelGenerate	This template generates a common package named *service-model* that works as repository of APIs exposed by aggregate services. This package is not a stand-alone application. Instead, it stores commands, events and infos classes for each aggregate service (see Figure 5.3) to allow the communication between services. In addition, a common class named *Channels.java* is also generated by this template. It establishes the name of a channel for each service, this is a requirement for EDA systems.
modelGenerate	This template does an extraction of the models that belong to each service in order to generate classes with their respective attributes. Moreover, a *DomainEventPublisher* class is created to allow operations publish events when it's necessary. A partial code is shown in Listing 5.2.
commandHandlerGenerate	This template creates a package called CommandHandlers. Inside, it generates a class in which all commands related to aggregate services will be managed. In particular, this class will receive messages from different sagas' Orchestrations.
eventHandlerGenerate	This template creates a package called Messaging. Inside, it generates a class in which all events associated with aggregate services will be managed. In particular, this class will receive messages from different sagas' Choreography.
daoGenerate	This template creates a package named DAO for each aggregate service. Moreover, it does an extraction of the operations that belong to each service to generate interfaces. Those interfaces will be used for the next template.
implGenerate	This template creates a package called impl for each aggregate service. Inside, it generates classes that implement the operations listed in DAO. The user just needs to develop the logic for each operation.
controllerGenerate	This template creates a package named controller in which there will be a class with the implementation of the operations associated with a service. This operations will be shown in a Swagger UI.
saga	This template creates a package called Saga in which there will be all saga operations associated with a service. The dotted lines shown in Figure 5.6 refer that this package will not appear in services without saga operations. A partial code is shown in Listing 5.3.

Table 5.2: Description of the Acceleo templates.

```
[comment encoding = UTF-8 /]
[module modelGenerate('http://it.univa.disim.micro')]

[template public generateElement(aMicroserviceArchitecture : MicroserviceArchitecture)]
[comment @main/]

[for (s : Service | aMicroserviceArchitecture.services)]
[if (s.oclIsTypeOf(micro::AggregateService))]
  [for (model : Model | s.oclAsType(micro::AggregateService).models)]
  [comment model classes /]
    [file (s.name+'/src/main/java/org/'+aMicroserviceArchitecture.name.toLower()+
    '/' + s.fullname.toLower()+'/model/'+model.name+'.java', false, 'UTF-8')]

package org.[aMicroserviceArchitecture.name.toLower()/].[s.fullname.toLower()/].model;
...
@Document(collection="[model.name/]s")
public class [model.name/]{

  [for (attribute : Attribute | model.attributes)]
    [if (attribute.isId)]
    @Id
      [if (attribute.isGenerated)]
    @GeneratedValue(strategy = GenerationType.IDENTITY)
      [/if]
    private [attribute.oclAsType(micro::PrimitiveTypeAttribute).type/] [attribute.name.toLower
         ()/];
    [else] ...
    [/if]
  [/for]
  ...
}
  [/for]
  ...
[/if]
[/for]
[/template]
```

Listing 5.2: Code generator of models

```
[comment encoding = UTF-8 /]
[module sagaGenerate('http://it.univa.disim.micro')]

[template public generateElement(aMicroserviceArchitecture : MicroserviceArchitecture)]
[comment @main/]

[for (s : Service | aMicroserviceArchitecture.services)]
  [if (s.oclIsTypeOf(micro::AggregateService) )]
    [if (s.oclAsType(micro::AggregateService).eAllContents(Operation).eAllContents(Saga)->size
         () > 0 )]
      [for (saga : Saga | s.oclAsType(micro::AggregateService).eAllContents(Operation).
           eAllContents(Saga))]
        [file (s.name+'/src/main/java/org/'+aMicroserviceArchitecture.name.toLower()+'/'+s.
             fullname.toLower()+'/saga/'+saga.name.toLower()+'/'+saga.name+'Saga.java', false,
              'UTF-8')]
package org.[aMicroserviceArchitecture.name.toLower()/].[s.fullname.toLower()/].saga.[saga.
     name.toLower()/];
...

@Component
public class [saga.name/]Saga implements SimpleSaga<[saga.name/]SagaData>{
  ...

    this.sagaDefinition =
        step()
          [for (step : Step | s.oclAsType(micro::AggregateService).eAllContents(Operation)
              .eAllContents(Saga)->select(s | s.name.equalsIgnoreCase(saga.name)).Steps)]
            [if (step.eContainer(Saga).name.equalsIgnoreCase(saga.name))]
              ...
            [/if]
          [/for]
        .build();
  } ...
    [/if]
  [/if]
[/for]
[/template]
```

