University of Southern Denmark

IADM801

Spring - Master of Computer Science

Microservice Patterns in Jolie

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

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

Contents

Abstract

Microservices are a standard to implement large-scale distributed systems. Experts have defined and documented a catalogue of Microservice API Patterns, which represent recurring problems. Those documents does not reference implementations and this is the starting point of 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 generic implementations of the patterns and commenting how Jolie supports them.

Disclaimer

It is assumed the reader is familiar with some concepts of Jolie Language. Complete implementation of the examples used in this paper can be found at:
https://github.com/AlexanderFalk/microservicesinjolie

Introduction

Design Patterns are jargon for good practices within software development. The Patterns crystallise recurring solutions in real-world scenarios. Through experience you build and form a repertoire of principles in the creation of software. Concretely patterns [6] describe a problem with a solution attached, making it applicable in various contexts. A pattern advises – and considers – the pros and cons 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 insight into reusable templates for problems. Microservice API Patterns [13] are an extension to Design Patterns, that primarily focus on the communication between a client and a microservice. Lowering bandwidth usage, service-level agreements, and structure are some of the focus areas.
This paper focus' on three API patterns [13]:

- Service-Level Agreement - contract of the API operations to describe expectations
- Conditional Request - lowering bandwidth and processing
- Pagination - optimzing a response by lowering the amount of data returned

The patterns have been selected because they represent different classes of patterns. The Service-Level Agreement represents monitoring of the API; Conditional Request represents proxying of the request; and Pagination enhances operations of the API. The patterns are implemented in Jolie [7], a native microservice programming language. Experts have defined and documented a catalogue of Microservice API Patterns, which represent recurring problems. Those documents does not reference implementations and this is the focus of this paper.

Patterns & Implementations

Currently, there are five categories of microservice 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

The three chosen patterns are within the Quality and Structural categories and, therefore, not diving into every one of them.

The provided implementations follow the same architecture: a main service, providing the requested functionality by the client. It has an inputPort defined, deployed using the local medium [4], which uses the shared memory for communicating between main service and the pattern service. The pattern service uses two architectural constructs: (1) aggregation of the interfaces from the main service, which exposes its operations using the outputPort; and (2) the Courier [3] to intercept the requests and responses before and after forwarding the request to the main service. The pattern service embeds – Listing 1 – the main service, meaning, when the pattern service starts, it starts the main service.

```
1 outputPort MainService {Interfaces: MainServiceInterface}
2 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. The main service can be modified to run in a remote medium. This can be done by removing the embedding and changing the protocol. By doing so, the pattern services can utilize every available main services instead of the one coupled through embedding. Embedding is used because it eases up the development, knowing the main service starts when the pattern service starts.

Quality Pattern: Service Level Agreement

A Service Level Agreement (SLA) [8] is a contractual agreement between a provider and one or more clients. An SLA defines conditions over Quality-of-Service (QoS) [12] characteristics, e.g. the maximal time the service takes to respond to a request - also called Time-To-Respond (TTR). Those measurements are used to regulate the

relationship between the client and the provider of the service. For example, the provider guarantees response time within x seconds and within a given request limit (e.g., 99% of times) or a time span (within a given month). If the provider breaches the SLA, a penalty has to be paid to the client.

In our reference implementation, there is defined three QoS:

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

The main service used for this example is a Calculator service, which provides the ability to make addition, substraction, multiplication, and division. The pattern service is an SLA service and it has a sub-service called SLA Storage Service, providing metric analytics and persistent storage. When the client sends a request to the Calculator service, the SLA service intercepts it, starts a timer, and forwards the request. When the request comes back, it stops the timer, and executes a request against the SLA Storage Service. The code is encapsulated in the Courier, which is a Jolie construct for an aggregated port, allowing requests to be intercepted and conditions of forwarding. The behaviour above is implemented in lines 1-10 of Listing 2:

