South Dakota State University

Open PRAIRIE: Open Public Research Access Institutional Repository and Information Exchange

Electronic Theses and Dissertations

2018

Service Integration Design Patterns in Microservices

Meng Wang
South Dakota State University

Follow this and additional works at: https://openprairie.sdstate.edu/etd

Part of the Computer and Systems Architecture Commons

Recommended Citation

Wang, Meng, "Service Integration Design Patterns in Microservices" (2018). *Electronic Theses and Dissertations*. 2944.
https://openprairie.sdstate.edu/etd/2944

This Thesis - Open Access is brought to you for free and open access by Open PRAIRIE: Open Public Research Access Institutional Repository and Information Exchange. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of Open PRAIRIE: Open Public Research Access Institutional Repository and Information Exchange. For more information, please contact michael.biondo@sdstate.edu.

SERVICE INTEGRATION DESIGN PATTERNS IN MICROSERVICES

BY

MENG WANG

A thesis submitted in partial fulfillment of the requirements for the

Master of Science

Major in Computer Science

South Dakota State University

2018

SERVICE INTEGRATION DESIGN PATTERNS IN MICROSERVICES

MENG WANG

This thesis is approved as a creditable and independent investigation by a candidate for the Master of Science in Computer Science degree and is acceptable for meeting the thesis requirements for this degree. Acceptance of this does not imply that the conclusions reached by the candidates are necessarily the conclusions of the major department.

Yi Liu, Ph.D.
Thesis Advisor Date

George Hamer, Ph.D.
Acting Department Head Date
Dept. of Electrical Engineering and Computer Science

Dean, Graduate School Date

ACKNOWLEDGMENTS

I always believe that your choices define your life. Life is like a program that includes a lot of selection statements, but no looping statements or rerun functionality. Making a decision each time means the beginning of the next new journey. As I look back, one of the most impressive choices is that I started to pursue a computer science master's degree at South Dakota State University. The university was an entirely new place for me three years ago, so was the computer science major. And I do enjoy this journey of life. The quietness of Brookings makes me feel peaceful, and the rigor of computer science program reshapes my mind. What's more, I have got lots of help from the professors, friends and my family during the trip.

I would like to express my deepest thanks and sincere appreciation to Dr. Yi Liu for her patient guidance in my study. Dr. Liu gave me the valuable opportunity to participate in her project, providing a large number of research resources and various academic activities. Working for the project offers a basis and direction for my thesis and has enabled me to improve myself in practice. Besides, Dr. Liu is also a good friend who always cares about my life especially when I got pregnant. Without Dr. Liu's help, I would have not studied as smoothly as I am doing now.

My gratitude also goes to my friends. Thank you for talking to me and helping me exclude the anxiety whenever learning difficulties or setbacks in life. Thank you for encouraging and cheering me when seeing I obtain the success.

Finally, I am indebted to my family. My husband has been taking care of me by my side, accompanying me, and supporting me. My baby in my tummy has been growing up healthily although I pushed hard and was anxious sometimes. My parents have been encouraging and bringing me confidence.

I am so lucky that my choices make me grow up. I appreciate everybody who has helped me. The warmth you bring to me is the most precious treasure that I will ever cherish.

CONTENTS

LIST OF FIGURES viii

ABSTRACT x

Chapter 1 Introduction 1

1.1 Introduction 1

1.2 Motivation 2

1.3 Objectives 2

1.4 Thesis organization 3

Chapter 2 Background 4

2.1 EPIDEMIA System 4

2.2 Microservices 8

2.3 Cloud computing & Amazon Elastic Compute Cloud 10

2.4 Related Work 12

2.4.1 Design patterns for microservices on Microsoft Azure [15] 12

2.4.2 Microservices design patterns [22] 15

Chapter 3 Service Integration Design Patterns 18

3.1 Synchronous messaging design pattern 19

3.1.1 Problem 19

3.1.2 Context 19

3.1.3 Solution ... 19
3.1.4 Example implementation ... 20
3.1.5 Consequences ... 21
3.2 Asynchronous messaging design pattern ... 22
3.2.1 Problem ... 22
3.2.2 Context ... 22
3.2.3 Solution ... 22
3.2.4 Example implementation ... 23
3.2.5 Consequences ... 24
3.3 Hybrid messaging design pattern ... 24
3.3.1 Problem ... 24
3.3.2 Context ... 24
3.3.3 Solution ... 24
3.3.4 Example implementation ... 26
3.3.5 Consequences ... 28
3.4 Guidelines ... 29
Chapter 4 Case Study ... 32
4.1 EPIDEMIA overview ... 32
4.2 Microservices with Spring Boot ... 33
4.3 Applying the design patterns ... 34

4.3.1 The design pattern for each request 34

4.3.2 The recipe file settings 38

4.3.3 The design pattern implementation.. 40

4.3.4 Deployment in Amazon EC2 41

Chapter 5 Evaluation.. 42

5.1 Addressing the challenge of services integration....................................... 42

5.2 Simple and flexible patterns.. 44

5.3 General guidelines 45

5.4 Easily modifying the relationships... 45

5.5 Related work 46

5.5.1 Design patterns for microservices on Microsoft Azure [15] 46

5.5.2 Microservice design patterns [22]... 46

Chapter 6 Conclusion... 48

6.1 Conclusion 48

6.2 Future Work 49

References 50

LIST OF FIGURES

Figure 1 The top-level Architecture of EPIDEMIA 4

Figure 2 The high-level architecture of the public health interface 5

Figure 3 Screenshot of the Upload page 6

Figure 4 The high-level architecture of EASTWeb [7] 6

Figure 5 The high-level architecture of the data integration subsystem 7

Figure 6 Monolith and Microservices 9

Figure 7 Implementation of the design patterns in microservice architecture 15

Figure 8 Synchronous messaging design pattern 20

Figure 9 Method “callWithBlock” in java 21

Figure 10 Asynchronous messaging design pattern 23

Figure 11 Method “callWithoutBlock” in java 23

Figure 12 Hybrid messaging design pattern 25

Figure 13 Adding “level” and “parent” attributes in the pattern 26

Figure 14 Method “hybrid” in Java 26

Figure 15 Method “parseByLevel” in Java 27

Figure 16 Method “callByParent” in Java 28

Figure 17 Example of XML recipe configuration file 30

Figure 18 The workflow of fulfilling a request 32

Figure 19 Screenshot of Spring Initializr 33

Figure 20 Service integration for epidemiological data upload 34

Figure 21 Service integration for environmental data update 35

Figure 22 Service integration for data update 35

Figure 23 Service integration for data integration 36

Figure 24 Service integration for report generation... 37

Figure 25 XML recipe file for EPIDEMIA system 38

Figure 26 Method "locateRequest" in Java 39

Figure 27 Method "fetch" in Java 40

Figure 28 EPIDEMIA system in microservices on Amazon EC2 41

Figure 29 The user interface of EPIDEMIA system in microservice 43

Figure 30 The response after running request "Upload Epidemiological Data" .. 44

ABSTRACT

SERVICE INTEGRATION DESIGN PATTERNS IN MICROSERVICES

MENG WANG

2018

"Microservices" is a new term in software architecture that was defined in 2014 [1]. It is a method to build a software application with a set of small services. Each service has its process to serve a single purpose and communicates with other services through lightweight mechanisms. Because of a great deal of independently distributed services, it is a challenge to integrate the loose services fully. Too many trivial relationships can be messed up easily during deployment. Also, it is hard to modify the relationships if the services are updated as the source codes need to be re-edited and tested.

The microservices architecture is attracting much attention recently. More and more software-developers are producing continuous applications and microservices deliveries [2]. There is a need to develop a mechanism to better integrate the scattered services during the application delivery process.

The thesis proposes three general design patterns to integrate services in microservices architecture. These patterns are classified by the inter-service communication mechanisms and described with specific problems, contexts, solutions, example implementations and consequences. Besides, the informative guidelines are provided to make these patterns apply in different applications quickly.

The service integration design patterns help compose services and facilitate the process of building applications in microservices. All the patterns are helpful tools to address the service integration issues in microservices. Each approach is simple and flexible to apply generally. The structures can be easily modified through these approaches.

Chapter 1 Introduction

1.1 Introduction

"Microservices" is a new term in software architecture and has attracted more and more advocates recently, such as Amazon, Netflix, The Guardian, the UK Government Digital Service, realestate.com.au, Forward and comparethemarket.com and so on [1]. A microservices system contains a set of independent services and each service delivers a single business capability to support high cohesion and loose coupling. The microservices architecture is characterized by modularity, fine-grained service, and simple communication mechanisms and brings the benefits of flexible development, painless evolution, and elastic scalability. However, it also comes with multiple issues in integration due to the independently distributed services. Simple point-to-point connections between services may become a pitfall as too many trivial relationships can be messed up easily during deployment. In addition, after the relationships are built, it is hard to make changes in the structure as the source codes are required to modified and tested. These issues bring us to the question: how do we fully integrate the loose collection of components once we have split the whole system into a bunch of microservices?

Service integration is a challenge in microservices architecture. There could be a large number of services in a system and a service may be reused within different scopes which will intensify the complexity of the services composition. Thus, the goal of this thesis is to introduce general integration design patterns to facilitate the process of application building. The thesis presents three integrating services approaches according to the basic inter-service communication styles. By applying the design patterns, the

distributed autonomous microservices are organized together in a particular integration framework to cleanly offer all the functionalities in a system.

1.2 Motivation

As Gartner predicted, the hyper-converged integrated systems would capture a rapidly expanding market in the next few years and produce continuous applications and microservice deliveries [2]. More and more software developers will use microservices architecture in the development and have to search for solutions to better integrate the scattered services during the application delivery process.

The thesis proposes three general design patterns with simple classifications. These patterns provide straightforward and flexible solutions regarding the service integration problem. By applying the patterns to specific contexts, the components of an application can be wholly organized and contribute to implementing all the functionalities.

1.3 Objectives

This thesis aims to introduce microservices integration approaches to help organize services during building application process.

The objectives of the thesis are:

1. To address the service integration issues in microservices by applying classified integration design patterns.
2. To apply the integration design patterns in applications simply and flexibly.
3. To modify the structures easily through the integration approaches.