Listing 5.3: Code generator of sagas

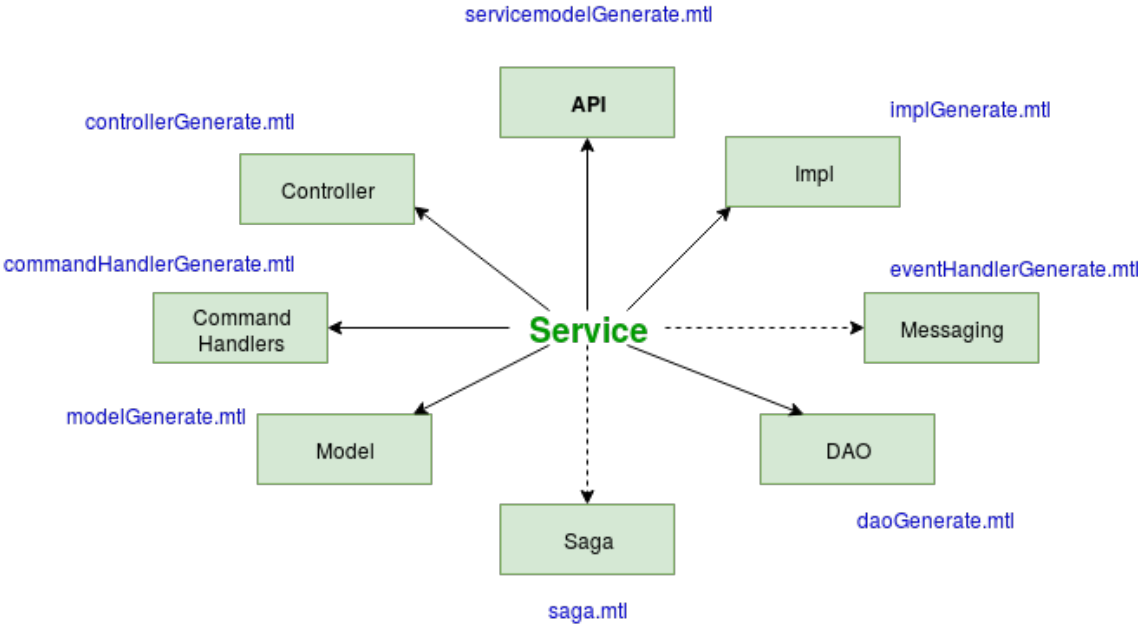


Figure 5.6: Acceleo - File code generators.

5.3 Use Case: An Order System

Consider a simple e-commerce system that involves three aggregate services and one view service as shown in Figure 5.7. The three aggregate services are called Customer Service, Order Service and Invoice Service respectively, while a view service is called Order View Service. Even though, this is a simple example of an e-commerce system, all the patterns described in previous sections such as database per service, publish-subscribe, sagas, among others, have been included. In addition, each service exposes an API by means of events, commands and information in order to establish a communication between them.

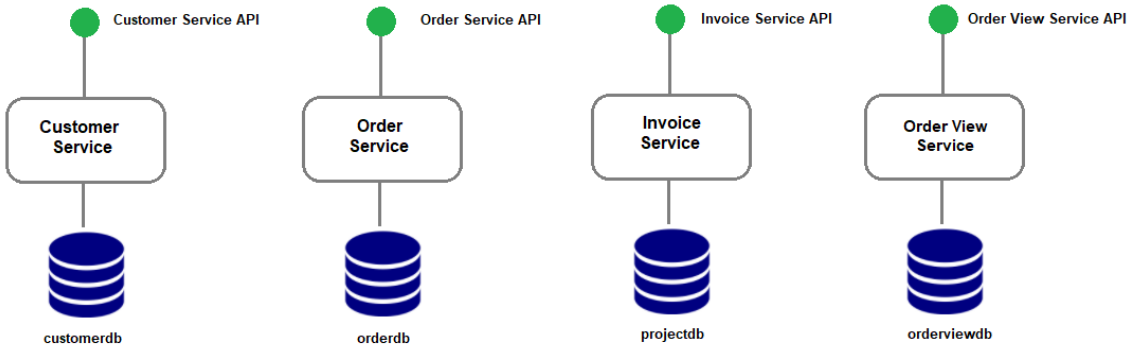


Figure 5.7: A simple e-commerce example.

A description for every service of this e-commerce example is presented as follows:

Order Service

- Create an order requires the following operations.
  - Validate Customer ID.
  - Request an invoice.
  - Complete the order.
  - Reject order in case of failure.
- Update an order requires the following operations.
  - Validate Customer ID.
  - Validate Invoice ID.
  - Update the order.
- Delete an order.
- Exposes an API using Swagger UI to carry out its operations.

Customer Service

- CRUD operations are performed.
- Exposes an operation to validate a Customer ID.
- Exposes an API using Swagger UI to carry out its operations.

