A linguistic approach to microservices

Claudio Guidi, italianaSoftware

Distilling the core concepts of microservices into a programming language
Interesting ...but why?
The nature of computational resources is going to change

A machine

The computational resource so far

Naturally not distributed.

Processes focused on computation.

Functions, procedures and objects.

A cloud of machines

The reference computational resource in the next years

Naturally distributed.

Processes focused on communication.

Services, microservices, nanoservices, ...

italianaSoftware
Containers are going to replace the idea of machine we know so far.

- Containerization (cloud independence)
- Virtualization (hardware independence)
- Real Computational Resource
Layers of abstraction

The cloud is here

italianaSoftware
How long are we seeing behind the surface?

Today

We can’t access the real computational level yet
italianasoftware

Today/Tomorrow?
The new machine

This is the new machine we are going to use

The cloud is here
What does it imply?

Which are the short terms and the long terms consequences of such a scenario?
The applications

The applications will be obtained as a composition of existing components
Computational resource

Software functionality

Ideally, the concept of computational resource will be more and more interchangeable with the concept of software component.

In the classical idea where a computational resource is a computer machine, the software is concretely installed into the machine and it is strongly tight to the technical features of the machine.

In the new machine the metrics of the system will be managed by the cloud platform provider as a all which will optimize the loads, by relocating resources or replicating them. The software functionalities will be actually decoupled from the available technical resources.
Communication is more important than computation

Communication and coordination will be more important than computation

The management of the connections and the dependencies among the components is the key point!
Containers today
Core concepts of a container

System Level
• Lighter than Virtual Machines
• Requires a software layer installed in the hosting machine, ex: Docker

Application Level
• Instantiated starting from an image
• Usually requires connections with other containers
• Remotely controlled status: start/stop/pause/destroy/…
Images

- Internally coded as a set of script to be executed
- A container is an instance of an image. Multiple copies of the same container can be generated from the same image.
- **Technology agnosticism:** different applications made with different technologies can be used within a container
- Images can depend on other images
Every application can be installed within a container. Web application, database servers, load balancers, etc. It is sufficient to properly prepare the correspondent image.
Deploying

NEW!

The main impact engineers are facing so far is about application deployment.

- Easy to create
- Easy to start/stop
- Easy to move
- Easy to replicate

PROGRAMMING
Programming still continue to exploit traditional paradigms.
What’s programming?

**Remember**: communication and coordination are the main features to deal with

Two aspects:

1. **Single component** programming
   1. Communication endpoints
   2. Communication protocols
   3. Communication primitives
   4. Activity flow
   5. Fault handling

2. **Architecture** programming.
   1. Component composition
   2. Message routing
   3. Protocol transformation
   4. Asynchronous vs synchronous communication

Depending on the technologies there are different languages, different tools, different methods.
Single component programming

Access endpoints for providing functionalities
- Communication protocols
- Interfaces of the functionalities

Other components dependencies
- Communication protocols
- Interfaces of the functionalities

Business logics
- Actual coding of the provided functionalities

Libraries and Frameworks + Programming Languages =
Frameworks and languages

Communication and coordination features are usually demanded to the usage of frameworks and libraries

- Several tools and frameworks
- They change very often.
- They require upgrades which could have impacts on the code.
- Specific skills required by the developers
- No actual standards for defining interfaces
- Usually structurally designed for specific protocols (ex: http)

The syntax is quite stable
- Once trained, a developer has all the important knowledge to be productive

Unfortunately, the code related to the communication activities is strictly bound to the chosen framework and libraries.
Programming languages are irrelevant

The knowledge required to manage frameworks and libraries eats up the knowledge required to manage the programming language which become quite irrelevant.
Knowledge is money

- More than one
- They change often
- High dinamicity

- Syntax usually stable in the long terms. Backward compatibility usually guaranteed
How to reduce the knowledge?

Reducing the required knowledge by moving the programming concepts hidden in the frameworks into a stable syntax of a programming language.
A new programming language

Our choice is to introduce a new programming language. Jolie represents our solution.

Our aim is to move the knowledge trapped within the frameworks into the programming language.
Jolie

http://www.jolie-lang.org
Once upon a time...

SOA standard specifications analysis (WSDL, SOAP, WS-BPEL,...)

Development of a Milner’s CCS based process calculus (SOCK).
   Definition of operational semantics

Development of a language based (Jolie) and of its interpreter engine

Exploitation in industry and real life
Jolie is interpreted

A jolie service is described by a script which is interpreted at runtime by the jolie engine.

No compilation required.

The engine is developed in Java

We call **microservice** the running instance of a service

![Diagram](image)
Services

The main idea behind Jolie is that

*A service is a single unit of programmable software.*

It is not obtained as a customization or a specialization of an existing server, but it is the only possible programmable entity.

