General-Purpose Languages vs Language-based for Microservice Development

By Anthony Genson & Alexander Lerche Falk

Table of contents

Abstract 0
Introduction 1
Microservices 1
Languages 2
Circuit Breakers 2
API Gateway 3
Jolie 4
Golang 6
Node.js 8
Conclusions and Future Work 9
Travaux cités 11

Abstract

General-Purpose Languages have been used for decades to create applications. They are not focused on anything specific but are rather generic when it comes to different domains. Language-Based are domain-specific and tries to focus on one domain instead of multiple. In this review, the focus is made on Microservices, and how to implement good practices and guidelines in a language and compare it against two General-Purpose Languages. The Language-Based is Jolie and the two General-Purpose Languages are Nodejs and Golang.

By having a simple service, patterns, and middleware, we created scenarios to be implemented in the three languages. Jolie showed advantages of how it interacted with the request/response communication and how one service communicated with others. The General-Purpose Languages gave us flexibility when we needed it, where Jolie was more restricted, but they also gave us problems, when we had to interact with the request/response communication. Jolie has features, which are helpful when you develop microservices. You are restricted but it gives you a good design by construction of your microservice, meaning you can think less about the pitfalls, which are common when developing applications, where network handling is included.

Introduction

When it comes to distributed systems, software engineers are challenged. The popular languages, such as Java, C#, and C/C++, were built with the design-thoughts of developing single executables – also known as monolith applications [1]. This meant, the ability to distribute the modules of the software was unmanageable due to the monolith design philosophy, where the application is not responsible for *one* task but for every task, which is needed to complete specific functionality.

Later, Service-Oriented Architecture (SOA) [2] became the new black. The philosophy was to divide each service/component of the application. They still shared the communication layer and data storage. The common communication bus – also defined as Enterprise Service Bus (ESB) [3] – is seen as an integration service, controlling the communication between various services. The dividing of services did help software engineers manage applications better. But a new problem rose single point of failure and blocking requests on the ESB. Shared data storage is a pro and con. Pro, when your services must reuse the same data. Con, since your services are tightly coupled with the data storage. This is leading towards other methods on how to tackle the problems occurring in SOA.

Microservices

To accommodate the issues with SOA, a new term dawned: microservices. This is where it gets tricky. What is the difference between SOA and Microservices? [4] Microservices are more independent. They are encapsulated down to the memory, database, and processing power. They do *one* job and they do it well. A real-life example of this is Facebook. Loading friends, retrieving adds, retrieving notifications, retrieving messages, and retrieving posts. Each functionality is independent and gives the flexible of easier maintenance, development, and loose coupling. Even if the service for retrieving friends went down, you still have the possible to use the rest of Facebook.

Dividing every functionality into a service can become tedious to structure. Therefore, you need a microservice architecture [5], which is a distributed application where each module is a microservice. Each microservice is, as said, its own functionality. By having it isolated in such a way, we can limit the amount of code pr. application, leading to less bugs. This also let the developer of the microservice directly test the functionality without having to worry about breaking the system. And as a consequence, version controlling [6] makes continuous integration [7] and software maintenance manageable, as you do not have to worry about every other functionality of the system.

Microservices are following an architectural pattern, which "… *provides a guideline to partition the components of a distributed application into independent entities, each addressing one of its concerns.*" [5]. This means, you are not bound to a specific programming language, but are free to use whatever language you want to.

Languages

We have chosen to compare Jolie against Node.js and Golang. The reason for this decision is: the languages are popular when it comes to building microservices. From Golang's own website [8] it states: "*Go is an open source programming language that makes it easy to build simple, reliable, and efficient software"*, where "simple" is one of definitions of a microservice. The paper from "Microservices: A Language Based Approach" [9] concluded Golang as good examples of being able to develop concurrency and scalable applications, which – also – were some of the reasons for trying out this language.

