Sliceable Monolith: Monolith First, Microservices Later

Fabrizio Montesi
Department of Mathematics and Computer Science
University of Southern Denmark
Odense, Denmark
fmontesi@imada.sdu.dk

Marco Peressotti
Department of Mathematics and Computer Science
University of Southern Denmark
Odense, Denmark
peressotti@imada.sdu.dk

Valentino Picotti
Department of Mathematics and Computer Science
University of Southern Denmark
Odense, Denmark
picotti@imada.sdu.dk

Abstract

We propose Sliceable Monolith, a new methodology for developing microservice architectures and perform their integration testing by leveraging most of the simplicity of a monolith: a single codebase and a local execution environment that simulates distribution. Then, a tool compiles a codebase for each microservice and a cloud deployment configuration. The key enabler of our approach is the technology-agnostic service definition language offered by Jolie.

1 Introduction

Microservices are the prominent software paradigm for building distributed applications that strive for scalability, maintainability, and tight development and deployment cycles [2]. Microservices enforce strong boundaries and interact by message passing, leading to modular and independently executable software components. Multiple versions of a service can be deployed next to each other so that developers can plan progressive transitions to new versions.

However, the aforementioned benefits of microservices come at a serious cost in terms of complexity. Microservices are distributed systems and typically require dealing with multiple codebases (one for each microservice). This makes prototyping and testing much more challenging compared to monoliths—standard applications that consist of a single executable. These costs can easily outweigh the benefits and, especially when it comes to greenfield project development, experts have very mixed opinions on whether to start with microservices or with a monolith [4, 7, 9]. Thus we ask: Can we recover some of the simplicity of monoliths in the development of microservices? A positive answer would contribute to making the greenfield development of microservice systems more approachable, which is particularly relevant considering the difficulty of migrating monoliths to microservices [9].

In this article, we propose a new development methodology where an entire microservices architecture is coded in a single codebase, which reduces drastically the complexity of reaching a working prototype to iterate on. We depict our methodology in Figure 1 and outline it in the following.

The main artifact in the codebase is a "sliceable monolith": a microservice system definition that looks like a monolith, but where all components are enforced to be services with clear boundaries and data models (e.g., the structures of Data Transfer Objects). We achieve these features by using the Jolie programming language [5], which enforces some best practices for microservice development linguistically: interaction among components happens necessarily through formally-defined service interfaces. Thanks to the built-in facilities of the Jolie interpreter, the application can then be tested locally straight away, enabling fast refinement cycles of the prototype.

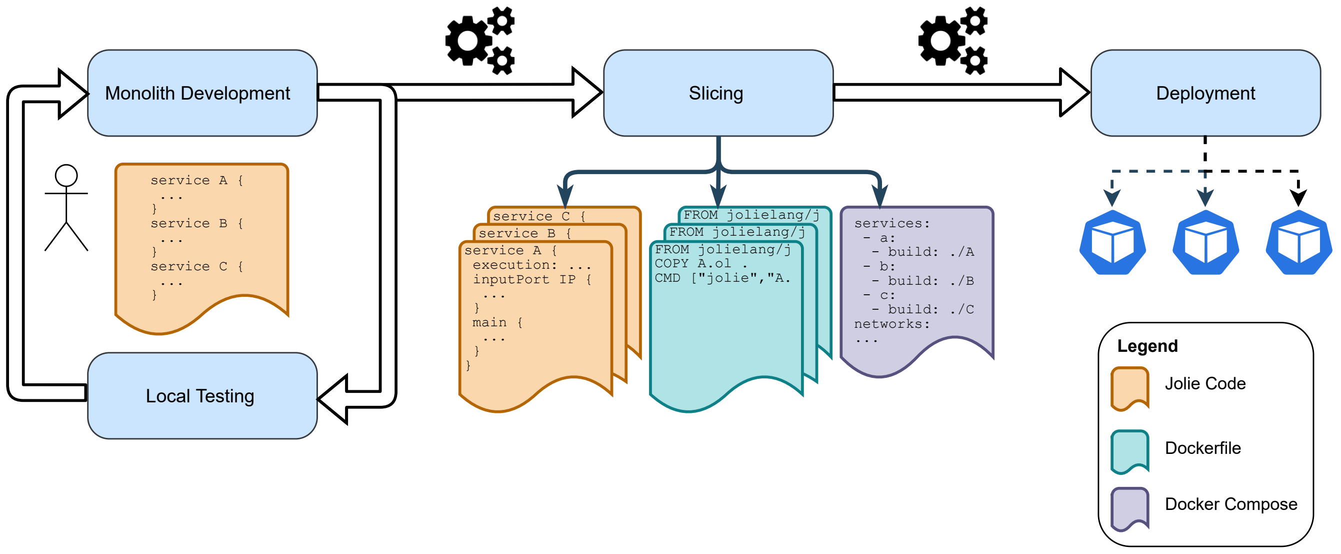
The structure of a sliceable monolith make it possible to automatically extract the implementation of each microservice into its own codebase. We implement this procedure with an automatic *slicer* tool, emphasising the fact that the sliceable monolith is cut alongside the sharp module boundaries defined by the microservices architecture. Our slicer tool also produces the necessary configuration for the containerisation and distributed deployment of the microservice system on the cloud, by using Docker and Docker swarm mode. At this point, developers are free to choose between iterating on the sliceable monolith codebase, or to start developing some (even all) of the microservices independently. The technology-agnostic nature of Jolie interfaces makes it possible to mix different languages for the implementation of each microservice (Jolie currently supports its own behavioural language, Java, and JavaScript, with a plug-in architecture for adding more [5]).

