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

25th 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

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 aims to 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 and concepts of the microservice architecture paradigm, some of the benefits and consequences, 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 requires a lot of careful considerations when it comes to choosing a software architecture. Many developers will choose a more monolithic architecture where all the functionality of the application is in one codebase. This seems like 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.

For this section and the following subsections, the book *Microservices Patterns* by Chris Richardson [1] will serve as a good foundation. The definitions and concepts described by him will be used throughout this thesis.

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 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 aims 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 API which other services can use for communication. This provides some 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 also 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

This section aims to highlight some microservice API patterns which have been discussed in the book by Olaf Zimmermann et al.

As mentioned before, microservices often communicate with each other through defined APIs and often the client of a microservice application will also communicate using an API. Thinking about and incorporating good design patterns when creating any service-oriented architecture can be a big benefit. The design patterns can be partitioned into five categories of design patterns each trying to solve different problems.

The first category of patterns is the *Foundation Patterns.* These patterns aim to deal with executive decisions of the APIs, including: Where the API should be accessible from, how a client interacts with the system through an API and how is the system landscape with multiple services handled. Some of the patterns in this category include frontend- and backend integration which aims to address the communication between the client and the system and the communication between systems or services, respectively. Another set of patterns in this category is concerning the visibility of an API.

The second category is the *Responsibility Patterns*, which aims to clarify the architectural roles and responsibilities of API endpoints. This can be further partitioned into operational responsibilities, which aim to handle state changes from the API client, and information holder types which aim to handle the exchange of data between APIs and clients.

Diving deeper down the levels of abstraction, the third category, *Structure Patterns*, addresses the structure of messages between APIs. This includes request parameters and responses.

The last category, which will be quickly explained in this part of the section, is the *Evolution Patterns* which will address how the APIs evolve and how the API provider will handle versioning, compatibility and deprecation.

2.2.1 Quality Patterns

This category of patterns is highlighted in a subsection because they are more relevant in the context of the thesis. Quality patterns focus on cost-effectively providing high-quality services. These patterns can be partitioned into *Refecence Management*, *Data Transfer Parsimony*, and *Quality Management and Governance.*

Relevant patterns in this category include the *Conditional Request* pattern which aims to eliminate unnecessary server-side processing, the *Rate Limit* pattern that prevents excessive usage of an endpoint and the *Pricing Plan* pattern that allows the provider to monetize the API,

Many of the patterns can be used in conjunction to enhance the effect of the desired functionality. For example: *Pricing Plan* can use *Rate Limit* to allow the provider to have different levels of pricing plans. To further enhance the functionality of the pricing plan pattern, *API Keys* (A structure API pattern which allows the API provider to assign a unique token/key to each client which can be used for authorization purposes) can be used together to allow the API provider to safely and reliably monetize the API using different billing plans. Adding this to an API Gateway is good to ensure that all clients are monetized accordingly.

Generally, most of the API patterns described in the book are best utilized in conjunction, and when looking at Jolie in a moment, it is crucial to understand that these patterns, both API and non-API patterns, should be seen as building blocks which are used together to implement the desired architecture. It is also worth mentioning that the patterns discussed in this section will not be implemented in this thesis.

2.3 Jolie

Jolie is a service-oriented programming language, where (micro)services are the basic building blocks. Everything in Jolie is contained in services and services communicate with each other through APIs which the programmer needs to design and implement in the code. [2]

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. The 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

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

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. The syntax of a RequestResponse operation is: `opName(RequestType)(ResponseType)`, and for OneWay operations it is: `opName(RequestType)`

