

UNIVERSITY OF SOUTHERN DENMARK

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

SPDM801: MASTER'S THESIS IN COMPUTER SCIENCE

A tool for eliciting patterns in microservice architectures written in Jolie

Author

Emil Ovcina
emove18@student.sdu.dk
emilovcina@gmail.com

Supervisor

Marco Peressotti
peressotti@sdu.dk

4th April 2023



Abstract

Nostrud non elit velit dolor pariatur velit nostrud sint ad exercitation amet amet proident proident. Sit ipsum laboris elit excepteur culpa elit sit cupidatat minim amet cupidatat. Officia nostrud cillum adipisicing amet pariatur do tempor in laboris eu. Occaecat excepteur ut aliquip incididunt officia enim sit tempor.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Scope & Aim	1
2	Preliminaries	2
2.1	Microservice Architecture	2
2.1.1	Some of the Problems with Monoliths	2
2.1.2	Utilizing the Microservice Architecture	3
2.2	Microservice API Patterns	3
2.2.1	The Embedded Entity Pattern	4
2.2.2	The API Gateway Pattern	4
2.3	Jolie	4
2.3.1	Basic Building Blocks of a Jolie Program	4
2.3.2	Embedding Services in Jolie	5
2.3.3	Programming the Communication Topology	6
2.4	Docker & Docker Compose	9
2.4.1	Jolie in Docker	9
2.5	Current Tools	9
2.5.1	Visualization Tools	10

Chapter 1

Introduction

In this introductory chapter, I will go into the motivation behind the thesis.

1.1 Motivation

1.2 Scope & Aim

Chapter 2

Preliminaries

This chapter will provide a context for the thesis. The reader will be able to gain the necessary background knowledge in order to understand the purpose of the thesis and why it can be useful for developing Jolie applications. This essentially builds on the motivation described in the previous chapter.

This chapter will highlight some of the relevant definitions of the microservice architecture paradigm, some of the benefits and consequences, and alternatives, as well as provide the reader with a quick overview of the Jolie programming language, what Docker Compose is, and what tools exist both for Jolie and other programming languages which can be used in some way to visualize.

2.1 Microservice Architecture

Building software today requires a lot of careful considerations when it comes to choosing a software architecture. Many developers will choose a more monolithic architecture where all functionality of the application is in one codebase. This is the simpler approach because everything is deployed as one solution, however, there are many drawbacks with this approach when the software starts getting bigger, and a larger number of users starts interacting with the platform.

In this section and the following subsections, I will adopt the definitions from "Microservices patterns" by Chris Richardson. [Ric19]

2.1.1 Some of the Problems with Monoliths

Drawbacks exist in all parts of using monolith software architecture, everything from development to deploying and maintaining the production application. From a development standpoint, it can be slow to introduce new features into a monolithic application's codebase. As the project grows, so will the complexity, and trying to somehow weave in a new feature in a large, cluttered project can seem almost impossible.

After developing a new feature, or fixing a bug, the developer would ideally like to see their change in production as fast as possible. This can, however, be quite a long and tedious process

when developing in one large codebase. First, all tests must run, which can take a long time. The codebase is complex so the likelihood of a test failing is big, meaning that the tests must be run multiple times.

When the project runs in production, a whole new set of issues can quickly arise. It can be difficult to scale an application when the whole application is one big instance. The only thing to do is give the machine running the application more processing power and memory storage capacity, in other words, vertically scale the application. Another significant problem with having the application be one instance of everything, is that a single point of failure exists. If one functionality of the program is faulty, it can affect other parts of the application even if the other parts seemingly have nothing to do with the faulty code.

2.1.2 Utilizing the Microservice Architecture

To avoid all the problems with the monolith architecture, developers can try to go for a more distributed approach, where the microservice architecture is one of those approaches.

