Packaging Microservices

Fabrizio Montesi and Dan Sebastian Thrane

University of Southern Denmark

Abstract. We describe a first proposal for a new packaging system for microservices based on the Jolie programming language, called the Jolie Package Manager (JPM). Its main features revolve around service interfaces, which make the functionalities that a service provides and depends on explicit. For the first time, JPM supports binding a service to an externally-provided package, and a notion of interface parametricity that can be used to develop generic service libraries that can modify the behaviour of arbitrary services. We illustrate the latter with a generic circuit breaker package.

1 Introduction

Microservices is an emerging paradigm where that components (even the internal ones) are autonomous and reusable services [4]. Applications are built by composing services as black boxes, using message passing.

The nature of microservices fosters granularity, and a MicroService Architecture (MSA) typically consists of many individual services. Since services are independent and their coordination happens only through message exchanges, code reuse (the focus of this work) takes a different form than that found in standard approaches based on software packages. Typically, in other paradigms, packages are software libraries, i.e., pieces of source or compiled code that become a part of the execution of the main application (e.g., through source inclusion, or static/dynamic linking). While this approach can be used for developing an "atomic" microservice (a service that does not contain other services), it falls short of capturing the essence of the paradigm and how it is used.

There are two key aspects that we need to keep in mind when dealing with code reuse in microservices. First, a common pattern in service development is to resolve the dependency of a service simply by *binding* it to an externally-provided service (available somewhere else in the network), instead of importing code to be run locally. Second, if we do decide to import some code to be run locally, that code should still be run as a separate and independent "local" service. This way, if we need to change strategy later on (say, when we go from development to production) and switch from running a dependency locally to binding our service to an external provider, we can do it without changing our implementation.

Package managers for mainstream technologies were not built with MSAs in mind, so these two patterns are not natively supported. Microservice developers must instead typically resort to ad-hoc conventions to deal with these problems.

In this paper, we report on the development of a package management system for the Jolie programming language [5][1]: the Jolie Package Manager (JPM). Jolie

[1] http://www.jolie-lang.org/

supports microservices natively, so it is a prime case study for the development of a package system for microservices that deals with the aforementioned aspects. We illustrate how JPM supports the configuration and use of service packages. Furthermore, JPM supports a notion of interface parametricity (polymorphism), which can be used to develop services whose behaviour is determined by the interfaces of the other services that they are bound to at deployment time. Parametricity is necessary because these interfaces are known only when packages are "linked" to each other (to solve dependencies).

2 A Simple Example

We briefly introduce Jolie with a small e-shop example.

Listing 1.1. `shop.ol`: A small microservice for a shop.

```
1  include "paymentprocessor.iol"
2
3  inputPort Shop { ... }
4  outputPort Warehouse { ... }
5  outputPort PaymentProcessor {
6      Location: "socket://paymentprocessor.com:443"
7      Protocol: https
8      Interfaces: IPaymentProcessor }
9
10 main {
11     checkout(order)(response) {
12         charge@PaymentProcessor( /* ... */ )()
13     }
14 }
```

Listing 1.1 shows a simple `Shop` service. It has a single `checkout` operation, defined in Lines 11–13. The `Shop` has two dependencies: the `PaymentProcessor` and the `Warehouse`, given as output ports. An output port dictactes how we can invoke another service. The output port `PaymentProcessor` is defined in Lines 5–8. This includes a `Location` attribute, which defines where the service can be contacted, a `Protocol` attribute, which defines the transport protocol to be used, and a list of statically defined `Interfaces`, which types the API of the service.

The current practice to make Jolie services configurable is based on ad-hoc conventions. For example, we may include a file named `config.iol` that contains some constant definitions (representing configuration parameters). Listing 1.2 shows the configuration file for the `PaymentProcessor`.

Listing 1.2. `config.iol`: Configuration for the `PaymentProcessor`.

```
1  constants {
2      PAYMENT_PROCESSOR_TEST_MODE = false,
3      PAYMENT_PROCESSOR_ACCOUNT_ID = "xxxxx-xxxxx-xxxxx",
4      PAYMENT_PROCESSOR_LOC = "socket://localhost:443",
5      PAYMENT_PROCESSOR_PROTOCOL = "https"
6  }
```

Here we provide a few fields for the behaviour of the service (`TEST_MODE` and `ACCOUNT_MODE`) and configuration for the input port of the service (`LOC` and `PROTOCOL`). The main problem of this approach is that this file needs to be included as source code by the service that we are configuring. This hides what the parameters mean (Are they bindings or not? What is the resulting architecture?). It also opens to security risks: since we are importing source code, an attacker may insert arbitrary malicious code in the configuration file and it would be executed.