We usually exploit a trapezoid for representing a service.
The linguistic constructs for communication

Listener endpoints and sender endpoints are a specific construct in the language. They are called ports: inputPorts and outputPorts.
The location represents the place, in the space of the Internet, where the service can be reached.
How

The protocol specifies the application protocol to be used. Sodep is a protocol released with Jolie but also http, http/json, http/soap and https.
What

outputPort OPName {
Protocol: sodep
Interfaces: MyInterface
}

inputPort IPName {
Protocol: sodep
Interfaces: MyInterface
}

The interface expresses all the available operations provided by a service.
A service can invoke different services on different ports.

A service can provide its functionalities on more than one port. Different protocols and locations are possible for a service.
The interface definition is part of the language

```plaintext
outputPort OPName {
  Protocol: sodep
  Interfaces: MyInterface
}

inputPort IPName {
  Protocol: sodep
  Interfaces: MyInterface
}

type TestRRequest: void {
  .field: string
}

Interface MyInterface {
  RequestResponse:
    testRR( TestRequest )( string )
  OneWay:
    testOW( TestRequest )
}
```
Communication primitives

OneWay: asynchronous

```java
outputPort OPName {
    Protocol: sodep
    Interfaces: MyInterface
}

main {
    request.field = "hello world!"
    testOW@OPName(request);
    ...other activities...
}

inputPort IPName {
    Protocol: sodep
    Interfaces: MyInterface
}

main {
    testOW(request);
    ...other activities...
}
```
Communication primitives

RequestResponse: synchronous

```
outputPort OPName {
  Protocol: sodep
  Interfaces: MyInterface
}

main {
  request.field = "hello world!"
  testRR@OPName( request )( response );
  ...other activities...
}

inputPort IPName {
  Protocol: sodep
  Interfaces: MyInterface
}

main {
  testRR( request )( response ) {
    ...other activities...
  }
}
```
The behaviour

The behaviour of a service is represented by a **workflow**. 
*Potentially, every service is an orchestrator.*

A behaviour structured as a workflow permits to focus on the message flows instead of computation.
Other

- Tree format for data and messages
- Standard computation constructs
  - if then
  - for
  - while
- Other constructs
  - spawn
  - foreach
  - provide until
  - linkIn and linkOut
- Fault handling
  - Termination handlers
  - Compensation handlers
  - Dynamic installation handlers
- Sessions
  - Session correlation (correlation sets)
  - Global data and session data
  - Critical sections programming (synchronize)
- Execution modalities
  - Single
  - Sequential
  - Concurrent
## Important distilled single component programming concepts

<table>
<thead>
<tr>
<th>Programming Constructs</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ports</td>
<td>They permit to define in a standard way endpoints: protocols and available operations are just parameters.</td>
</tr>
<tr>
<td>Interface</td>
<td>It permits to formally define the interface of the service without exploiting external tools.</td>
</tr>
<tr>
<td>Communication primitives</td>
<td>They permit to define message communication actions (send and receive) independently from protocols and formats.</td>
</tr>
<tr>
<td>Workflow behaviour</td>
<td>It permits to define the activities of a service by focusing on the message flow instead of computation.</td>
</tr>
<tr>
<td></td>
<td>Sessions are instantiated automatically.</td>
</tr>
</tbody>
</table>
If you remember...

**PROGRAMMING WITH JOLIE**

No specific packaging is required or libraries. Just prepare the container with the Jolie Engine.

The code is readable.

The syntax is stable.

**Less knowledge required.**

Common concepts and constructs immediately shared by all the team members. Everybody can read and understand what a service is doing without any additional knowledge.
Programming architectures
Where is the architecture?

Containers

Creating connections
(they must still available at the level of service)

Moving

Scaling

Services

?
Programming services means programming architectures

Since the single programmable unit is a service, it does not matter the conceptual designing model you choose for designing your system or your application because the result will always be an architecture of services!

Each modelled service has an immediate and direct representation into code
Prommaning services means programming architectures

The relation among entities are directedly modelled as service dependencies
Service size is not a problem

You can easily fragment your entities in sub-entities by keeping the mapping with services because services can be EMBEDDED.
Everything you think is an architecture!