1.4 Thesis organization

The thesis focuses on introducing the service integration approaches. Chapter 2 gives an overview of microservices architecture, EPIDEMIA system and Amazon Web Service (AWS), and reviews others works related to microservices integration. Chapter 3 introduces the integration design patterns, describes the problem, context, solution, implementation, and consequences in each pattern, and provides the guidelines for applying the integration mechanism. Chapter 4 presents the process of using the design patterns to EPIDEMIA system as the case study. Chapter 5 evaluates the integration approaches by moving EPIDEMIA system to microservices with the proposed design patterns and compares our work with others' related work. Chapter 6 concludes the thesis and discusses other directions for future work.

Chapter 2 Background

This chapter overviews the EPIDEMIA system that is used as the case study in the thesis, introduces microservices with its concept, key characteristics, and benefits, depicts cloud computing and Amazon Elastic Compute Cloud (Amazon EC2), and reviews the related literature on microservices integration.

2.1 EPIDEMIA System

The Epidemic Prognosis Incorporating Disease and Environmental Monitoring for Integrated Assessment (EPIDEMIA) is a malaria information system [5]. It integrates weekly malaria surveillance data and remote sensing environmental data to support malaria detection and forecasting in the Amhara region of Ethiopia. The EPIDEMIA system is built upon a collection of standalone subsystems including *public health interface, remote sensing data processing, data integration, modeling, and reporting* subsystems, which were developed separately in fulfilling specific objectives. The top-level architecture is shown in Figure 1.

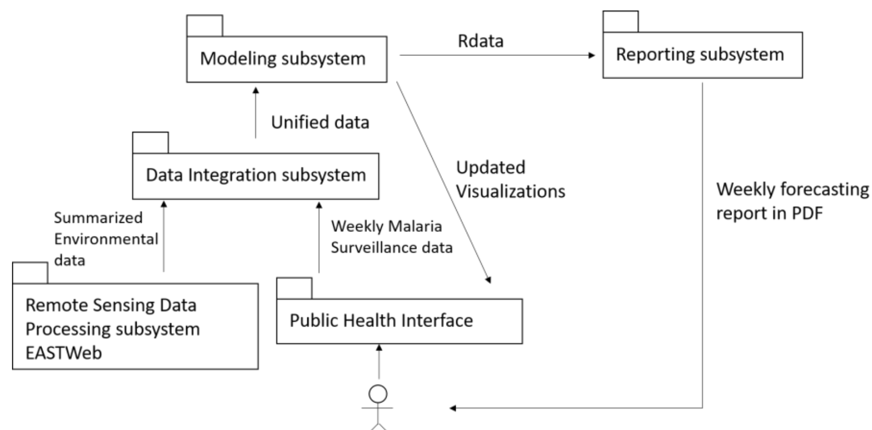

Figure 1 The top-level Architecture of EPIDEMIA

1. Public health interface

The *public health interface* was designed and implemented as the EPIDEMIA project website [6] sitting on a Linux web server. The architecture of the *public health interface* is shown in Figure 2. Given access to the website, the users can upload, query and download surveillance data, and view the interactive and static visualizing epidemiological and environmental data.

Figure 3 displays the upload page of the EPIDEMIA project website. The public health users can upload the disease surveillance data in the format given on the webpage. The submitted surveillance data are inserted into the corresponding epidemiological table in the EPIDEMIA database after being cleaned by the epidemiological data processing component in the *data integration* subsystem.

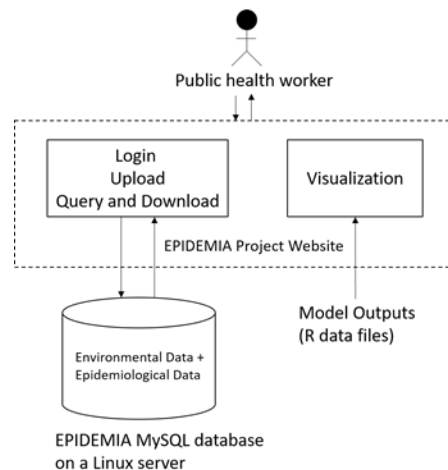

Figure 2 The high-level architecture of the public health interface

Figure 3 Screenshot of the Upload page

2. Remote sensing data processing subsystem

The *remote sensing data processing* subsystem is the EASTWeb system [7, 8]. The datasets used in the EPIDEMIA system are IMERG and IMERG RT rainfall products from GES DISC and MODIS MCD43A4 reflectance, MCD 43A2 quality, and MOD11A2 land surface temperature products from the LP DAAC. Utilizing the components shown in Figure 4, EASTWeb automatically downloads these datasets from the online archives repository, processes them and stores the generated statistical summaries into PostgreSQL database on the Windows Server where the EASTWeb system is running.

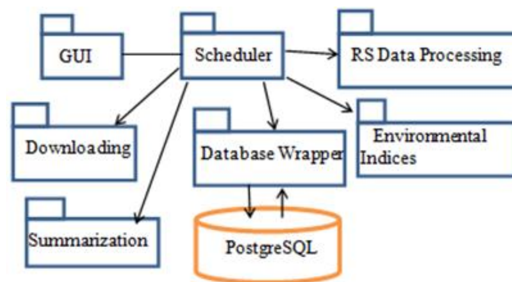

Figure 4 The high-level architecture of EASTWeb [7]

3. Data integration subsystem

The *data integration* subsystem is composed of three components: the Epidemiological Data Processing Component, the Environmental Data Transfer Component, and the Data Unification Component to integrate the epidemiological and the environmental datasets. The high-level architecture of the *data integration* subsystem is given in Figure 5.

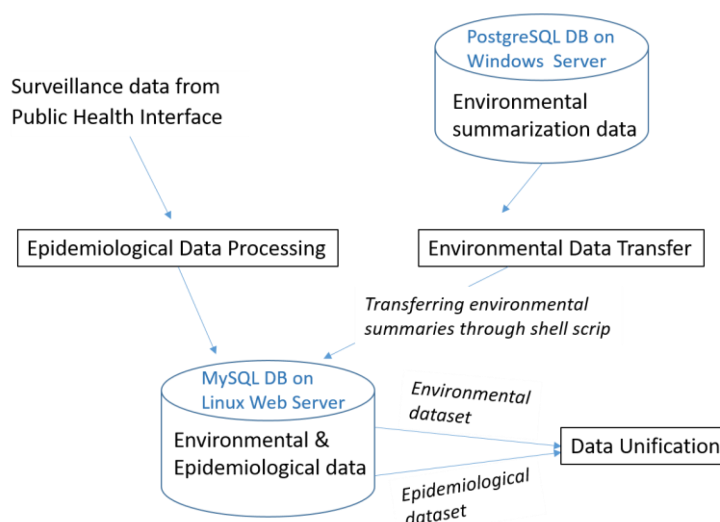

Figure 5 The high-level architecture of the data integration subsystem

The Environmental Data Transfer component ships the summarized environmental data generated from the *remote sensing data processing* subsystem on a Windows server to the Linux web server and stores the data in the MySQL database along with the summarized epidemiological data. A shell script is written to transfer the environmental data. The users need to log in to the Linux web server and enter a command to invoke the shell script to transfer the summarized environmental data.

The Data Unification component is developed using several R packages. It plays a vital role in temporally harmonizing the epidemiological and environmental datasets into a unified dataset when the *modeling* subsystem requests the dataset.

4. Modeling subsystem

Two models, an early detection model and an early warning model, were developed in the *modeling* subsystem. The early detection model uses surveillance data from July 2012 to identify seasonal patterns and longer-term trends of increasing or decreasing malaria incidence [5]. The early warning model is used to forecast malaria risk in future weeks based on recent short-term trends and environmental anomalies that influence mosquito populations and transmission dynamics [5].

Both models were implemented in R on a local machine via RStudio, a free and open-source IDE for R [9]. The *modeling* subsystem invokes the data unification component in the data integration subsystem for a unified dataset and uses it in the detection and warning models. The model outputs are saved as R data files.

5. Reporting subsystem

The *reporting* subsystem converts the model outputs from the *modeling* subsystem to an RNW file, from which PDF reports are generated using knitr [10], an engine for dynamic report generation with R. The generated reports are saved in the Linux web server.

2.2 Microservices

Martin Fowler defined the term “Microservices” in 2014 [1] as a method to build a software application with a set of small services. Each service has its process to serve a

single purpose and communicates with other services through application programming interfaces (API) [1]. There are three key characteristics in microservices.

1. Modularity

The microservices architecture contains a suite of independent modules in a system. Each module, also called a microservice, encapsulates its domain logic and contributes to the whole functionalities of a system, unlike the monolith which puts all the functionalities in a single process [1]. Figure 6 compares a monolithic system with a microservices system. The modularity improves the flexibility of the development and the deployment of each service flexible and increases the comprehensibility of the system.

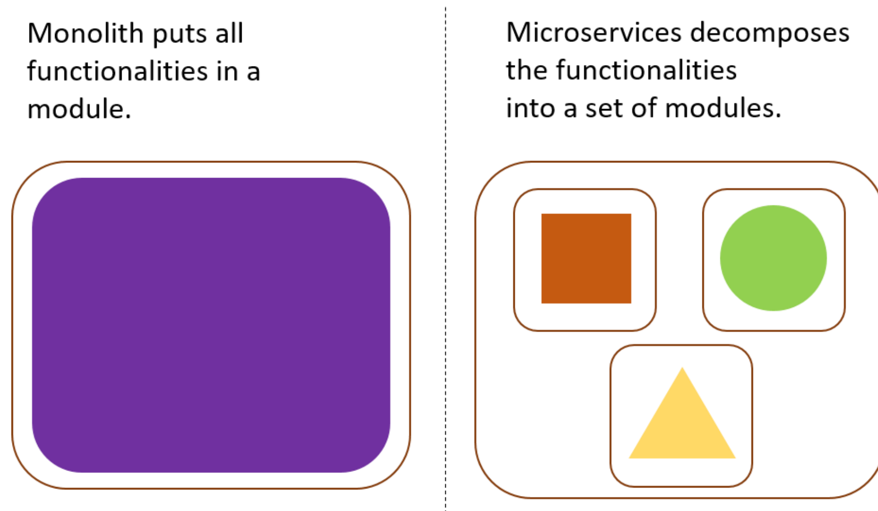

Figure 6 Monolith and Microservices

2. Fine-grained service

As the name “Microservices” suggests, the size of each service should be comparatively small. If a service serves more than one purposes, it should be broken down into smaller units according to the purposes. Each service should focus on a single

business capability to support low coupling in the system. The independent fine-grained services bring a low cost of system maintenance and evolution in the future.

