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

Microservices are surging through the IT industry, trying to accomodate the issues from Service-Oriented Architecture. The Microservice Architecture keeps an overview of the microservices, but it does not show how to implement recurring problems microservices comes with. Microservice API Patterns tries to help with this. In this paper the focus lies on three patterns: Service Level Agreement, Conditional Request, and Pagination, which helps with quality and optimization of microservices. We implement them using the native microservice language, Jolie, showing the ease of implementing patterns in Jolie and the advantages of doing so.

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 [6], 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" [2], 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 focus' on three API patterns [12]:

- Service Level Agreement
- Conditional Request
- Pagination

The patterns are implemented in Jolie [7], 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 [4], 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 [3] architecture to intercept the request and respond messages between the client and the main service. The pattern service embeds – Listing 1 – 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) [8] is a contractual binding between a provider and one or more clients. SLA's defines the Quality-of-Service (QoS) [11] 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 Listing 2, and executes a request against the SLA Storage Service. This service handles everything related to the SLA conditions, functions, and storage.

```
getCurrentTimeMillis@Time(void)(start); // Start Timer
forward( request )( response );
getCurrentTimeMillis@Time(void)(stop); // Stop timer
responsetime = double(stop - start);
buildReport@SLAStorageService( { .responsetime = responsetime } )(
    reportingResponse );
response.report << reportingResponse
```

Listing 2: Courier operation for the SLA service

The SLA Storage Service uses a JSONfile as basic persistent storage and has three operations are defined:

- buildReport - analyzes the response time of the calculation done for the client, checks for contractual breaches, and updates the JSONfile
- getReport - retrieving the information from the JSONfile and returns the response to the client
- reset - resets the file. Can be useful when shifting month

The "buildReport" operation is called by the SLA Service, which provides the response time as a request parameter. The operation calls a function – Listing 3 – providing the logic behind the SLO's. On line 2 it opens and reads the persistent storage file, using the content of the file as a data structure for the lifespan of the function. On line 4 a function is called to calculate the average response time SLO. From line 6-19 it is checking if any SLO has been breached or if everything is fine. From line 20-27 it is updating the persistent storage file with the new information, and lastly returns the response back to the user.

```
define build_sla_reporting {
    readFile@File( { .filename = STORAGE_FILENAME, .format = "json" } )(
        fileresponse );
    fileresponse.numberofrequests++;
    calculateTheAvgFailedResponseTime; // function to calculate the total average
        response time
    fileresponse.avgresponsetime = avgsum
    if ( request.responsetime > SERVICE_LEVEL_OBJECTIVE_RESPONSE_TIME ) {
        fileresponse.numberofbreaches++;
        // Breach of response time SLO
        if ( TotalNumberOfBreaches >=
            SERVICE_LEVEL_OBJECTIVE_NUMBER_OF_BREACH_RESPONSE_TIMES )
            // Breach of total number of breaches SLO
    } else {
        // Response time is fine
        if(fileresponse.avgresponsetime < SERVICE_LEVEL_OBJECTIVE_AVG_RESPONSE_TIME
            ) {
            // No breach of the average response time
        } else {
            fileresponse.numberofbreaches++;
            // Breach of average response time SLO
        }
    }
    ...
    // Construct the response format to the file
    ...
    writeFile@File( file )();
    with( objectives ) {
        .responsetime = request.responsetime;
        .avgresponsetime = fileresponse.avgresponsetime;
        .breaches = fileresponse.numberofbreaches
    };
    with( response ) { .servicelevelagreement.objectives << objectives };
    response.statusCode = 200;
}
```

Listing 3: Updates persistent storage file and checks for contractual breaches

The response is seen in Listing 4:

```
{
    "result": 1305,
    "report": {
        "servicelevelagreement": {
            "objectives": {
                "breaches": 4,
                "responsetime": 12,
                "avgresponsetime": 6
            }
        },
        "statusCode": 200
    }
}
```

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. It has been chosen to move the persistent storage to its own service instead of incomporating it into the SLA Service. The SLA Storage Service is running sequentially, which guarantees every client exclusive access to a resource, and data races can be minimized. By separating the persistent storage logic to its own service we also gain more loose coupling. Since the SLA is bound to the service, we can have a global perspective of the client satisfaction. One problem with this is if two clients are located in two distinct regions, and one client is satisfied, the second is not, but the SLA shows everything being fine. In this case you can apply use the SLA Storage Service and distribute it to be applicable based on regions. If the SLA Service goes down, the SLA Storage Service is not affected, which is important.