The languages have been used to build one simple service – a calculator – with the possibility to add other services / patterns in front of it – such as authentication. The Circuit Breaker proxies the requests to the calculator, while preserving error rates and timeouts, and giving it the possibility to be extended with other services, such as authentication. The idea behind this project was to see if Jolie made it easier for the developer to create microservices and tackled some of the hurdles from General-Purpose languages.

We have tested five different scenarios by composing the services / patterns:

1. A simple calculator service alone
2. An authentication system around the calculator
3. The Circuit Breaker pattern over the Calculator service
4. An authentication system covering the Circuit Breaker pattern above
5. The Circuit Breaker pattern over the authenticated system and Calculator

These scenarios are used to determine how the different languages differs and their complexity to achieve a Microservice Architecture (MSA). All the code from the three implementations – which are working versions – can be found on the Github repository [10].

Circuit Breakers

A Circuit Breaker (CB) [11] is a structural design pattern, which sits in between a service and a client. It can either be on the service-side, the client side, or function as a proxy.

CB's are trying to comprehend issues of overloaded – or unresponsive – services. The pattern can be thought of as a failure "wrapper", whose job is to monitor whether the service is having issues, when requests are created. An example is when a client is calling a service, and the response-time of the service is above a certain threshold, the circuit breaker must stop forwarding requests to the service, and takeover the response to the client, which is going to be a fault.

The CB has three states:

- **Closed:** requests are passed to the target service. Meanwhile, the CB is monitoring the behavior that the requests are causing and if it determines one or more thresholds have been breached, it trips to an open state.
- **Open:** the CB is going to return an unavailability error and not execute any request it receives. Then after a given timeout, it changes the state to half-open.
- **Half-Open**: the first request it receives in this state is going to determine the next state. If it succeeds, the state will transit back to closed. Otherwise, the state goes back to being open.

There are certain functionalities, which we have implemented in our applications, which determines when to swap between the three states. (1) When a request is taking longer than a certain amount of time – example 20 seconds – we are going to throw a timeout, which transcends to the Open-state. The next request coming in is immediately going to receive the unavailable response (HTTP Status Code 503). (2) Every request is getting a timestamp and a state: success or failure. This is stored in a certain amount of seconds interval – example 60 seconds. The amount of failures compares to success' determines the error rate, which is a rate describing the allowance of errors in the CB. If the error rate is above ex. 5 %, the state of the CB is changed to Open. Since we are only interested in the past 60 seconds of errors, we are cleaning up all errors before this (rolling window).

In our applications, when it comes to the CB, we are using it as a proxy gateway. It is a service containing the path of the services behind it.

API Gateway

The API Gateway is a service, functioning as a proxy, sitting in between multiple microservices, to ensure the request is being send to the correct destination. We have chosen to include the API Gateway pattern in the Circuit Breaker (CB). Since we are using CB as a proxy service, meaning, it is a standalone microservice, it made sense to include the operations to be performed from the API Gateway.

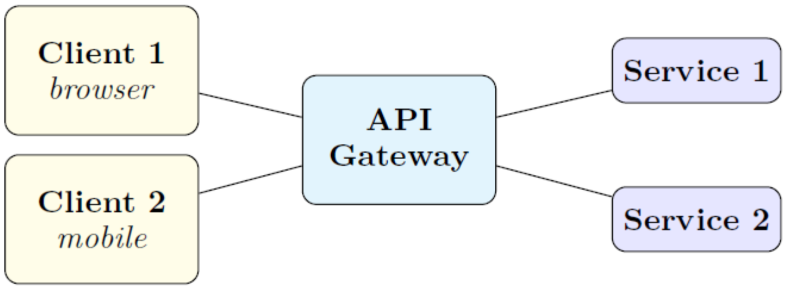


Figure 1 - API Gateway

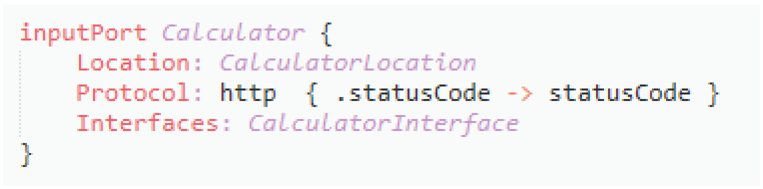
The one problem by doing so is errors are shared among the microservices. But in our case, if the calculator is not working, and we are having the authenticator in front of it, we accept this problem.