3 Packages and the Package Manager

We introduce a *package* abstraction to the Jolie language and provide a tool that combines packages with configurations to achieve our aims from the Introduction, the Jolie Package Manager (JPM). A package is a folder containing Jolie source code. The code of a package is read-only when used as a dependency, to enable potential updates and integrity checks when packages are installed.

JPM distributes packages following a relatively standard approach. A packages in a repository is equipped with a package manifest. A manifest contains information about the package, used for indexing (e.g., name, description, purpose, etc.) and package management (e.g., dependencies and version). We omit the details of manifests and how they are used to install packages from repositories, since these are similar to those in mainstream package managers. In the remainder, we focus on features that are peculiar to JPM.

3.1 Configuration

Jolie packages are configured by *configuration profiles*, which we introduce here. Crucially, profiles do not need to be included as source code by packages. They are instead given in separate *deployment files* (written by the user of the package) that are processed by the Jolie tool-chain in a controlled way, when we need to run the services given inside of the package. The syntax of profiles recalls that of the Jolie constructs that can be configured. In Listing 1.3, we show an example of a configuration profile for the `PaymentProcessor`, which replaces the ad-hoc source-included configuration file given in Listing 1.2.

Listing 1.3. `pp.col`: Configuration for the `PaymentProcessor` in JPM.

```
1 profile "pp-production" configures "PaymentProcessor" {
2     inputPort PaymentProcessor {
3         Location: "socket://localhost:443"
4         Protocol: https }
5     PAYMENT_PROCESSOR_TEST_MODE = false
6     PAYMENT_PROCESSOR_ACCOUNT_ID = "xxxxx-xxxxx-xxxxx"
7 }
```

This snippet shows a single profile named `"pp-production"`. A profile provides binding information (location, protocol) for communication ports and configuration values to a particular package. A user can provide different configuration profiles, e.g., one for development and one for production, and select among them at deployment time.

We require that the configurable elements of a Jolie program are marked with a new keyword, `#ext`. This allows the developer to omit binding information in

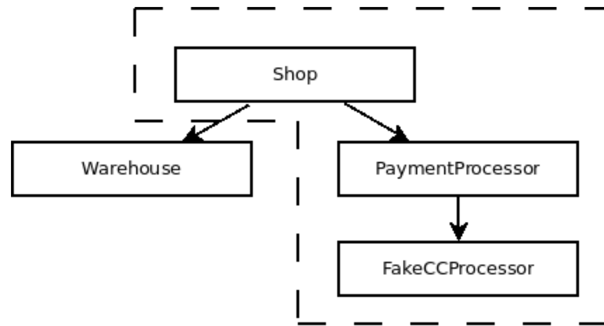

Fig. 1. Desired deployment configuration for our development build. The dashed region represents services inside of the same VM.

communication ports; the omitted field need then to be provided externally by a configuration profile. We do not allow setting the `Interfaces` part of a port externally, since this would prevent type checking of programs until they know their deployment setup (typing how they use ports inside of their behaviours). Thus in the `PaymentProcessor` its input port is defined as:
`#ext inputPort PaymentProcessor { Interfaces:IPaymentProcessor }`.

3.2 Embedded Dependencies

In Jolie, all components are services that run independently. Sometimes, for performance or convenience, it is useful to *embed* a service in the same local VM (Jolie is implemented in Java). Services in the same VM still exchange messages, but they can use efficient in-memory channels, as opposed to the network.

Our new package system allows us to embed pre-configured Jolie packages in two ways. These two ways give a system administrator the freedom to choose and apply the best deployment strategy without any changes required to the services. We start by looking at the externally configurable approach.

Figure 1 shows a development configuration for the `Shop`. In this configuration, we embed the `PaymentProcessor` and its dependencies inside the `Shop`. Listing 1.4 depicts the desired deployment. When we state that a service should be embedded, we simply pass the name of the configuration profile to be used.

Listing 1.4. Embedding services from a configuration.

```
1 profile "shop-development" configures "Shop" {
2     outputPort PaymentProcessor embeds "pp-development"
3 }
4 profile "pp-development" configures "PaymentProcessor" {
5     inputPort PaymentProcessor { Location: "local" }
6     outputPort FakeCCProcessor embeds "fake-processor"
7 }
8 profile "fake-processor" configures "FakeCCProcessor" { ... }
```

Sometimes, it may be desirable to embed services statically in the source code, instead of through configurations. In these cases it is useful to propagate the need for configuration parameters up the tree of embedded services. In the previous example, we might want the `Shop` service to contain configuration for the

PaymentProcessor. In order to solve this problem we allow for configuration needs to be republished to the parent service, but *only* when the embedding is defined statically. We illustrate this in Listing 1.5.