Invoice Service

- Exposes an operation to create and reply an invoice.
- Exposes an operation to reject an invoice.
- Exposes an API using Swagger UI to get information about its invoices.

Order View Service

- Handles the events of Order Service, Customer Service and Invoice Service.
- Updates a common database.
- Exposes an API using Swagger UI to get information about all services.

The first step is to describe the components that conform the ecommerce example using the metamodel presented in section 5.1. The end-user must identify in advance the set of models, commands and events to use, relationships between services as well as the mechanism to execute Sagas. Clearly, Order Service needs to perform coordination of Sagas to carry out its operations since they are involved with Customer Service and Invoice Service. Figure 5.8, shows one representation of this MSA model where most of the operations are presented (i.e. the Update Order and Delete Order operations that belong to the Order Service are omitted).

The green boxes represent the aggregate services such as Order Service, Customer Service and Invoice Service, while an orange box represents the unique view service (i.e. Order View Service) presented in this example. Each aggregate service has an API where they expose a set of events, commands and information. The latter has been omitted in the model. Each aggregate service also has a model, a model event and a set of operations. Each operation that changes the status of a model publishes an event that can be handled by other services.

Order Service has an operation called *createOrder* that creates a new order with an status as pending or a variable completed as false, then it uses a Saga's orchestrator. This orchestrator or transaction is composed by 4 steps. Each step is related with a command exposed by an API. The first step is related to *CompensateOrder* command that rejects the order in case of failure. The second step is associated with *ValidateCustomer* command that validates whether a Customer ID exists or not. The third step is related to *RequestInvoice* and *CompensateInvoice* commands in which a request for an invoice is asked. In case of failure *CompensateInvoice* command will be activated to reject the created invoice. The last step is associated with *CompleteOrder* command that changes the state of the order to completed which means that the transaction has been successfully terminated.

On the other hand, Order View Service represented by an orange box, handles the events of all the services and stores the information in a common database. Clearly, the CQRS pattern is carried out using this service.