3. Simple communication mechanism

The microservices architecture focuses on lightweight communication mechanism instead of hiding complexities in the communications. HTTP request-response with resource API's and lightweight messaging are commonly used in microservices to provide "dumb pipes" [1]. This distinct characteristic makes microservices different from service oriented architecture (SOA). SOA commonly implements its communication structure with Enterprise Service Bus (ESB), which is a centralized service bus and promotes a high-level protocol communication [3]. The ESB may bottleneck integration and limit elasticity. Simple communication approach enables changes in services without modifying the central service communication bus and offers the system better scalability [4].

2.3 Cloud computing & Amazon Elastic Compute Cloud

- **Cloud computing**

Cloud computing is a model that provides the configurable information technology (IT) sources to consumers in an on-demand self-service manner through the internet rapidly [11]. The IT sources include computing power, storage capacity, bandwidth, Domain Name System (DNS), etc.

There are five common characteristics in cloud computing as explained below.

1. Large-scale distributed servers. The cloud providers generally have large-scale cloud services platforms. Most of the famous enterprises own millions of servers such as

Google, Amazon, IBM, Microsoft and so on. Building upon these distributed servers, the cloud platforms provide vast computing power.

2. Virtualized data center. By using cloud computing, the customers move their emphasis from burdensome work on hardware purchase and maintenance to the application deployment that really matters. After choosing and registering accounts in a cloud service platform, the users can log in the cloud platform to purchase and configure the required services like database service, storage service, etc. These services work as virtualized data centers that can be managed through your PC or mobile devices over the network anywhere at any time. With the virtualized data centers, the developers can implement and deploy their applications easily.
3. Reliability and elasticity. In general, the cloud service platforms adopt data replica fault-tolerant, computing node isomorphism and interchangeable and other methods to support high reliability [12]. Moreover, the capabilities can be provisioned and released elastically to scale up and down according to your demand [11].
4. Pay on the demand. The consumers can purchase the services as their demand or usage. For example, Amazon Web Service offers the "Pay-as-you-go" mode allowing you to pay base on the changing needs. The cost is comparatively low and reasonable as the computing time and provisioned storage mainly determines it.
5. Security. The well-known cloud providers have their professional teams to protect the data's security and reduce the risk of data leakage. Besides, the critical identity and access management provide customers confidence in safety.

Cloud computing has three service models: Software as a Service (SaaS), Infrastructure as a Service (IaaS), and Platform as a Service (PaaS) [11].

SaaS provides applications to consumers through a standard interface such as web browser directly. Google Docs is the application belonging to SaaS. The providers are responsible for the layers underlying the cloud infrastructure [11].

IaaS lets customers purchase the cloud infrastructure to manage the operating system and deploy their applications, for example, AWS Elastic Compute Service.

PaaS is similar to IaaS, but the users only deploy the applications by the operating system and environments that are managed and controlled by the cloud providers. AWS Elastic Beanstalk is an example of PaaS.

- **Amazon Elastic Compute Cloud (Amazon EC2)**

Amazon EC2 is a web service that allows users to rent the cloud servers to deploy their applications easily and rapidly. EC2 adopts open source Kernel-based Virtual Machine (KVM) hypervisor technology for virtualizing compute infrastructure [13]. Each virtual machine, also knowns as an instance, runs small, large, extra-large virtual private servers and supports both Windows and Linux operating systems. Amazon EC2 passes multiple benefits such as elastic web-scale computing, completely controlled instances by consumers, flexible cloud hosting services, integrated cloud services, reliable environment, cloud security, inexpensive cost and easily start [14].

2.4 Related Work

2.4.1 Design patterns for microservices on Microsoft Azure [15]

The AzureCAT patterns & practices team presented nine design patterns that can be especially applied to microservices. Each pattern is described with the problem,

context, solution, issues and considerations, and when to use the pattern [15]. These patterns are summarized as below.

- Ambassador pattern.

By creating helper services to send requests and give a response, it works as a proxy between the application and remote services. The pattern is helpful when building a common set of client connectivity tasks or supporting the connection in a legacy application [16].

- Anti-Corruption Layer pattern.

The layer isolates different subsystems and translates the communications among subsystems that don’t have uniform semantics [17]. It acts as a façade that integrates the legacy and new systems.

- Backends for Frontends pattern.

It creates separate backend services for each type of user interface, such as desktop backend service, mobile backend service, etc [15]. These simpler, finer, faster backends bring flexibility in functionality backend integration.

- Bulkhead pattern.

The pattern splits service instances into different pools based on consumer load and availability requirements [18]. By isolating resources and services, the pattern protects the system against defeated by a service failure.

- Gateway Aggregation pattern.

To avoid chatty communication between the client and the services, the pattern uses a gateway to receive the client requests and dispatch them to the corresponding backend services, then combines and gives the responses [19].

- Gateway Offloading pattern.

It offloads some responsibilities from the services to a gateway proxy, such as certificate management, monitoring, protocol translation [20].

- Gateway Routing pattern.

It places a gateway with a single endpoint between the consumer and services. The gateway is responsible for routing requests from the consumer to appropriate services.

- Sidecar pattern.

It integrates part of the services in the application into a separate process or a container that works as an attached sidecar [21]. The sidecar pattern makes the application more extensibility by adding multiple components and technologies.

- Strangler pattern.

It describes a process to replace the legacy services with new services inch by inch and finally, a new system is built up entirely.

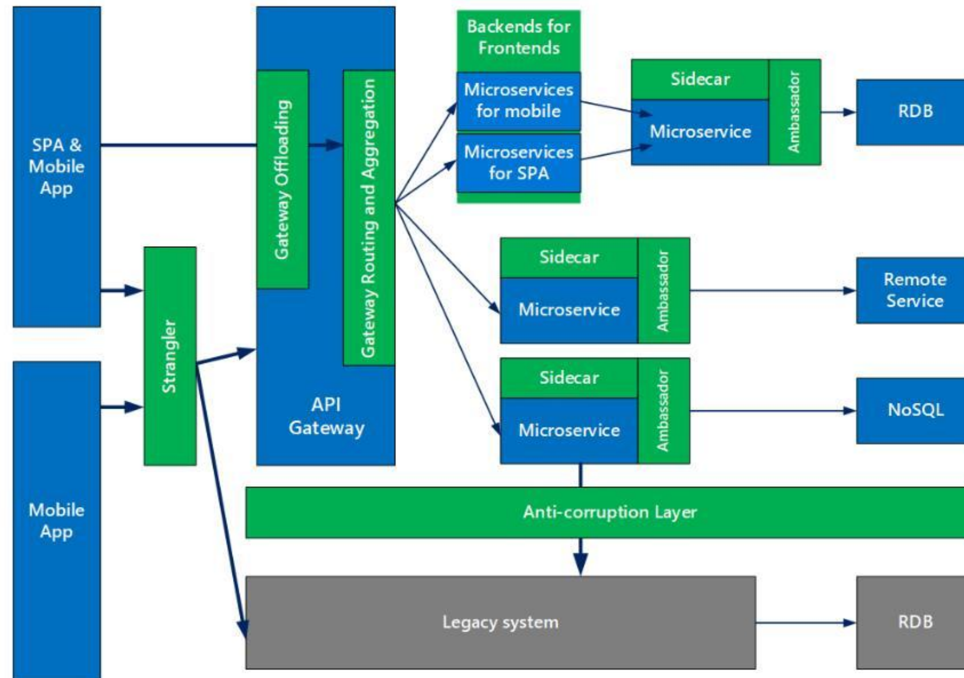

Figure 7 Implementation of the design patterns in microservice architecture

All of these patterns are used to solve small and fine issues in differing composition situations. They could be applied together to integrate the services as shown in Figure 7 [15].

2.4.2 Microservices design patterns [22]

Arun Gupta [22] introduces six design patterns to compose services.

- Aggregator microservice design pattern.

There are two aggregate strategies in this pattern. One is that the aggregator is a single interface that aggregates different services together and can invoke all the services to fulfill a particular request. The other strategy is that the aggregator works as a high-level composite microservice. Instead of aggregating the microservices themselves, the aggregator collects data from each service and applies business logic to the services [22].

- Proxy microservice design pattern.

It is similar to the aggregator microservice design pattern. The difference is that there is no microservice aggregation but microservice delegation in the pattern. The proxy can be either dump to delegate the request only or smart to do the data transformation between the services and requests.

- Chained microservice design pattern.

All the related services are invoked one by one to complete the request in the pattern. Each service will give a response to the calling service to add its business value.

- Branch microservice design pattern.

The pattern extends the aggregator microservice design pattern and the chained microservice design pattern. The request is sent to one service which acts like an aggregator invokes one or more chained services.

- Shared data microservice design pattern.

Although microservice favors a separate database for each service to preserve its independence, there may be a shared database between high coupling services according to the specific business needs in microservice. Different from the branch microservice design pattern, the shared data microservice design pattern adds a shared database.

- Asynchronous messaging microservice design pattern.

Base on the branch microservice design pattern, this pattern applies asynchronous messaging communication technique like pub/sub messaging. It integrates services with reduced coupling between services.

All of these patterns are prevalent and useful, providing multiple integration approaches from varying perspectives when designing and implementing microservices.

Chapter 3 Service Integration Design Patterns

This chapter presents general service integration design patterns to speed building applications up in microservices. The design patterns are synchronous messaging design pattern, asynchronous messaging design pattern and hybrid messaging design pattern. Each pattern provides a detailed solution and example implementation towards a certain problem and context. In addition, the guidelines of applying these patterns to implement microservices are demonstrated.

