JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

MaGiC: a DSL Framework for Implementing Language Agnostic Microservice-based Web Applications

Antonio Bucchiarone[*], **Claudiu Ciumedean**[†], **Kemal Soysal**[•], **Nicola Dragoni**[†], **and Václav Pech**[‡]
[*]Fondazione Bruno Kessler, Trento, Italy
[†]Technical University of Denmark, Kongens Lyngby, Denmark
[•]LS IT-Solutions GmbH, Berlin, Germany
[‡]JetBrains, Czech Republic

ABSTRACT The status quo of software applications is in a constant evolution due to emerging of new technologies, performance improvements, and new business requirements. In the past years a new architectural style named microservice architecture emerged which takes an approach to develop application characterised by a suite of small services each running in its own process being decoupled from the other application's components. Nevertheless, implementing a microservice architecture is not trivial and it also comes with several downsides such as a higher complexity of configuring and developing the services, proper componentisation for supporting a single business capability, and implementing the appropriate related software design patterns. Additionally, it requires writing of boilerplate code templates to configure the communication with the services and their deployment. State of the art research tries to address these issues by providing domain-specific languages that enable users to specify and generate microservice applications. However, these solutions do not provide a tool for specifying and generating a microservice-based application similar to any other software application. This paper proposes MaGiC, a DSL framework for implementing language agnostic microservice-based web applications. The framework can be used to specify and deploy to Docker containers microservice-based software applications from end-to-end which can be used as any other application on the internet.

KEYWORDS Microservices, Domain Specific Language, MPS, Web Applications

1. Introduction

Software applications are constantly evolving due to business requirements, performance improvements, and emerging of new technologies (Okwu & Onyeje 2014). One of the emerging architectural trends for building software applications is the *microservice architecture* (Di Francesco et al. 2017). The microservice architecture paradigm is an approach for developing an application as a suite of small services, each running in its own process and communicating through lightweight mechanisms. Division of the application in a suite of services allows for implementing loosely coupled components which can be scaled and deployed independently. Furthermore, it allows for development flexibility as each service can be owned and maintained by different teams, and the services are technology and language agnostic, meaning that each team can implement each service using different technologies and programming languages (Irudayaraj & P. 2019) (Bucchiarone et al. 2019). Nevertheless, implementing a microservice architecture properly is not a trivial task as the componentisation of the software application is highly dependent on how well the software can be split in components and it is difficult to define their context boundaries (Martin Fowler 2014). Moreover, in a complex microservice-based application with a big number of services the communication might become an overhead and potentially a source of errors. Additionally, configuring the microservice architecture is a complex, cumbersome, and time consuming

JOT reference format:
Antonio Bucchiarone, Claudiu Ciumedean, Kemal Soysal, Nicola Dragoni, and Václav Pech. *MaGiC: a DSL Framework for Implementing Language Agnostic Microservice-based Web Applications.* Journal of Object Technology. Vol. 23, No. 1, 2023. http://dx.doi.org/10.5381/jot.2023.23.1.a2

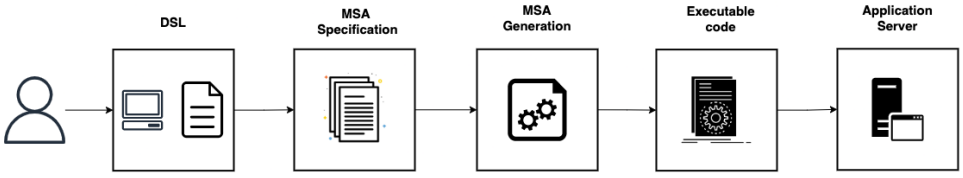

Figure 1 Diagram illustrating the workflow for specifying and generating MSA identified in the state of the art research.

task requiring writing of boilerplate code (Terzić et al. 2017). The literature consists of several attempts which try to address the above mentioned issues from different angles. (Guidi et al. 2017) proposes a language-based approach which focuses on making available in the Jolie (Montesi et al. 2014) programming language, a mechanism to provide first-class citizens that can be used for building microservice architectures.
A different method is put forward by MicroBuilder (Terzić et al. 2017) and AjiL (Sorgalla et al. 2018) which provide *domain-specific languages (DSLs)* that enable users to specify and generate microservice based applications with ease. Figure 1 illustrates an example of the process designed by the researchers (Terzić et al. 2017) when using the DSL. From the picture, it can be observed that the user of the DSL is able to specify the microservice-based application (MSA) then use the generator to produce an actual executable code which can be built and run as the application server.

However, the generated application is not an application which can be used similar to any other web application available on the internet but rather just a part of it, particularly the server side of the application.
Therefore, this article aims to address this issue further and to propose **MaGiC**, *a DSL framework for implementing language agnostic microservice based web applications*, that allows users to specify and generate end-to-end MSA web applications using a suite of DSLs which have the same lexical root.
Figure 2 presents the workflow process of the proposed MaGiC framework where its users are able to use the three DSLs to specify and generate a MSA consisting of an application server, a communication layer (gateway), and an user interface.

The structure of this article is as follows. Section 2 introduces the essential concepts used in this research. Section 3 illustrates state of the art research into the area of MSA generation by using a DSL. Section 4 presents a set of considerations for developing the MaGiC framework, then in section 5 the design and implementation process of the MaGiC framework is described. Respectively, section 6 illustrates the validation process of the MaGiC framework, and in section 7 the conclusion of the overall work is drawn. Lastly, Section 8 depicts the future work of the project.

2. Concepts and Background

To set the article terminology, this section briefly describes the context of the work and the main concepts used.

2.1. Domain-Specific Language

A domain-specific language is a programming language or a specification language which offers through proper notations and abstractions an expressive capacity focused on, and typically restricted to, a certain problem domain (Mernik et al. 2005) (Gray et al. 2008). As DSLs are tailored for a particular application domain (Borum et al. 2021), through notations and constructs they offer substantial gains for expressiveness and ease of use compared to general-purpose programming languages (GPLs) .
The literature (Mernik et al. 2005) provides some general guidelines about *when* and *how* to design a DSL, and splits the development phases into several stages *decision, analysis, design*, and *implementation*, and the main takeaways are:

- To develop a new DSL is not easy and evident of its necessity (Bennett & Rajlich 2000).
- The most accessible way to design a DSL is to base it on an already existing language as the implementation might be easier (Mernik et al. 2005).
- Implementing a domain-specific language is a hard task as one has to deal with compilers and interpreters. However, language workbenches can be used to ease the process (Costa et al. 2018).

2.2. Metaprogramming System

The following section describes the concept of metaprogramming and metaprogramming systems. It important to note that there are several metaprogramming systems available however in the scope of this document only Jetbrains' MPS[1] will be described.
A language workbench is a way of doing language oriented programming such as metaprogramming (Fowler 2005). Metaprogramming is a programming method that allows computer programs to treat other programs as data. With other words, a metaprogram reads another program, manipulates it, and returns a modified program.
One example of such a program is Jetbrains' MPS, which is a language workbench and supports the design and implementation of DSLs. Languages written in MPS are projected rather than textually represented, known as projectional editing, implying that instead of entering plain text to program a language, the user of the language workbench selects from a list of concepts that are valid in their respective contexts.
A DSL developed in MPS is formed of a multitude of entities that represent different aspects of the language. The entities can

[1] https://www.jetbrains.com/mps/

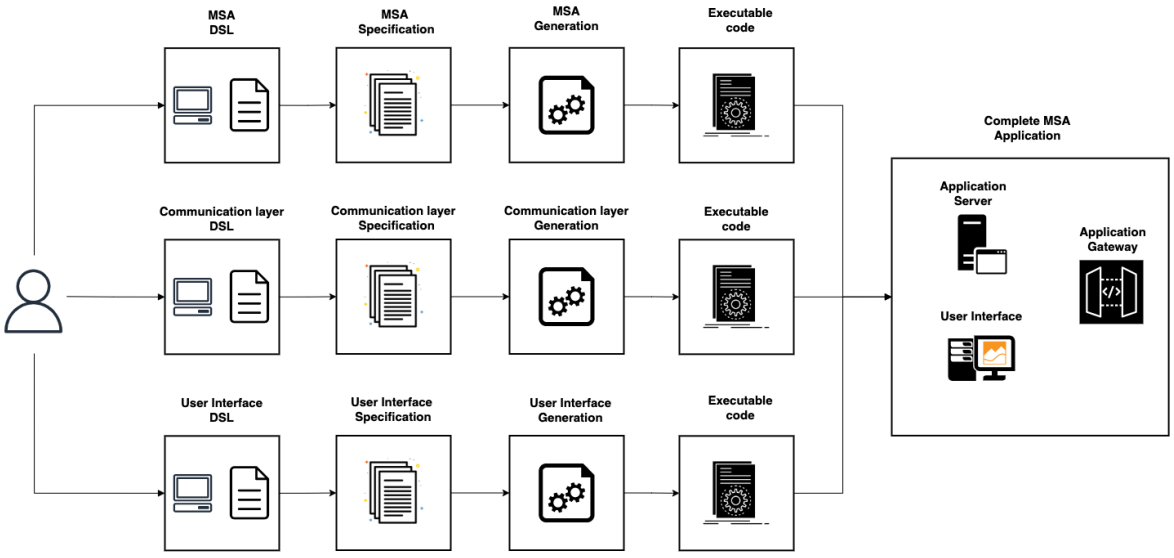

Figure 2 Diagram illustrating the workflow for specifying and generating a MSA using the MaGiC framework.

be split in two categories, the semantical category and structural (Bucchiarone et al. 2021).

2.3. Microservice Architecture

Microservices have been around for a while now, nevertheless they are still considered to be a new trend despite their popularity and that the software design principles on which they are based on have been developed before this trend emerged.
When describing microservices they are often compared to a different architecture style in particular the monolith architecture paradigm, which is on the opposite spectrum, as the microservice architecture is characterised by small independent services, while the monolith architecture is defined as a single indivisible unit (Blinowski et al. 2022). An illustration showing the difference between the two architectures can be seen below in Figure 3.

The microservice architecture enables componentization of an application via services by design, meaning that an application is divided in a suite of components, often called services, that are different than the traditional components (e.g. libraries) in an object-oriented (OO) program as they run in different processes and communicate through a mechanism such as a web service request (e.g. HTTP request) or remote procedure call, rather than using in-memory function calls (Karabey Aksakalli et al. 2021).
By splitting the system in multiple services rather than libraries brings benefits to the overall application as each single service is an autonomous unit which can be deployed and scaled independently. In addition to this, the boundaries between the components, in this case services, are better defined leading to loosely coupled components and with more explicit interfaces preventing misuse of services and possible production outages (Martin Fowler 2014) (Blinowski et al. 2022).