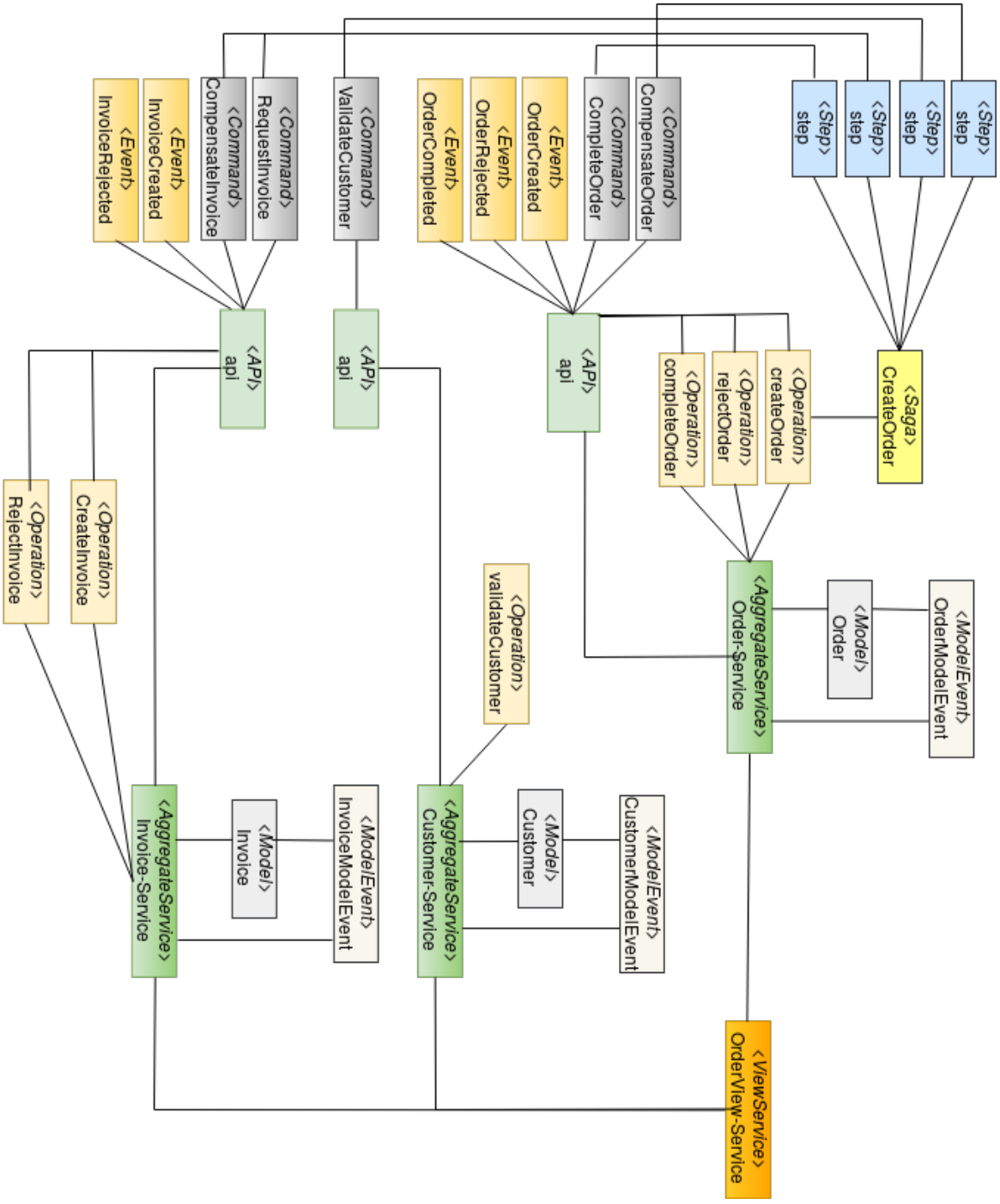


Figure 5.8: E-commerce example.

The second step refers to the validation of the MSA model and then the generation of file codes for each service's component as shown in section 5.2.1. In fact, this process guarantees a system that can be builded and deployed without errors. The coordination of sagas will be presented as code. For instance, Figure 5.9 shows a snippet code that represents the saga's orchestrator presented in Order Service. Clearly, it shows that the *CreateOrderSaga* has 4 steps, the same displayed in Figure 5.8. The entire code is located in this github repository[6].

```
@Component
public class CreateOrderSaga implements SimpleSaga<CreateOrderSagaData>{

    private static final Logger log = LoggerFactory.getLogger(CreateOrderSaga.class);

    private SagaDefinition<CreateOrderSagaData> sagaDefinition;

    public CreateOrderSaga(InvoiceServiceProxy invoiceService, OrderServiceProxy orderService, CustomerServiceProxy customerService){

        this.sagaDefinition =
                step()
                    .withCompensation(orderService.rejectOrderCommand, this::makeRejectOrderCommand)
                .step()
                    .invokeParticipant(customerService.validateCustomerByOrder, this::makeValidateCustomerByOrder)
                .step()
                    .invokeParticipant(invoiceService.requestInvoiceCommand, this::makeRequestInvoiceCommand)
                    .onReply(InvoiceInfo.class, this::handleRequestInvoiceCommand)
                    .withCompensation(invoiceService.compensateInvoiceCommand, this::makeCompensateInvoiceCommand)
                .step()
                    .invokeParticipant(orderService.completeOrderCommand, this::makeCompleteOrderCommand)
                .build();
    }

    @Override
    public SagaDefinition<CreateOrderSagaData> getSagaDefinition() {
        return sagaDefinition;
    }
```

Figure 5.9: Saga DSL.

There are two ways to build and run this MSA system. One is carried out using the Jenkins files generated during this process such as *EventuateComponentsJenkinsfile* and *ServicesJenkinsfile*. The first one refers to the Eventuate Framework components and the second corresponds to the continuous delivery pipeline of this e-commerce example services as shown in Figure 5.10. The second way is to execute a docker compose file (also generated) that defines multi-container docker applications, which means, the e-commerce services. It can be performed following the *README.md* file which contains the installation steps, prerequisites and services information.

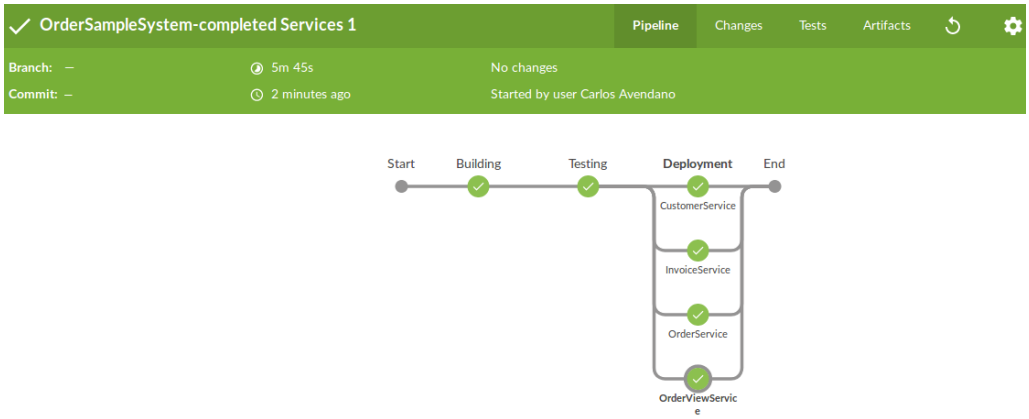


Figure 5.10: Jenkins pipeline of E-commerce example.