The novel service integration design patterns have distinctive features. First, the classify criteria are innovative and make the approaches apply generally. These design patterns are proposed based on the inter-service communication mechanisms, synchronous and asynchronous. The communication is synchronous if a request requires a response from the service and is blocked while waiting for the response [23]. The communication is asynchronous if a request is not blocked while waiting for the response if there is any [23]. Some requests can be completed through either synchronous messaging or asynchronous messaging, and the others may need a combination of these two messaging approaches. The synchronous and asynchronous messaging are basic communication methods in microserivces. Therefore, the design patterns can be applied in different applications widely. Second, the solutions presented in the patterns are evolved from the current prevalent design patterns summarized by Arun Gupta. For example, the synchronous messaging design pattern is improved by the chained microservice design pattern [22] via discarding the features of separate databases and two-way communications between services, but adding the feature of single-way

communications to reduce coupling. Third, the example implementation of each pattern and the guidelines to apply the patterns make the approaches complete and unique.

3.1 Synchronous messaging design pattern

Synchronous messaging design pattern is a variation of the chained microservice design pattern [22]. One difference is that each service can have either its own database or a shared database if there is any. The other is that the services communication is one-way instead of two-way. It can be applied when services are interacting via synchronous messaging.

3.1.1 Problem

A set of services are required to add their values by executing the business logics inside, and a service instance will be blocked until it is triggered.

3.1.2 Context

After sending a request, the client is expecting a response from the called service. To complete the request command, more than one services are waiting in a queue to execute. The client will not activate the next service until receiving a response.

3.1.3 Solution

Suppose we have three services, Service A, B, C to execute by the request. As shown in Figure 8, the external request is sent to service A and is waiting for a response. Service B is blocked until Service A is completed, and Service C starts to execute when Service B communicates with Service C after running. Since the data are updated at different stages and will not be messed up by applying the services functionalities, the databases of the services can be isolated or shared.

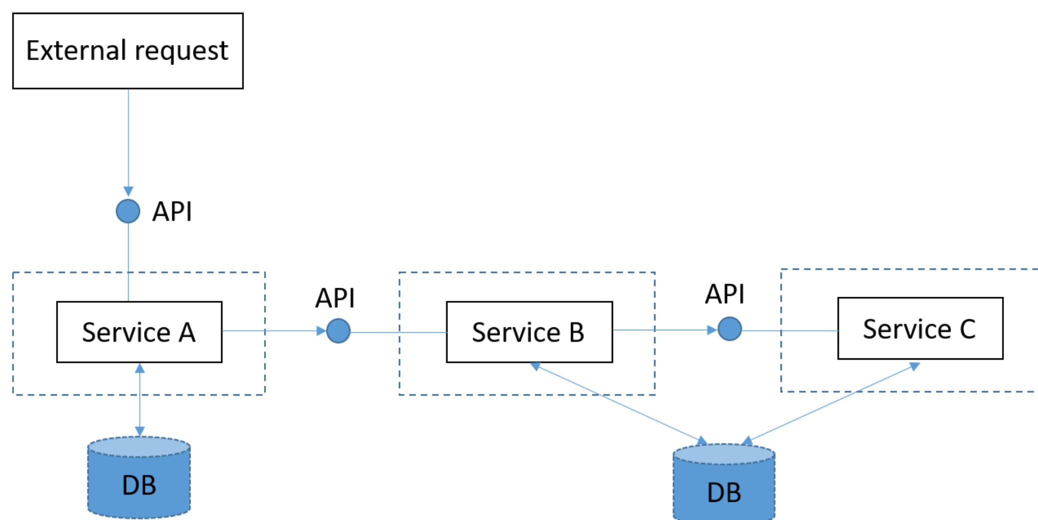

Figure 8 Synchronous messaging design pattern

3.1.4 Example implementation

Figure 9 shows the method "callWithBlock" implemented in Java. It blocks other remained services in the list while executing a service via a block file. The block file will be held during the execution of the service.

```
/************************************************************
*** DESCRIPTION : <call a service and block others >        ***
*** INPUT ARGS  : <services>                                ***
*** OUTPUT ARGS : <None>                                    ***
*** IN/OUT ARGS : <None>                                    ***
*** RETURN      : <None>                                    ***
************************************************************/
public void callWithBlock (List<String> services) throws
Exception {
      // No services is invoked
      if (services.isEmpty()) {
            return;
      }
      // call first service
      invoke(services.get(0));

      String lockFile;
      Path path;
      boolean exists;

      for (int i = 1; i<services.size(); i++) {
            // put the name and path of the lockFile to check
            lockFile = ".\\"+services.get(i-1)+".lock";
            path = FileSystems.getDefault().getPath
                  (System.getProperty("user.dir"), lockFile);
            exists = Files.exists(path, new LinkOption[]
                     {LinkOption.NOFOLLOW_LINKS});

            // if lockFile exists then wait
            while (exists) {
                  Thread.sleep(3000);
                  exists = Files.exists(path, new LinkOption[]
                     {LinkOption.NOFOLLOW_LINKS});
            }
            // invoke next service when the lockFile is deleted
            invoke(services.get(i));
      }
}
```

Figure 9 Method “callWithBlock” in java

3.1.5 Consequences

The synchronous messaging design pattern ensures the services communicating through synchronous messaging. However, the total response time is a concern especially when the waiting queue is composed of many services. Moreover, the cascading failures of the system may occur if one service fails.

3.2 Asynchronous messaging design pattern

Asynchronous messaging design pattern is applied in asynchronous messaging that doesn't block the client while waiting for a response.

3.2.1 Problem

The thread of the request will not be blocked after it calls a service to run. The client may invoke another service without getting any response.

3.2.2 Context

The client sends a request to a service, and the response is not necessarily needed. Other services may be invoked by the request as well without any block.

3.2.3 Solution

An external request goes through a set of services before receiving a response as shown in Figure 10. No service is prevented while running other services. In this pattern, all of the services will not share their databases to avoid messing up data. When a request is received, the invoked services will update their data respectively if they have.

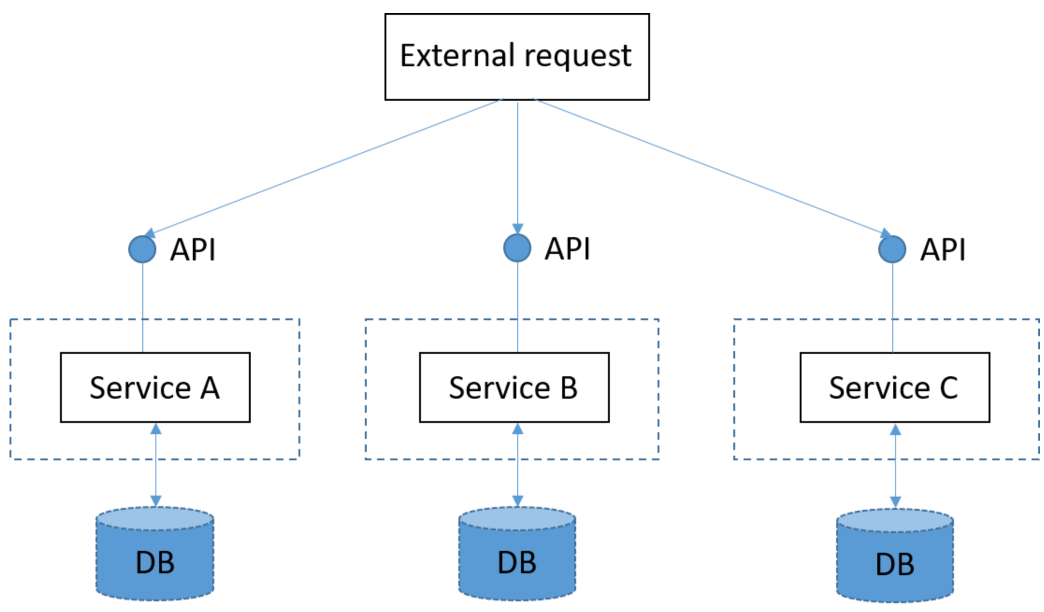

Figure 10 Asynchronous messaging design pattern

3.2.4 Example implementation

```
/*****************************************************************
*** DESCRIPTION : <call a service without blocking others > ***
*** INPUT ARGS  : <services>                                ***
*** OUTPUT ARGS : <None>                                    ***
*** IN/OUT ARGS : <None>                                    ***
*** RETURN      : <None>                                    ***
*****************************************************************/
public void callWithoutBlock (List<String> services) throws
Exception {
      // No services is invoked
      if (services.isEmpty()) {
            return;
      }
      // invoke all related services
      for (int i = 0; i<services.size(); i++) {
            invoke(services.get(i));
      }
}
```

Figure 11 Method “callWithoutBlock” in java

3.2.5 Consequences

The asynchronous messaging design pattern is quite powerful and well understood. It minimizes the interaction between the services. Even if one service fails, other services will not be affected.

3.3 Hybrid messaging design pattern

To realize a business need, a combination of synchronous and asynchronous messaging is typically required. The hybrid messaging design pattern provides a solution by assembling the synchronous messaging design pattern and asynchronous messaging design pattern.

3.3.1 Problem

A part of the interested services is postponed to execute, while others are free to implement.

3.3.2 Context

The contexts can be various as the interactions strategies can be organized randomly. Take this context as an example, the client publishes a request message, multiple services receive the request and start to apply the business logic. At the same time, a set of other services are waiting in a queue for one service completion.

3.3.3 Solution

Based on the example context above, the pattern allows Service A and Service B to invoke by the external request simultaneously before getting a response, which is applying asynchronous messaging, and Service C and Service D are blocked while

Service A running in which the synchronous messaging works. A service can have more than one "parents" that make the service suspend.