Quality Pattern: Conditional Request

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

The Time-Based Conditional Request [9] is the approach chosen for the follow Jolie example, where the response header called: "Last-Modified" is used to show a timestamp, which can be used in subsequent requests. The setup is identical with the SLA example, except the main service is now a DataService microservice, which can respond with a chunk of random data. The first time the user creates a request towards the DataService microservice, we respond with the "Last-Modified" header. It is expected this header is used in the request header called: "If-Modified-Since" [1] at the next request to calculate the difference and return the latest data if necessary. In the Listing 5 on line 1 we are checking for existence of the "If-Modified-Since" request header. If it is defined, but it is empty, we are not making any checks, assuming the client got no data from earlier, responding with everything, and append the "Last-Modified" response header. From line 8-13 we get the "If-Modified-Since" header in the request, containing the clients current timestamp. On line 15-21 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.("If-Modified-Since") ) ) {
    length@StringUtils(request.("If-Modified-Since"))(headerLength);
    if ( headerLength <= 0) {
        getCurrentDateTime@Time( {.format = "dd-MM-yyyy kk:mm:ss"} )( lastModified
            );
        forward( request )( response )
        response.statusCode = 200
    } else {
        request.("If-Modified-Since").format = "dd-MM-yyyy kk:mm:ss";
        getTimestampFromString@Time( request.("If-Modified-Since") )(
            timestampFromString );
        timestampFromString = timestampFromString / 1000;
        getCurrentTimeMillis@Time( void )( currentTimestamp );
        currentTimestamp = currentTimestamp / 1000;
        // New data is returned
        if ( (currentTimestamp - timestampFromString) > MODIFIED_TIMESTAMP ) {
            getCurrentDateTime@Time( {.format = "dd-MM-yyyy kk:mm:ss"} )(
                lastModified );
            forward( request )( response )
            response.statusCode = 200
        } else {
            response.statusCode = 304 // "No New Content. Saving Bandwidth";
        }
    }
}
```

Listing 5: Courier operations for the Conditional Request service

There are other alternatives to be used for the Conditional Request pattern. One of them is called: Fingerprint-Based Conditional Request [1], and is somewhat similiar to TimeBased. Fingerprint-Based generates a tag for the client, where to get this tag you can use a hash function on the response to generate a unique value. The tag becomes the unique value and is returned as a response to the client. From here it can be used as in TimeBased, where the client sends it along the request, and then a check can verify whether the response has changed from the last time. Since the Conditional Request Service for now is tightly coupled with the TimeBased approach, it could be an idea to abstract the checking functionality into a "Condition Service", where the client can specify which approach to use, giving more flexibility, and less coupling.

Structural Pattern: Pagination

In the Conditional Request pattern, we are checking whether it is necessary to respond with all data to renew the client-side. In some situations the client is not interested in getting all data but only a chunk of it. This is where the Pagination pattern [10] comes into the picture. Pagination is about cut-off of data and provide the client freedom to specify what data to retrieve. As in the Conditional Request pattern, we have multiple variants to choose from. In the Jolie example it is the Offset-Based Pagination which has been chosen. In this approach, we have three meta-data values:

- Offset - The number of documents to skip before start picking documents to return
- Limit - The amount of documents to return
- Page - The total number of documents is broken into pages of a certain size. This specify which page to start on

If the Offset and Limit values have been defined in the request, we start by counting documents from the Offset and until the Limit, and then return the chunk of data as response to the client. If one of them are missing, and Page is defined, we start from the Page times the PAGE_SIZE, which is a constant defined server-side. As an example if Page = 2 is defined, and PAGE_SIZE = 25, our starting index is (Page - 1) * PAGE_SIZE, and end index is PAGE_SIZE * Page. With this approach, we can move to the next page of documents and not interfere with previous documents, avoiding duplications. We show this in Listing 6. From line 3-9, we check whether Offset and Limit are set. If they are, we respond with the documents within their defined ranged. Else from line 11-20, we check whether Page is defined, and proceed with above condition calculation.

```
 1  forward( void )( newresponse );
 2  // Store meta-data values
 3  if ( request.offset != "" && request.limit != "" ) {
 4      runUntil = request.offset + request.limit;
 5      for ( i = request.offset , i < runUntil , i++ ) {
 6          response.data[i-request.offset].chunk = newresponse.data[i].chunk
 7      }
 8      ...
 9      response.statusCode = 200
10  } else {
11      if ( request.page != "" ) {
12          runUntil = (PAGE_SIZE * request.page);
13          if (request.page != 1) {
14              request.page = (request.page - 1) * PAGE_SIZE
15          }
16          for ( i = request.page-1, i < runUntil , i++ ) {
17              response.data[i-request.page+1].chunk = newresponse.data[i].chunk
18          }
19          ...
20          response.statusCode = 200
21      }
22  }
```

Listing 6: Courier operations for the Pagination Service

If we set the Offset to 50 and Limit 50 3, we get the response as in Listing 7

```
{
"paginationdetails": {
    "offset": 50,
    "limit": 3
}, "data": [
        {"chunk": "438b100f-afcd-4b54-a491-1d1fed677cf4"},
        {"chunk": "b750e10c-6d4e-4ed5-8fff-1d6ff4092150"},
        {"chunk": "1b92843e-8340-459e-ade6-88e32a271298"}
    ],
    "statusCode": 200
}
```

Listing 7: The response to the client based on Offset and Limit meta-data

One drawback of using Offset-Based is the scaling of documents. The farther you increase your Offset, the more intensive does the processing get, because you have to read through all documents up to the Offset index. An alternative to the Offset-Based approach is the Cursor-Based [5]. Here you have a cursor pointing to an unique identifier, liking to the next identifier. The chosen meta-data value for the Cursor-Based has to be sequential. Have the linking provides a knowledge-mechanism of where to go next with little or none processing. This comes with a drawback: you cannot jump back to a previous page.

Conclusion

Three Micoservice API Patterns have been shown to provide mechanisms to increase the quality of the services, and at the same time optimize how data is handled and transferred. The examples shows how Jolie is capable of utilizing these patterns and with minimal code. Because of the way Jolie's handles data structures, we can have one-to-one mappings between our types and requests/responses, and it eases up the implementations of Microservice API Patterns. The examples show one way of implementing the patterns, but depending on the use-case, the alternatives can be a better fit.

References

[1] R. Fielding and J. Reschke. Conditional request, June 2014.

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

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

[4] Saverio Giallorenzo. Jolie local location.

[5] Michael Hahn. Evolving api pagination at slack, August 2017.

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

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

[8] Sonia Pearson. Service level agreement.

[9] Various. Conditional request.

[10] Various. Pagination.

[11] Various. Quality patterns.

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