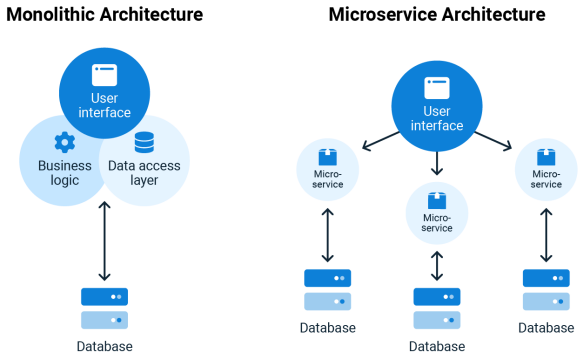

Figure 3 Illustration between the monolithic and microservice architecture. In the monolithic architecture it can be seen that the system as a whole is an indivisible unit which consists of the user interface, business logic, and data access layer. While in MSA the application the system is divided into small divisible units, each unit having a single focus. Image source https://octopus.com/blog/monoliths-vs-microservices.

Moreover, the microservice architecture provides decentralisation governance, meaning that the software does not have to be built on a single standard technology stack which is used across allover the company but rather on multiple stacks. To stress this further, microservices are organised around business capabilities meaning that each service focuses on the implementation of software for a specific business area. Different business areas have different problems which can be better solved with specific technologies, thus making the decentralisation governance aspect of the microservice architecture quite beneficial (Martin Fowler 2014).
Lastly, in a microservice architecture the communication infrastructure between the services is characterised by the "*smart endpoints and dumb pipes*" (Martin Fowler 2014) approach,

meaning that communication between the different microservices is choreographed using simple REST protocols rather complex protocols (Martin Fowler 2014)

2.4. Microservice Architecture Design Patterns

There are several design patterns for implementing the microservice architecture and when the MaGiC framework was developed a handful of design patterns were researched. Patterns such as Circuit Breaker, Load Balancer, are a good fit for a complex system where the suite of microservices is large and horizontal scaling techniques are employed, but in the context of the MaGiC framework this is not the case. In addition to this, patterns such as Publish/Subscribe were not implemented due to the scope of this project as the focus was not on facilitating asynchronous communication with the services (Di Francesco et al. 2017).
Nevertheless, the generated MSA can be deployed on both mobile and desktop clients, and the user of the DSL might choose to serve a different UX depending on the client. Therefore, BFF design pattern fits perfectly for this scenario. Henceforth, only the microservice software design patterns employed for the project will be described.

2.4.1. API Gateway The API Gateway pattern is characterised, as its name asserts, by a gateway though which a number of various services are accessible. The gateway acts as a single entry point for all client requests and based on the request type the message is forwarded to the appropriate microservice (Zhao et al. 2018). An illustration of the pattern can be seen below in Figure 4.

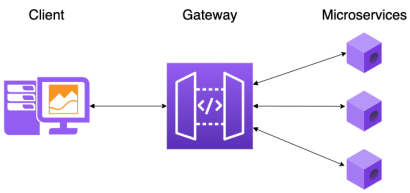

Figure 4 The image illustrates the API Gateway pattern where the client communicates with the gateway which in turn communicates with the suite of microservices in order to fulfil the client's request.

Implementing the API Gateway in the MSA has many benefits some of them being, the client is not concerned about communicating with each microservice at the right endpoint as there is only the gateway endpoint, thus simplifying the client logic. The overall performance is increased as the number of requests and round-trips are significantly reduced in a scenario where data is aggregated from multiple services. Additionally, by using a variation of this pattern it can provided specifically tailored APIs for each particular client (Zhao et al. 2018).

2.4.2. Backend For Frontend (BFF) Backend For Frontend (BFF) is an architectural pattern is an extension of the API Gateway pattern and proposes an approach where each client application (frontend) has a dedicated server-side component (backend) in furtherance of enhancing and improving the overall UX. This pattern emerged as a consequence of the fact that different client applications (e.g. desktop, iOS mobile client, Android mobile client) provide different APIs (Application Programming Interfaces) and therefore the user experience from one platform to another changes.
Similar to the API Gateway pattern the BFF is used as a communication delegator between the client and the suite of microservices. However, in the BFF case the set of microservices provide specifically tailored user experience depending on the client platform as each client's request is propagated through a specific and isolated API (Pavlenko et al. 2020). An illustration of the BFF design pattern can be seen below in Figure 5.

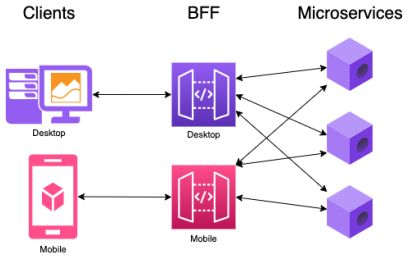

Figure 5 The image illustrates the BFF pattern where the desktop client communicates with its related service, the desktop BFF, and the mobile communicates with the mobile BFF, in turn both BFFs communicate with the suite of microservices providing a platform specific user experience.

2.5. Software Containerization

Software containerization is an approach in software development where an application along with its dependencies, and set of configurations, are bundled together as a container image and deployed to a host operating system. As the software is deployed together along with its dependencies the containerization approach is highly beneficial as it eliminates issues related to inconsistent environment setups and moreover, it enables fast application horizontal scalability due to the ease of spawning new container instances when required (Jaramillo et al. 2016). In the context of this article Docker[2] was used as a software containerization tool. Docker is a platform for shipping and running software applications with ease by offering a lightweight containerization platform making available a suite of tools which makes application deployment and management effortless.
Docker containers naturally fit MSA as each microservice can be deployed as a self-contained container unit. The microservice suite can be easily deployed and managed since the creation and launching of Docker containers is done through scripting thus enabling for automation techniques. Furthermore, as microservices are language and technology agnostic they can be independently deployed as separate containers and owned by different software development teams which can work on the implementation of a given service choosing the most suitable technology or language for solving a given problem, and in the same time allowing for collaboration with other services as the

[2] https://www.docker.com/

communication over the network is effortless (Jaramillo et al. 2016).

2.6. Microservices in Practice

Implementation of microservice architectures have become quite popular in the last years. However, despite the fact that the concept of a microservice is novelty, when it comes to the actual implementation of a microservice it relies on already established technologies such as HTTP protocol (Martin Fowler 2014).
The long term goal for the MaGiC framework is to be language agnostic so that in theory the microservice based web application can be generated to any general-purpose programming language. The current state of the framework supports generating the microservice suite to Node.Js and Python programming languages, and they were chosen due to their popularity, as according to a survey made by StackOverflow (Overflow 2021). The following listings showcase a minimal sample of a microservice implementation in both programming languages which serve as an example of how simple microservices are implemented in practice and can be used as a comparison with a microservice specified using MaGiC and its subsequent generated code which can be found in following GitHub Repository (Bucchiarone et al. 2023).

```
const express = require("express");
const app = express();

app.get("/order-list", (req,res)=>{
    res.status(200).json(orders);
});

app.get("/order", (req,res)=>{
    res.send(order);
});

const port = 8081;
app.listen(port)
```

Listing 1 Sample implementation of a microservice written in Node.js using the Express framework. The microservice deals with the orders business logic and exposes two HTTP methods which are used to send information related to orders.

```
from flask import Flask, request, jsonify,
    send_file
from flask_cors import CORS
import uuid
import json

app = Flask(__name__)
CORS(app)
app.run()
@app.route('/order-list', methods=['GET'])
def getOrders():
    return jsonify(orders)
@app.route('/order', methods=['GET'])
def getOrders():
    return jsonify(order)
```

Listing 2 Sample implementation of a microservice written in Python using the Flask framework. The microservice deals with the orders business logic and exposes two HTTP methods which are used to send information related to orders.

3. Related Work

The following section reports the relevant studies related to the DSLs for microservices. Additionally, it includes works which were used as inspiration towards developing the MaGiC framework.

3.1. MicroBuilder: A Model-Driven Tool for the Specification of REST Microservice Architectures

Terzić et al. (Terzić et al. 2017) developed a software tool, MicroBuilder, for generating code for a REST MSA using a model-driven approach. Their focus was on developing a software which provides a set of tools for developing a microservice based application while taking care of the challenging tasks such as configuring the microservice architecture, load-balancing, microservice auto-discovery and registering, but also reduce the amount of time spent to develop such an application by eliminating the need of writing redundant code templates. The MicroBuilder tool is divided in two modules, namely MicroDSL which is a DSL used for constructing the REST microservice software architecture, and MicroGenerator which consists of a set of generators that are used for generating actual executable code from the programme written using the MicroDSL.
In more details, the MicroDSL enables users of the language to specify, by using the DSL's concrete syntax, a microservice based application using a set of concepts related to the domain of REST MSA development. The programme specification written using the MicroDSL is then fed as an input for the MicroGenerator module of the MicroBuilder software tool and an executable program code is generated for a given platform, in this case for the Java programming language (Terzić et al. 2017). An illustration of the development process using the MicroBuilder software tool can be seen below in Figure 6.

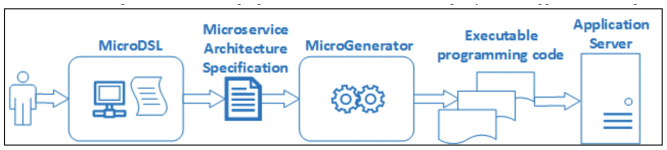

Figure 6 Illustration of the development process of a REST MSA using MicroBuilder (Terzić et al. 2017).

To validate the software tool, the research team developed the specification for a REST MSA web shop using the MicroDSL concrete syntax which later on was generated / transformed into the Java programming language.
Additionally, the team decided to validate their solution further and compared the lines of code for each microservice necessary to specify the web shop using the MicroDSL, with the lines of code required to program the web shop manually while using best programming principles and reducing the amount of empty lines (Terzić et al. 2017). The difference between the amount of lines of code can be seen below in Figure 8.

Microservice name	The number of lines of code in the MicroDSL specification	The number of manually written lines of code
User microservice	28	406
Payment microservice	17	270
Product microservice	18	335
ShoppingCart microservice	25	385
Total	98	1802

Figure 7 Table representing the difference between the amount of lines of code necessary to build the webshop application with and without using MicroDSL. Image source (Terzić et al. 2017).

3.2. AjiL: A Graphical Modeling Language for the Development of Microservice Architectures

Sorgalla et al. (Sorgalla et al. 2018) developed a graphical language and toolkit for implementing model-driven microservice architecture. The language name is AjiL (Aji Modeling Tool) and the research group aim is to reduce the effort of redundantly and inconvenient manual process when implementing a microservice architecture using a Model-driven Engineering approach. The tool allows developers to specify diagrams for a given microservice suite and generate *pre-configured system foundations* (Sorgalla et al. 2018), based on the respective diagrams.

AjiL consists of three main pillars, the Aji modelling language (AjiML) which is a lightweight graphical domain-specific modelling language (DSML), the AjiEditor responsible for creating diagrams based on the AjiML using a dedicated Eclipse workbench, and Aji Spring Cloud Generator used for transforming the modelled diagrams into actual source code written in the Java programming language using the Spring Cloud framework (Rademacher et al. 2020) (Sorgalla et al. 2018).