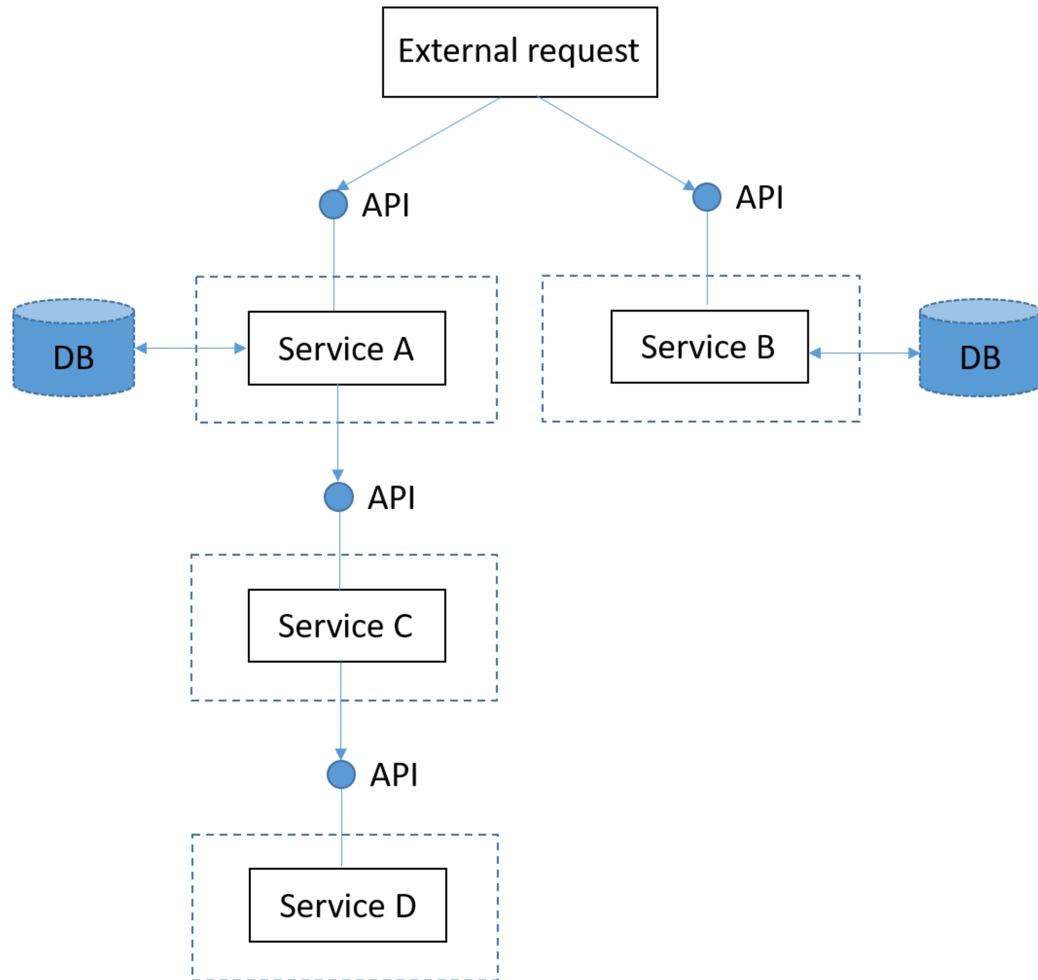

Figure 12 Hybrid messaging design pattern

3.3.4 Example implementation

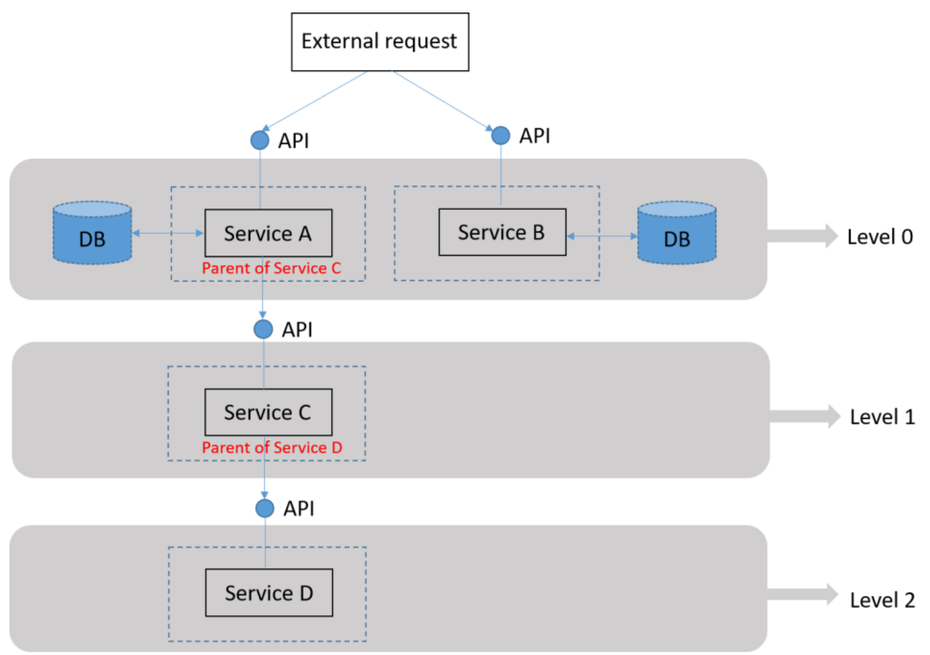

Figure 13 Adding “level” and “parent” attributes in the pattern

To ensure the services invoked in the correct order, we add the attributes “level” and “parent” in each service as shown in Figure 13.

```
/*******************************************************************
*** DESCRIPTION : <parse the services in a hybrid pattern >     ***
*** INPUT ARGS  : <request>                                     ***
*** OUTPUT ARGS : <None>                                        ***
*** IN/OUT ARGS : <None>                                        ***
*** RETURN      : <None>                                        ***
*******************************************************************/
public void hybrid(Element request){
      // step 1: grab the level list
      NodeList levelList =  request.getElementsByTagName("level");
      // step 2: parse all the levels
      for (int i = 0; i<levelList.getLength(); i++) {
            Node levelNode = levelList.item(i);
            if (levelNode.getNodeType() == Node.ELEMENT_NODE) {
                  Element level = (Element) levelNode;
                  // parse by level
                  parseByLevel(level);
            }
      }
}
```

Figure 14 Method “hybrid” in Java

```java
/*******************************************************************
*** DESCRIPTION : <parse the services in a level >              ***
*** INPUT ARGS  : <level>                                       ***
*** OUTPUT ARGS : <None>                                        ***
*** IN/OUT ARGS : <None>                                        ***
*** RETURN      : <None>                                        ***
*******************************************************************/
public void parseByLevel(Element level) throws Exception {
	// step 1: fetch all the services in the level
	NodeList serviceList = level.getChildNodes();
	List<String> services = new ArrayList<>();
	// step 2: parse all the services
	for (int i=0; i<serviceList.getLength(); i++) {
		Node n = serviceList.item(i);
		if (n.getNodeType() == Node.ELEMENT_NODE) {
			// fetch the parent nodes of a service
			if (n.getNodeName().equals("parent")) {
				services.add(n.getTextContent());
			// fetch the service itself
			}else {
				services.add(n.getTextContent());
				callByParent(services);
				services.clear();
			}
		}
	}
}
```

Figure 15 Method “parseByLevel” in Java

```
/*******************************************************************
*** DESCRIPTION : <call the service by checking its parents >    ***
*** INPUT ARGS  : < services >                                   ***
*** OUTPUT ARGS : <None>                                         ***
*** IN/OUT ARGS : <None>                                         ***
*** RETURN      : <None>                                         ***
*******************************************************************/
public void callByParent(List<String> services) throws Exception {
	// if the service has no parent
	if (services.size() == 1) {
		invoke(services.get(0));
	// if the service has parent(s)
	}else {
		int i;
		String lockFile;
		Path path;
		boolean exists;
		// check the parents' status
		for (i = 0; i<services.size()-1; i++) {
			// put name and path of the parents' lockFile
			lockFile = ".\\"+services.get(i)+".lock";
			path = FileSystems.getDefault().getPath
				(System.getProperty("user.dir"), lockFile);
			exists = Files.exists(path, new LinkOption[]
				   {LinkOption.NOFOLLOW_LINKS});
			while (exists) {
				Thread.sleep(1000);
				exists = Files.exists(path, new LinkOption[]
					   {LinkOption.NOFOLLOW_LINKS});
			};
		}
		// invoke the service
		invoke(services.get(i));
	}
}
```

Figure 16 Method “callByParent” in Java

3.3.5 Consequences

The hybrid messaging design pattern provides flexibility to enable different interaction styles to work together. It can be widely applied to a complicated system. The drawbacks are the same as those in the synchronous messaging design pattern: risk of cascading failure and long response time.

3.4 Guidelines

Below are the guidelines for applications following and applying the integration design patterns in microservices.

1. Establish the microservices with well-defined APIs.

Before integrating the services with the design patterns, the system must be broken down into independent services with well-defined APIs. Splitting up the monolithic system according to the business capabilities is the typical decomposition approach in microservices. API works as the communication protocol between the service and its consumers. The most prevalent two protocols are HTTP request-response APIs and lightweight messaging APIs [1].

2. Identify the design pattern applied to each event.

Various events are handled in an application. For each event, there are two steps to identify the design pattern. Step 1: To determine all the associated services for an event according to the functionality of each component. Step 2: To identify the inter-service communication mechanism among the associated services. If the request is blocked while waiting for a response then the communication is synchronous. If a request is free to call other services no matter receiving a response or not then asynchronous messaging is applied. If both of above situations happen during different phases then it is hybrid messaging.

3. Set the recipe configuration file.

The recipe configuration file is an event menu that aims to choreograph the services. It contains all the titles of the requests that may occur, the services with specific

performed orders, and the pattern that the event belongs to. The formats of the configuration file should be human readable and text-based such as XML and JSON. For example, Figure 17 shows the XML configuration file for the patterns in Figure 8, Figure 10 and Figure 12.

```xml
<?xml version="1.0" encoding="UTF-8"?>
      <RequestMenu>
            <request>
                  <title>figure_8_request</title>
                  <pattern>synchronous</pattern>
                  <microservice>service_A</microservice>
                  <microservice>service_B</microservice>
                  <microservice>service_C</microservice>
            </request>

            <request>
                  <title>figure_10_request</title>
                  <pattern>asynchronous</pattern>
                  <microservice>service_A</microservice>
                  <microservice>service_B</microservice>
                  <microservice>service_C</microservice>
            </request>

            <request>
                  <title>figure_12_request</title>
                  <pattern>hybrid</pattern>
                  <level id="0">
                              <microservice>service_A</microservice>
                              <microservice>service_B</microservice>
                  </level>
                  <level id="1">
                              <parent>service_A</parent>
                              <microservice>service_C</microservice>
                  </level>
                  <level id="2">
                              <parent>service_C</parent>
                              <microservice>service_D</microservice>
                  </level>
            </request>
      </RequestMenu>
```

Figure 17 Example of XML recipe configuration file

The syntax of the above XML recipe configuration file is specified in Table 1.