2 A Use Case from Smart Cities

In this section, we present our development methodology with a running example based on (the relevant parts of) the implementation of a use case from smart cities. We model a scenario in which a microservice architecture manages a set of private parking areas whose owners, in agreement with the power grid operators, have decided to share their charging stations in exchange for monetary incentives. The end-users of the application are owners of electrical vehicles looking for available charging stations near a given location. The described scenario is borrowed from [1, 8], where microservices interact according to the Domain Event pattern. The application follows the CQRS pattern (Command Query Responsibility Segregation): two services, one for querying data (`QuerySide`) and the other for updating data (`CommandSide`) interact indirectly through an event store service (`EventStore`).

2.1 Development and Local Testing

Service `QuerySide` allows clients to obtain information about parking spaces equipped with charging stations, performing queries based on geolocation. Service `CommandSide` offers an API for updating information about parking spaces. Service `EventStore` supports the coordination of the other two services by offering an API for event-driven communication.



```
service A {
 ...
}
service B {
 ...
}
service C {
 ...
}
```

```
service C {
service B {
service A {
 execution: ...
 inputPort IP {
  ...
 }
 main {
  ...
 }
}
```

```
FROM jolielang/j
FROM jolielang/j
FROM jolielang/j
COPY A.ol .
CMD ["jolie","A.
```

```
services:
 - a:
  - build: ./A
 - b:
  - build: ./B
 - c:
  - build: ./C
networks:
...
```

Figure 1. The development methodology of Sliceable Monolith.

Following the Sliceable Monolith approach, in this subsection we define the interfaces (data models and APIs) and implementations of these services in a single codebase.

We start by writing a Jolie program that defines a service block for each one of our services:

```
service QuerySide( config ) { ... }
service CommandSide( config ) { ... }
service EventStore( config ) { ... }
```

Each service is parameterised on an externally-provided configuration, called `config`.

We show the most interesting parts of the definition of service `CommandSide`, which is the most involved service. First, we use Jolie data types to define the data model of the messages that the service exchanges. These include parking area identifiers and structures for providing information about each area: name, the time period in which it is available, the speed supported by the charging station in the area, and geolocation.

```
type PAID:long // Parking Area IDentifier
type ParkingArea {
  id:PAID
  info:ParkingAreaInformation
}
type ParkingAreaInformation {
  name:string
  availability*:TimePeriod
  chargingSpeed:ChargingSpeed
  geolocation:Location
}
```

Using the data model, we build the API of `CommandSide` as a Jolie interface that comprises three RPCs for creating, updating, and deleting parking areas (`RequestResponse` is Jolie for RPC):

```
interface CommandSideInterface {
RequestResponse:
 createParkingArea( ParkingAreaInformation )( PAID ),
 updateParkingArea( ParkingArea )( string ),
 deleteParkingArea( PAID )( string )
}
```

In the definition of service `CommandSide`, we offer the API that we have just defined to clients (through a Jolie `inputPort`) and we declare a dependency towards the `EventStore` service (through a Jolie `outputPort`). The locations at which these communication ports should be deployed at are parameters that we get from the externally-provided configuration.

```
/* ... data types and API definitions ... */
service CommandSide( config:Configuration ) {
  execution: concurrent //Handle clients concurrently
  inputPort InputCommands {
    location: config.CommandSide.location
    protocol: http { format = "json" }
    interfaces: CommandSideInterface
  }
  outputPort EventStore {
    location: config.EventStore.location
    protocol: http { format = "json" }
    interfaces: EventStoreInterface
  }
  main { /* business logic implementation */ }
}
```

Observe that the definitions of data types and APIs appear outside of the **service** block that defines the implementation of `CommandSide`. This allows us to share these same types across the implementations of all services, which aids in keeping the data models of interacting services consistent. For example, we can conveniently reuse the type `ParkingArea` in the data types of events that can be exchanges through the `EventStore` service. This convenience comes at zero cost: our slicer tool (next subsection) uses a dependency analysis to produce optimised code for each microservice.