Using the AjiEditor the user of the Aji modelling language is able draw the microservice suite along with their interactions and internal specifications, which in turn are generated to the Java programming language and the Spring Cloud framework, a lightweight event-driven framework to quickly build microservice based applications. It is worth mentioning, all the modelled interfaces of the services are generated to REST controllers exposing CRUD (Create Read Update Delete) operations.

Furthermore, despite the fact that the study team focused on generating the code base to the Java programming language the models were designed so that they are language and technology agnostic thus enabling for generation to other programming languages if the functionality was to be added (Sorgalla et al. 2018).

3.3. Sliceable Monolith: Monolith First, Microservices Later

A different approach called *sliceable monolith* is put forward by (Montesi et al. 2021) who chose to implement a MSA based application by first implementing it as monolith then use an automated tool, called *Jolie Slicer*, to slice it into its subsequent microservices. Additionally, the tool outputs the required configuration for containerisation and deployment of the services to the cloud using Docker. Another important aspect of the software tool is its technology-agnostic characteristic which is enabled by the nature of the Jolie programming language, thus allowing the users of the sliceable monolith software to use different general purpose programming languages for developing the microservice suite. Furthermore, the tool also allows for unit and integration testing of the entire application architecture with minimal effort by providing a configuration JSON file therefore ensuring the correct functionality of the implemented application (Montesi et al. 2021).

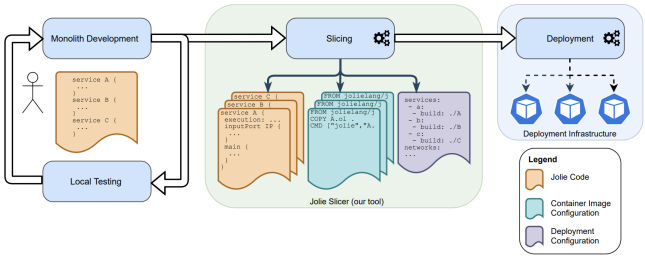

Figure 8 Illustration of the workflow methodology for using the Sliceable Monolith software tool. Image source (Montesi et al. 2021).

While the Sliceable Monolith tool deals with the development of MSA based applications the approach proposed by the research team is different than the methodology proposed by the MaGiC framework. The research team put forward a way of deploying a microservice-based application by implementing it as a monolith first then use an automated tool to slice it and deploy it as suite of microservices but the application consists of the microservice suite only. Additionally, the methodology and the generated application is different compared to the one offered MaGiC. Nevertheless, the technology-agnostic and Docker containerisation aspects are valuable for both approaches.

3.4. Microservice DSL (MDSL)

Microservice DSL represents a domain-specific language to describe microservice contracts along with their data representation and API endpoints, and was constructed based on abstractions from domain-specific interface definition language. The design goals of the MDSL are to facilitate agile modelling practices, that is to focus on the readability of the DSL over the parsing efficiency of the abstract syntax of the language, as well as to be platform decentralised meaning that the language features which aid in the specification of a given microservice suite should not be limited to a single communication protocol (e.g. HTTP). The abstract syntax of the DSL is based on the domain model and concepts of MAP[3] consisting of several features such as endpoints, operations, and elements for representing data. The concrete syntax of API endpoints is elaborate consisting of a set lexicon elements to enable a detailed API specification, and in contrast the data representation lexicon is compact and simple abstracting from data formats such as

[3] https://microservice-api-patterns.org/

XML[4] and JSON[5], but also from service-oriented languages (Zimmermann et al. 2022).
Figure 9 provides an example of an API specification using the Microservice domain-specific language.

```
API description SampleCRMScenario
version "1.0"
usage context PUBLIC_API for FRONTEND_INTEGRATION
overview "Example in MDSL Primer"

data type Customer {"name": D<string>,  "address": D<string>}
data type StatusCode "success": MD<bool> default is "true"

endpoint type CustomerRelationshipManager serves as
   PROCESSING_RESOURCE
exposes
  operation createCustomer with responsibility
     STATE_CREATION_OPERATION
    expecting payload "customerRecord": Customer
    delivering payload "customerId": D<int>
    compensated by deleteCustomer
  operation deleteCustomer with responsibility
     STATE_DELETION_OPERATION
    expecting payload "customerId": D<int>
    delivering payload "success": StatusCode
    transitions from "customerIsActive" to "customerIsArchived"
```

Figure 9 API example specified using MDSL. Image source https://microservice-api-patterns.github.io/MDSL-Specification/primer.

According to the author of MDSL (Zimmermann et al. 2022) the language ought to be used in contexts where a suite of microservices are to be described along with their several communication protocols (e.g. HTTP, message queuing). Furthermore, the DSL can be used to represent their subsequent request and response messages, as well as the interface endpoint. The different language specifications of MDSL represented using hexagon diagrams can be see below in Figure 10.

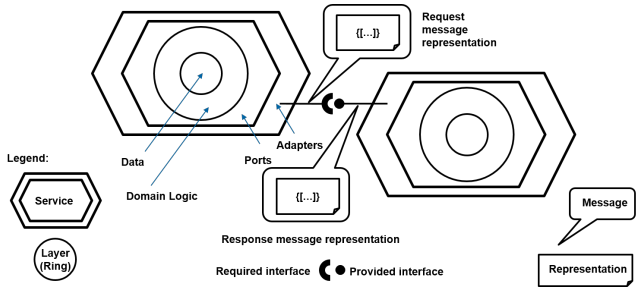

Figure 10 Illustration of the usage context of MDSL specification. Image source https://microservice-api-patterns.github.io/MDSL-Specification/primer.

Nevertheless, MDSL is a tool intended for only specifying microservices and not actual development, the generated code to either Java or Jolie (Montesi et al. 2014) programming languages represents a skeleton of given specified service and the actual functionality needs to be implemented by a developer (Zimmermann et al. 2022).

[4] https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction
[5] https://www.json.org/json-en.html

3.5. LEMMA: A Language Ecosystem for Modeling Microservice Architecture

LEMMA is an extensive language ecosystem used for modeling MSA based applications in terms of their design, development, and deployment. The language ecosystem has a distributed nature as it consists of multiple modeling languages called viewpoint-specific MSA modeling languages, each language addressing a different sub-domain of microservice architecture (Rademacher, Sachweh, & Zündorf 2019).

Domain Data Viewpoint, represents the modelling language designated for domain experts and service developers for describing domain-specific data models using the language structures and concepts (Rademacher, Sorgalla, et al. 2019).

Service Viewpoint, as its name asserts the *service viewpoint* is the modelling language for specifying the complete behaviour of a microservice in terms of its communication protocol with other services and the data format, its endpoints, and its operations (Rademacher, Sorgalla, et al. 2019).

Operation Viewpoint, deals with the configuration and deployment of microservices to Docker containers (Rademacher, Sorgalla, et al. 2019).

Technology Viewpoint, is the modelling language for the MSA technology. LEMMA, similar to MSA, promotes technology heterogeneity thus it offers two technology specific aspects in the form of a *built-in technology* and *custom technology*. These two technology aspects can be divided into different elements such as the deployment technology, data format, infrastructure, programming languages, and communication protocol (Rademacher, Sachweh, & Zündorf 2019).

In terms of the generation of the modeled MSA application using LEMMA, the ecosystem enables users of the language to use a generator to derive Jolie APIs (Montesi et al. 2014) from the defined LEMMA models. However, similar as in the case of MDSL (Zimmermann et al. 2022) the generated APIs do not provide actual functionality but rather the skeleton of the MSA application in terms of its operation concepts which developers can use as a guidance for the implementation (Giallorenzo et al. 2022).

3.6. Wasp

Wasp is a software tool for building full-stack web applications and it can be used to develop all three main application's pillars: (1) client-side; (2) server-side; (3) deployment. It was developed with the aim to enable developers to build web applications writing less code and without web technology-specific knowledge. The software tool consists of a DSL for specifying web application specific terms (e.g. page, route) for declaring a high-level overview structure of the application. The web application specification written using the DSL represents the input of the Wasp's compiler / generator which along with the

actual programming language code written by the developer, are compiled into a functional web application (see Figure 11). As the Wasp authors mentioned, the tool is not a *no-code solution* (Wasp 2022) meaning that the DSL is used only for describing the structure of the web application in terms of pages and routes, and as for the functionality of the application actual development is necessary (Wasp 2022).

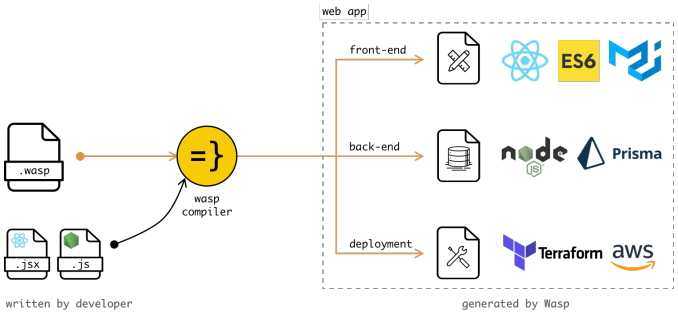

Figure 11 Diagram illustrating the development process of a web application using the Wasp software tool (Wasp 2022).

The abstract syntax of Wasp DSL is declarative and statically typed, each declaration describing a part of the web application. The declaration syntax consists of three main components: (1) declaration type, representing one of the types offered by the Wasp DSL (e.g. app, route, page); (2) declaration name, the identifier of the declaration; (3) declaration body, the definition of the declaration itself which is specific to the declaration type. (Wasp 2022). Figure 12 provides and example of the Wasp DSL syntax.

```
app MyApp {
  title: "My app"
}

route RootRoute { path: "/", to: DashboardPage }

page DashboardPage {
  component: import Dashboard from "@ext/Dashboard.js"
}

action createTask {
  fn: import { createTask } from "@ext/operations.js",
  entities: [Task]
}
```

Figure 12 Example declaring an application with a route, a page, and an action using the Wasp DSL (Wasp 2022).

According to the authors of Wasp, the tool is in alpha version not production ready and there are still a suite of features that need to be developed and tested (Wasp 2022). However, the Wasp DSL for specifying the web application structure in terms of pages, routes, and other declaration types, is simple and declarative allowing the user to specify the application structure with ease.

4. Considerations

To begin with, the reason it was decided use MPS to create the MaGiC framework rather than reusing some of the existing DSLs from the literature which focus on the microservice domain is due to the fact that in general DSLs are unfortunately not reusable or extensible when created with different software tools, for example the Eclipse integrated development environment (IDE) used by (Terzić et al. 2017) to develop MicroBuilder. However, due to the evolution of DSLs the language workbench paradigm was born to provide a common platform to reuse. Therefore, Jetbrain's MPS language workbench was a great option for implementing the MaGiC framework since it is specialized to provide needed aspects for defining concepts, editors, behaviors, constraints, type system, refactoring, intentions, find usages, data flow, tests, migrations, versioning, runtimes, generators, build, deployment, and more importantly it allows for reusability.