Listing 1.5. shop.ol: Republishing configuration from PaymentProcessor to the Shop.

```
1 constants { PROPAGATED_PARAM: string }
2 outputPort PaymentProcessor { ... }
3 embedded {
4     JoliePackage: "PaymentProcessor" in PaymentProcessor {
5         PARAM republish as PROPAGATED_PARAM
6     }
7 }
```

4 Parametric Interfaces

Proxy services delegate the computation of replies for their requests to other services. A notable example is circuit breaker [8]. We summarise this pattern in the following (see [7] for a thorough discussion in Jolie).

Circuit breakers attempt to protect against some of the problems that occur when using remote calls, such as connection problems, timeouts, and critical faults. During normal operation, a CircuitBreaker functions like a normal proxy between a Client and a TargetService. Monitoring code inside of the CircuitBreaker attempts to detect problems. If enough problems are detected, the CircuitBreaker will start failing immediately without attempting to proxy the call. After a period of time it will start allowing some calls through, and eventually transition back to the normal state and allow all calls through.

Packaging proxy services like circuit breakers raises a particular problem. Intuitively, the proxy should only accept calls for operations that are declared in the interface of the target service. However, this interface is known only at deployment time, since we may want to reuse the same circuit breaker package to protect different services. Proxy services are thus inherently *parametric* on the interfaces of the target services that we choose at deployment time. To address this problem, we introduce the notion of configurable interface to Jolie.

Listing 1.6. cb.ol: An externally defined interface.

```
1 #ext interface ITarget
```

In Listing 1.6 we define an externally configurable interface. A concrete interface is bound to it at deployment time by reading the configuration, giving the service, in this case CircuitBreaker, the information needed to correctly proxy operations. Observe that since we want Jolie services to be type-checkable without knowing their configuration (since configuration may change in different deployment setups), this means that the behaviour of the service is necessarily defined as polymorphic, i.e., it cannot assume any specific operation in the configurable interfaces (this is obtained through aggregation in Jolie, see [5]).

To create a circuit breaker running in a client, we would embed the circuit breaker locally and have it bound to our external payment processor, this is shown in Listing 1.7.

Listing 1.7. `cb.col`: Shop configuration with client-side circuit breaker for `PaymentProcessor`.

```
1 profile "shop-production" configures "Shop" {
2     outputPort PaymentProcessor embeds "cb-pp"
3 }
4 profile "cb-pp" configures "CircuitBreaker" {
5     interface ITarget = PaymentProcessIface from "PaymentProcessor"
6     outputPort TargetSrv { ... }
7 }
```

We can just as easily create a circuit breaker that operates server-side (intercepting incoming calls), or as a proxy in the network, by adopting different deployment files for `"cb-pp"`.

5 Related Work

A common approach to simulate bindings to external services in mainstream languages is to communicate with services via stub libraries. A stub library provides an interface that resembles that of the target service, and internally delegates all work to the latter. In web architectures, these libraries can be synthesised from specifications, for example using OpenAPI [1] specifications in tools like Swagger [9]. Our approach applies directly to web development through Jolie [6].

Seneca [2] is a microservice toolkit for Node.js, where business logic is enclosed in a plugin typically distributed as a Node.js module. By including these plugins as dependencies, a server can essentially embed the logic of these plugins, similarly to JPM. Seneca uses pattern matching to determine which service should handle a particular message. This makes the bindings of a Seneca service implicit, in contrast to JPM, where bindings are explicit. A similar mechanism may nevertheless be implemented in Jolie and JPM by adopting proxy services, cf. [3].

References

1. Open api website. `https://www.openapis.org/`. Accessed 22-02-17.
2. Seneca Authors. Seneca official website. `http://senecajs.org`. Accessed 22-02-17.
3. Mila Dalla Preda, Maurizio Gabbrielli, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. Interface-based service composition with aggregation. In *ESOCC*, volume 7592 of *Lecture Notes in Computer Science*, pages 48–63. Springer, 2012.
4. Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present And Ulterior Software Engineering (PAUSE)*. Springer, 2017. To appear. Available at https://arxiv.org/abs/1606.04036.
5. F. Montesi, C. Guidi, and G. Zavattaro. Service-oriented programming with Jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.
6. Fabrizio Montesi. Process-aware web programming with jolie. *Sci. Comput. Program.*, 130:69–96, 2016. Also: SAC, pages 761–763, 2013.
7. Fabrizio Montesi and Janine Weber. Circuit breakers, discovery, and API gateways in microservices. *CoRR*, abs/1609.05830, 2016.
8. Michael T. Nygard. *Release It!: Design and Deploy Production-Ready Software (Pragmatic Programmers)*. Pragmatic Bookshelf, 2007.
9. SmartBear Software. Swagger website. `http://swagger.io/`. Accessed 22-02-17.