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
emovc18@student.sdu.dk
emilovcina@gmail.com

Supervisor

Marco Peressotti
peressotti@sdu.dk

30th March 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

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, as well as provide the reader with a quick overview of the Jolie programming language, what Docker Compose is, and what other similar tools exist both for Jolie and other programming languages.

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 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.

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.

2.3 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

```jolie
inputPort IP {
    Location: "socket://localhost:9999"
    Protocol: sodep
    Interfaces: SomeInterface
}
```

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

```jolie
type SomeType: void {
    content: string
    id: int
}

interface SomeInterface {
    RequestResponse:
        handleSomething(SomeType)(int)
}
```

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"

```jolie
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.

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.

```jolie
inputPort AggregatorPort {
    Location: "socket://localhost:8888"
    Protocol: sodep
    Aggregates: OP1, OP2
}
```

Here is a simple example of how the aggregator pattern works:

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 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.

Listing 2.5: Input port which redirects requests using resource names

```jolie
inputPort RedirectorPort {
    Location: "socket://localhost:8888"
    Protocol: sodep
    Redirects:
        rss1 => OP1
        rss2 => OP2
}
```

Here is a simple example of how the redirector pattern works.

Couriers are can be used in extension with aggregators. Couriers allow the developer to append functionality to a set of operations. 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.

Collections are 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 IP 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

Chapter 3

Chapter

3.1 Section

`service S`

```
1 service {}
```
Jolie

```
1 function f(): void { return 5; }
```
TypeScript

Title goes here!

```
1 service {}
```
Jolie

Listing 3.1: Caption goes here!

```
1 service {}
```
Jolie

Listing 3.1

Bibliography

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