The research projects MicroBuilder (Terzić et al. 2017) and AjiL (Sorgalla et al. 2018) are trying to address the several issues encountered when implementing a microservice-based application. Their solution is to provide a DSL which can be used to specify a suite of microservices and generate them to a lower level programming language.
In the case of LEMMA (Rademacher et al. 2022), it represents an extensive language ecosystem which offers users a large variety of functionalities the current state of the language does not allow for generating actual functional applications but rather only the interfaces of the MSA based on the specified models which aid the developers with the implementation of the application.

To the authors best knowledge there is no state of the art research which focuses on providing a DSL framework for specifying and generating a microservice-based web application which consists of a client-side, a communication layer / gateway, and a microservice suite, which allows for generation of a fully functional MSA based web application.
Starting from the work of MicroBuilder (Terzić et al. 2017) and AjiL (Sorgalla et al. 2018), the focus of our research is to develop a DSL-based framework for specifying and generating all three parts of the MSA, which means that for each of the three parts of the application a DSL needs to be provided.

Each abstraction level to a specific domain builds its own ubiquitous language. The domain in this article is software in general. But software does not exist without the need for it. Its stakeholders are not only developers. The non developer stakeholders should use a DSL to provide the main requirements that the targeted software application should fulfill. Each domain expert has an own set of DSLs to describe his domain in the whole project. So refinement and reuse of already modeled domain objects or interactions by other domains experts is easy. Such a collaboration of DSLs is provided in our MaGiC paper. Due to the time limitation we concentrated on DSLs we strictly needed, others DSLs could or should be of interest or beneficial.

Therefore, the goal of this article is to present the MaGiC framework for developing MSAs which consists of three different domain-specific languages:

- A microservice DSL that allows for specifying the microservice suite.
- A client side DSL used for constructing the UI of the application.
- A gateway DSL for specifying an efficient communication design pattern between the client-side and microservice suite.

Similar as in the case of the Sliceable Monolith approach, and LEMMA, the focus for our research is to develop a tool able to generate applications programmed in various programming languages, thus leveraging the technology heterogeneity aspect of microservices. Additionally, in MSA software containerisation is a highly beneficial approach where the produced generated code artefacts are deployed into Docker containers.
Moreover, since one of the research goals is to ease the development of microservice-based web applications which can be translated into the effort of building and maintaining the application, the DSL readability is ought to be considered. According to Aggarwal et al. (Aggarwal et al. 2002): "*the readability of source code or quality of various documents greatly influences software maintainability*", thus when designing the syntax of the three DSLs of the MaGiC framework an emphasis on readability is placed in furtherance of creating a framework which aligns with the research's goals.
Ultimately, to the extent of the author's knowledge, it can be stated that the research project in concern has a novelty aspect since there are no projects which focus on building a DSL framework to enable users to model a microservice based web application from end-to-end including the functionality.
Table 1 presents the differences between our proposal and the most relevant solutions proposed in the previous sections.

	MB	**AjiL**	**MaGiC**	**SM**	**LEMMA**
REST MSA DSL	✓	✓	✓	✗	✓
Gateway DSL	✗	✗	✓	✻	✓
Client-side DSL	✗	✗	✓	✗	✗
Docker deployment and containerisation	✗	✗	✓	✓	✓
Language / Technology Agnostic	✱	✱	✓	✓	✓

Table 1 Comparison in terms of features between state of the art research projects and the MaGiC framework.
MB = MicroBuilder; **SM** = SliceableMonolith; ✱= supported, but not implemented; ✻= supported due to language primitives but not human readable;

To conclude, based on the research and considerations previously presented a set of functional and non-functional requirements were constructed for the MaGiC framework illustrated in Table 2.

Functional	**Non-functional**
Provide a DSL for generating the client side of the MSA	Ease the process of implementing a MSA
Provide a DSL for generating the gateway of the MSA	The generated microservices should be organised around business capabilities
Provide a DSL for generating the microservice suite of the MSA	The building of the MSA and deployment to Docker containers should be done with ease
Allow for deploying and building in Docker containers	The DSLs syntax should focus on readability
Allow for generation of the MSA to different programming languages	

Table 2 The set of functional and non-functional requirements for the MaGiC framework.

5. Design and Implementation

The following section presents the design and implementation of the MaGiC framework. To remind the reader, it consists of three DSL pillars and all of them were implemented using Jetbrains' MPS. The complete MaGiC framework implementation as well as the user manual of the framework can be found at the following GitHub repository (Bucchiarone et al. 2023). The manual consists of the syntax of the language, as well as the language documentation. Additionally, it provides a guide for using MaGiC to specify, generate, and deploy a **Hello World** microservice based web application.

- Microservice DSL, which is used to model and generate the microservice suite.
- Gateway DSL, used for specifying the communication layer between the client-side / UI and the microservice suite.
- Client DSL, representing the DSL for specifying the client-side of the web application.

It is important to mention that the MaGiC framework is uniform. Particularly, MaGiC is an abstraction which can be used to describe a microservice, BFF, and UI, in any general-purpose programming language and the syntax of each pillar was designed so that it describes the entities of the system from a higher-level and abstract perspective.
Moreover, the decision of splitting the MaGiC framework into three DSL pillars was inspired from the concepts of microservice architecture. In MSA each service is focused on a single business capability, therefore as MaGiC addresses the MSA domain the authors decided on splitting the framework so that each DSL pillar is focused on a single capability. That is a DSL for building the microservice suite, gateway / BFF, and the user interface.

5.1. Microservice DSL

The Microservice DSL represents the domain-specific language used to specify a REST MSA, particularly users can express a microservice suite where each microservice exposes CRUD operations which are used to interact with a particular set of data.

5.1.1. Syntax The syntax of the DSL is inspired from a subset of features of MDSL (Zimmermann et al. 2022) due to its high readability and microservice domain-specific focus, to which several alterations were added. The DSL

consists of several properties which are used to specify metadata about the microservice. In addition to this, a microservice can be configured to expose several operation methods each operation supporting CRUD operation types. Figure 18 from the Appendix illustrates a formal representation of the DSL features' abstract syntax using Backus–Naur form (BNF) notation (McCracken & Reilly 2003).

The abstract syntax features are then incorporated in the DSL concrete syntax where a strong focus on readability over parsing efficiency is emphasised similar as in the MDSL (Zimmermann et al. 2022) case. Figure 19 from the Appendix shows the specification of a microservice expressed through the Microservice DSL concrete syntax.

5.1.2. Design

The design choices are strongly related to the concepts of language readability, and technology and programming language decentralisation. Therefore, as stated before, the syntax of the language is inspired from a subset of features from MDSL (Zimmermann et al. 2022) but also several additions were made in furtherance of constructing a verbose enough language and simple in the same time to allow DSL users to specify a microservice with ease. In addition to this, microservice implementation examples were surveyed from which common patterns were abstracted in order to come up with a model which can be used as a matrix for generating a microservice into any GPL.
The GPL chosen for microservice generation are Node.Js [6] using the Express framework [7], and Python [8] using the Flask framework [9]. Moreover, as one of the requirements for the MaGiC framework is to enable software containerisation, the Microservice DSL comes with an additional feature, particularly deployment of microservices to Docker containers. Additionally, it was decided to automatically provide the documentation of the microservice API endpoints using SwaggerUI [10] so that DSL users can easily interact with the microservice endpoints and set up the communication layer.

5.1.3. Implementation

The implementation was done using DSL Jetbrains' MPS which made the process easier due to its several helpful features. The MPS aspects were heavily employed in implementing the DSL as the language consists of several structures which use several editors, behaviours and generators. Thus, as the Microservice DSL is split in several structures they are then integrated in the microservice main DSL structure using OO (object-oriented) principles as MPS follows the object-oriented paradigm. As for the syntax of the DSL, it was defined using MPS editor aspect which allows for declaration of the concrete syntax and integration of the abstract syntax defined as MPS structures. An illustration of MPS editor aspect used to declare the syntax of the Microservice DSL can be seen below in Figure 20 from the Appendix.

[6] https://nodejs.dev/learn
[7] https://expressjs.com/
[8] https://www.python.org/about/
[9] https://flask.palletsprojects.com/
[10] https://swagger.io/tools/swagger-ui/

5.1.4. Generation

The generation of the microservice is done to a lower level programming language, as well as the configuration for deployment to Docker containers specified in a *Dockerfile*. Additionally, the SwaggerUI configuration files, are produced which are used for documenting the microservice's REST API endpoints exposing CRUD operations and can be visualised by accessing the */api-documentation* microservice route. Furthermore, in order for the user of the DSL to deploy the microservice with ease a Shell script build is provided, thus hopefully enhancing the overall UX of the DSL.
In terms of how the generation process works, MPS implies the model-to-model transformation approach, with other words the MPS generator specifies translation of constructs encoded in the input language, the DSL in this case, into constructs encoded in the output language usually denoted by a general purpose programming language (Team 2021). However, for the generation of the MaGiC framework DSLs the model-to-text generation approach was used since the Python language has not been implemented for MPS, thus the MPS native model-to-model generator could not be used. Therefore, *MPS Plaintextgen* [11] model-to-text generator was used which makes this direct model-to-text transformation easier rather than the MPS model-to-text generator.

Figure 13 provides an example of the form of a given microservice specified in the MaGiC DSL, with its generated counterpart in the Node.js programming language. It is important to note that this generation functionality applies to all three DSL pillars that the MaGiC framework consists of.

5.2. Gateway DSL

The Gateway DSL is used to specify and generate the communication layer between the server-side / suite of microservices and the client-side / application's user interface, and it is implemented using Backend for Frontend (BFF) design pattern.
The BFF design pattern was chosen due to the fact that the generated MSA can be deployed on both mobile and desktop clients, and the user of the DSL might choose to serve a different UX depending on the client.
It is important to note that when designing the Gateway DSL for implementing BFF a strong focus was placed on the performance of the application as the user is able to specify the exact information to be sent from the microservice to the client.
Best practices for mobile web application include conservative use of resources and to implement a highly performant application that runs smoothly with low latency the memory of the device, processor power, and network bandwidth ought to be minimized (Aldayel & Alnafjan 2017). Therefore, the Gateway DSL was designed so that when specifying the communication layer DSL users are able to specify what exact properties are expected to be sent to the client-side, rather than sending all the properties, thus reducing the network bandwidth. An example of this would be to query a given service for a user's name, in this case the DSL user would be able to specify that only the name should be sent over to the client and not all the user information.

[11] https://github.com/DSLFoundry/mps-plaintextgen

Figure 13 Illustration exemplifying a microservice written using the MaGiC DSL (left) and its generated form written in Node.js (right). As it can bee seen, in the left side of the illustration the *CustomerService* microservice is specified, exposing a method for fetching a particular *customer* from the stored list of *customers* based on its *ID*. Respectively on the right it can be seen that the behaviour specified using MaGiC is translated and generated into the Node.js programming language.
It is important to note that users of MaGiC are able to choose to generate the microservice to either Node.js or Python programming languages. For a more detailed documentation please refer to the user manual from the following repository (Bucchiarone et al. 2023).