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

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

interface SomeInterface {
    RequestResponse:
        someOperation(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 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"

```
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.

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

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 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.

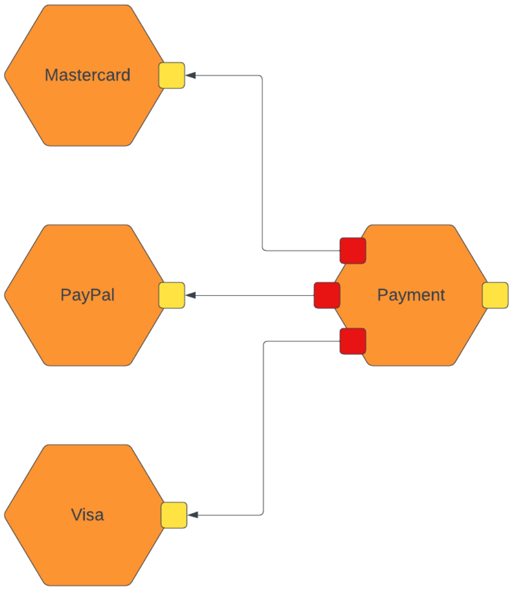

Figure 2.1: A group of microservices handling different payment methods. The payment service acts as an aggregator obscuring the underlying services. The orange hexagons depict services, the yellow boxes depict input ports and the red boxes depict output ports. Sending a request to the payment service 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.

Listing 2.5: Input port which redirects requests using resource names

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

Jolie

A simple example of what the redirector could be used for is: imagine that an e-commerce application wants to have one point-of-entry for the system. The application could set up an *API Gateway* which will act as that point of entry. When a client wants to get information for a product page, it can specify what it needs in the resource names. This also handles protocol transformation so the client does not need to know the internal service's required protocols. The example is showcased in figure 2.2. The client makes requests to the redirector specifying the resource name to fetch the relevant data.

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

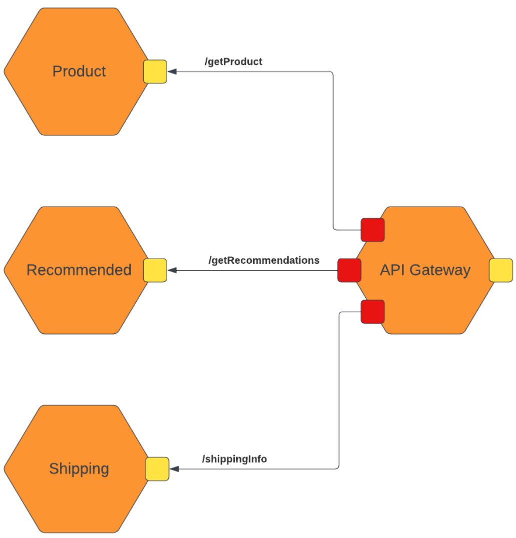

Figure 2.2: A group of microservices which can be accessed through an API gateway. The client can specify a resource name to specify which service to send the request to. The product service fetches information about a product. The recommended service fetches the recommended products based on the user and the shipping service fetches shipping information given the user's location.

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. Since collections are an extension of aggregates, they can have the same use cases, but the collections can group services that share the same interfaces but can have much different underlying business logic.

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.

JPM is a package management tool for Jolie which, similarly to NPM, keeps track of dependencies of a Jolie program.

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* aims 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 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.

Chapter 3

Using the Visualization Tool

With the foundational knowledge and context for the microservice architecture, microservice patterns, the Jolie programming language, and the current tools both for Jolie and for visualization in general in place, it is time to get familiar with the tool developed for this thesis.

This chapter will go into how the tool is used to enhance the development experience by going through a simple example microservice application. Firstly, an application is defined. The application for this example is a simple e-commerce platform consisting of seven microservices:

- **User service** - Handles user authentication. Lets the client create, update and delete their account and also log in to get an authentication token.
- **Product service** - Handles product information. This service exposes a basic CRUD (Create Read Update Delete) API for products sold on the platform.
- **Recommended service** - Handles fetching recommended products. A client can query this service which will, based on the user's id, return a list of products which the user might also like.
- **Order service** - Handles grouping of selected products and all the information required for the user to place an order. This includes shipping and tax fees.
- **Payment service** - Handles transactions of orders. When the user places an order they must provide some kind of payment method as well as the necessary information to complete the transaction.
- **Analytics service** - A monitoring service. This service can keep track of popular products, frequent shoppers, what products get bought together, and much more.
- **Notification service** - Handles sending e-mails to users when the transaction is approved. Can also send promo codes, offers, and discounts to users.

This example is very simplified, and only the architectural aspect of the microservices will be implemented, meaning the connections between services and the APIs. This means that no business logic will be implemented, and the types of interface operations will be simplified since the functionality of the application is not in focus in this chapter. Other details such as deployment, data persistence, and security will also be omitted from this example. A simple diagram can be seen in figure 3.1 which displays the desired architecture of the application.

To showcase the visualization capabilities of the tool the application will not be developed using any architectural programming features from Jolie such as embeddings, aggregation, redirection, etc. These will, however, be introduced later.

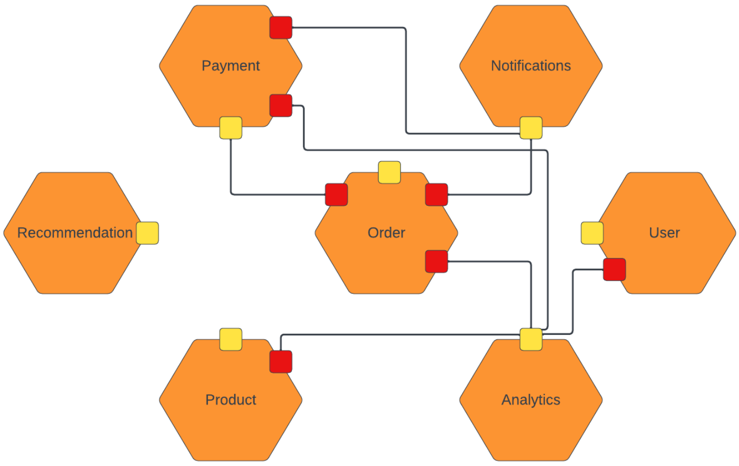

Figure 3.1: A diagram of the example architecture of the e-commerce application. The output ports (red) represent internal communication between the services via their input ports (yellow).

3.1 Setup and Requirements

In order to use the tool, Jolie must be set up correctly with version `1.11.0-git`. This section will not specify how this is done, but the Jolie website goes more in detail on how the correct git version is installed [1]. From this point onwards it is assumed that the environment variables are set correctly to point to the git version of Jolie.

The tool is developed to be used with Visual Studio Code[2], aka VS Code, which is a code editor developed by Microsoft. VS Code has a large ecosystem of plugins which are used to enhance the developing experience. To start using the tool, it must be downloaded and installed either from the VS Code marketplace under the name: `JolieVisualize`, or compile the source code and extract a `.VSIX` file to install manually.

The development starts with creating a `.JSON` file which JolieVisualize uses to know which services are at the top level of the application. The plugin has a default file name it will look for, namely, `architecture.jolie.json` located in the root folder, which is where the developer will define all top-level services, networks, and properties of the services. The plugin comes with

[1]Downloading the newest version of Jolie - `https://www.jolie-lang.org/downloads.html`
[2]Visual Studio Code - Code Editing. Redefined. - `https://code.visualstudio.com/`

a command to initialize the architecture file, and this creates the file and populates it with a template service contained in a network.

3.1.1 Structure of the Architecture File

The architecture file is a JSON file which consists of an array of arrays of services. The outermost array can semantically be understood as the list of all networks. Listing 3.1 shows this structure where the `{...}` represents the services. Each network can have any number of services, and it is up to the developer to specify what a network represents depending on where the services will deploy.

Listing 3.1: Structure of the architecture JSON file showing two networks.