Jolie

From the Jolie homepage [12] it states:

"Jolie crystallises the programming concepts of microservices as native language features: the basic building blocks of software are not objects or functions, but rather services that can always be relocated and replicated as needed. Distribution and reusability are achieved by design".

Jolie is by design created to develop microservices, where it is focusing on communication, interfaces, and dependencies [9]. Jolie handles the communication between microservices by two port types: inputPort and outputPort [9]. inputPorts exposes the ports to be communicated with different microservices, whereas outputports can invoke the functionality provided by the inputPort. There is support for multiple communication protocols such as: Bluetooth, sockets, HTTP, SOAP, and Sodep. Jolie sets up its communication by writing:



```
inputPort Calculator {
    Location: CalculatorLocation
    Protocol: http  { .statusCode -> statusCode }
    Interfaces: CalculatorInterface
}
```

Figure 2 - normal inputPort in Jolie

Where the location defines how and where the communication is specified. The protocol defines which protocol to use for transferring data. And the interface defines "... *the functionalities that services provides to and require from the environment"* [9], where functionalities are operations which can be invoked via remote or local calls. The smart thing about this setup – and an advantage in Jolie – is the abstraction of being able to swap out location and protocol without breaking all your code.

Jolie also offers an aggregation system directly in the inputPort, which plays the role of a Proxy for the outPorts it aggregates. It is something really handy for creating proxies, allowing to change protocols on the way, or even add some behavior before forwarding a request. That way we can easily add an authentication system in the proxy:

```
inputPort AuthenticatedCalculator {
    Location: ProxyLocation
    Protocol: http
    Aggregates:
        Calculator with ProxyInterface_Extender,
        Authentificator
}
```

Figure 3 - inputPort in Jolie with aggregation and interface extension

In the example above, the Calculator service don't even have to be touched, the proxy can extend the interface, adding information to the required data (api key) and can even modify the data answered. This behavior is added via the "courier":

```
courier AuthenticatedCalculatorSodep {
    [ interface CalculatorInterface( request )( response ) ] {
        install(
            InvalidKey => println@Console("Error: InvalidKey")(),
            ZeroDivisionError => println@Console("Error: ZeroDivisionError")()
        );
        check_key@AuthentificatorSodep( { .key = request.key } )( key_info );
        forward( request )( response );
        with( response ) { .key_info << key_info }
    }
}
```

Figure 4 - courier in Jolie as middleware for authentication

This system can be applied for a lot of different patterns, like the circuit breaker pattern:

```
courier CircuitBreaker {
    [ interface CalculatorInterface( request )( response ) ] {
        install( CircuitBreakerFault => println@Console("Error: CircuitBreakerFault")() );
        getState;

        if (state == State_Open) throw( CircuitBreakerFault, "Server not available." )
        else {
            scheduleTimeout@Time( int(CallTimeout*1000) { .operation = "callTimeout" } )( timeoutID );
            install( TypeMismatch => handleError );

            forward( request )( response );
            handleSuccess
        }
    }
}
```

Figure 5 - courier in Jolie as middleware for Circuit Breaker pattern

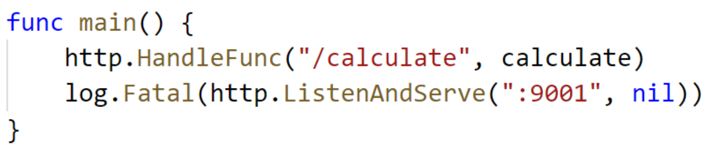
While implementing the Jolie example, we first created an operations service. Then we thought of having only one path for the calculator, and just specify the operator in the body of the request. To do that, we just had to create the calculator service which aggregates the operations service without really touching.