Externally-provided configurations for Jolie services can be given as JSON files. By using a JSON file that provides locations at the local host, we can test the entire architecture locally, both with unit and integration tests. While the latter is typically problematic in general, in our case it is simply a matter of writing a few lines of Jolie code that can be run locally. For example, the following test checks that deleting a parking area triggers the right event notification from `EventStore`.

Jolie

```
subscribe@EventStore( {
  location = testLocation
  topics[0] = "PA_DELETED"
} )( res )
deleteParkingArea@CommandSide( 123L )()
notify( event )
if( event.type != "PA_DELETED" || event.id != 123L )
  throw( AssertionFailed )
```

2.2 Slicing and Deployment

To switch to a distributed architecture, we write another configuration file where each location is now an abstract DNS name (resolved by Docker) and we run our Jolie slicer tool.

Running the slicer on the monolith produces an output directory that contains a subfolder for each service and a `docker-compose` file. The following is the interesting snippet from the Docker Compose file.

Docker Compose

```
version: "3.9"
services:
  queryside:
    build: ./queryside
  commandside:
    build: ./commandside
  eventstore:
    build: ./eventstore
```

Each subfolder contains the code of its respective service. Unnecessary code that pertains to other services is thrown out by the slicer by using a dependency analysis, to avoid namespace pollution and preserve loose coupling. Additionally, each subfolder contains a `Dockerfile` that instructs Docker on how the service should be containerised, like the following.

Dockerfile

```
FROM jolielang/jolie
COPY CommandSide.ol .
COPY ../config.json .
CMD ["jolie", "-p", "config.json", "CommandSide.ol"]
```

The generated Docker Compose file is compatible with Docker swarm mode. It can be used as is, but the programmer can also refine it, e.g., with the desired load balancing configuration. Launching Docker in swarm mode with our Docker Compose file will deploy the entire architecture as a composition of independent microservices, as expected.

3 Conclusion

When developing a new service system, developers have to choose whether to start with a monolith or jump directly to microservices. The first choice leads to quick prototyping, but poses the risk of discarding the monolith entirely when migrating to microservices [3, 4]. The second choice gives immediately a more flexible architecture, but at the cost of slowing down early development tremendously [6, 9].

Sliceable Monolith is a middle ground between these approaches, which retains some of the simplicity of developing a monolith and automates the migration to a full-fledged distributed system of microservices.

For conciseness, we have omitted the business logic implementations of our services in the example (they are not surprising). Our methodology abstracts from the technologies used for such implementations. Our tool already supports different technologies through Jolie, which has a plug-in architecture for implementing the business logics of services (the "**`main`**" block) in other languages, like Java and JavaScript [5].

Our experience gives a first positive answer to the question posed in the introduction. An interesting future work would be to study the impact of our methodology more systematically, using a larger set of use cases and objective metrics.

Acknowledgments

We thank Florian Rademacher for useful discussions about the use case.

References

[1] 2020. PuLS: Parken und Laden in der Stadt. https://parken-und-laden.de/
[2] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*. Springer, 195–216.
[3] Martin Fowler. 2014. Sacrificial Architecture. https://martinfowler.com/bliki/SacrificialArchitecture.html
[4] Martin Fowler. 2015. Monolith First. https://martinfowler.com/bliki/MonolithFirst.html
[5] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. 2014. *Service-Oriented Programming with Jolie.* Springer, 81–107.
[6] Sam Newman. 2015. *Building Microservices: Designing Fine-Grained Systems* (first ed.). O'Reilly.
[7] Sam Newman. 2015. Microservices For Greenfield? https://samnewman.io/blog/2015/04/07/microservices-for-greenfield/
[8] Florian Rademacher. 2021. *A Language Ecosystem for Modeling Microservice Architecture.* Ph.D. Dissertation. University of Kassel. Forthcoming.
[9] Stefan Tilkov. 2015. Don't start with a monolith. https://martinfowler.com/articles/dont-start-monolith.html