```json
[
    [
        { "file": "svc1.ol" }
    ],
    [
        { "file": "svc2.ol" }
    ]
]
```

JSON

The services have different properties which the user can specify. All properties are showcased in appendix A. For the example in this chapter, only the file needs to be specified for each service, for now.

3.2 Developing the Services

All seven services in this example will have a unique folder each in the root directory of the project. This is done to separate which files are belongs the which service. In each folder, JPM can be initialized if the developer needs it, and the interfaces, types, and services will all be separated in files as well.

The tool will be utilized as much as possible to create and refactor code, all while every service being developed will be visualized next to the source code in VS Code. Every time a service has been declared it is added to the architecture file. This can be done quickly by the developer using code snippets added by the tool. All architecture JSON file snippets can be seen in appendix B.

3.2.1 The User Service

The interface which specifies what operations the user service implements is called `UserIFace` in this example. Listing 3.2 shows how the interface is set up. The ErrorResponse type is a more universal response type which gives a message and status code, which can be used in case of an error, but with no error, it will simply set the error flag to false. Each of the request types represents what data the operation needs, in order to be invoked. For registering users and logging in an email and password are needed. The update request type represents the user to be updated and what is going to be updated for the user. The delete request simply needs the user ID of the user.

Listing 3.2: The interface for the user service

```
interface UserIFace {
    RequestResponse:
        register(RegisterRequest)(ErrorResponse),
        login(LoginRequest)(LoginResponse),
        updateUser(UpdateRequest)(ErrorResponse),
        deleteUser(DeleteRequest)(ErrorResponse),
}
```

Jolie

The service is created in the main file and the execution is set to *concurrent*. The service can now be added to the architecture file. Since only one service is present in the main OL file, only the file name needs to be specified in the service JSON. Now the visualization tool can be run using the command *Jolie: Visualize* in VS Code, and a single service called *User* can be seen.

To use the tool to add the ports, double-click on the service to open the service in the sidebar. Next to the empty lists of ports, a `"+"` button can be clicked to add either an input port or an output port. This will bring up a pop-up window, shown in figure 3.2, where the user can specify the details of the port such as name, location, protocol and a list of interfaces.

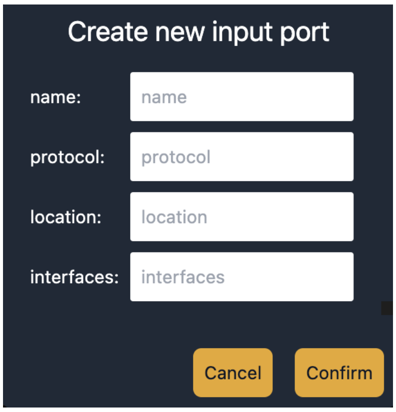

Figure 3.2: Pop up in the visualization tool where a user can specify details about a port, and by clicking 'confirm' will add that port in code.

Using this feature. The service needs to have an input port which uses the interface specified before, `UserIFace`, and before creating the output port for the analytics service, the interface for the analytics service should be defined to prevent a parsing error from the Jolie parser. The analytics interface is called `AnalyticsIFace`, and the operations will be discussed in another section. For now, it is just an empty interface.

After clicking 'confirm' on the popups, the interfaces are correctly imported if they exist. The code for the user service is displayed in listing 3.3, where some of the implementation details are omitted, but the general structure of the service can be seen.

Listing 3.3: The user service after the ports have been created with omitted implementation details.

```
from .userInterface import UserIFace
from ..analytics.analyticsInterface import AnalyticsIFace
service User {