Do you remember? We said we want to reduce the knowledge!
Programming architectures with Jolie
Redirection

M

/a

/b

/c
Redirection, the code

```plaintext
outputPort SubService {
    Location: "socket://localhost:8001/
    Protocol: soap
}

outputPort SumService {
    Location: "socket://localhost:8002/
    Protocol: soap
}

inputPort M {
    Location: "socket://localhost:8000/
    Protocol: sodep
    Redirects:  a => A,
                b => B
}
```
Aggregation, the code

outputPort A {
  Location: "socket://localhost:8001/"
  Protocol: soap
}

outputPort B {
  Location: "socket://localhost:8002/"
  Protocol: soap
}

inputPort M {
  Location: "socket://localhost:8000/"
  Protocol: sodep
  Aggregates: A, B
}
Couriers

The extended information can be used for invoking other services (ex: authentication services)

Extended message which carries additional context infos (ex: security tokens)

Message without extension, forwarded to the target service
Ms1, ms2 and ms3 they all have the same interface. Forwarding is decided by using the extended information of the message.
More services can be instantiated within the same microservice.
Jolie & Docker
Jocker

docker pull jolielang/jocker

http docker API on localsocket

Docker
Developing
(experimental)

1. Download project

2. Automatic extraction of dummy dependencies

3. Creation of dummy services images

4. Dummy services containers deployment in local machine

5. Binding service under development to the dummy dependencies

6. Development
Developing
(experimental)

Git repo

commit

9

Destruction of the dummy environment and commit

7

8 destroy images and commit

Docker
**Testing**  
(experimental)

1. Project
2. Download project
3. Download test suite
4. Deploying system  
   And run the test suite
5. Performing tests
6. Test results
7. Destroy system
Dynamic Architectures

(Experimental)
From synchronous…

```plaintext
outputPort OPName {
  Protocol: sodep
  Interfaces: MyInterface
}

main {
  request.field = "hello world!"
  testRR@OPName(request)(response);
  ...other activities...
}

inputPort IPName {
  Protocol: sodep
  Interfaces: MyInterface
}

main {
  testRR(request)(response){
    ...other activities...
  }
}
```
Dynamic Architectures

(Experimental)

…to asynchronous

```
outputPort OPName {
  Protocol: sodep
  Interfaces: MyInterface
}

main {
  request.field = "hello world!"
  testRR@OPName( request ){ response);
  ...other activities...
}
```

```
inputPort IPName {
  Protocol: sodep
  Interfaces: MyInterface
}

main {
  testRR( request )( response ){
    ...other activities...
  }
}
```

Keeping the code synchronous!!!!
Integration with other technologies
Just a brief list of possibilities

- **Java**
  - There is a native possibility to integrate Java code into a Jolie service (JavaServices)
  - It is possible to run a Jolie service within an hosting application server (Ex: JBoss)
  - It is possible to send sodep messages to Jolie from a third party Java application
- **Javascript**
  - As for Java, it is possible to embed javascript code into a Jolie service
- **Text Editors**
  - Plugins for Atom and Sublime Text
- **HTTP and HTTPs**
  - HTTP and HTTPs are supported protocols
- **SOAP Web Services**
  - SOAP is a supported protocol
  - jolie2wsdl and wsdl2jolie are tools which permit to convert Jolie interface into WSDL documents and viceversa
- **JSON**
  - JSON format is supported as format for the http Protocol
- **Web**
  - Leonardo is a web server written in Jolie.
- **REST**
  - Thanks to the usage JSON messages can be exploited just parameterizing a port
  - If necessary, it is possible to implement a standard REST services by exploiting Jester, a REST router for jolie services
- **Databases**
  - SQL databases can be easily connected to a Jolie service using JDBC libraries.
  - MongoDB connector (by Balint Maschio)
Jolie in industry
Jolie Enterprise

- Central control panel
  - Deployment
  - Start/stop
- Logging and monitor
  - All the Jolie service can be monitored and logged
- Used as a digital platform for system integration and business process development
RFC client: it permits to call SAP RFC

IDOC client: it permits to send IDOC to SAP

IDOC server: It permits to receive IDOC documents from SAP

RFC server: It permits to receive RFC calls from SAP
A Jolie service receives documents from SAP
A Jolie service receives documents from SAP.

- Customers usually adopt JSAP for archiving documents from SAP to a third party Software.
Conclusion
A new computational resources paradigm has been introduced by cloud computing.
Software distribution is the natural paradigm for approaching such a kind of systems.
The high level of complexity of distributed systems require to reduce the required knowledge for managing them.
A new generation of languages which crystalize the basic microservice programming principles could help in reducing the required knowledge.
Jolie is a good candidate for representing this new generation of languages.
Jolie is already used as a technology in the industry with successful results.
Thank you
Some interesting architectural programming patterns
Slave Service Mobility
Master Service Mobility
Master Service Mobility
Recursion

```
func(x)
if (x == 0)
    return 0
else
    return x + func(x - 1)
```
Sessions

Sessions are automatically spawned at the reception of the first message. Such a mechanism is transparent w.r.t. the developer.
Example

Creating a REST microservice in Java with Spring Boot
(from a web tutorial)

• Setting up initial **Spring Boot** structure: creation of a controller which registers the method for managing the POST request and its response

• Writing the application by annotating the application class. In the main we need to use method SpringApplication.run(…)

• Documentation generation exploiting **Swagger**

• Packaging Spring Boot for Docker deployment using **Gradle Docker Plugin**

A huge part of the code is tight to the used frameworks.