Except from having to specify the ports and interface manually, for example for the courier, this model allows real loose coupling between the services. The existing services don't have to be touched for adding new behaviors, they stay completely independent. This can also be done in other languages of course, but Jolie just provides this feature naturally instead of having to find out loopholes.

Jolie takes some time to understand while learning it, as the logic differs from the other languages we can be used to. But in a way, it is really well thought for microservices purpose, even creating a set of rules for them.

Golang

Golang (we are using Go for short) comes with a standard library "*net/http*", which is used to perform the HTTP communication. It is easy to setup a service in Go, as illustrated in Figure 6:

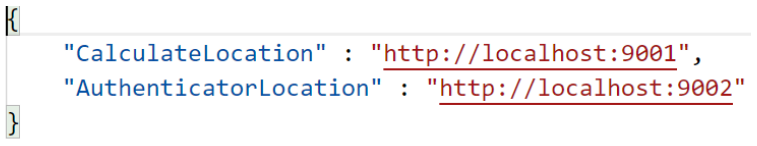


```go
func main() {
    http.HandleFunc("/calculate", calculate)
    log.Fatal(http.ListenAndServe(":9001", nil))
}
```

Figure 6 - Setup HTTP Communication with Go by performing two lines of code

With two lines of code, we can listen and serve our microservice. The first line uses the operation "*HandleFunc"* from the net/http library, which creates a new HTTP route and adds the functionality to be performed, when requested. The second line, the "*http.ListenAndServe"* operation, is starting the server and is wrapped in a logging framework, giving us the opportunity to log errors to the console, if it appears.

The CB forwards the requests if the state is Closed and the error rate is not breached. The information regarding where to forward the requests comes from a json file - Figure 7:

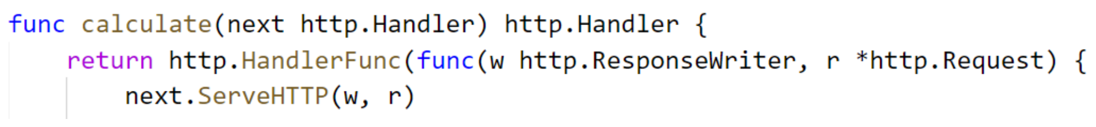


```json
{
    "CalculateLocation" : "http://localhost:9001",
    "AuthenticatorLocation" : "http://localhost:9002"
}
```

Figure 7 - JSON file containing the information regarding the service locations

One disadvantage of using Go is its http library. When we are putting the authenticator in front of the calculator, we are passing the request and response as parameters for creating a new request. When a request comes into the calculator it is going to check if the user is logged in or not. If not logged in, it is going to try and login. But the response body is closed after the login check, so we cannot log the user in immediately. This is by design [13] so, you are not surprised when the response body has been changed, but it creates duplication of response bodies. You now have the original response body and the copied version. And for every middleware you are applying, you need to do this copy-trick. It can become memory intensive if the request body is large.

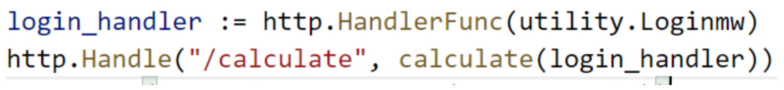
In Go, when you need to perform middleware actions on your functionality, you must ensure your functions is setup in a specific way. The first thing you need to do is to have a 'Handler' – responds to HTTP requests – as a parameter and return the same. A 'Handler' contains one function called ServeHTTP, which has one job: call the middleware (the argument given to the function) and signal when done – seen in Figure 8.



```go
func calculate(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        next.ServeHTTP(w, r)
```