    execution{ concurrent }

    inputPort IP {
        ...
        Interfaces: UserIFace
    }

    outputPort Analytics {
        ...
        Interfaces: AnalyticsIFace
    }

    main {
        ...
    }
}
```

Jolie

The implementation details of the types are omitted for all services, but the source code for the application example can be found in the GitHub repository[3].

3.2.2 The Product Service

The product service handles all operations of the products. This is a basic CRUD service, which also will send analytics to the analytics service, which can be used to monitor popular products etc. The create and update operation's request types require a product, which is a type containing information about the product, and a user ID because only users who own the product should be able to alter it. The read operation request type is only an integer corresponding to the product ID. The delete operation request type is a type containing the user ID and product ID to, once again, check if the user is allowed to delete the product and the ID of the product.

When the types and interface (`ProductIFace`) have been created, a service named *Product* is created in the main `.ol` file and once again the execution context is set to *concurrent*. The service is added to the architecture file in the same network as the user service, and can now be seen in the visualization user interface.

By opening the product service in the sidebar the input and output port can be created similarly to the user service previously. The input port will use the `ProductIFace` interface and the output port will use the `AnalyticsIFace` interface. The two services displayed in the visualization UI can be seen in figure 3.3 with the two ports created for each service.

[3]Source code: `https://github.com/EmilOvcina/jolievisualize/tree/main/microservice_example`

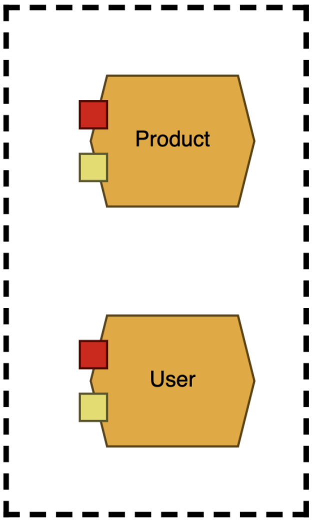

Figure 3.3: The product and user services displayed in the visualization tool's UI. They have an input port (yellow) and an output port (red) each and they both are considered to be in the same network.

3.2.3 The Analytics Service

This service supports one operation for now. This is `addToLogs(AnalyticsRequest)`, which is a one-way operation. The interface for the service is already defined, so the operation is added there with the request type being `AnalyticsRequest`. This type holds information about what content will be logged and from which service.

This service is created in the main file, and the execution is also *concurrent*. The analytics service is added to the architecture file and is displayed in the visualization UI. The input port can be created using the tool just like the last two services, and the interface for the port is `AnalyticsIFace`.

The input port is using the TCP/IP communication medium, and the output ports in the product and user services must be connected to the input port of the analytics service. If the ports are not connected in the visualization UI, it is possible to change the location of the ports directly in the UI. By clicking on the ports, the sidebar will open displaying information about the ports. Double-clicking on the location field allows the user to change it. Doing this for both output ports to make sure they have the same value in the location field will connect the output ports to the input port in the UI, as well as, change the code to match the changes made in the code.

After connecting the ports, the connections is be represented in the visualization UI. This can be seen in figure 3.4 where the user and product services are connected to the newly created analytics service.

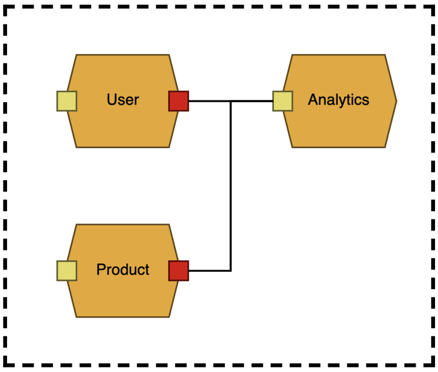

Figure 3.4: The product, user, and analytics services displayed in the visualization tool's UI. The user and product services have an output port (red) each of which is connected to the input port (yellow) of the analytics service.

3.2.4 The Notification Service

The notification service will get requests from other services and send an email to the specified user. This can be when a transaction is completed, a product is back in stock etc. This service supports one one-way operation, `sendNotification`. The request type contains the user ID to send the notification to, and the content of the notification. Adding the service to the architecture file will render it in the UI.

Using the UI, once again, to add an input port with the interface called `NotificationIFace`. This is the only port needed for this service.

3.2.5 The Payment Service

The payment service's only job is to process payments from orders. The interface for this service is a single request-response operation, `processPayment`. The request type for this operation needs information about the payment method, amount to pay, user ID and credit card details. The response type has an error flag, date of purchase and optionally a receipt if the transaction was successful.

Same with all other services, this is added to the architecture file to render it in the UI. The payment service needs one input port for the interface `PaymentIFace`, an output port connected to the analytics service, and another output port connected to the notification service. Once again, by opening the ports in the sidebar, the user can change the location of the ports to match so the correct ports are connected.

3.2.6 The Order Service

This service handles the user's orders. A user can place an order, see that order and cancel it. The interface, `OrderIFace`, has these three request-response operations: `placeOrder`, `getOrder`, and `cancelOrder`. The request type of placing an order requires a list of products and a payment request, which could be populated by the shopping basket in the application UI. The response type simply lets the user know if the order was placed correctly with an error flag and a status code. The "get order" operation requires an order ID of the order to return and will simply respond with an order type containing information about the order. Cancelling an order also requires the order ID and will respond with a type with an error flag and status code.

When the order service is defined in the main file, and the service is added to the architecture file, the input port is created using the `OrderIFace` interface. The service needs three output ports. One for the analytics service, one for the notification service and one for the payment service. The analytics that the service can log is what products get bought together and how many orders a user cancels etc. The notification service needs to be invoked to send confirmation about cancelled orders, or if the status of the order changes. The payment service will be invoked through the order service, so when the user places an order the payment service will get the request and send the response back to the order service which creates a response for the client.

The tool can either help in connecting the correct ports as done with the other services, but the developer can do that in the code as well, and when the developer does a change in the code manually and saves the document, the visualization UI is automatically updated to reflect the changes made in the code.

3.2.7 The Recommended Service

The last service is a standalone service, which only the client needs to invoke. The recommended service interface consists of one request-response, `getRecommendations`, which based on the user ID given finds the most recommended products as a list of products.