```
1  courier SLA {
2      [interface CalculatorInterface( request )( response ) ] {
3          getCurrentTimeMillis@Time(void)(start); // Start Timer
4          forward( request )( response );
5          getCurrentTimeMillis@Time(void)(stop); // Stop timer
6          responsetime = double(stop - start);
7          buildReport@SLAStorageService( { .responsetime = responsetime } )(
               reportingResponse );
8          response.report << reportingResponse
9      }
10 }
```

Listing 2: Courier operation for the SLA service

The SLA Storage Service uses a JSON-file as basic persistent storage and has three operations defined:

- buildReport - analyzes the response time of the calculation done for the client, checks for contractual breaches, and updates the persistent storage
- getReport - retrieving the information from the persistent storage and returns the response to the client
- reset - resets the persistent storage. 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 procedure – Listing 3 – defining the logic behind a Service-Level Objective (SLO) i.e, the conditions applied over the QoS of a given API. At line 2 the procedure opens and reads from the persistent storage, using the content of the file as a data structure for the lifespan of the request. At line 4 a procedure is called to calculate the average response time SLO. From line 6-19 the procedure is checking if any SLO has been breached. From line 20-28 the procedure is updating the persistent storage file with the new information, and lastly returns the response back to the user.

```
1  define build_sla_reporting {
2      readFile@File({.filename = STORAGE_FILENAME, .format = "json" })(fileresponse);
3      fileresponse.numberofrequests++;
4      calculateTheAvgFailedResponseTime; // function to calculate the total average
           response time
5      fileresponse.avgresponsetime = avgsum;
6      if ( request.responsetime > SL_OBJECTIVE_RESPONSE_TIME ) {
7          fileresponse.numberofbreaches++;
8          // Breach of response time SLO
9          if(TotalNumberOfBreaches >= SL_OBJECTIVE_NUMBER_OF_BREACH_RESPONSE_TIMES)
10             // Breach of total number of breaches SLO
11     } else {
12         // Response time is fine
13         if(fileresponse.avgresponsetime < SL_OBJECTIVE_AVG_RESPONSE_TIME) {
14             // No breach of the average response time
15         } else {
16             fileresponse.numberofbreaches++;
17             // Breach of average response time SLO
18         };
19     };
20     ...
21     // Construct the response format to the file
22     ...
23     writeFile@File( file )();
24     with( objectives ) {
25         .responsetime = request.responsetime;
26         .avgresponsetime = fileresponse.avgresponsetime;
27         .breaches = fileresponse.numberofbreaches
28     };
29     with( response ) { .servicelevelagreement.objectives << objectives };
30     response.statusCode = 200;
31 }
```

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 with its clients. It can be on the whole API service or specific operations, but it outlines the terms for the client with relation to quality. 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. Another advantage is: if the SLA Service goes down, the SLA Storage Service is not affected, which is important to keep data intact and avoid losing data being processed by the SLA Service.

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 solve this problem by tracking client requests and if the request matches a specified condition, then no data is returned, because the client can use a cached copy of a previous request.

The Time-Based Conditional Request [10] is the approach chosen for the following 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 similar to 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. The client is expected to use header 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 Listing 5 at line 1 we are checking for existence of the "If-Modified-Since" request header. If it is defined, but 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. At 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.