[6]https://github.com/carloselpapa10/OrderSampleSystem-GeneratedCode

Although, a great part of the code is generated, the end-user needs to complete the business logic of the system. The complete code is located in this github repository[7].

Moreover, each service of the system, exposes an API using Swagger UI as shown previously in section 5.2.2, which uses a visual documentation that makes easy the implementation of the operations and provides a client side consumption. Figure 5.11 shows the Swagger UI of Order Service.

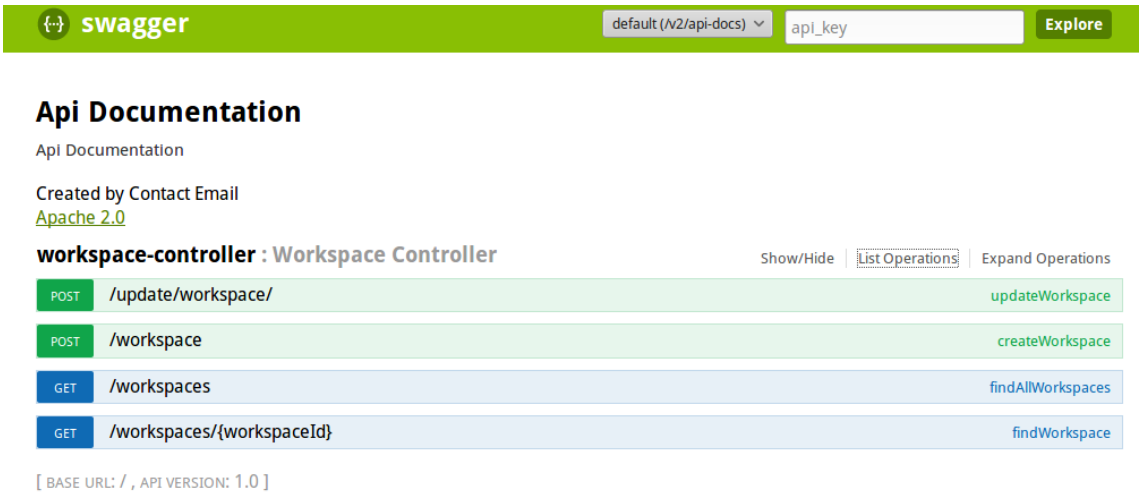


Figure 5.11: Swagger UI of Order Service.

Finally, an interesting component called Spring Boot Admin or Admin Server is used to manage and monitor the metrics for every service presented in the model. Some of the metrics are disk space, memory consumption, garbage collection, servlet containers, databases, logs, status (i.e. up or off-line), among others. Figure 5.12 shows the services of the e-commerce example through this useful component.

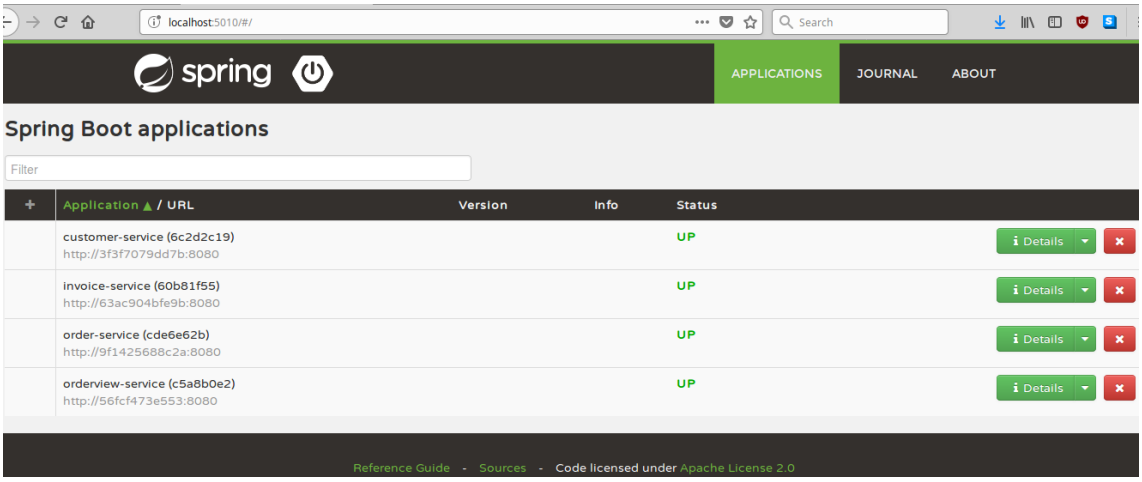


Figure 5.12: Spring Boot Admin.