The microservice architecture tries to make the application modular. This means that all business logic is broken up into different services where each service serves only one cohesive set of purposes, and the services can be replicated. The services will have some kind of API which other services can use for communication. This provides some kind of forced boundary because a service can never access the internal classes and code of another service unless the API allows it. This helps in preserving modularity and keeps services decoupled. The definition of a microservice can be a bit indistinct, so it is often up to the individual development team what a microservice entails.

The microservice architecture addresses the more *non-functional* aspects of the application. This includes maintainability, extensibility and testability, as well as the important aspect of *horizontal scaling* where multiple instances of the same business logic can be deployed giving faster response times and eliminating the single point of failure mentioned before.

There can be some difficulties in working with microservices, so it is not a one-solution-fits-all. The communication between microservices can be a whole new dimension of complexity. The developer has several ways of implementing communication and all have their benefits and consequences. The microservices can communicate through event channels, they can expose a REST API, and one could set up a service mesh to handle inter-service communication. Another set of problems is data consistency. Different patterns can be used to ensure that data between services stay consistent. This includes *sagas*, *event sourcing* and many more.

2.2 Microservice API Patterns

In this section, I will go through some microservice API patterns which will become relevant in the context of Jolie programs.

2.2.1 The Embedded Entity Pattern

2.2.2 The API Gateway Pattern

2.3 Jolie

Jolie is a service-oriented programming language, which aims to abstract away the communication between services to a certain degree. The programmer will only need to design the API of the service in order to have other services communicate with it. A Jolie program can be seen as a composition of services. Jolie code is always contained in services, where some building blocks are used to design properties of the specific service.

2.3.1 Basic Building Blocks of a Jolie Program

This subsection will quickly describe some of the relevant building blocks which are needed for a Jolie service.

Service: the service block is sort of the key element of Jolie programs. Everything inside this block is what that specific service will handle. It is also in this block where a developer will create business logic and provide information about the API. Services usually consist of some main business logic block and any number of ports.

Ports: ports are the means of communication between services. Not just between Jolie services but also external communication. Jolie-services differentiate between in-going and out-going communication. So building blocks for both exists, namely `inputPort` and `outputPort`. Ports have their properties which a developer needs to specify. The three main properties of any port are: *location*, *protocol*, and *interfaces*. Where location specifies where the port will be listening for messages and where the port will send messages. Protocol specifies how the port sends and receives messages. Lastly, for input ports, interfaces specify which operations the service is capable of performing, and for output ports, the interfaces specify which operations the service is capable of invoking internally. Listing 2.1 shows a basic input port in Jolie, where all the key properties are set.

Listing 2.1: Simple input port in Jolie

```
1 inputPort IP {  
2   Location: "socket://localhost:9999"  
3   Protocol: sodep  
4   Interfaces: SomeInterface  
5 }
```

Jolie

The location of the port can utilize different media for communication, but the two most common are TCP/IP sockets and Jolie in-memory communication. Specifying the location of ports with `socket://` followed by an IP address means that the ports are communicating via TCP/IP. Specifying the location with `local` means that the ports will use an in-memory channel to communicate, this also means that services on different networks cannot communicate. A port using the local communication medium will be referred to as a "local port".

Interfaces: interfaces define a list of operations, which a service needs to implement in order to use. These operations can be of type `RequestResponse`, where the operation takes in some request type and return some response type, and `OneWay` where the operation takes in a request type but returns nothing.

Type: interfaces use types to define what is expected to be given to an operation, and in some cases what the operation will return. In Jolie, there are basic data types which include: `void`, `int`, `string`, `bool`, etc. However, it is also to define custom types which can contain subtypes. This could be used when an operation needs more data types. Listing 2.2 is an example of a custom type used in an interface in Jolie:

Listing 2.2: Custom type used in an interface in Jolie

```
1  type SomeType: void {
2      content: string
3      id: int
4  }
5
6  interface SomeInterface {
7      RequestResponse:
8          someOperation(SomeType)(int)
9  }
```

Jolie

Besides the four things mentioned, there are some other things which are needed for a Jolie program, however, for this thesis, there is no reason to dive further into the specifics. If one wishes to get more familiar with the language, the programming language does have a documentation page which goes more into the specifics of each component.