Figure 8 - Shows Go setup for allowing a function to run the next middleware

To see how it works in action, Figure 9 shows how the calculator takes in the middleware for execution. The '*utility.Loginmw*" is an utility created for sending requests against the authenticator service for easier maintaining.



```go
login_handler := http.HandlerFunc(utility.Loginmw)
http.Handle("/calculate", calculate(login_handler))
```

Figure 9 - The setup for calling the middleware functionality on Calculator

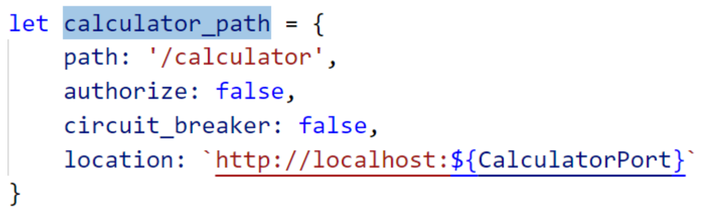
Another disadvantage of using this practice to use middleware functionality is to nest handlers into handlers. In our example, we are only using one handler for the '*Calculator'* parameter, but if we needed some middleware for the login handler, we had to use the same approach, and have it takes in a handler as a parameter as well.

When you start getting into the practices and techniques with Go, microservice development is an ease. But you are required to handle errors, which can be cumbersome and tedious. Other thing you must take care of is, since Go has focus on performance and bandwidth, the request body you get from HTTP comes in bytes format. This mean, you must do transformations on it to get readable, when you want to perform decisions based on the input. This create duplication of handlings for the same tasks. Example: transform the bytes to json to extract information.

Node.js

Node.js comes with an advantage of having data structures confirming as JSON. This create ease of use when it comes to handling the input from the HTTP request, which is JSON. We are using the Express framework, which is a popular framework for Node.js to develop microservices and web services in general. Node.js was also built with the intention of developing applications for the web, which you can feel when you are starting to develop in the language.

In our Node.js architecture, we are having a proxy service, which when initialized / started, it pushes the paths it needs to proxy. In Figure 10 you can see the structure, which holds another service to include:



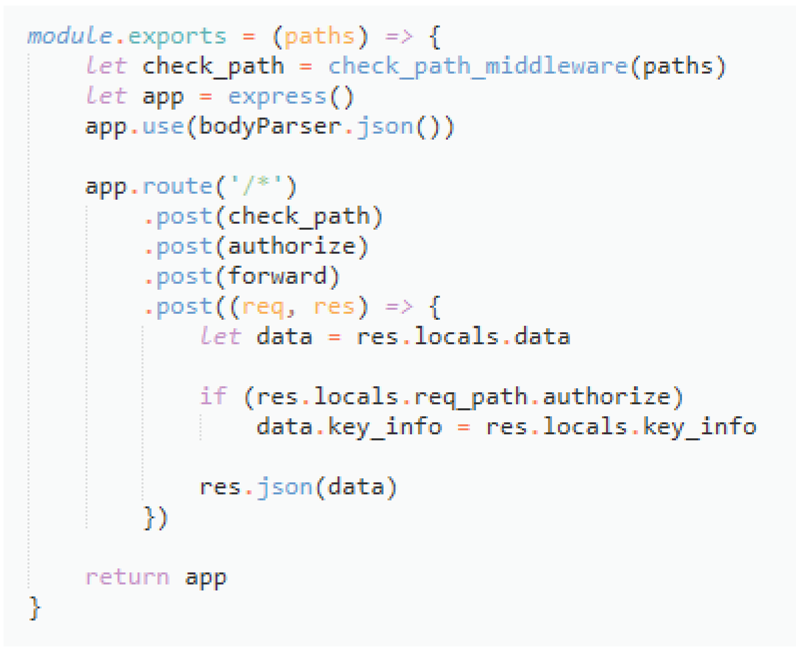
```
let calculator_path = {
    path: '/calculator',
    authorize: false,
    circuit_breaker: false,
    location: `http://localhost:${CalculatorPort}`
}
```

Figure 10 - The data structure of a service path. This is used for the proxy service to figure out where to proxy the requests to

Based on this information given to a structure, the proxy can determine what to do and how to react. We can add or remove behaviors without hurdles. The Express framework comes with a way to nest middleware, which means, we can add behavior for handling the request if needed. **Erreur ! Source du renvoi introuvable.**1 gives the implementation, where it checks if the path is correct, authorizing the user if the "authorize" field of the path structure from Figure 11 is set to true, and then forwards the request either normally or through a circuit breaker if specified in the data structure.