Table 1 The syntax description of the XML recipe configuration

Tag Name	Syntax Description
<request>	The event to handle by the application.
<title>	The name of an event.
<pattern>	The type of the design pattern applied to the event. It can be synchronous, asynchronous, or hybrid.
<microservice>	The service's name.
<level>	The execution order. It has an attribute "id" that starts with 0 and increments by one each time.
<parent>	The predecessor of a microservice.

4. Implement the design patterns to perform the request.

Before implementing the design patterns, there is a need to parse the recipe configuration file in the application. Figure 18 specifies the workflow of fulfilling a request.

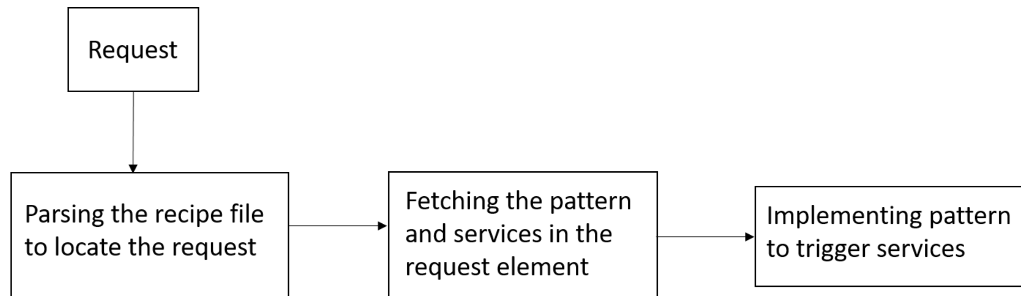

Figure 18 The workflow of fulfilling a request

Chapter 4 Case Study

This chapter demonstrates the process of applying the design patterns proposed in Chapter 3 to the EPIDEMIA system. The EPIDEMIA system is migrating to microservice architecture and facing the challenge of integrating the distributed services.

4.1 EPIDEMIA overview

The EPIDEMIA system consists of five subsystems, which are *public health interface, remote sensing data processing (EASTWeb), data integration, modeling, and reporting* subsystems. Each subsystem serves one single business capability. *Public health interface* is developed to update the epidemiological data; *EASTWeb* is implemented to update the environmental data; *data integration* is designed to integrate the epidemiological data and the environmental data; and *modeling* subsystem runs detection and warning models in R to forecast the malaria outbreak, *reporting* subsystem generates the PDF forecasting report. Therefore, every subsystem can be wrapped up as an independent microservice. Below are five autonomous services in the EPIDEMIA system:

1. New epidemiological data service

2. New environmental data service
3. Data integration service
4. Forecasting service
5. Report service

4.2 Microservices with Spring Boot

Spring Boot [24] evolves from Spring framework [25] and provides a convention-over-configuration solution for creating stand-alone, production-grade Spring based Applications that you can "just run" [24]. Rather than define a boilerplate configuration, a simple Spring Boot framework uses a file named “pom.xml” to import all the needed packages. Figure 19 demonstrates Spring Initializr (http://start.spring.io/) that offers a robust tool to bootstrap the Spring Boot project swiftly.

Figure 19 Screenshot of Spring Initializr

Due to the features of automatical configuration and directly dependencies embedding, Spring Boot is chosen to build the five services in the EPIDEMIA system.

4.3 Applying the design patterns

4.3.1 The design pattern for each request

According to the end users' requirements, there are five events to handle in the EPIDEMIA system.

1. The users upload the epidemiological data to the system.

The new epidemiological data are uploaded through the *public health interface*. Only the new epidemiological data service is called in responding the request. Since no thread is blocked when running the service, the asynchronous messaging design pattern can be applied to the request as shown in Figure 20.

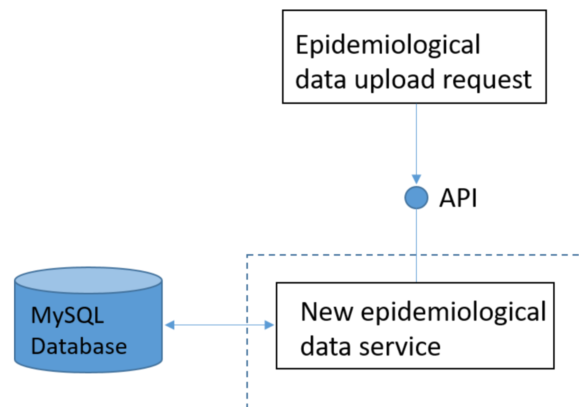

Figure 20 Service integration for epidemiological data upload

2. The users update the environmental data.

EASTWeb is a stand-alone application and is responsible for updating environmental data. The new environmental data service is the component that works for the request. Thus, the asynchronous messaging design pattern is utilized to the request (Figure 21).

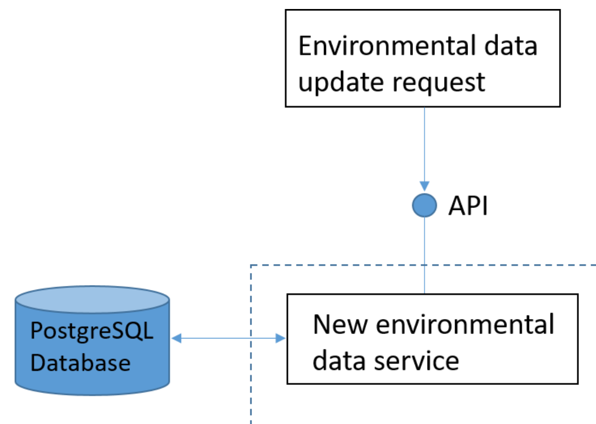

Figure 21 Service integration for environmental data update

3. The users update data in the system.

The request contains two tasks, epidemiological data upload and environmental data update. Both of the new epidemiological data service and the new environmental data service are needed to add outputs to the request, and each service has its own database. Asynchronous messaging is the right option here as shown in Figure 22 because the request is not blocked while waiting for the response from a service.

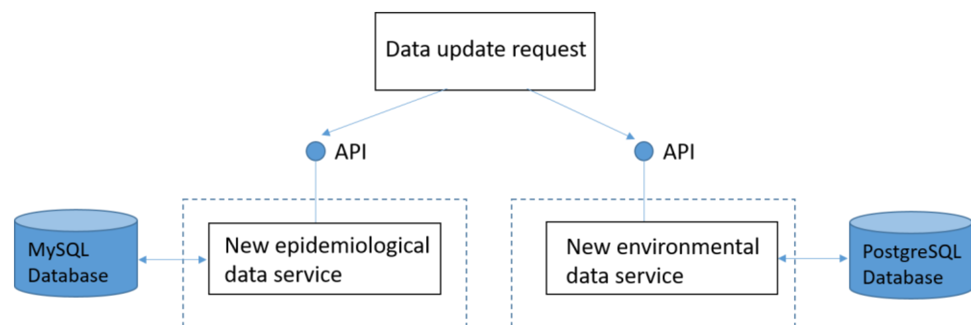

Figure 22 Service integration for data update

4. The users integrate the new environmental data with epidemiological data.

The request has a desire to activate the new environmental data service and the data integration service. The new environmental data service is invoked to update the

environmental data first. The request is held until receiving the response from the service. Then, the request enables to run the data integration service. The synchronous messaging design pattern is applied to the request as shown in Figure 23.

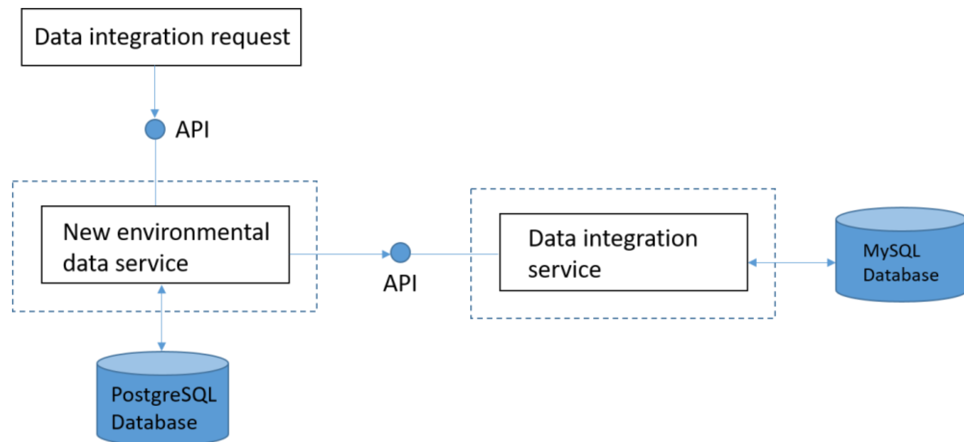

Figure 23 Service integration for data integration

5. The users generate the forecasting report.

All of the services are required to generate the forecasting report. Firstly, the request is sent to the new epidemiological data service and the new environmental data service via asynchronous messaging. Secondly, after the two services give responses to the client, the data integration service is executing in which the client will be blocked. Thirdly, the forecasting service is running after data integration service completes. At last, the report service generates the PDF report after forecasting service is over. Both of the synchronous and asynchronous messaging are utilized, so the hybrid design pattern is suitable in this case (Figure 24).