A possible extension to the service can involve fetching analytics data from the analytics service and doing calculations with that information to avoid sharing databases between services. This will require the analytics service to expose an operation allowing the internal services to get the data, and an output port in the recommended service to invoke that operation. This extension will be omitted in this example.

3.2.8 The Completed Application

With all services and ports in place, the application can be seen as one microservice architecture in the visualization UI, as seen in figure 3.1.

Having the whole application in the visualization tool allows the developer to inspect the application in detail to get an overview of what is happening between services and what APIs the different services expose. Clicking on ports opens the sidebar and displays information about the ports including the interfaces. All the interfaces can be clicked on to further inspect the operations in the interface. All types used for the operations are also clickable and will display information about the type, including the subtypes, cardinality and root type.

From here the developer can start using architectural patterns in Jolie as discussed in the previous chapter. Some refinements of the current services can also be made, for example, the payment service should probably have underlying services to take care of different payment methods.

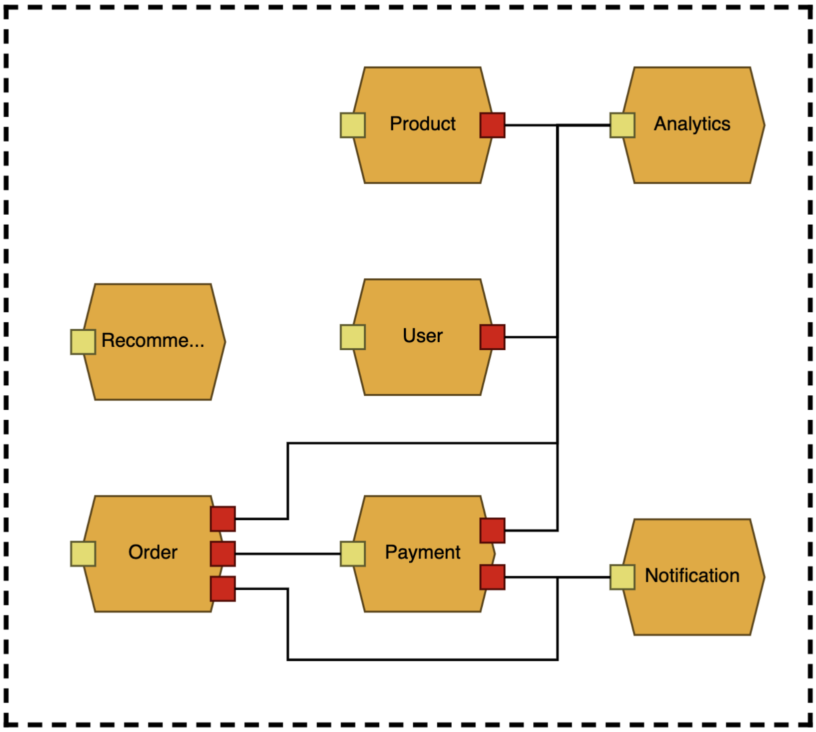

Figure 3.5: The entire e-commerce application is displayed in the visualization tool's UI. All services exist in the same network and all are top-level.

3.3 Improvements

Now that the application has all seven services implemented, the developer can apply some of the architectural features allowed in Jolie to improve the application and implement some of the microservices (API) patterns discussed in the last chapter.

3.3.1 Embedding

It is not necessarily the best idea to have all seven services as top-level services. The services not accessed by the client directly should be embedded in the services which need them. The analytics service is used by four services: Order, Payment, User, and Product. To embed the analytics service in these four services the input port's location can be changed to `local`. This is just one way of embedding using local in-memory communication and is not the only way of embedding services. Another method of embedding will be used for another service later. Changing the port location to `local` can be done in the sidebar. The output ports in the mentioned services using the analytics interface must also be deleted which, at the moment only can be done in the code, but instead of looking for the main files which contain the ports, opening the port in the sidebar and clicking `{}` at the top of the sidebar opens the correct file in the editor.

Now in the architecture file, for the analytics service, the field `instances:4` can be added to the JSON object representing the analytics service. This will add 4 instances of the analytics service, and by using the mouse and dragging the services, the developer can drag the services onto the four services and embed them one after one. This will add the code in the correct files to specify that the analytics service instances are embedded.

The notification service is also used by other services, and should not be accessed by a client directly. For this service, the ports will not use local in-memory communication as the analytics service did. In the architecture file the notification service will need to have two instances, so the field for instances is added to the service JSON. Now two instances of the notification service exist as top-level services. Each instance can be dragged into its respective parent service, namely, the order and payment services and the embedding code will be added to both services. The embedding will use the already existing ports to connect so no local ports will be created.

The last service to embed is the payment service in the order service. This can be done by either changing the existing port to use local communication or just dragging the payment service as it is now into the order service to use the existing ports as communication and the code will again be changed to reflect the embedding.

The new architecture can be seen in figure 3.6 and it shows only the services which are accessed by a client. The services which now have embeddings can be seen with a little "plus" sign, which can be clicked on to expand the visualization of the service revealing local ports and embedded services. The embedded services can be removed by dragging them out of their embedder. Doing this with local communication will cause the local ports to be removed from the code.

3.3.2 Aggregation

Now all the top-level services are services which a client is directly requesting. It is now possible to add some of the architectural patterns to the system in order to implement some of the ideas mentioned in the previous chapter. The tool facilitates the possibility to select services by holding the shift key and clicking the service shapes. The sidebar will show which services are selected and a list of patterns which can be applied based on some criteria. Selecting the four top-level services of the example application shows that the aggregator pattern is applicable.

Clicking on the "aggregator" button will open a pop-up allowing the developer to enter all necessary information for creating an aggregator service, and choosing if the aggregator should connect to existing ports or create new input ports for the aggregated services, or if the aggregated services should be embedded in the aggregator service. For this example, the aggregator will just connect using the existing ports requiring that the developer types in the locations of the input port for each service correctly. After clicking "confirm", the aggregator service will be created in the file of the first service selected. The new architecture can be seen in figure 3.1 and shows

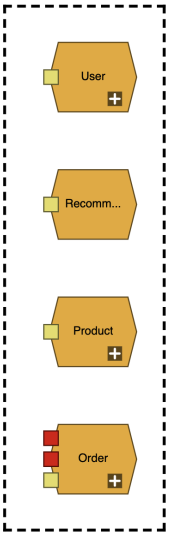

Figure 3.6: The e-commerce application after the payment, notification, and analytics services have been embedded.

that the aggregator service called `Aggregator` is connected to the services and is displayed in a slightly more orange colour to illustrate that the aggregator is a generated service and is only a scaffolding service which the developer needs to implement the functionality for.

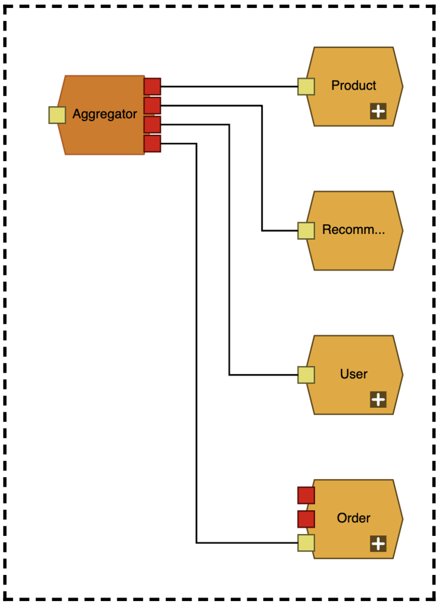

Figure 3.7: The e-commerce application with an aggregator service that serves as a reverse proxy for the services that are invoked by clients.

3.3.3 Networks

The final part of the application is determining which networks each top-level service belongs to. This is relevant when the application is deployed. The services can be dragged out of their dashed box to create a new network and place the service in the network. Other services can then be dragged into the new network to add the service to that network, or services can be dragged out of their network to create more networks.

Creating new networks and moving services around to different networks does nothing in the Jolie code. It changes the structure of the architecture file, but that is only for visualization purposes. For this example, the networks can be arbitrary so the services will all reside in the same network for now.

3.4 Prototyping