2.3.2 Embedding Services in Jolie

Jolie facilitates the possibility of programming the execution contexts. This means that, in code, the developer can handle the execution of other services. This is called *Embedding*.

A service can embed another service, which means that the embedder can launch the embedded service. This is useful when a service depends on another service to be running. The service can simply embed another service to ensure that when the embedder is running, so is the embedded services. Another benefit of embedding is that the language will take care of the connection, meaning that the developer does not necessarily need to worry about how a service is connected to the embedded services. This also potentially hides the connection from the outside world.

The in-memory communication channel can be utilized here. The developer can specify if the embedder should create a new local port and connect automatically via that. This requires that the embedded service contains an input port which shares protocol and location. Listing 2.3 showcases this functionality.

Listing 2.3: Embedding of a service called "svc" via a local output port "OP"

```
1  embed svc as OP
```

Jolie

It is not required that the embedder specifies some local output port for communication. In some cases, where the two services already have ports which use TCP/IP sockets, it can be the intention to just keep using those ports for communication. If this is the intention, the developer can simply remove the `as OP` part of the embed line, and this will still ensure that the embedded service is running and available when the embedder is executing.

In the quality pattern category of the microservice API patterns mentioned before, embeddings are a way of implementing *Embedded Entity* reference management.

2.3.3 Programming the Communication Topology

Jolie also facilitates the programming of the communication topology. This includes aggregation, redirection, couriers, and collections, which is useful when implementing some of the microservice design patterns mentioned before.

Aggregation is an architectural pattern where a service has an input port which exposes all operations of a specified set of output ports. This essentially works like a reverse proxy, where the aggregated service distributes requests to aggregated services without requiring the aggregated services to be publically exposed or known. Listing 2.4 displays how an input port can expose the API of two other output ports. The API of the output ports being aggregated can be extended by the aggregator service. This is done by using *interface extenders* which does as the name implies and allows any client to access more operations than the output port's interfaces specify. This is a simple architectural pattern but can be used to implement some widely used architectural strategies. This includes load balancing, caching, encryption and cyber attack prevention.

Listing 2.4: Input port which aggregates requests to some output ports, OP1 and OP2.

```
1 inputPort AggregatorPort {
2   Location: "socket://localhost:8888"
3   Protocol: sodep
4   Aggregates: OP1, OP2
5 }
```

Jolie

To Showcase this feature in an example: imagine that an app wants to support different payment methods but doesn't want the client to know the location and protocol of each of them. An aggregator can be set up between the client and the services, and depending on the payment method, the aggregator can send the payment request to the correct service. Figure 2.1 shows a system with a service which can aggregate requests by having an output port to each of the aggregated services and a single input port where the aggregation is specified. The aggregated services can either be embedded or not.

Redirection is a pattern which works similarly to the aggregator, but architecturally is very different. A service with an input port can specify that a resource name gets redirected to a specific service via an output port. Listing 2.5 displays how an input port can specify resources and map them to an output port. This means that a client sending a request to the redirector can specify a resource name in the communication media, and the redirector will forward the