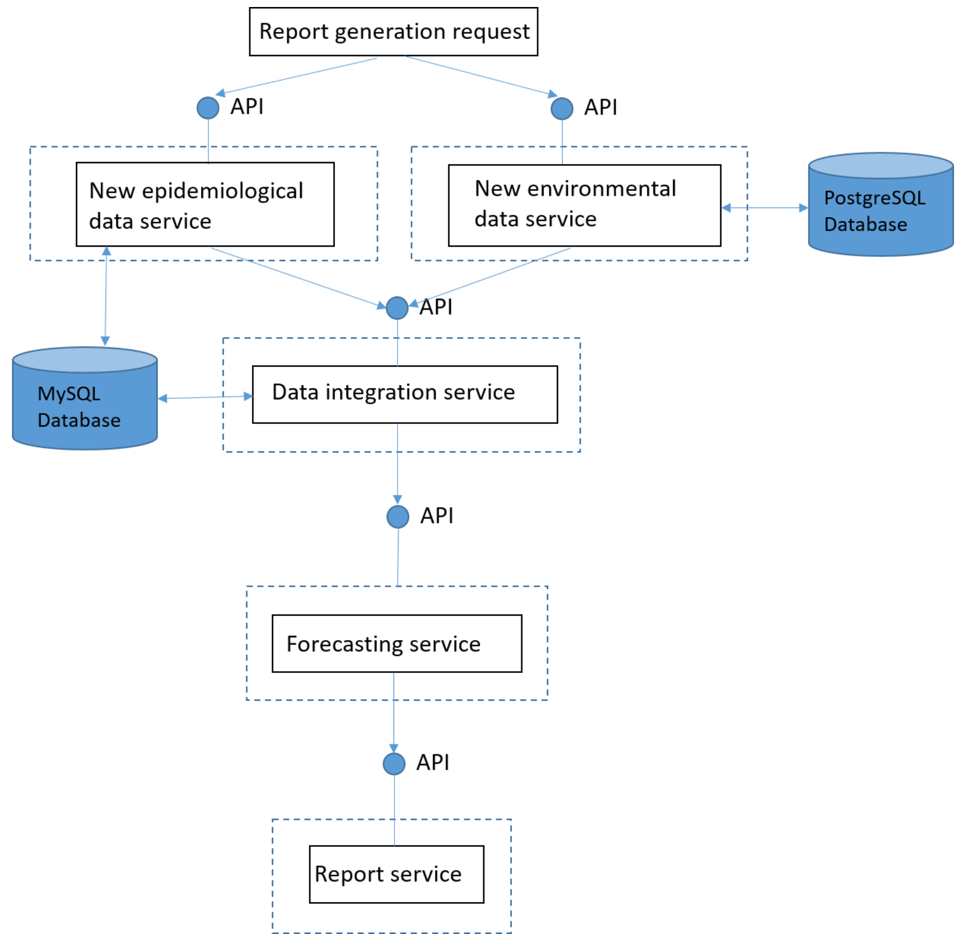

Figure 24 Service integration for report generation

4.3.2 The recipe file settings

```
<?xml version="1.0" encoding="UTF-8"?>
      <RequestMenu>
            <request>
                  <Title>epidata_update</Title>
                  <pattern>asynchronous</pattern>
                  <microservice>epidata</microservice>
            </request>

            <request>
                  <Title>envdata_update</Title>
                  <pattern>asynchronous</pattern>
                  <microservice>eastweb</microservice>
            </request>

            <request>
                  <Title>data_update</Title>
                  <pattern>asynchronous</pattern>
                  <microservice>epidata</microservice>
                  <microservice>eastweb</microservice>
            </request>

            <request>
                  <Title>data_integration</Title>
                  <pattern>synchronous</pattern>
                  <microservice>eastweb</microservice>
                  <microservice>dataInt</microservice>
            </request>

            <request>
                  <Title>report_generation</Title>
                  <pattern>hybrid</pattern>
                  <level id="0">
                        <microservice>eastweb</microservice>
                        <microservice>epidata</microservice>
                  </level>
                  <level id="1">
                        <parent>eastweb</parent>
                        <parent>epidata</parent>
                        <microservice>dataInt</microservice>
                  </level>
                  <level id="2">
                        <parent>dataInt</parent>
                        <microservice>forecast</microservice>
                  </level>
                  <level id="3">
                        <parent>forecast</parent>
                        <microservice>report</microservice>
                  </level>
            </request>
      </RequestMenu>
```

Figure 25 XML recipe file for EPIDEMIA system

The recipe file presented in Figure 25 configures the types of design patterns and the services with defined execution order for all the events in EPIDEMIA system. It is written base on the integration solutions displayed in Figure 20, 21, 22, 23, and 24.

```
/**************************************************************
*** DESCRIPTION : <parse the recipe to locate the request >
*** INPUT ARGS  : <requestName>
*** OUTPUT ARGS : <None>
*** IN/OUT ARGS : <None>
*** RETURN      : <String>
**************************************************************/
public String locateRequest(String requestName) {
      try {
            // give the recipe file name
            File recipeFile = new File("recipe.xml");

            DocumentBuilderFactory dbFactory =
                        DocumentBuilderFactory.newInstance();
            DocumentBuilder dBuilder =
                        dbFactory.newDocumentBuilder();
            // parse the xml file
            Document doc = dBuilder.parse(recipeFile);
            doc.getDocumentElement().normalize();

            // retrieve the request element
            NodeList requestList =
                  doc.getElementsByTagName("request");

            for (int i = 0; i < requestList.getLength(); i++) {

                Node nNode = requestList.item(i);
                if (nNode.getNodeType() == Node.ELEMENT_NODE) {
                    Element request = (Element) nNode;
                   //fetch the title from a request
                    String title = request.getElementsByTagName
                        ("Title").item(0).getTextContent();
      // locate the title that is in line with request name
                    if (title.equals(requestName)) {
                        sendRequest(request);
                        break;
                    }
                }
            }
            return "Your request is processed.";
      } catch (Exception e) {
            e.printStackTrace();
            return "Error";
      }
}
```

Figure 26 Method “locateRequest” in Java

4.3.3 The design pattern implementation

Before implementing the design pattern, it is necessary to parse the recipe configuration file as stated in Figure 18. First, the recipe configuration file is read to detect the title that is line with the request name, the example method is given in Figure 26. Next, the patterns and services in the request element are fetched, the example method "fetch" is described in Figure 27. At last, the design pattern implementation is called in the "fetch" method.

```
/************************************************************
*** DESCRIPTION : <fetch patterns, services from the request***
***                and implement the design pattern>        ***
*** INPUT ARGS   : <requestName>
*** OUTPUT ARGS  : <None>
*** IN/OUT ARGS  : <None>
*** RETURN       : <None>
************************************************************/
public void fetch(Element request) throws Exception {
      String pattern = request.getElementsByTagName("pattern")
                                  .item(0).getTextContent();
      if (pattern.equals("synchronous")) {
        NodeList msList =  request.getElementsByTagName
                                        ("microservice");
        List<String> services = new ArrayList<>();
        for (int i = 0; i<msList.getLength(); i++) {
            String service = msList.item(i).getTextContent();
            services.add(service);          }
        // implement the synchronous messaging design pattern
        callWithBlock(services);
    }else if(pattern.equals("asynchronous")) {
        NodeList msList =  request.getElementsByTagName
                                        ("microservice");
            List<String> services = new ArrayList<>();
            for (int i = 0; i<msList.getLength(); i++) {
            String service = msList.item(i).getTextContent();
            services.add(service);        }
        // implement the asynchronous messaging design pattern
        callWithoutBlock(services);
     }else if(pattern.equals("hybrid")) {
        // implement the hybrid messaging design pattern
        hybrid(request);
    }
}
```

Figure 27 Method "fetch" in Java

4.3.4 Deployment in Amazon EC2

Amazon EC2 is adopted to deploy the EPIDEMIA system in microservices. Amazon EC2 provides sorts of instance types with varying combinations of CPU, memory, storage and networking capacity [26]. For example, M5 instances are the latest generation of General Purpose Instances [26]. The M5 instance is utilized to set up the EPIDEMIA system.

To run EASTWeb application, the hardware requires at least four cores available in CPU, up to 0.5GB memory, and at least 100GB of free space. Based on the requirements, the instance, m5.xlarge offering 4 vCPU, 16GB of memory, EBS instance memory and up to 3500 Mbps network bandwidth is launched to deploy the EPIDEMIA system. Additionally, 200GB of EBS volume is created and attached to the instance to make the EPIDEMIA system work well. Figure 28 depicts the EPIDEMIA system in microservices architecture on AWS EC2.

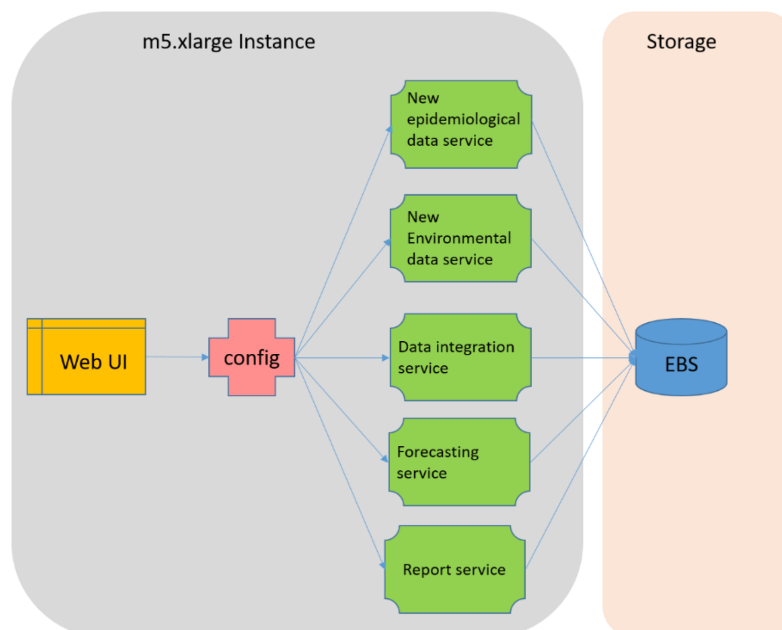

Figure 28 EPIDEMIA system in microservices on Amazon EC2

Chapter 5 Evaluation

This chapter evaluates the service integration design patterns. The three general design patterns are proposed to promote the process of service composition. The chapter shows the evaluation of the design patterns by applying them in the case study. In addition, the comparison of our work with others' work is illustrated at the end of the chapter.

5.1 Addressing the challenge of services integration

To verify the feasibility of the proposed design patterns, there is a need to apply the patterns to a case study. The goal of the thesis is to facilitate services composition in microservices. The EPIDEMIA system is built upon a collection of independent subsystems and still requires manual operations on executing the subsystems due to the lack of integration. Therefore, the EPIDEMIA system is suitable to apply the service integration design patterns.

The five subsystems of EPIDEMIA were developed separately in fulfilling specific objectives in different stages over the past ten years. Before being incorporated into EPIDEMIA, each subsystem was independently developed and operated. For example, the *EASTWeb* subsystem was originally developed in 2013 and then upgraded in 2015 for automatically downloading earth observation datasets, processing them and generating statistical summaries for a given area and time period [8]. Moreover, as a standalone application since it was developed, *EASTWeb* has its own user interfaces and performs independently. The other four subsystems, *public health interface*, *data integration*, *modeling*, and *reporting* are in similar situations. The data transferring and interactions among the subsystems still require the user's interference. Therefore, how to

successfully integrate the loose collection of the subsystems is a big challenge in EPIDEMIA project.

