University of Southern Denmark

IADM801

Spring - Master of Computer Science

Microservice Patterns in Jolie

Submitted To:
Saverio Giallorenzo, Marco Peressotti,Fabrizio Montesi
IMADA
Mathematics & Computer Science Department

Submitted By :
Alexander Lerche Falk
alfal19@student.sdu.dk
171093

Contents

Abstract

Disclaimer

It is assumed the reader is somewhat familiar with the concepts of Jolie Language.

Introduction

Design Patterns are philosophy jargon for good practices within software development. The patterns mirrors reoccurring issues in real-world scenarios and tries to accomodate software developers reinventing the wheel. Through experience you build a repertoire of principles in the creation of software. These principles are called patterns [4], which describes a problem with a solution attached, making it applicable in various contexts. A pattern advises – and considers – the forces and trade-offs for the solution. Design Patterns gained popularity after the release of the book "Design Patterns: Elements of Reusable ObjectOriented Software" [1], written by the Gang-of-Four (referral to the four authors), which gave the insight into reusable templates for problems. Microservice API Patterns are an extensions to Design Patterns, primarily focusing on the communication between a client and a microservice. Lowering bandwidth, service agreements, structure, and API qualities, are some focus areas.
This paper focusón three API patterns [8]:

- Service Level Agreement
- Conditional Request
- Pagination

The patterns are implemented in Jolie [5], which is a native microservice programming language, focusing on services, distribution, and reusability. The last two are obtained by design.
The goal is to show the patterns in play with Jolie and the ease of applying them.

Patterns & Implementations

There are five categories — for now — of patterns:

- Foundation Patterns - related to execution decisions of the API
- Responsibility Patterns - related to roles of API endpoints
- Structural Patterns - related to interface representation design
- Quality Patterns - related to efficiency and cost-effectiveness
- Evolution Patterns - related to life-cycle management / versioning

We are not diving into every category. The three chosen patterns are within the Quality and Structural categories.

In Jolie our examples have the same structure - like a template. There is a main service, providing the requested functionality by the client. It has an inputPort defined, deployed using the local medium [3], using the shared memory for communicating between main service and the pattern serice. The pattern service functions as middleware, sitting infront, and aggregates the interface, of the main service, providing it with the ability of using the courier [2] architecture to intercept the request and respond messages between the client and the main service. The pattern service embeds – **??** – the main service, meaning, when the pattern service starts, it starts the main service as well.

```
outputPort MainService {Interfaces: MainServiceInterface}
embedded {Jolie: "<PATH_TO_MAIN_SERVICE>" in MainService}
```

Listing 1: Embed main service into the pattern service

The difference in the implementations are how the requests and responds are manipulated in the courier.

Quality Pattern: Service Level Agreement

A Service Level Agreement (SLA) [6] is a contractual binding between a provider and one or more clients. SLA's defines the Quality-of-Service (QoS) [7] characteristics through Service Level Objectives, ensuring clients are getting what they are paying for. A single SLA can be described as: Time-To-Respond (TTR). This is the time it takes for – in our case – the microservice to respond to requests. If an agreement from the provider was a response time within x seconds, and it breaches for more than y times per month, then a penalty has to be paid to the client.

Three SLA's are defined:

- Response Time
- Average Response Time
- Number of breached response times

The main service used for this example is the Calculator service, which provides the ability to make addition, substraction, multiply, and division. The pattern service is the SLA service, which provides metrics for the performance analytics of the service. When the client sends a request to the calculator service, the SLA service intercepts, starts a timer, and forwards the request. When the request comes back, it stops the timer **??**.

```
global.number_of_requests++; // Keeping track of every request
getCurrentTimeMillis@Time(void)(start);
forward( request )( response );
getCurrentTimeMillis@Time(void)(stop);
sla_reporting // A function
```