[7]https://github.com/carloselpapa10/OrderSampleSystem-completed

5.4 RESULTS

The use of MDE approaches to address the complexity of microservices architecture has come with benefits. Indeed, most of the drawbacks mentioned in previous chapters have been resolved. Below are some of these problems and how they are being tackled.

"Distributed systems are complex"

A challenge with using the microservice architecture is that developers must deal with the additional complexity of creating a distributed system. Developers must use an inter-process communication mechanism. Implementing use cases that span multiple services requires the use of unfamiliar techniques. However, using the model-driven approach proposed during this chapter, great part of the code is generated through the MSA metamodel explained in section 5.1. In fact, Cloc[8] app was used to compare both the generated and the complete code of the previous e-commerce sample. The result was grateful, the 87% of the java code was generated, which means, the coordination of Sagas, the event and command handlers, configurations, APIs and others were already builded for this tool. Thus, the complexity to develop the MSA application was minimum. Table 5.3 shows the comparison between the codes.

E-commerce App	Language	Files	Code
Generated Code	Java	95	2146
	Maven	8	754
	YAML	5	392
Complete Code	Java	95	2467
	Maven	8	754
	YAML	5	392

Table 5.3: E-commerce application comparitions.

"Event-Driven Architecture is potentially error-prone since the developer must remember to publish events"

EDA-based applications need to publish events for each operation that modifies a business object (i.e. such as create, update and delete). Otherwise, the system will not be consistent. This is one of the major problems using this approach. However, thanks to this tool, the developer won't care about to publish events anymore, since this is validated for some OCL constraints in order to associate each operation of an aggregate service with an event provided by its API as explained in section 5.1. So, any system generated

[8]https://github.com/AlDanial/cloc

by this tool must be consistent and ready to run using few commands.

"Deploying a microservices-based applications is much more complex"

A monolithic application is simply deployed on a set of identical servers behind a traditional load balancer. Each application instance is configured with the locations (host/port) of infrastructure services such as the database and a message broker. In contrast, a microservice application typically consists of a large number of services [45]. Therefore, this tool uses the Jenkins' pipelines and the Docker Compose files as two ways to carry out the deployment of the services as shown in section 5.2.2. So, once the MSA system is generated, few commands are needed to run all the services.

CHAPTER 6

CONCLUSION

This thesis introduced a model-driven approach to construct a metamodel that covers the main concepts, features and components of the MSA systems. Hence, it was necessary to describe some important microservices patterns that allow the decomposition of monolithic application using DDD, the communication between microservices performing coordination of Sagas such as choreography and orchestration, the division of a traditional database model to store and query information as CQRS, among others. Most of these patterns were adopted using the core concepts of MDD such as abstraction and automation.

The main contribution of this work can be summarized as the definition and development of a microservices tool, which allows the creation, testing and deployment of MSA systems based on EDA. To this end, some characteristics of this model-driven approach are listed below.

- It allows to generate applications' code which can be used afterward to increase the elasticity of large services, or to provide more flexible compositions with other services.
- It reduces some MSA drawbacks such as complexity, inter-process communication mechanisms, error-prone, among others.
- It allows the implementation of API Composition and CQRS patterns through View Services.
- It allows a continuous deployment using Jenkins pipelines and Docker compose.
- It allows to manage and monitor each service of a generated MSA system.

The future works are oriented to empower the deployment of containerized applications. This means the implementation of a container-based technology as Kubernetes[1], which is an open-source system for automating deployment, scaling, and management of containerized applications.

[1] https://kubernetes.io/

REFERENCES

[1] Acceleo, "Acceleo - transforming models into code". 2011.

[2] Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi, and Attila Vizhanyo. The design of a language for model transformations. *Software and System Modeling*, 5(3):261–288, 2006.

[3] Shaukat Ali, Muhammad Zohaib Z. Iqbal, Andrea Arcuri, and Lionel C. Briand. A search-based OCL constraint solver for model-based test data generation. In Manuel Núñez, Robert M. Hierons, and Mercedes G. Merayo, editors, *Proceedings of the 11th International Conference on Quality Software, QSIC 2011, Madrid, Spain, July 13-14, 2011.*, pages 41–50. IEEE Computer Society, 2011.

[4] K. Bakshi. Microservices-based software architecture and approaches. In *In 2017 IEEE Aerospace Conference*, pages 1–8, March 2017.

[5] Guruduth Banavar, Tushar Deepak Chandra, Robert E. Strom, and Daniel C. Sturman. A case for message oriented middleware. In Prasad Jayanti, editor, *Distributed Computing, 13th International Symposium, Bratislava, Slovak Republic, September 27-29, 1999, Proceedings*, volume 1693 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1999.