5.2.1. Syntax The syntax of the Gateway DSL is similar to the Microservice DSL and the reason for using the same syntax is due to the fact that the two domains are closely related and their concepts can be expressed naturally using the same lexicon. Additionally, by sharing a similar syntax same concepts of the two DSLs can be reused thus easing the implementation. An additional benefit is that by constructing the DSLs with more or less the same syntax users of the framework would not have to accommodate to three different language syntaxes but rather make use of one syntax for declaring a MSA from end-to-end.
Similar as in the Microservice DSL case, the DSL consists of several properties which denote metadata about the gateway but also configuration specific information. Furthermore, the DSL exposes different operations each supporting CRUD operations communicating with a referenced microservice API endpoint. Figure 21 from the Appendix illustrates the Gateway DSL's abstract syntax in Backus-Naur form.

The abstract syntax features of the Gateway DSL are then incorporated in the DSL concrete syntax where similar in the Microservice DSL and MDSL (Zimmermann et al. 2022) a strong focus on readability is placed. Figure 22 from the Appendix shows the syntax of specifying a gateway using the Gateway DSL.

5.2.2. Design The design choices are strongly related to the Microservice DSL, as they were made in order to promote readability over parsing efficiency thus fulfilling one of the non-functional requirements, particularly technology and programming language decentralisation. Nevertheless, as the Gateway DSL is used to specify the communication layer between the microservice suite and the client-side several simplifications and changes were made. Furthermore, the BFF gateway can be generated to either Node.Js using the Express framework, or Python using the Flask framework. In addition to this, same as in the Microservice DSL case the generated code can be deployed to Docker containers using the generated Shell script.

5.2.3. Implementation The implementation of the DSL was done using Jetbrains' MPS making use of its several useful features. In addition to this, the already created models for the Microservice DSL were employed since the two languages share common aspects. Figure 23 from the Appendix showcases the defined structure of the Gateway operation which uses the same concepts as the ones declared for the Microservice DSL.

5.2.4. Generation The generation of the BFF based on the specification written using the Gateway DSL are the same as in the case of the Microservice DSL. Thus, the BFF can be generated to either Node.Js or Python programming languages, and additionally a *Dockerfile* is automatically generated which can be used to deploy the BFF to Docker containers using a Shell script. The generation phase of the BFF DSL, as well as the other DSLs, is heavily relied upon on MPS template switchers which allow implementing decision trees for generating the right code depending on the context.

5.3. Client DSL

The Client DSL represents the language for specifying and generating the client-side / UI of the microservice based web application. In section 3 several projects were presented and one of them focusing on developing a DSL for generating a clide-side application is Wasp (Wasp 2022) from which inspiration was drawn to design the Client DSL of the MaGiC framework. Additionally, as the Client DSL communicates with the gateway which forwards client requests to the microservice suite, microservice-domain specific aspects from the other two DSL pillars were extracted and incorporated in the Client DSL.

5.3.1. Syntax The syntax of the DSL is based on concepts extracted from the Wasp DSL, particularly the structure of the client-side application which is split in pages and routes, and additionally it consists of microservice-domain rest specific aspects extracted from the Gateway DSL and respectively

Microservice DSL, particularly the client pages support CRUD operations, but also from client-side application specific features. The Client DSL also consists of a configuration structure used to specify the port in which the client application is running and which one of the client types (desktop / mobile) it exposes. Figure 24 presents the abstract syntax of the Client DSL.

Furthermore, the Client DSL has an additional syntax for specifying the UI components, denoted by <*Component*> in Figure 24 from the Appendix, of the application which is inspired from HTML, which the standard markup language for documents designed to be displayed in a web browser, thus fitting the requirements for the MaGiC framework as it is used to build microservice based web applications. The syntax of each of the UI component is illustrated in Figure 25 from the Appendix.

In terms of the concrete syntax of the Client DSL, it is important to mention that same as in the case of the other two DSLs of the MaGiC framework a strong emphasis was placed upon readability. Figure 26 from the Appendix illustrates the abstract syntax used to specify the client-side of the MSA.

5.3.2. Design First of all, as it was previously mentioned the Client DSL has features extracted from the Wasp DSL and microservice domain-specific aspects. Additionally, the user of the DSL is able to specify UI components using a syntax similar to how HTML elements are declared.
Furthermore, as the focus of the project is to build a framework for implementing language and technology agnostic microservice based web applications the design choices are strongly related to this concept.
In terms of the generated application it was decided to generate a single-page application (SPA), which is a web application implementation that loads only a single HTML document and then its content is updated accordingly via JavaScript APIs which serve resources such as JSON. The reason why it was chosen to generate a SPA is due to the fact that SPAs are somewhat the new normal in the web applications world and they offer a more dynamic experience and performance gains compared to traditional web application with multiple pages (Fink & Flatow 2014).

It is important to mention that there are several JavaScript UI frameworks which aid with the development of single-page applications using best practises. However, due to the scope of this project and time constraints it was decided to generate the specified SPA using the React framework. React is an open-source JS library for building interactive UIs with ease and it is component-based meaning that each UI component is encapsulated and manages its own state allowing for composability and building of highly complex and performant user interfaces (Gackenheimer 2015).

Additionally, for the actual design of the client-side application it was decided to use the design system of Bootstrap (Gaikwad & Adkar 2019), which is one of the most popular UI frameworks for styling web applications. Particularly, React Bootstrap was used which is a UI Boostrap library specifically develop for styling React based single-page applications.

It is highly important to mention that the name of the UI components which are part of the Client DSL are based on the naming used by the Boostrap design system, and this is due to the fact that no general taxonomy of naming UI components was found. Therefore, despite the fact that the focus is to build a technology agnostic framework when it comes to the actual names of the UI components this goal is not necessarily reflected. Lastly, similar to the other two DSLs of the MaGiC framework, the generated client-side application can be deployed to Docker containers using the Shell script, in furtherance of eliminating the need for the user to manually deploy the application and hopefully increasing the overall UX of the framework.

5.3.3. Implementation The implementation of the Client DSL is similar as in the two other DSLs, Jetbrains' MPS was used due to its numerous helpful features. The Client DSL uses certain MPS structures declared in the Microservice and BFF DSLs, however since the underlying technology is different than the one used to implement the microservice suite and gateway there are several client domain-specific structures.

5.3.4. Generation The generation of the client-side application using the Client DSL outputs *.jsx* files, a React specific file format. These files are then compiled and built by Webpack, an open-source module bundler for modern JavaScript applications, using the *CreateReactApp* command line framework. Additionally, once the application is compiled and built it is deployed to Docker containers and can be accessed using the chosen process destination port.

6. Validation

The following section presents the validation of the software artefact created for the project in concern, namely the MaGiC framework.

6.1. Application Development

For the validation of the framework, a REST microservice based web shop application was implemented. Furthermore, to evaluate the language and technology agnostic aspect of the DSLs, both the microservice suite and gateways were generated to Node.js and Python programming languages.
Lastly, in order to test the functionality of the application, the application was deployed and built into Docker containers. The MSA consists of several microservices which are in charge of different business capabilities, such as:

- `CustomerService`, responsible for handling the data related to the customers of the web shop.
- `OrdersService`, responsible for the data related to customer orders.
- `ItemsService`, responsible for the data related to items available in the web shop.

Additionally, the MSA includes two BFF gateways components for each client platform of the application (mobile and desktop), in furtherance of employing best practices when it comes to implementing a microservice architecture and fully validating the web shop application on multiple platform types.
Figure 14 presents an overview of the architecture of the microservice based web shop application.

It is important to mention that for implementing the microservice based web application using the MaGiC framework the bottom-up approach must be employed. The bottom-up approach refers to the implementation order of each component of the microservice based web application, particularly the microservice suite has to be specified first, then the BFF gateways, and lastly the mobile and desktop clients (see Figure 15).

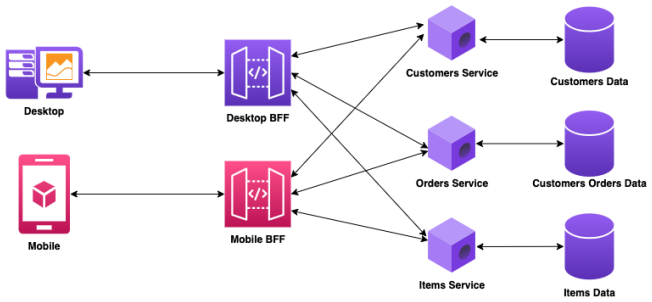

Figure 14 Illustration of the architecture of the microservice based web shop application specified using the MaGiC framework.

The complete MaGiC framework implementation, as well as the specified MSA code base can be accessed in the respective GitHub repository (Bucchiarone et al. 2023). Additionally, the repository contains a user manual for MaGiC which presents a tutorial of building a *Hello World* application using the framework.

6.2. Results

In order to validate the MaGiC framework and determine whether or not the initially set goals of this project were achieved several approaches documented in the literature for software implementation metrics were explored. Additionally, since the framework is focused on the MSA domain, the generated application is analyzed based on the microservice architecture key design principles identified by (Neri et al. 2020) to assess whether the MaGiC framework meets the criteria for being microservice oriented.

6.2.1. Microservice-Orientedness

In the study conducted by (Neri et al. 2020), the research team put forward a comprehensive analysis for microservice architectures in terms of key design principles and architecture problems, which can be used to determine whether the implementation of a MSA meets the criteria for being microservice oriented.

Independent deployability In MSA each microservice should be operationally independent from other services in terms of its deployment. A common issue identified in MSA is the deployment of multiple services in a single container which is against the independent deployability principle since services deployed to the same container are affected when one of the microservices in the container needs to be redeployed. In the context of MaGiC when generating a microservice the Docker configuration is generated as well which configures the packaging and deployment of the service to its own container.

Horizontal scalability The term refers to the ability of a system to add or remove copies of a microservice, a functionality usually implemented by a load balancer. Under this concept, the research team identified to architecture smells, endpoint-based service interactions and no API gateway.
The endpoint-based service interactions traces back directly to the load balancing ability of a system and occurs when the communication between the services is hardcoded and a specific microservice is always called rather than its replicas. For the MaGiC framework this is not an issue due to the fact that the current state of the system does not implement load balancing, but since the infrastructure for enabling it is in place with further development this could be achieved.
Secondly, since MaGiC includes a DSL for generating a BFF which by design is an API gateway from this perspective MaGiC meets the criteria of microservice-orientedness.

Isolation of failures As its name asserts the concept refers to the failure resilience property of a microservice suite so that when a failure happens a service is still able to serve another microservice without creating a snow-balling effect where all the services become unresponsive due to their dependability on each other. When it comes to the MaGiC framework isolation of failure was not a focus point since due to its current state the microservices do not communicate with each other and only the gateway communicates with a given service resulting in a small number of communication requests.