Listing 2: Courier operations for the SLA service

When the timer has stopped, the SLA analytics is calculated **??**. The values of the SLA's have been defined as global constants. On line 1, we find the difference between start and stop time. On line 2, we are executing a function, which returns the average response time of all stored response times. On line 5 we check whether the response time is not living up to the expectation, and increment the total breaches by 1. On line 9, we are checking the average response time, and on line 16 we are aggregating the SLA values into the response to the client.

```
objectives.responsetime = double(stop - start);
calculateTheAvgFailedResponseTime; // A function
objectives.avgresponsetime = sum; // The sum variable is defined by above
objectives.breaches = global.number_of_breaches; // Set to 0, when servic

if ( objectives.responsetime > SERVICE_LEVEL_OBJECTIVE_RESPONSE_TIME ) {
    global.number_of_breaches++
    // Response Time has been breached. Increment the global value.
} else {
    if(sum < SERVICE_LEVEL_OBJECTIVE_AVG_RESPONSE_TIME) {
        // No breaches for response time or the average respones time
    } else {
        // Average response time breached
    }
}
// Unfold the response data structure and deep copy the values into the t
with( response ) { .servicelevelagreement.objectives << objectives };
undef ( objectives )
```

Listing 3: slaOperation

The response is seen in **??**:

```
"result": 100,
"servicelevelagreement": [
    {
        "objectives": {
            "breaches": [
                6
            ],
            "responsetime": [
                3
            ],
            "avgresponsetime": [
                4.375
            ]
        }
    }
]
```

Listing 4: The response to the client when a calculation is requested

By building a ”contract” into the microservice, the API provider implementing the SLA pattern establishes transparency between its clients. It can be on the whole API service or specific operations, but it outlines the terms for the client regards to quality. The downside of implementing the SLA pattern in the microservice is: the transparency makes it easier to be held accountable on the quality of the API. Now the agreement is ”public” and in some cases you do not want this.

Quality Pattern: Conditional Request

In some situations you have client requesting the same data over and over again without changes. This increases the processing time server-side but also the bandwidth required to transfer data back and forth. The Conditional Request (CR) pattern tries to accomodate this problem by timestamping client requests and if the request is within a time period, then no data is returned.

In the Jolie example, we are adding a response header called: "lastModified". When the user creates a request towards the microservice, we respond with this header the first time. It is expected this header is used as a request header at the next request to calculate the difference and return the latest data if necessary.
In the listing **??** on line 1 we are checking for existence of the lastModified header. If it is defined but it is empty, we are not making any checks, and assumes the client got no data from earlier. The lastModified header is returned in the response as a header, and it is going to be used for the next request. On line 7-12 we get the lastModified header in the request and current timestamp. On line 14-16 we execute the check if new data is needed for the client, and respond with the new data, otherwise no new data is needed.

```
if ( is_defined( request.lastModified ) ) {
    // First time a request comes in
    if ( request.lastModified != "" ) {
        getCurrentDateTime@Time( {.format = "dd-MM-yyyy kk:mm:ss"} )( las
        forward( request )( response )
    }
    request.lastModified.format = "dd-MM-yyyy kk:mm:ss";
    getTimestampFromString@Time( request.lastModified )( timestampFromSt
    timestampFromString = timestampFromString / 1000;

    getCurrentTimeMillis@Time( void )( currentTimestamp );
    currentTimestamp = currentTimestamp / 1000;
    // New data is returned
    if ( (currentTimestamp - timestampFromString) > 300 ) {
        getCurrentDateTime@Time( {.format = "dd-MM-yyyy kk:mm:ss"} )( las
        forward( request )( response )
    } else {
        statusCode = 304
        // "No New Content. Saving Bandwidth";
    }
}
```

Listing 5: Courier operations for the Conditional Request service

Using this pattern we can save bandwidth, which can be crucial in situations where your hosted microservice has a limit on data usage.

Structural Pattern: Pagination

The Pagination pattern is

Conclusion

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addision-Wesley, 1994.

[2] Saverio Giallorenzo. Jolie - architectural composition: Couriers.

[3] Saverio Giallorenzo. Jolie local location.

[4] Craig Larman. *Applying UML and Patterns.* John Wait, 2004.

[5] Fabrizio Montesi, Claudio Guidi, and Saverio Giallorenzo. Jolie language, July 2019.

[6] Sonia Pearson. Service level agreement.

[7] Various. Quality of service.

[8] Olaf Zimmermann, Marko Stocker, Uwe Zdun, Daniel Lübke, and Cesare Pautasso. Microservice api patterns, July 2019.