[6] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[7] BOMAN, M., W., BUBENKO JR, J. A., JOHANNESSON, P., AND WANGLER, B. Conceptual modelling. In ICSE (1976), pp. 592âĂŞ605.

[8] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[9] Zaharah Allah Bukhsh, Marten van Sinderen, and Prince M. Singh. SOA and EDA: A comparative study - similarities, differences and conceptual guidelines on their usage. In Mohammad S. Obaidat, Pascal Lorenz, Peter Dolog, and Marten van Sinderen, editors, *ICE-B 2015 - Proceedings of the 12th International Conference on e-Business, Colmar, Alsace, France, 20-22 July, 2015.*, pages 213–220. SciTePress, 2015.

[10] Antonio Cicchetti, Davide Di Ruscio, and Romina Eramo. Towards propagation of changes by model approximations. In *Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006), 16-20 October 2006, Hong Kong, China, Workshops*, page 24. IEEE Computer Society, 2006.

[11] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[12] Juan de Lara and Hans Vangheluwe. Atom^3: A tool for multi-formalism and metamodelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 2002.

[13] Jim des Rivières and John Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[14] DIDONET DEL FABRO , M., J.BEZIVIN , JOUAULT , F., BRETON , E., AND G.GUELTAS. AMW: A generic Model Weaver. In International Conference on Software Engineering Research and Practice (SERP05). 2005.

[15] Nicola Dragoni, Schahram Dustdar, Stephan Thordal Larsen, and Manuel Mazzara. Microservices: Migration of a mission critical system. *CoRR*, abs/1704.04173, 2017.

[16] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. In Manuel Mazzara and Bertrand Meyer, editors, *Present and Ulterior Software Engineering.*, pages 195–216. Springer, 2017.

[17] Daniel Escobar, Diana Cardenas, Rolando Amarillo, Eddie Castro, Kelly Garcés, Carlos Parra, and Rubby Casallas. Towards the understanding and evolution of monolithic applications as microservices. In *XLII Latin American Computing Conference, CLEI 2016, Valparaíso, Chile, October 10-14, 2016*, pages 1–11. IEEE, 2016.

[18] Eric Evans. *Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional, 1st edition*. 2004.

[19] Luca Florio and Elisabetta Di Nitto. Gru: An approach to introduce decentralized autonomic behavior in microservices architectures. In Samuel Kounev, Holger Giese, and Jie Liu, editors, *2016 IEEE International Conference on Autonomic Computing, ICAC 2016, Wuerzburg, Germany, July 17-22, 2016*, pages 357–362. IEEE Computer Society, 2016.

[20] Martin Fowler and James Lewis. Microservices a definition of this new architectural term. In *https://martinfowler.com/articles/microservices.html, 2014*, Online; accessed 22 April 2018.

[21] M. Gautier, M. Monsion, and J. P. Sagaspe. Détermination d'une représentation des noyaux de volterra pour un système physiologique non-lineéaire. In Jean Cea, editor, *Optimization Techniques: Modeling and Optimization in the Service of Man,*

Part 1 - Proceedings, 7th IFIP Conference, Nice, France, September 8-12, 1975, volume 40 of *Lecture Notes in Computer Science*, pages 110–115. Springer, 1975.

[22] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988.

[23] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The missing link of MDA. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2002.

[24] Jean-Philippe Gouigoux and Dalila Tamzalit. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In *2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017*, pages 62–65. IEEE, 2017.

[25] Kevin Hoffman. Beyond the twelve-factor app: Exploring the dna of highly scalable, resilient cloud applications. In *O'Reilly Media, Inc, 1st edition*, 26 April, 2016.

[26] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.

[27] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained - the Model Driven Architecture: practice and promise*. Addison Wesley object technology series. Addison-Wesley, 2003.

[28] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6, May 2009.

[29] Ivan Kurtev. *Adaptability of model transformations*. PhD thesis, University of Twente, Enschede, Netherlands, 2005.

[30] Duc Minh Le, Duc-Hanh Dang, and Viet-Ha Nguyen. Domain-driven design patterns: A metadata-based approach. In Tru Cao and Yo-Sung Ho, editors, *2016 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future, RIVF 2016, Hanoi, Vietnam, November 7-9, 2016*, pages 247–252. IEEE, 2016.

[31] R. C. Martin. The single responsibility principle. In *http://programmer.97things.oreilly.com/wiki/index.php/The_Single_Responsibility_Principle*, 2009, November.

[32] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20(5):14–18, 2003.

[33] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[34] Meta Object Facility (MOF). *http://www.omg.org/spec/MOF/2.0/*. 2006.