Decentralisation By definition microservice architecture must implement decentralization by design in terms of the system functionality, business logic where each service serves a single domain logic, and a dedicated database. Furthermore, decentralisation not only applies to the functionality of a microservice but also to the implementation technologies and the teams that own a suite of microservices. In the case of MaGiC, the Microservice DSL allows for specifying a service which deals with only a single business domain logic and interacts with a specific database. Additionally, the framework is language agnostic and currently supports generation of services to both Node.js and Python programming languages, additionally it allows for different teams to own different services.

6.2.2. End-To-End testing

End-To-End testing has been used to determine whether or not the generated MSA actually works, once the application was generated the two clients were tested using the end-to-end software testing (Waseem et al. 2020) approach.
This approach has been selected due to the fact the method

validates the entire software from starting to end, as its purpose is to test the whole application dependencies, communication with different services, databases, in a production like scenario (Waseem et al. 2020).
Moreover, it can be argued that by implementing the web shop MSA using the MaGiC framework, an end-to-end test was performed on the framework itself as all the different components of the framework were used in order to specify and generate the microservice based web shop application. Additionally, the deployment and building of the application into Docker containers was tested as well.
Furthermore, by performing the end-to-end tests the language and technology agnostic aspect of the MaGiC framework was evaluated as well due to the fact that the different components of the application were generated to different underlying technologies. Thus, by performing the test cases it was determined whether or not despite the fact that the services are implemented in different technologies they are able to work seamlessly together. The end-to-end test cases for both desktop and mobile platforms can be seen below in Tables 3 and respectively 4.

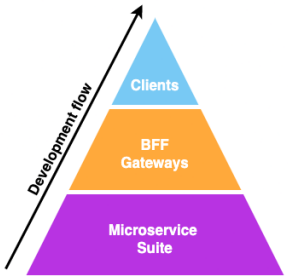

Figure 15 Illustration of the bottom-up approach for implementing a microservice based complete application using the MaGiC framework.

Scenario	Test Case
Login	Login in the webshop with one of the customers credentials, email "test@john.dk" and password 12345.
Navigation	Test navigation to Items and Cart pages.
Single item view	Access a single item page by clicking the "See more button" in the Items page.
Buy item	Access single's item page by clicking the "See more button" in the Items page. Buy an item by clicking "Buy Item" button and navigate to the Cart page to determine that the item was added to cart.
Delete item	Access single's item page by clicking the "See more button" in the Items page. Delete an item by clicking "Delete" button and navigate to the Items page to determine that the item was deleted.

Table 3 Desktop end-to-end test cases.

Scenario	Test Case
Login	Login in the webshop with one of the customers credentials, email "test@john.dk" and password 12345.
Navigation	Test navigation to Items, Cart, Create Item pages.
Single item view	Access a single item page by clicking the "See more button" in the Items page.
Buy item	Navigate to Items page and buy an item by clicking "Buy Item" button and navigate to the Cart page to determine that the item was added to cart.
Create item	Navigate to Create Items page and create an item by introducing the required information and pressing "Create item" button. Navigate to Items page to determine the item was added in the shop.

Table 4 Mobile end-to-end test cases.

Additionally, as the Client DSL allows for declaring UI components which use the Bootstrap design language an emphasise was placed on the aspects of the two clients, mobile and desktop, in terms of their design and use of responsive web design (RWD), which refers to the development of web pages that look good on devices with different sizes (Bryant & Jones 2012). Figure 16 presents two illustrations of the client platforms where it can be seen that the layout of the page shifts from one client platform to another indicating a responsive web design.
Lastly, it is important to mention that despite the two clients communicate with their own respective BFF and the operation APIs were designed for data performance, the network bandwidth of the different client platforms was not measured due to the minimal amount of data sent over the network.

The results of the end-to-end tests were positive meaning that all the performed test cases on the generated microservice based web shop application passed. Additionally, as the MaGiC framework was used to implement, deploy, and build the whole MSA it can be stated that end-to-end test for the MaGiC framework passed as well. Furthermore, the technology and language agnostic aspect of the framework was validated as well due to the fact the end-to-end tests of the MSA passed which components are implemented using different technologies. In addition to this, deployment and building of the application into Docker container was also validated when end-to-end testing the implemented web shop MSA.

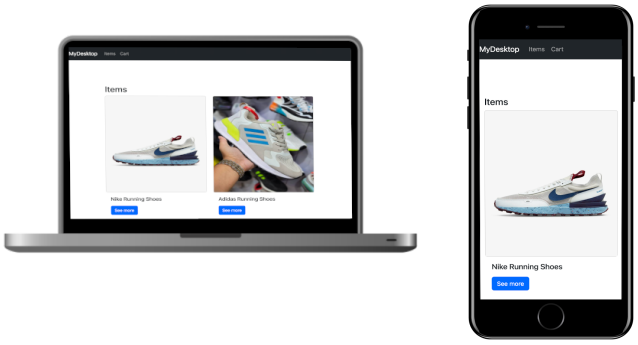

Figure 16 Illustration of the web shop application running on two different client platforms.

6.2.3. Cycle Time Cycle time refers to the span of time required for an engineer to deploy the implemented code to production from the moment he started working on it, that

is after the formal requirements have been identified. With other words, cycle time indicates the time required to complete a particular development task. The cycle time is used for determining the development velocity of teams as well as their efficiency and ability to deliver a working piece of software within a confined time frame (Agrawal & Chari 2007).

In the context of validating the MaGiC framework the cycle time was tracked for implementing the microservice based web shop application using the three main DSL pillars. The evaluation was conducted by only one of the authors of this paper on an Apple MacBook Pro 2017, and the following required software was pre-installed:

MPS , version 2021.2
Node.js , Node.js 16.14.0
Docker , 4.11.1

In the context of the evaluation the cycle time is defined as the span of time required from the creation of the sandbox solution in MPS to the point in time when the application was deployed to Docker containers and ready to be used, and it was tracked using the Apple iOS Stopwatch application.
The results of the evaluation concluded that the required cycle time to implement the MSA web shop for a single developer was approximately ***3h 27m***.

The cycle time can be considered low for developing the whole application compared with the manual process, nevertheless the result is influenced by several variables and therefore is biased. This is due to the fact that the skill of person using the MaGiC framework is highly impacting the cycle time result, meaning that a more skilful person would require less time to implement the microservice based web application.
However, implementing the MSA manually, without using the DSL, would require several skills such as knowledge of microservice architecture and its related software design, Node.js and Python programming languages, React SPA development, and lastly Docker deployment configuration. Thus, it can be stated that as the user would have to have experience with the MaGiC framework to implement a microservice based web application, which requires time, making the learning curve steeper and therefore increasing the cycle time.

7. Conclusion

In this article we propose MaGiC a DSL-based framework which allows for specifying a microservice-based web application that consists of a microservice suite (server-side), a communication layer / gateway, and a client-side. Additionally, it was decided that the framework must be language agnostic, eliminate the need of writing boilerplate code templates for configuration the microservice suite, as well as to provide a tool for building and deploying the application to Docker containers. Furthermore, it was determined that the generated MSA must be developed employing the best practices and software design patterns when it comes to implementing a microservice architecture, and the MaGiC framework should ease the overall development process of a MSA.
In terms of the syntax of the three DSLs of the MaGiC framework only textual notation was considered. However, Jetbrains' MPS allows for other notation as well such as tabular or graphical notations. Due to the scope of this article, other notations than textual were not considered however they could aid at increasing the overall readability of the DSLs thus accomplishing one of the projects goals even further.
Having implemented the MaGiC DSL framework, in order to determine whether the framework meets the outlined objectives of the paper it was decided to perform a validation process by implementing an MSA based web application from end-to-end. Initially, the microservice-orientedness of the generated MSA was assessed based on the criteria put forward by (Neri et al. 2020) and it can be concluded that in the current state of the framework meets most of the requirements but further MaGiC versions, horizontal scalability and isolation of failures ought to be implemented.
In terms of the end-to-end testing, it was completed without issues, and based on the fact that the different components of the MSA were generated to different programming languages the language agnostic aspect of the MaGiC framework was validated as well. Additionally, the results for the required cycle time to implement the MSA, which was approximately ***3h 27 m***, suggests the framework does ease the development process of MSA.
It is worth mentioning that another experiment that could have been performed is to compare the readability of the DSLs with the readability of the actual generated programming language, which would give the authors of the article additional insights about the developed framework.
To conclude, it can be stated that the objective of developing a framework which eases the process when it comes to develop a MSA was met due to the fact that implementing a MSA manually would require knowledge of microservice architecture, suitable software design patterns, single-page application development, as well as Docker deployment and configuration.

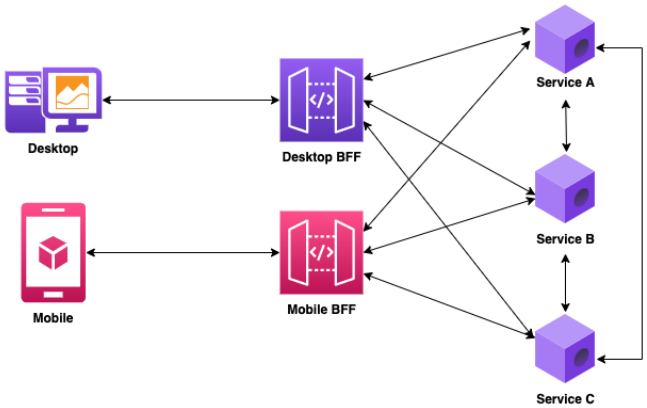

Figure 17 Illustration of the architecture of the MSA supporting both East/West and North/South microservice communication.

8. Future work

Due to the scope of the article aspects such as security, availability, asynchronous message queuing, and monitoring were not a focal point. Therefore, for the further development of the MaGiC framework these aspects of MSA development ought to be considered and incorporated in the framework.
The current implementation of the microservice suite facilitates only a North/South communication, that is the services are able to communicate only with external components, in this case the BFF gateway. However, in a more complex MSA the services communicate with other services as well, East/West communication (Amaral et al. 2015) (see Figure 17). Additionally, the communication between the microservices could be extended to use an asynchronous communication model such as publish / subscribe messaging (Sachs et al. 2010) to align with industry modern standards of implementing MSA.

Moreover, the Client DSL provides a limited amount of UI components therefore the complexity of the UI in terms of its design its also limited. Thus, in order to further add value to the MaGiC framework more UI components could be implemented in the language. The current implementation of the MaGiC framework does not support the use of database management systems. Particularly, the data with which the microservices interact is stored in a locally stored JSON file which is somewhat limited. Therefore, it would be highly beneficial for the MaGiC framework to allow storing of data in a database system and expose the necessary operations for interacting with the data.

Additionally, despite the fact that throughout the paper the concept of DevOps was not described, the MaGiC framework would highly benefit on employing DevOps techniques. DevOps is a concept for integration of operational and development infrastructure that merges software roles in order to enhance communication, improve deployment rate, and maintain a high quality of the software. Moreover, it helps teams to increase confidence in the applications they build, respond better to customer needs, and achieve business goals faster, as well as continually provide value to customers by producing better, more reliable products.