Compared to Go, this is less verbose to use, and does not require lots of code to be refactored when changes are to be done. For example, removing middleware. In Go, you must provide the middleware, since it is not an optional parameter.

This example was actually made to try and imitate the Jolie example, the middlewares being a way to add behavior for handling requests. The fact of having only one flexible proxy controlling everything isn't loose coupling, but the focus was rather made on the flexibility of Node.js to implement such architectures. Node.js, and because of JavaScript, comes really handy to create chunks of code and use it when needed, which is not that obvious with Jolie. Here, the implementation of the circuit breaker was made as a JavaScript class object. It could easily be a package available online ready to use. We just have to import it and use it in a middleware here. In Jolie, we must manually provide the interface the circuit breaker should handle. On the other hand, Node.js don't have any restrictions or rules concerning the development of microservices as Jolie has, which can cause failure more easily.



```
module.exports = (paths) => {
    let check_path = check_path_middleware(paths)
    let app = express()
    app.use(bodyParser.json())

    app.route('/*')
        .post(check_path)
        .post(authorize)
        .post(forward)
        .post((req, res) => {
            let data = res.locals.data

            if (res.locals.req_path.authorize)
                data.key_info = res.locals.key_info

            res.json(data)
        })

    return app
}
```

Figure 11 - Nesting of middleware from the proxy. It checks the path, checks for authorization, and forwards the request as middleware

Conclusions and Future Work

With General-Purpose Languages you get adaptability, meaning, you can create functionality in different ways, which can be great in cases, where you want to create functionality to be imported in different files. This gives no restrictions or typing system to ensure your microservice is working as expected. On the opposite end, you have Language Based, which comes with a set of rules and a syntax for developing microservices. The problem here – at least for Jolie – is: Language Based lacks flexibility in terms of creating functionality to be imported in different ways. In Jolie, you would create a new service, because everything is service oriented. The reason for this choice – at least from the FAQ on the Jolie website [14] – is the insurance of developers not being able to break loose coupling and having hidden data structures, which can break thread-safety. The good thing about this: if you follow the syntax and the set of rules, you are going to have microservices, which is going to ensure it works by construction. From the General-Purposes Languages, you are using a good amount of time to debug, when your microservice is not working as expected.

Go was not a language we knew a ton about. Therefore, it could be the case, not best practices have been used to develop the microservices. If we had a professional Go developer creating this instead of us, there is a chance better practices would have been used and providing less coupling. Node.js on the other hand was closer to us, and therefore easier to create flexibility and loose coupling in the microservices. Beside this, Node.js is built for this type of work, easing up the development of microservices.

Jolie comes with a feature called aggregation, allowing composing services, and is a generalization of network proxies. No behavior must be added; it only delegates the operations to other services. This functionality is smart, when you want to create a proxy service. Aggregation also comes with what is called "courier" in Jolie, which intercepts the messages during aggregation and here you can perform behavior if needed. Jolie also allow an interface extension, which makes possible adding additional data in the message and only on the proxy side, without informing the service it aggregates. This allows to add behavior by nesting and leaves the aggregated services independent. It is possible to implement it in other General-Purposed languages, but it does not come naturally.

Another advantage for Jolie is how it can swap the protocol without affecting how the functionalities of the microservice are working. This saves many hours of development, if you are thinking about swapping protocol in the middle of a project. If we had to change protocol in Node.js or Golang, it would require refactoring a lot of the code, since there are dependencies of how behavior is determined based on the output from the protocol – in this case, the HTTP.