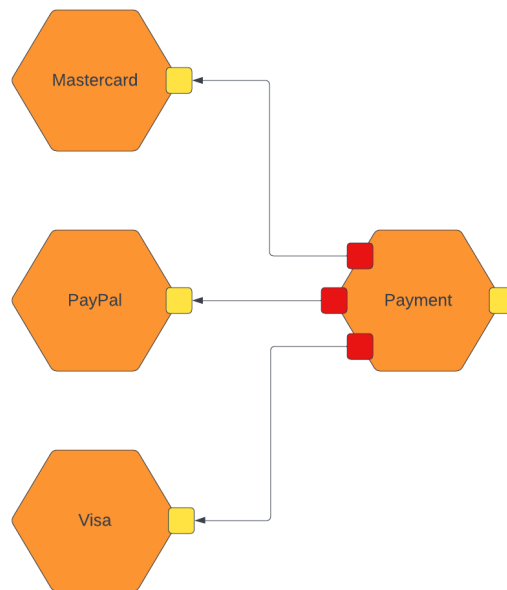


Figure 2.1: A group of microservices handling different payment methods. The payment service acts as an aggregator obscuring the underlying services. The orange hexagons depict services, the yellow boxes depict input ports and the red boxes depict output ports. Sending a request to the payment service will aggregate the message to the correct payment service depending on the user's chosen payment method. This is done without the client needing to know the correct service's location and protocol.

message to the correct service based on that resource name. To specify a resource name the client simply needs to specify it in the URL, e.g `socket://localhost:9000/!/rss` where the `/!/rss` part is what specifies the resource name.

Redirection can be used to implement several different microservice (API) patterns since it essentially is a proxy. Generally, a lot of API structures can be implemented using redirectors, because of how a client can specify a specific resource. API Gateway is one of the API patterns which can be implemented using redirectors, which are used a lot by heavily visited sites like Netflix.

Listing 2.5: Input port which redirects requests using resource names

```

1 inputPort RedirectorPort {
2   Location: "socket://localhost:8888"
3   Protocol: sodep
4   Redirects:
5     rss1 => OP1
6     rss2 => OP2
7 }

```

Jolie

Here is a simple example of what the redirector pattern could be used for: imagine that a client wants to load a webpage displaying all relevant information about a certain product. Instead of

the client needing to call different services with different locations and protocols, a redirector could be used to act as a single point of entry. The client would then specify the resources needed for a product page. similarly, if another service in the system would like to get the current stock of some product it could call the product service with the correct resource name. The example is showcased in figure 2.2. The client makes requests to the redirector specifying the resource name to fetch the relevant data.

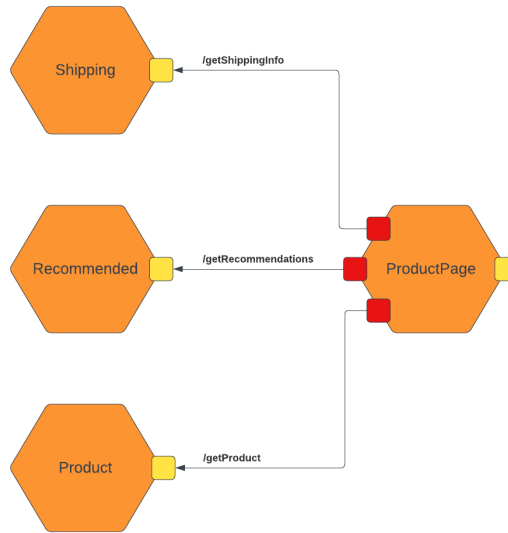


Figure 2.2: A group of microservices handling different elements of product information. The *Product* service is a redirector service which supports three resource names: `getSimilar`, `getRecommendations`, and `getStock`. The client can choose what resource to get for the specific product. The *Similar* service’s job is to know what other products are similar to the searched product. The *Recommended* service finds what products are recommended together with this product and *Warehouse* gets warehouse information about the product.

Couriers allow the developer to append functionality to a set of operations. They work well in extension with other communication topologies like aggregators. The developer defines a courier process by specifying an input port and a set of operations. When that input port receives a request using any of the operations, the courier process executes some code before forwarding the request along to the main operation implementation.

Couriers can be used to implement any type of middleware functionality. From the book by Olaf Zimmermann et al., many of the quality patterns can be implemented using couriers. This includes *conditional requests*, *rate limit*, *pricing plan* etc. Besides the quality microservice API patterns, other microservice patterns can be implemented using couriers, namely, *API key* security and authorization.