Two important aspects of DevOps are Continuous delivery and Continuous integration (CI / CD) techniques (Jha & Khan 2018). In the context of MaGiC, CI / CD techniques ought to be used for the framework itself allowing the developers of the framework to frequently provide updates and prevent issues by detecting potential integration errors through automated build pipelines. Furthermore, same practices could be facilitated by the framework for the users of the MaGiC DSL allowing them to easily deploy their MSA based application and verify their correctness.

When it comes to the development practice envisioned for the MaGiC framework, the goal is to maintain the language-agnostic aspect of the framework. This means that in the future the framework will have to evolve so it supports building more complex applications. Nevertheless, the research group does understand that the framework cannot accommodate for all development use cases, therefore, in the future, the framework could support embedding of add-ons written in a general-purpose language so it allows framework users to achieve the desired functionality.

Lastly, the MaGiC framework does not provide any guarantees that the functionally of the specified MSA based web application works as expected. Therefore, similar as in the case of Sliceable Monolith approach by (Montesi et al. 2021) , the framework would highly benefit of automatically generating integration tests and model checking approaches to determined that the generated architectural model is sound.

References

Aggarwal, K., Singh, Y., & Chhabra, J. (2002). An integrated measure of software maintainability. In *Annual reliability and maintainability symposium. 2002 proceedings (cat. no.02ch37318)* (p. 235-241). doi: 10.1109/RAMS.2002.981648

Agrawal, M., & Chari, K. (2007, 04). Software effort, quality, and cycle time: A study of cmm level 5 projects. *Software Engineering, IEEE Transactions on*, *33*, 145-156. doi: 10.1109/TSE.2007.29

Aldayel, A., & Alnafjan, K. (2017). Challenges and best practices for mobile application development: Review paper. In *Proceedings of the international conference on compute and data analysis* (p. 41–48). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3093241.3093245

Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M., & Steinder, M. (2015). Performance evaluation of microservices architectures using containers. In *2015 ieee 14th international symposium on network computing and applications* (p. 27-34). doi: 10.1109/NCA.2015.49

Bennett, K., & Rajlich, V. (2000, 05). Software maintenance and evolution: a roadmap. In (p. 73-87). doi: 10.1145/336512.336534

Blinowski, G., Ojdowska, A., & Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, *10*, 20357-20374. doi: 10.1109/ACCESS.2022.3152803

Borum, H. S., Niss, H., & Sestoft, P. (2021). On designing applied dsls for non-programming experts in evolving domains. In *2021 acm/ieee 24th international conference on model driven engineering languages and systems (models)* (p. 227-238). doi: 10.1109/MODELS50736.2021.00031

Bryant, J., & Jones, M. (2012). Responsive web design. In *Pro html5 performance* (pp. 37–49). Berkeley, CA: Apress. doi: 10.1007/978-1-4302-4525-4_4

Bucchiarone, A., Cicchetti, A., Ciccozzi, F., & Pierantonio, A. (2021). *Domain-specific languages in practice with jetbrains mps* (1st ed.). Springer Publishing Company, Incorporated.

Bucchiarone, A., Ciumedean, C., Soysal, K., Dragoni, N., & Pech, V. (2023, May). *magic-dsl-framework.* Zen-

odo. Retrieved from https://doi.org/10.5281/zenodo.7971803 (MaGiC: a DSL Framework for Implementing Language Agnostic Microservice-based Web Applications) doi: 10.5281/zenodo.7971803

Bucchiarone, A., Dragoni, N., Dustdar, S., Lago, P., Mazzara, M., Rivera, V., & Sadovykh, A. (2019). *Microservices: Science and engineering* (1st ed.). Springer Publishing Company, Incorporated.

Costa, P. H. T., Canedo, E. D., & Bonifácio, R. (2018). On the use of metaprogramming and domain specific languages: An experience report in the logistics domain. In *Proceedings of the vii brazilian symposium on software components, architectures, and reuse* (p. 102–111). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3267183.3267194

Di Francesco, P., Malavolta, I., & Lago, P. (2017). Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *2017 ieee international conference on software architecture (icsa)* (p. 21-30). doi: 10.1109/ICSA.2017.24

Fink, G., & Flatow, I. (2014). Introducing single page applications. In *Pro single page application development: Using backbone.js and asp.net* (pp. 3–13). Berkeley, CA: Apress. doi: 10.1007/978-1-4302-6674-7_1

Fowler, M. (2005). *Language workbenches: The killer-app for domain specific languages?* Retrieved 2022-09-16, from https://martinfowler.com/articles/languageWorkbench.html

Gackenheimer, C. (2015). What is react? In *Introduction to react* (pp. 1–20). Berkeley, CA: Apress. doi: 10.1007/978-1-4842-1245-5_1

Gaikwad, S. S., & Adkar, P. (2019). A review paper on bootstrap framework. *IRE Journals*, *2*(10), 349–351.

Giallorenzo, S., Montesi, F., Peressotti, M., & Rademacher, F. (2022). Model-driven generation of microservice interfaces: From lemma domain models to jolie apis. In M. H. ter Beek & M. Sirjani (Eds.), *Coordination models and languages* (pp. 223–240). Cham: Springer Nature Switzerland.

Gray, J., Fisher, K., Consel, C., Karsai, G., Mernik, M., & Tolvanen, J.-P. (2008, 10). Dsls: The good, the bad, and the ugly.
doi: 10.1145/1449814.1449863

Guidi, C., Lanese, I., Mazzara, M., & Montesi, F. (2017, 07). Microservices: A language-based approach.. doi: 10.1007/978-3-319-67425-4_13

Irudayaraj, P., & P., S. (2019, 09). Adoption advantages of micro-service architecture in software industries. *International Journal of Scientific & Technology Research*, *8*, 183-186.

Jaramillo, D., Nguyen, D. V., & Smart, R. (2016). Leveraging microservices architecture by using docker technology. In *Southeastcon 2016* (p. 1-5). doi: 10.1109/SECON.2016.7506647

Jha, P., & Khan, R. (2018, 06). A review paper on devops: Beginning and more to know. *International Journal of Computer Applications*, *180*, 16-20. doi: 10.5120/ijca2018917253

Karabey Aksakalli, I., Çelik, T., Can, A., & Tekinerdogan, B. (2021, 06). Deployment and communication patterns in microservice architectures: A systematic literature review. *Journal of Systems and Software*, *180*, 111014. doi: 10.1016/j.jss.2021.111014

Martin Fowler, J. L. (2014). *Microservices.* (https://martinfowler.com/articles/microservices.html (visited: 2022-09-13))

McCracken, D. D., & Reilly, E. D. (2003). Backus-naur form (bnf). In *Encyclopedia of computer science* (p. 129–131). GBR: John Wiley and Sons Ltd.

Mernik, M., Heering, J., & Sloane, A. (2005, 12). When and how to develop domain-specific languages. *ACM Comput. Surv.*, *37*, 316-. doi: 10.1145/1118890.1118892

Montesi, F., Guidi, C., & Zavattaro, G. (2014). Service-oriented programming with jolie. In A. Bouguettaya, Q. Z. Sheng, & F. Daniel (Eds.), *Web services foundations* (pp. 81–107). New York, NY: Springer New York. doi: 10.1007/978-1-4614-7518-7_4

Montesi, F., Peressotti, M., & Picotti, V. (2021). Sliceable monolith: Monolith first, microservices later. In *2021 ieee international conference on services computing (scc)* (p. 364-366). doi: 10.1109/SCC53864.2021.00050

Neri, D., Soldani, J., Zimmermann, O., & Brogi, A. (2020). Design principles, architectural smells and refactorings for microservices: a multivocal review. *Software-intensive Cyber-physical Systems*, *35*(1-2), 3-15. doi: 10.1007/s00450-019-00407-8

Okwu, P. I., & Onyeje, I. N. (2014). Software evolution: Past, present and future..

Overflow, S. (2021). *Stack overflow developer survey 2021.* Retrieved 2022-09-14, from https://insights.stackoverflow.com/survey/2021?_ga=2.115807297.675239690.1628167975-2066421306.1628167975#technology

Pavlenko, A., Askarbekuly, N., Megha, S., & Mazzara, M. (2020, 05). Micro-frontends: application of microservices to web front-ends.
doi: 10.22667/JISIS.2020.05.31.049

Rademacher, F., Sachweh, S., & Zündorf, A. (2019). Aspect-oriented modeling of technology heterogeneity in microservice architecture. In *2019 ieee international conference on software architecture (icsa)* (p. 21-30). doi: 10.1109/ICSA.2019.00011

Rademacher, F., Sorgalla, J., Sachweh, S., & Zündorf, A. (2019). Viewpoint-specific model-driven microservice development with interlinked modeling languages. In *2019 ieee international conference on service-oriented system engineering (sose)* (p. 57-5709). doi: 10.1109/SOSE.2019.00018

Rademacher, F., Sorgalla, J., Wizenty, P., Sachweh, S., & Zündorf, A. (2020). Graphical and textual model-driven microservice development. In A. Bucchiarone et al. (Eds.), *Microservices: Science and engineering* (pp. 147–179). Cham: Springer International Publishing. doi: 10.1007/978-3-030-31646-4_7

Rademacher, F., Sorgalla, J., Wizenty, P., & Trebbau, S. (2022, 09). Towards holistic modeling of microservice architectures using lemma..

Sachs, K., Appel, S., Kounev, S., & Buchmann, A. (2010). Benchmarking publish/subscribe-based messaging systems. In M. Yoshikawa, X. Meng, T. Yumoto, Q. Ma, L. Sun, & C. Watanabe (Eds.), *Database systems for advanced applications* (pp. 203–214). Berlin, Heidelberg: Springer Berlin Heidelberg.

Sorgalla, J., Wizenty, P., Rademacher, F., Sachweh, S., & Zündorf, A. (2018, 09). Ajil: Enabling model-driven microservice development.. doi: 10.1145/3241403.3241406

Team, J. M. (2021). *Generator.* Retrieved 2022-09-04, from https://www.jetbrains.com/help/mps/mps-generator.html#overview

Terzić, B., Dimitrieski, V., Kordić (Aleksić), S., Milosavljevic, G., & Luković, I. (2017, 03). Microbuilder: A model-driven tool for the specification of rest microservice architectures..

Waseem, M., Liang, P., Márquez, G., & Salle, A. D. (2020). Testing microservices architecture-based applications: A systematic mapping study. In *2020 27th asia-pacific software engineering conference (apsec)* (p. 119-128). doi: 10.1109/APSEC51365.2020.00020

Wasp. (2022). *Wasp (web application specification language).* (https://wasp-lang.dev (visited: 2022-09-14))

Zhao, J., Jing, S., & Jiang, L. (2018, 09). Management of api gateway based on micro-service architecture. *Journal of Physics: Conference Series*, *1087*, 032032. doi: 10.1088/1742-6596/1087/3/032032

Zimmermann, O., Stocker, M., Lübke, D., Zdun, U., & Pautasso, C. (2022). *Patterns for api design - simplifying integration with loosely coupled message exchanges*. Boston: Addison-Wesley Professional.