The tool facilitates testing in a local deployment. Using the command *Jolie: Docker-Compose* will generate a build folder where all top-level services reside in their respective subfolder containing all dependencies, a package.json file if the project is using JPM, and a Dockerfile to create an image when running the deployment. A `docker-compose.yml` file will also be generated specifying each service and the deployment configurations. In the generated deployment file the networks are also specified and added to each service depending on what network the developer added the service to in the architecture file and what networks the service is connected to via ports. This is just a generic deployment file and should not be used for anything other than testing the application locally. The tool only supports Docker-Compose at the moment, so Docker is required for this feature to be effective.

To make the Jolie service accessible from outside clients it is required that the services follow the requirements for running Jolie services in Docker mentioned in the preliminaries section about Jolie in Docker 2.4.1. This includes changing the location to use the container name as the hostname if TCP/IP communication is used between services and adding an extra field in the architecture file. The field is a `ports` field which is a list of Docker ports the container will expose using the syntax of Docker ports, e.g. `"3000:3000"` meaning that port 3000 is exposed externally and mapped internally to port 3000. Other fields can be added to customize the deployment. The container name can be changed using the field `"container"` giving more freedom in terms of what the services are called in the code because the default container name is the service name. If a service in the deployment needs any dependencies outside of Jolie, e.g. configuration files, a list of files can be specified in a `"volumes"` field. This will add the files to a resource subfolder in the build folder and will be bind mounted when the container is running.

In the architecture file, external dependencies can also be added to the architecture which allows other services to be displayed alongside the Jolie services. The developer can specify a service in a network and give an `image` field which links to an image on Dockerhub or other image repositories. In the visualization UI, this will be shown with a blue shape which indicates that it is an external service, and the ports also need to be specified in the architecture file. Building the Docker-Compose file, external services are also added.

When everything is set up and docker is running on the machine, the developer can start the local deployment with Docker-Compose by running `docker-compose up` in a command line tool or terminal.

Chapter 4

Implementation

The tool has now been used in a practical example to give an idea of how it is used to enhance the development experience. This chapter will go into the design decisions and implementation details to make each part and feature of the tool function, as well as, explain what the tool consists of and how the components work together to create the finished product.

The specifics of the code will be omitted, but the general control flow of each part of the tool will be explained.

4.1 Overview of the Components

The tool essentially consists of four components: The VS Code extension, the Svelte user interface, the Java program, and the NodeJS utility program. The Java program is used to run the Jolie parser, which is also written in Java, and that will gather all information from the Jolie code needed by the visualization tool.

The Svelte user interface is the component which takes in the user input and renders the services, ports and connections. In order for the UI to get the relevant data from the Jolie code, the VS Code extension invokes the NodeJS utility program, with the correct parameters, which invokes the Java program. The NodeJS program listens for the output of the Java program, meaning that the NodeJS program functions as a wrapper for the Java tool so the VS Code extension does not have to invoke the Java tool directly, and sends the data to the VS Code extension, which sends the data to the Svelte UI.

Figure 4.1 shows the process of initializing the tool in VS Code, which is done using the *visualize* command in VS Code.

4.2 Data Representation

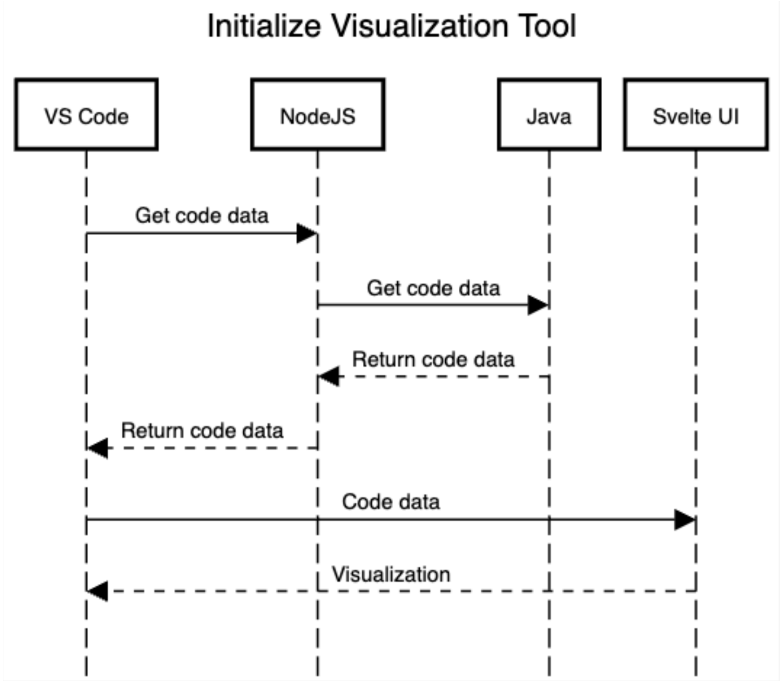

Figure 4.1: A sequence diagram showing the sequence of requests and invocations between the four components that the tool consists of. The VS Code extension invokes the utility program using NodeJS which will handle running the Java program with the correct parameters, and afterwards, the VS Code extension sends that data to the Svelte UI which will visualize it in VS Code.

The data which the Java program will output is consistent through all four components of the tool. The chosen data format is JSON data because it is easy to parse in TypeScript, which all components consist of except for the Java program. The data needed for the tool is a list of interfaces, a list of types and the list of services.

4.2.1 Interfaces & Types

Each interface in the list of interfaces has a name, identifier, file path, and two lists of operations. One of the lists contains all one-way operations, and the other contains all request-response operations. The request-response operations consist of the operation name, the request type name-file path pair and the response type name-file path pair, and for one-way operations it is similar but with the response type omitted. Having the type name and file path pair and a separate type object list instead of the type object directly in the specific interface object is a design decision made at the very start of the development process. Each interface could have a list of types which will save computation when the user clicks on the type in the sidebar and the type is displayed because only the interface's list of types needs to be searched for the correct type. This, however, has the drawback of a lot of duplication, only if all interfaces in the system use a unique set of types can this issue be avoided. This is why a global list of types is used. The interfaces then reference what types are used for operation and the global list of types can be used when using the sidebar to view interfaces and types, and by using the Jolie language constraint of no duplicate type names in the same OL file, the file path and type name are enough to find the correct type when searching for it.

The list of types, as mentioned, contains all types used by all interfaces in the architecture. Each type object in the list represents the information about a type in the code, this includes the name of the type, the file in which the type is defined, the root type, subtypes which is a list of types as well, and cardinality of the type in case of this being defined. If the type is a *choice type*, the two choices are also represented in the type object as strings which reference other types in the global list. [1] Types from *interface extenders* are also added to this list.

4.2.2 Services & Networks

The last list in the global JSON is the list of services. The list of services is also used to infer how the networks are set up and which services reside in each network. This approach draws inspiration from how bigraphs are used to model software architecture. Bigraphs are a type of graph consisting of two sub-graph components. The link graph and the place graph. The link graph models the connection between nodes in the system while the place graph models the topology. The nodes in the system of the bigraph can be directly mapped to Jolie services. Bigraphs also model ports which are a component of Jolie services as well. The data representation used for the tool draws the most inspiration from the place graph since the link graph is inferred by the locations and protocols of the ports in Jolie. A place graph models the topology and the networks, which are called *roots* in bigraph terminology, and also embeddings, which are called *nestings*. The inspiration with bigraphs is taken from Context-Aware Systems (Cherfia, Belala & Barkaoui, 2014)[2], where bigraphs are formally introduced and explained but are not necessary for this use-case.

