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

29th 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, 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 Architectural Patterns

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.

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
}
```

Jolie

Interfaces: interfaces define a list of operations. 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)
}
```

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 Architectural Programming 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.

2.4 Docker-Compose

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.