About the authors

Antonio Bucchiarone is a senior researcher in the Motivational Digital Systems (MoDiS) research unit of FBK in Trento, Italy. His main research interests are: Self-Adaptive Systems, Domain Specific Languages for Socio-Technical System, and AI planning techniques for Automatic and Runtime Service Composition. He has been actively involved in various research projects in the context of Self-Adaptive Systems, Smart Mobility and Constructions and Service-Oriented Computing. He is an Associate Editor of IEEE Software, IEEE Transactions on Intelligent Transportation Systems, and IEEE Technology and Society Magazine. You can contact him at bucchiarone@fbk.eu.

Claudiu Ciumedean is a software engineer at Microsoft and a fresh graduate from the Technical University of Denmark with a M.Sc. degree in Computer Science and Engineering. He is currently involved in several software development projects and has a strong interest in web technologies, distributed systems, and virtual reality publishing a couple of research papers within the area. You can contact him at claudiuciumedean@gmail.com.

Kemal Soysal is a software professional at LS IT-Solutions GmbH, Berlin, Germany. His main research interests are: Model Driven Software Development, Domain Specific Languages, Software Architecture, Infrastructure Architecture. You can contact him at kemal.soysal@ls-it-solutions.de.

Nicola Dragoni Nicola Dragoni is Professor in Secure Pervasive Computing at DTU Compute, Technical University of Denmark, where he also serves as Head of Section (Cybersecurity Engineering), Head of the DTU Center for Digital Security (DIGISEC) and Deputy Head of the DTU Compute's PhD School. Nicola Dragoni received the M.Sc. (cum laude) and Ph.D. degrees in Computer Science from University of Bologna, Italy. His main research interests centre around pervasive computing and security, with latest focus on Internet-of-Things, Fog/Edge computing and mobile systems. He has co-authored 140+ peer-reviewed scientific papers in international journals and conference proceedings. He has edited 3 journal special issues and 1 book. He has been active in several national and international projects. You can contact him at ndra@dtu.dk.

Václav Pech is a senior software developer at JetBrains and a part-time lecturer at the Charles University in Prague. With a M.Sc. degree from Charles' University in Prague and more than 20 years experience working as a software developer/consultant, he's keenly interested in server-side Java technologies, modern programming languages, domain specific languages, distributed and parallel systems and agile methodologies. He's currently involved in the JetBrains MPS project, developing a projectional DSL workbench and building customized DSLs. You can contact him at vaclav.pech@jetbrains.com.

A. Appendix

A.1. Microservice DSL

```
microserviceName ::= <string>
version ::= <string>
description ::= <string>
maintainerEmail ::= <string>

port ::= <integer>
language ::= Python | NodeJs
dataReference ::= <DataType>

Operation features syntax:

method ::= CreateEntity | DeleteEntity | GetEntities | GetEntitiesBy |
    GetEntity | GetEntityBy | UpdateEntity
type ::= CREATE | READ | UPDATE | DELETE
entityType ::= <EntityType>
queryParams ::= <key> { " " <key> }
expectedPayload ::= Empty | Entity | EntityID | Entities
deliveredPayload ::= Empty | Entity | EntityID | Entities
successMessage ::= [ <string> ]
errorMessage ::= [ <string> ]

The DataType is represented by the following syntax:

<DataType> ::= <value>, where <value> represents a valid JSON array
   data structure

Respectively, the EntityType is represented by the following syntax:

<EntityType> ::= { <key> : <value> } { "," <key> : <value> }, where <
   key> is a string constant that defines a data property, and <value>
    ::= string | number | integer | boolean
```

Figure 18 Representation of Microservice DSL features using BNF notation.

```
Microservice name <microserviceName>
version <version>
description <description>
maintainer email <maintainerEmail>

port <port>
language <language>
data reference:
 <dataReference>

exposes
  operation <method> with type <type> on <dataType> data
    with entity type <entityType>
    at endpoint location with query params <queryParams>
    expecting payload <expectedPayload>
    delivering payload <deliveredPayload>
    with success message <successMessage>
    and reporting error message <errorMessage>
```

Figure 19 Microservice DSL syntax.

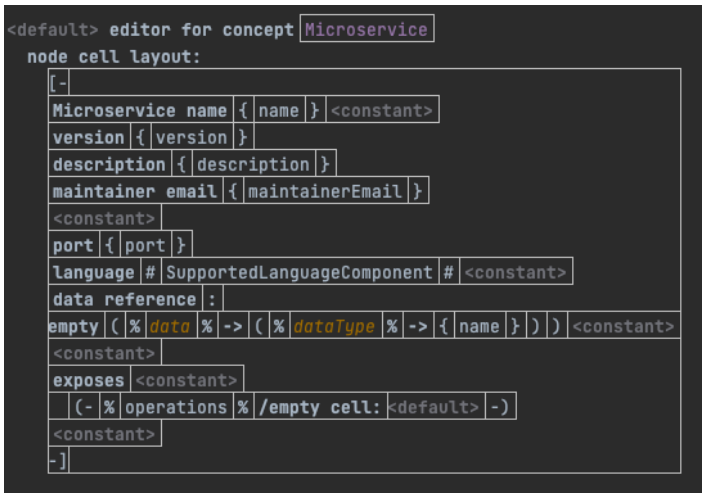

Figure 20 Illustration of the MPS editor aspect used for declaring the syntax of the Microservice DSL.

A.2. Gateway DSL

```
BFFName ::= <string>
version ::= <string>
description ::= <string>
maintainerEmail ::= <string>

port ::= <integer>
language ::= Python | NodeJs

Operation features syntax:

method ::= CreateEntity | DeleteEntity | GetEntities | GetEntitiesBy |
    GetEntity | GetEntityBy | UpdateEntity
type ::= CREATE | READ | UPDATE | DELETE
route := <string>
entityType ::= <EntityType>
queryParams ::= <key> { " " <key> }
deliveredPayload ::= Empty | Entity | EntityID | Entities
expectedProperties ::= <key> { " " <key> }

microservice ::= <MicroserviceName>
microserviceEndpointLocation ::= <string>
```

Figure 21 Representation of Gateway DSL features using BNF notation.

```
BFF name <BFFName>
version <version>
description <description>
maintainer email <maintainerEmail>

port <port>
language <language>

exposes
  operation <method> with type <type> at endpoint location <route> on
      entity type <entityType>
    delivering query params <queryParams> or payload type
    <deliveredPayload>
    expecting properties <expectedProperties>
  which communicates with
    microservice <microservice>
    at endpoint location <microserviceEndpointLocation>
```

Figure 22 Gateway DSL syntax.

```
concept BFFOperation extends    BaseConcept
                    implements IOperationType
                               IOperationMethod
                               IDeliveredPayloadType
                               ScopeProvider

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  route             : string
  microserviceRoute : string

  children:
  microservice         : MicroserviceReference[0..1]
  entityTypeRef        : EntityTypeReference[0..1]
  expectedQueryParams  : KeyValuePairReference[0..n]
  deliveredQueryParams : KeyValuePairReference[0..n]
```

Figure 23 Illustration of the defined structure of the Gateway operation using MPS.

A.3. Client DSL

```
Client configuration syntax
    appName ::=  <string>
    port ::= <string>
    mobileClient ::= <ClientName>
    desktopClient ::= <ClientName>

Client syntax
    name ::= <string>
    maintainerEmail ::= <string>
    description ::= <string>
    version ::= <string>
    clientType ::= Mobile | Desktop

    globalState ::= <string> { " " <string> }

Page syntax
    pageName ::= <string>
    bff ::= <BFFName>
    bffRoute ::= <string>

    pageRoute ::= <string>
    queryParameters ::= <key> { " " <key> }
    globalState ::= { <globalState> : <key> } { "," <globalState> :
    <key>}, where <globalState> is one of the specified global state
       properties, and <key> refers to a string constant that is a
       property  of an <EntityType>.
    entityType ::= <EntityType>
    showInNavigation ::= boolean
    hasInternalState ::= boolean

Page operation syntax
    operationName ::= <string>
    bff ::= <BFFName>
    bffRoute ::= <string>
    type ::= CREATE | READ | UPDATE | DELETE
    entity ::= <EntityType>

    deliveredQueryParams ::= <key> { " " <key> }
    deliveredGlobalStateProperties ::= <globalState> { " " <
       globalState> }
    deliveredPayload ::= Empty | Entity | EntityID | Entities
    redirectsTo ::= <PageName>
    globalState ::= { <globalState> : <key> } { "," <globalState> : <
       key>}, where <globalState> is one of the specified global state
        properties, and <key> refers to a string constant that is a
       property  of an <EntityType>.

    components =  <Component> { " " <Component> }

<EntityType> ::= { <key> : <value> } { "," <key> : <value> }, where <
   key> is a string constant that defines a data property, and <value>
    ::= string | number | integer | boolean
```

Figure 24 Representation of the Client DSL abstract syntax using Backus-Naur form notation.

```
<Title>
    text ::= <string>
</Title>

<Text>
    text ::= <string>
 </Text>

<Link>
    text ::= <string>
    links to ::= <PageName>
    queryParms ::= <key> { " " <key> }
</Link>

<Input>
    name ::= <string>
    label ::= <string>
    type ::= email | number | password | tel | text
    required ::= boolean
 </Input>

<Image>
    imgSrc ::= <string>
    altText ::= <string>
</Image>

<Form>
    interactsWithOperation ::= <OperationName>
    actionText ::= <string>
    inputs ::= <Input> { " " <Input> }
</Form>

<CrudAction>
    text ::= <string>
    variant ::= primary | secondary | success | danger | warning |
       info | link
    interactsWithOperation ::= <OperationName>
 </CrudAction>

<Card>
    image ::= <string>
    title ::= <string>
    action ::= <CrudAction>
 </Card>
```

Figure 25 Representation of the Client DSL UI component syntax.

```
Client configuration abstract syntax

App name:  <appName>
Port: <port>
Mobile client: <mobileClient>
Desktop client: <desktopClient>

Client abstract syntax

Name: <name>
Maintainer email: <maintainerEmail>
Description: <description>
Version: <version>
Client type: <clientType>

Global state {
  <globalState>
}

Page <pageName> {
  Communicates with <bff> BFF and fetches state from <bffRoute>
  Route <pageRoute> with query params  <queryParams> and global state
     <globalStateRef> of entity type <entityType>

  Show in navigation: <showInNavigation>
  Has internal state: : <hasInternalState>

  Operations {
    <operationName> {
      Communicates with <bff> BFF
      at endpoint location <bffRoute> with operation type <type>
      interacts with entity <entityType>

      delivers {
        query params <deliveredQueryParams>
        global state properties <deliveredGlobalStateProperties>
        payload type <deliveredPayload>
      }

      on success {
        redirects to <redirectsTo>
        updates global state <globalState>
      }
    }
  }

  Components {
    <Component>
  }
}
```

Figure 26 Representation of the Client DSL concrete syntax.