Jolie eases up the development of microservices compared to Go and Node.js. It is not "difficult" to develop microservices in Go and Node.js, but you are getting some headaches which would be nice to avoid. The aggregation/courier-model of requests from Jolie helps with the problems of closed responses or multiple requests, which is what we saw in Go. Even if Jolie is great for microservices development, it still lacks a little flexibility we are used to in other languages. We also experienced a problem with forwarding HTTP errors from the other microservices. This meant, if the calculator failed, it would be the error of the Circuit Breaker you were going to see. This is something we suggest should be investigated. There might be a solution, but it is not something we have stumbled upon. Other than this, if you are interested in building microservices, and want to avoid trivial problems from the General-Purpose Languages, give Jolie a chance.

It is to be stated; Jolie takes hours to understand. The concepts and syntax are different from other languages. Most General-Purpose Languages have similarities and you can "guess" how it works. It is not a language you pick up and learn in two hours. But when you learn it, you can create microservices in a good pace and without worrying too much.

For future work, it could be an idea to check the performance comparison, deployment, and general good software engineering practices.

References

[1] Various, «Monolith Applications,» Wikipedia - The Free Enclopedia, 04 02 2018. [En ligne]. Available: https://en.wikipedia.org/wiki/Monolithic_application. [Accès le 25 05 2019].

[2] Various, «Service Oriented Architecture,» Wikipedia - The Free Enclopedia, 13 05 2019. [En ligne]. Available: https://en.wikipedia.org/wiki/Service-oriented_architecture. [Accès le 25 05 2019].

[3] B. M. H. N. D. S. G. S. B. T. C. U.-U. T. W. Jürgen Kress, «Enterprise Service Bus,» Oracle A/S, 07 2013. [En ligne]. Available: https://www.oracle.com/technetwork/articles/soa/ind-soa-esb-1967705.html. [Accès le 25 05 2019].

[4] I. Miri, «Microservices VS. SOA,» DZONE, 04 01 2017. [En ligne]. Available: https://dzone.com/articles/microservices-vs-soa-2. [Accès le 25 05 2019].

[5] S. G. A. L. M. M. Nicola Dragoni, «Microservices: yesterday, today, and tomorrow,» HAL, Odense, 2017.

[6] J. L. Scott Chacon, «About Version Control,» Git, [En ligne]. Available: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control. [Accès le 25 05 2019].

[7] D. Radigan, «Continous Integration, Explained,» Atlassian, [En ligne]. Available: https://www.atlassian.com/continuous-delivery/continuous-integration. [Accès le 25 05 2019].

[8] Google, «The Go Programming Language,» Google, [En ligne]. Available: https://golang.org/. [Accès le 06 06 2019].

[9] I. L. M. M. a. F. M. Claudio Guidi, «Microservices: A Language-Based,» Springer, Bologna, 2017.

[10] A. Genson et A. Falk, «GitHub - Microservises examples,» [En ligne]. Available: https://github.com/AGenson/microservises-examples.

[11] J. W. Fabrizio Montesi, «Circuit Breakers, Discovery, and API Gateways in,» arXiv, Odense, 2016.

[12] The Jolie Team, «Jolie,» [En ligne]. Available: https://www.jolie-lang.org/. [Accès le 06 06 2019].

[13] bradfitz, «Github Golang Issue Page,» 09 01 2013. [En ligne]. Available: https://github.com/golang/go/issues/4637. [Accès le 08 06 2019].

[14] The Jolie Team, «Jolie Language,» [En ligne]. Available: https://www.jolie-lang.org/faq.html. [Accès le 08 06 2019].

[15] The Jolie Team, «Communication Ports,» March 2019. [En ligne]. Available: https://jolielang.gitbook.io/docs/basics/communication-ports. [Accès le 06 06 2019].