Collections is another extension of aggregators. Collections are useful when an aggregator input port aggregates services which share the same interface. They are specified by grouping

output ports when defining aggregates. This together with courier processes can fully, and easily, implement a load balancer for services sharing interfaces because the courier can forward the requests to any of the aggregated services based on some condition.

2.4 Docker & Docker Compose

Docker is a containerization tool used for deploying applications. It builds an *image* which specifies how the container should build and start when it is created. Docker handles a single container, and *Docker Compose* is used to handle multi-container applications. Docker Compose will handle the networking between containers, so it is a great tool for testing and deploying applications using a microservice architecture.

Docker Compose is a container orchestration tool, essentially configuring multiple containers and allowing the developer to ensure that the correct files are mounted, the correct ports are exposed and the containers are bound to their specified networks. It also handles multiple instances/replicas of containers if needed.

2.4.1 Jolie in Docker

To utilize Docker and Docker Compose when developing a microservice architecture in Jolie, creating images can be done using the Jolie base image `jolielang/jolie`. Using this image when making a Dockerfile will set up Jolie when building the image, so only the exposed ports, source files and possible runtime arguments should be handled by the developer.

When running a container, the developer needs to specify what container ports to expose, what parameters should be parsed into the Jolie program, and if it needs to connect to other services the developer needs to first create the network and then assign each container to that network. This is where Docker Compose, or *Kubernetes* which is another container orchestration tool, can become helpful because it will take care of all this if the developer specifies it in the deployment configuration file.

Connecting ports over a Docker network needs some extra work from the developer. Ports which use TCP/IP sockets for communication cannot use "localhost" as seen in the previous examples, they need to use the container name as the host address so Docker can figure out where to send messages inside the network. This can look something like: `socket://auth:9999`, where `auth` is the name of the container.

2.5 Current Tools

Jolie, and other programming languages, do have some tools in order to enhance the developer experience. This section will go through some of the tools which have been developed for Jolie and then look at some of the counterparts in other languages.

Joliedoc generates a documentation page for a single Jolie service. This gives an overview of the API a Jolie service exposes. It shows the input ports and output ports of the service as well

as their location and protocol. This tool is useful when the developer wants a simple and easy-to-follow representation of a single service's API, which include operations, types, port information, and dependencies. For other programming languages, there are tools like *JSDoc* which look at the comments in the code to generate the API documentation. This requires the developer to write more lines for the same result. Tools like *Stoplight* and *Swagger* can do the same for all languages, but this requires the developer to set up a markdown or YAML file and specify the whole API in that. Because of Jolie's way of writing the API as a part of the language, Joliedoc can infer the API from the code without the developer needing to write more lines or comments to achieve this goal. Jolie does have another tool which generates OpenAPI specifications, which Swagger uses, but this is more to be used in conjunction with these other tools, and is not a standalone tool like Joliedoc is.

Slicer

LEMMA2Jolie

2.5.1 Visualization Tools

There are a lot of different visualization tools for software architecture which all fall into some categories and all have different use cases and intentions. The subset of tools which fall under the category of *modelling tools* tries to document a system on different levels of abstraction. This can be on the level of individual components to large-scale businesses with interconnecting components and sectors. Tools like *IcePanel* and *Aplas* allow the developer of a system to use C4 modelling to create a model of their system, on any level of abstraction.

Another subset of tools is the code-based tools which allow the developer to programmatically or textually create models. This is where tools like *mermaid*, *ELK* and *graphviz* belong. These tools allow the developer to write some kind of structured text or code and then they will render the diagram. Developers do not have to use any specific modelling technique, if they can write it in text or code it will be visualized.

The last relevant category of tools is the diagramming tools. Tools like *draw.io* and *lucidchart* allows the user to diagram everything from E/R, UML and FlowCharts to complex systems. This is often done in a drag-and-drop fashion.

One thing all these tools have in common is that they need developers to handle the modelling and diagramming. The developers need to have an overview of the system and then model it using any of these tools.

Bibliography

[Ric19] Chris Richardson. *Microservices Patterns*. Manning, 2019.