[35] MILLER , J., MUKERJI , J. *Mda guide version 1.0. 1*. 2003.

[36] A Mouat. Using dockers USA: O'Reilly Media, Inc. December 2015.

[37] MULLER , P.-A., FLEUREY , F., AND JÃL'ZÃL'QUEL , J.-M. *Weaving Executability into Object-Oriented Metalanguages*. (Montego Bay, 2005), pp. 264âĂŞ278.

[38] Sam Newman. *Building microservices - designing fine-grained systems, 1st Edition*. O'Reilly, 2015.

[39] Object Constraint Language Specification. *Version 2.2, Object Management Group (OMG)*. 2010.

[40] OBJECT MANAGEMENT GROUP (OMG). *MDA Guide version 1.0.1, 2003*. OMGDocument: omg/2003-06-01.

[41] OMG. *M. 2.0 query/views/transformations rfp*. 2002.

[42] OMG. *SPECIFICATION , M. Q. F. A. Omg document 05-11-01*. 2005.

[43] Florian Rademacher, Jonas Sorgalla, and Sabine Sachweh. Challenges of domain-driven microservice design: A model-driven perspective. *IEEE Software*, 35(3):36–43, 2018.

[44] Chris Richardson. *Microservices Patterns MEAP*. 2017.

[45] Chris Richardson. Microservices architecture. In *http://microservices.io/, 2017. Online*, accessed 22 April 2018.

[46] Chris Richardson. Eventuate Conceptual Model. In *http://eventuate.io/docs/concepts.html, 2017*, Online; accessed 19 May 2018.

[47] Denis Rosa. Saga pattern parti. In *https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part/, 2018. Online*, accessed 22 April 2018.

[48] Denis Rosa. Saga pattern parti". In *https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part-2/, 2018. Online*, accessed 22 April 2018.

[49] Louis M. Rose, Nicholas Drivalos Matragkas, Dimitrios S. Kolovos, and Richard F. Paige. A feature model for model-to-text transformation languages. In Joanne M. Atlee, Robert Baillargeon, Robert B. France, Geri Georg, Ana Moreira, Bernhard Rumpe, and Steffen Zschaler, editors, *Proceedings of the 4th International Workshop on Modeling in Software Engineering, MiSE 2012, Zurich, Switzerland, June 2-3, 2012*, pages 57–63. IEEE Computer Society, 2012.

[50] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[51] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[52] Dharmendra Shadija, Mo Rezai, and Richard Hill. Towards an understanding of microservices. In *23rd International Conference on Automation and Computing, ICAC 2017, Huddersfield, United Kingdom, September 7-8, 2017*, pages 1–6. IEEE, 2017.

[53] S. Suresh Kumar; P M Mallikarjuna Shastry. Database-per-Service for E-learning system with Micro-Service Architecture. 2017.

[54] José Luis Sierra, Baltasar Fernández-Manjón, Alfredo Fernández-Valmayor, and Antonio Navarro. An extensible and modular processing model for document trees. In *Proceedings of the Extreme Markup Languages® 2002 Conference, 4-9 August 2002, Montréal, Quebec, Canada*, 2002.

[55] Gabriel Costa Silva, Louis M. Rose, and Radu Calinescu. Cloud DSL: A language for supporting cloud portability by describing cloud entities. In Richard F. Paige, Jordi Cabot, Marco Brambilla, Louis M. Rose, and James H. Hill, editors, *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, CloudMDE@MoDELS 2014, Valencia, Spain, September 30, 2014.*, volume 1242 of *CEUR Workshop Proceedings*, pages 36–45. CEUR-WS.org, 2014.

[56] Jonas Sorgalla. Ajil: A graphical modeling language for the developmentof microservice architectures. Online; accessed 22 April 2018.

[57] Martin Stefanko. Saga implementations comparison. In *https://jbossts.blogspot.com/2017/12/saga-implementations-comparison.html, 2017*, Online; accessed 22 April 2018.

[58] Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors,

Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2003.

[59] Johannes Thones. Microservices. *IEEE Software*, 32(1):116, 2015.

[60] Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. *Sci. Comput. Program.*, 44(2):205–227, 2002.

[61] VOJTISEK , D., AND JÃL'ZÃL'QUEL , J.-M. *MTL and Umlaut NG: Engine and Framework for Model Transformation*.

[62] Philip Wizenty, Jonas Sorgalla, Florian Rademacher, and Sabine Sachweh. MAGMA: build management-based generation of microservice infrastructures. In Rogério de Lemos, editor, *11th European Conference on Software Architecture, ECSA 2017, Companion Proceedings, Canterbury, United Kingdom, September 11-15, 2017*, pages 61–65. ACM, 2017.

[63] XACTIUM. *Xmf-mosaic. http://xactium.com*.