```
1  if ( is_defined( request.("If-Modified-Since") ) ) {
2      length@StringUtils(request.("If-Modified-Since"))(headerLength);
3      if ( headerLength <= 0) {
4          getCurrentDateTime@Time({.format = "dd-MM-yyyy kk:mm:ss"})(lastModified);
5          forward( request )( response );
6          response.statusCode = 200
7      } else {
8          request.("If-Modified-Since").format = "dd-MM-yyyy kk:mm:ss";
9          getTimestampFromString@Time( request.("If-Modified-Since") )(
               timestampFromString );
10         timestampFromString = timestampFromString / 1000;
11         getCurrentTimeMillis@Time( void )( currentTimestamp );
12         currentTimestamp = currentTimestamp / 1000;
13         // New data is returned
14         if ( (currentTimestamp - timestampFromString) > MODIFIED_TIMESTAMP ) {
15             getCurrentDateTime@Time({.format="dd-MM-yyyy kk:mm:ss"})(lastModified);
16             forward( request )( response );
17             response.statusCode = 200
18         } else {
19             response.statusCode = 304 // "No New Content. Saving Bandwidth";
20         }
21     }
22 }
```

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 similar the to Time-Based one. In modern application installers an SHA-256 checksum [9] is used to verify whether the installation data has been tampered with, which increases the security of the installation. Fingerprint-Based is similar to modern application installers, since you can verify if the response has changed during multiple requests. Potentially, Fingerprint-Based can be used in reverse, where it ensures the data is not tampered with. 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 Fingerprint-Based can be used as in Time-Based, where the client sends the tag along the request, and then a check can verify whether the response has changed from previous request. Since the Conditional Request Service for now is tightly

coupled with the Time-Based 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 copy of the data at 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 [11] comes into the picture. Pagination is about cutting off data and providing 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 below we chose the Offset-Based Pagination. In this approach, we have three meta-data values:

- Offset - The number of documents to skip and starting index
- Limit - The amount of documents to return
- Page - The total number of documents is broken into pages of a certain size. This specifies which page to start from

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 is 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 the Page calculation:

```
forward( void )( newresponse );
// Store meta-data values
if ( request.offset != "" && request.limit != "" ) {
    runUntil = request.offset + request.limit;
    for ( i = request.offset, i < runUntil, i++ ) {
        response.data[i-request.offset].chunk = newresponse.data[i].chunk
    }
    ...
    response.statusCode = 200
} else {
    if ( request.page != "" ) {
        runUntil = (PAGE_SIZE * request.page);
        if (request.page != 1) {
            request.page = (request.page - 1) * PAGE_SIZE
        }
        for ( i = request.page-1, i < runUntil, i++ ) {
            response.data[i-request.page+1].chunk = newresponse.data[i].chunk
        }
        ...
        response.statusCode = 200;
    }
}
```

Listing 6: Courier operations for the Pagination Service

If we set the Offset to 50 and Limit to 3, we get the response showed 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

Smarter implementations are possible. For example, one drawback of the current implementation is that the greater the offset is increased, the slower the processing get, because you have to read through all documents up to the Offset index. In Listing 6 we are copying data into a new response instead of removing from the original response. This is good if you have a small amount of data you want to get, but if you need nearly all data, removing from the original response would be better, since it limits the processing, and you avoid a one-to-one copy. An alternative to the Offset-Based approach is the Cursor-Based [5]. Here you have a cursor pointing to an unique identifier, linking to the next identifier. The chosen meta-data value for the Cursor-Based has to be sequential. Having 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. As in Conditional Request,

an idea can be to abstract the implementation details into a "AbstractPagination Service", giving the client the freedom to specify which implementation to use.

Conclusion

Three Micoservice API Patterns [13] reference implementations have been shown to provide mechanisms to the services. The examples shows how Jolie is capable of implementing these patterns with minimal code. With the way Jolie 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. A new service can be created, which was done in the SLA Service implementation, where the persistent storage was moved into its own service, to gain more loose coupling, and comply with Microservice Architectural style.

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 official documentation - architectural composition: Couriers.

[4] Saverio Giallorenzo. Jolie official documentation - 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 official documentation - language, July 2019.

[8] Sonia Pearson. Service level agreement.

[9] A. Rijsinghani. Computation of the internet checksum via incremental update, May 1994.

[10] Olaf Zimmermann, Marko Stocker, Uwe Zdun, Daniel Lübke, and Cesare Pautasso. Conditional request.

[11] Olaf Zimmermann, Marko Stocker, Uwe Zdun, Daniel Lübke, and Cesare Pautasso. Pagination.

[12] Olaf Zimmermann, Marko Stocker, Uwe Zdun, Daniel Lübke, and Cesare Pautasso. Quality patterns.

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