The place graph component can be seen as a list of trees where the root is the network and contains all top-level services as immediate children and embeddings of the top-level services are the children of those services. This fits well with how a Jolie system can be modelled, so the list of services in the global JSON is essentially a place graph or a list of trees where each node is a service in the system. Each root in the list of trees is a network. In the sense of JSON, the network is a list of services since it can have any number of children, and the *embeddings* list of each service is also a list of services.

The services in the JSON also need to contain all other information which is used by the tool. This includes the input and output ports, execution target, name, identifier, and file path but also the information specified in the architecture file, which is container name, volumes, parameters etc.

[1] Types in Jolie - `https://docs.jolie-lang.org/v1.10.x/language-tools-and-standard-library/basics/data-types/index.html`

[2] Bigraphs paper Context-Aware Systems - `https://www.researchgate.net/publication/269703150_Towards_Formal_Modeling_and_Verification_of_Context-Aware_Systems`

4.2.3 Architectural Programming

The architectural programming features in Jolie also need to be represented in the data to visualize it in the tool's UI. This includes couriers, collections, aggregations, and redirections. All of these aspects are created at the Jolie port level in the code, so the information is also represented in the ports of the JSON.

Aggregations in the JSON are represented by the name of the output port to aggregate to, optionally the aggregate can have an interface extender and which be defined here. Collections are essentially a list of aggregations, so collections are also represented in the aggregates as a list of names of the output ports, and because multiple output ports of one service cannot have identical names, this will not create ambiguity.

Couriers are defined in Jolie based on an input port with aggregation, so the port also represents the couriers of that port. The couriers in the JSON have lists of operation names and interface names which the courier will act upon. This is both for request-response and one-way operations. The names refer to the operations and interfaces in the system.

Redirections for an input port are a list of pairs of resource and output port names. This represents how a resource is redirected to an output port. However, since the resource of the redirect is defined using the location of the output port, the output port will not be connected correctly unless the resource part of the string is removed, so the resource part of the location, namely, the part after `/!/` is removed and placed in a separate field in the JSON for the output ports.

4.2.4 Prototype Data

The data used to create the prototype is also represented in JSON and is split into two parts, the Docker-Compose YAML file content and the service folders which contains the Jolie files, dependencies, and Dockerfile. The Docker-Compose file content is simply a string, which will be written to the file. The build folders are used to generate this string, but this will be explained more in detail in a later section. The services folders, which are called build folders internally, are an array of objects that each represent a folder to be created when generating the prototype. The build folders have to represent what a Docker image will contain, so each folder can be seen as a Docker image. This has to be independent of the Jolie code since two identical services can have different parameters or runtime arguments.

Each build folder contains a list of files which will be added to the folder itself, but also a list of files which will be added to the shared resource folder. Information such as runtime arguments, exposed Docker ports, and the service name is also added to the build folder JSON because the Dockerfile needs this information when it is being generated.

4.3 NodeJS Utility Program

The NodeJS utility program functions as a wrapper for the Java program. It is created so the VS Code application does not have to execute the Java program directly and also if the tool should be extended to other code editors or run in the browser, this functionality should not be implemented in the VS Code extension. The NodeJS program also makes sure to return errors if a parsing error occurs or if Jolie is not installed correctly.

This part of the tool consists of three parts, the data fetching part which executes the Java program and checks for errors, the server part which allows the developer to visualize their program's architecture directly in the browser but without the code refactoring functionality, and prototyping which is identical to the functionality in the VS Code extension, but this can be used if the user of the tool installs only this part of the tool without the VS Code extension. The server uses the express framework[3] to open an endpoint to get the JSON data and also serve the webpage as static assets on localhost.

4.4 Java Program

The Java program is used to gather the required information from the Jolie code. The Jolie parser is written in Java and therefore the Java program here uses the parser directly to get the abstract syntax tree nodes and create the JSON objects.

The Java program has two types of outputs which is used by the tool. The JSON representation of the entire system and the information used by the prototyping functionality. Both functionalities of the Java program require the whole system to be parsed using the Jolie parser. Firstly, the tool creates the networks by parsing the architecture JSON file and making an internal representation of what it contains. The networks contain a map of top-level services mapped to a `ServiceNode` which is the abstract syntax tree (AST) node of a service from the Jolie parser. The top-level service is a representation of a Jolie service in the architecture file, as seen in appendix A. The Java program uses a lightweight JSON library, *json-simple*, to parse files and create JSON objects [4]. To get the ServiceNode AST node from the parser, the Jolie parser is invoked using its module parsing method. This returns the whole AST of a file, called a *program*. All ServiceNodes are then captured and added to the network's map object.

When the networks have been created, depending on if the program should send visualization data or prototyping data, all the necessary information about the services, ports, etc. is created.

4.4.1 Parsing the Jolie System

A system inspector has been created in order to parse the networks. The system inspector uses the created list of networks containing the top-level services and ServiceNodes. The inspector goes through each network's map object and creates a service object which is an internal representation of the services used by the tool. It creates the services by looking at all the child nodes of the ServiceNode AST node in a loop and using type comparison to check what child node is being looked at and how to parse the information correctly. Relevant child nodes for a service are: *ExecutionInfo*, *CourierDefinitionNodes*, *EmbedServiceNodes*, *InputPortInfo*, and *OutputPortInfo* nodes.

Service objects can also have a list of services which are considered embeddings. When looping through the AST nodes of a service, embedded service nodes will be used to create services, which

[3]Express Framework - `https://expressjs.com/`
[4]JSON-simple - `https://code.google.com/archive/p/json-simple/`

are added to the parent's list of services instead of the global list of services. This will enforce the place graph structure of services mentioned in the data representation section.

The most complex of the child nodes are the two types of port nodes. Input and output ports also have an internal representation in the Java program which keeps track of all the necessary information about the ports used by the tool. Ports have to contain information about the location, protocol, name, interfaces and operations. When the ports are being created, the interfaces which are used by the ports will also be created. These interfaces are also represented internally and will store information about the operations. The operations are also parsed using type comparison, since two types of operation declaration AST nodes exist in Jolie, namely, request-response and one-way. The interfaces are added to a global object *JolieSystem* which has information about all parsed services and interfaces.

For input ports specifically, information about aggregates, couriers and redirects is also stored. Redirects can be seen as a mapping where a resource is mapped to a port name. This is how the Jolie parser represents redirects as well, so this is simply copied over to the tool's representation. Aggregates are simply a name of an output port but can contain collections and interface extenders, however, Jolie's internal representation of aggregates contains this information so the interface extender just has to be created to match the tool's representation of interfaces, and collections are a list of output ports which will be copied directly.

When the interfaces are created the types used in the operations are also parsed and created in an internal representation. This representation stores information about the root type, cardinality, subtypes, etc. The types in Jolie can also be three different AST node types, namely definition links, inline definitions and choice definitions. Type comparison is also used here to parse the type correctly depending on the AST node type. Types are, similarly to interfaces, added to the system's global list of types.

Each building block of a Jolie service has an internal representation in the Java program, and all of them have a `toJSON` method which takes all information stored in the objects and creates a JSON object. When the tool invokes the Java program to get the JSON data, the `getJSON` method is invoked on the global JolieSystem object which cascades and invokes the getJSON method for all internal objects which also carcades to its internal objects. This will result in a JSON object which resembles the data representation discussed in the previous section, and this JSON can be used by the other components of the tool.

4.4.2 Generating the Prototype Data

The Java program also creates data for making the prototype. This includes the content of the Docker-Compose YAML file and the information needed for creating the service folders and the resource folder. After the whole system has been parsed and a JolieSystem is created, the system will be given as a parameter to a *build* object which will handle the logic for creating the JSON data. The builder will first create the service folders using the services in the networks from the JolieSystem object. Duplicate build folders will be checked for at this stage, which is done to make sure that duplicate images will not be generated when using Docker-Compose. However, as mentioned in a previous section, two identical services can be two different images if the runtime arguments are different, which is also handled here.

After the build folders have been created as internal objects, they are parsed onto another object which builds the Docker-Compose YAML string. The JolieSystem is also used here to create the networks in the Docker-Compose file. The Docker-Compose builder also has some internal representation of what a service is, since two services in Docker-Compose can use the same image but have different deployment properties, which means that they should be created as separate services in Docker-Compose, but they will not have two different service folders. The Docker-Compose builder uses a string builder to create the YAML file content. It goes through all services and creates an internal *compose service* and like with the service folders, composes duplicates and counts replicas.

The networks are also created in the Docker-Compose YAML, which can have another semantic meaning from the networks defined in the architecture file. In Docker-Compose networks are used to restrict services connecting, which is not a problem with a system created in Jolie, so to have the Docker-Compose networks work similarly to how Jolie, each service in Docker-Compose is set to be a part of its network, as specified in the architecture file, but also all other networks that the service has a port connecting to. [5]

4.5 Visualization User Interface

4.5.1 UI Library

4.5.2 Rendering

4.6 Visual Studio Code Extension

4.7 Refactoring

4.7.1 Code Refactoring

4.7.2 Embedding & Disembedding

4.7.3 The Aggregator Pattern

4.8 Prototyping

4.8.1 Dependencies

[5] Networks in Docker-Compose - `https://docs.docker.com/compose/networking/`

Appendices

Appendix A

Architecture File Structure

Field	**Desrciption**	**Type**	**Example**
file	The location of a Jolie file relative to the architecture file	String	`main.ol`
target	Name of the service in the file	String	`MainService`
name	Name of the service in the file	String	`MainService`
instances	Number of instances of the service to be visualized	Long	`2`
container	Name of the container in the deployment yaml file	String	`MainContainer`
args	Jolie arguments which get added to the Dockerfile after building	String	`–connlimit 10 –stackTraces`
params	Either path to a JSON file containing service parameters, or the parameters as JSON	String or JSON	`params.json` or `{ location: "localhost:1234"}`
env	Deployment environment variables. Gets added in the deployment yaml file	JSON	`{username: "test", password: "123"}`
image	Specifies a remote image which gets added in the deployment yaml file	String	`emilovcina/somejolieimage`
ports	List of strings defining Docker port mappings	String[]	`["4000:4000", "3444:9000"]`
volumes	List of file locations which will get bound as volumes when running the deployment	String[]	`["/config.ini", "assets/test.txt"]`

Appendix B

VS Code Architecture File JSON Snippets

jv

Listing B.1: Scaffolding architecture file.

```json
[
    [
        {"file": "svc.ol", "target": "name", "instances":1}
    ]
]
```

JSON

jvservice

Listing B.2: Scaffolding top-level service.

```json
{"file": "svc.ol", "target": "name", "instances":1}
```

JSON

jvdocker

Listing B.3: Scaffolding top-level Docker service.

```json
{"name": "svc", "image": "image", "instances": 1, "ports":["3000:3000"]}
```

JSON

Appendix C

VS Code Commands

Command	**Desrciption**
Jolie: Visualize	Opens the visualization UI
Jolie: Docker-Compose	Creates the build folder, in the root of the project, and sets up a deployment yaml file.
Jolie: Initialize Architecture File	Creates a scaffolding architecture JSON file in the root of the project.
Jolie: Choose Architecture File	Opens a file selector which allows the user to choose another JSON file as the architecture file.

Bibliography

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

[2] The Jolie Team. *The service-oriented programming language*. URL: `https://www.jolie-lang.org/`.