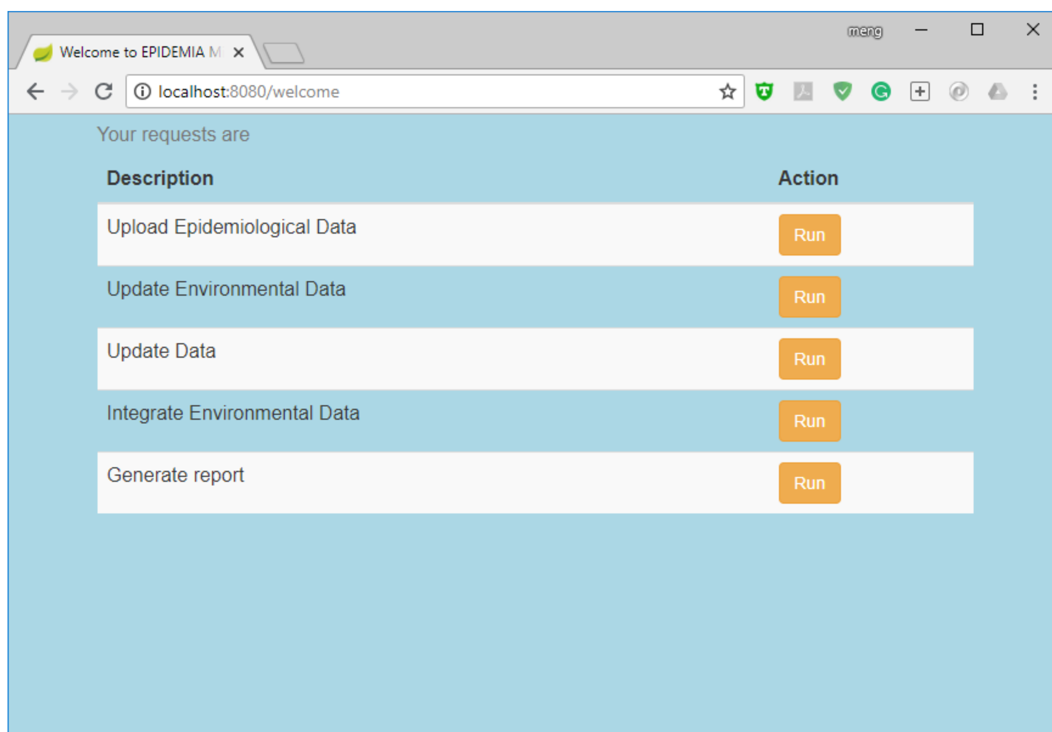

Figure 29 The user interface of EPIDEMIA system in microservice

In order to address the challenge, we wrap up the five subsystems as five services at first so that the subsystems are able to talk with each other through the APIs. The services are composed together by applying corresponding design patterns according to the inter-service messaging styles required in different events. For instance, all of the five services are integrated together via the hybrid messaging design pattern to fulfill the report generation request (Figure 24). To send a request, the users only need to click the button “Run” on the user interface as shown in Figure 29. After all the services complete its implementation successfully, the client will receive the response like “Your request is processed” in Figure 30. Otherwise, the error message will be displayed. As a result, the

challenge of the loose services integration in EPIDEMIA system has been addressed through the design patterns.

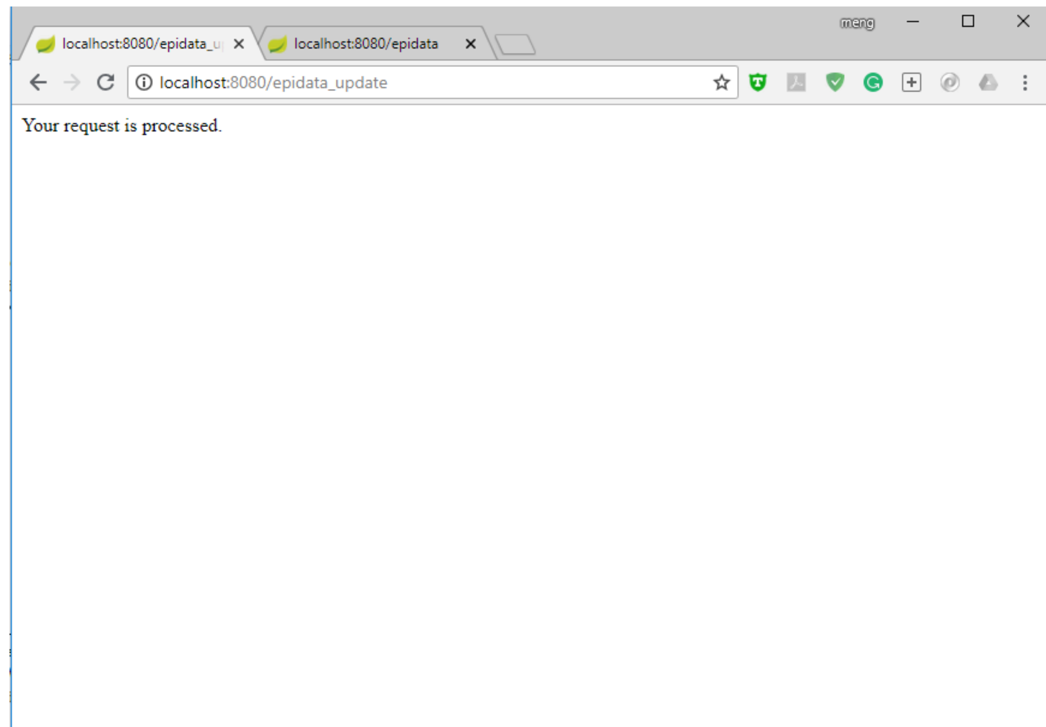

Figure 30 The response after running request “Upload Epidemiological Data”

5.2 Simple and flexible patterns

The service integration design patterns are considered simple and flexible to apply in applications.

- Simplicity

The more informative an approach is, the simpler it is applied. To make users easier to select an appropriate design pattern, each pattern is described with a particular problem and context. Also, the example diagrams are given in the approaches to make clients understand the patterns better. The example implementation of each approach is

presented to make users apply the approach simply. The consequences let users gain further insight into the pros and cons of each design pattern.

- Flexibility

The approaches can be applied to different applications with multiple requests. There is no limitation to apply the design patterns to integrate services as long as the applications are built upon microservices architecture. Additionally, the three design patterns can handle different events in an application because the patterns are clarified with basic inter-service messaging mechanisms that are applied commonly. Like the EPIDEMIA system, all of the events find a suitable solution in the approaches.

5.3 General guidelines

In Chapter 3, the guidelines for applying the design patterns are provided. The guidelines are general for systems facing the issue of service integration to follow. On one hand, the guidelines are composed of a steps list acting like an assembly line and each step contains a concise instruction. By following each step in the list, an application can achieve the goal of successful service integration. On the other hand, multiple styles of APIs are supported as long as the services can interact with the APIs, and varying formats of the recipe configuration file are acceptable if they are easy to understand and parse in the system.

5.4 Easily modifying the relationships

The microservices architecture is comprised of many independent units. It is common that a new unit is plugged in or a unit is deleted if there is a need. When the services are updated, the relationships among the services may be modified which could

be difficult as developers need to edit and test the source codes in the system. However, the recipe configuration file can help relieve the heavy lift of modifying relationships. The structure of the services can be re-built easily through editing the <request> element in the recipe configuration file. For instance, if a new service, prevention data service, is added to the EPIDEMIA system. The request of updating data is required to invoke the new service via asynchronous messaging. The changes can be quickly made by adding a <microservice> tag with the new service's name in the data_update request (Figure 25). Thus, the recipe configuration file is a helpful tool to modify the relationships easily.

5.5 Related work

5.5.1 Design patterns for microservices on Microsoft Azure [15]

Nine new design patterns are posted on Microsoft Azure to help relieve the challenges brought by microservices architecture. Each approach provides a solution towards a minor detailed integration problem when building an application. To entirely establish a system in microservices, it is necessary to utilize most of these patterns together (Figure 7). Different from the nine refined design patterns, the thesis proposes three general design patterns to integrate services fully. The communication structure among the services can be built with a single design pattern when the messaging mechanism criteria are met.

5.5.2 Microservice design patterns [22]

The author, Arun Gupta presents six microservice design patterns to compose services. These patterns provide multiple integration approaches from varying perspectives. Unlike the design patterns in our research, there are no clear clarified criteria among these patterns. The proxy microservice design pattern is a variation of the

aggregator pattern, the branch microservice design pattern extends the aggregator design pattern, the shared data microservice design pattern develops branch design pattern [22]. Moreover, the author briefly introduces the patterns without comprehensive descriptions. The design patterns proposed in the thesis have clear and simple clarified criteria: inter-service messaging mechanisms, so they are easy to distinguish and follow. Besides, each pattern is elaborated with a specific problem, context, solution, example implementation, and consequences.

Chapter 6 Conclusion

The service integration design patterns illustrated in the thesis can solve the challenges of microservices integration. This chapter summarizes the study and discusses the possible future work.

6.1 Conclusion

The service integration design patterns aim to help compose services during the process of building applications in microservices. By categorizing with the inter-service interaction mechanisms, three design patterns are proposed in this study. The benefits of the design patterns are summarized below.

1. The design patterns are helpful tools for applications facing trivial services integration issues like the EPIDEMIA system. After applying the approaches to the EPIDEMIA, all of its subsystems are entirely integrated to implement multiple requests.
2. Each design pattern is described with a corresponding problem, context, solution, example implementation, and consequences to make the approach informative.
3. The approaches are flexible to apply to different applications as long as they are established on microservices architecture.
4. The listed descriptive steps in the guidelines make the approaches general and easy to follow.
5. The recipe configuration file is helpful to modify the microservices structure.

6.2 Future Work

This thesis mainly focuses on service integration in microservices. Future work would explore more service integration mechanisms and strategies of choosing the service deployment platform. The following directions could be studied:

1. To consider more clarified criteria and develop more design patterns to integrate services. The study develops three design patterns according to the inter-service interaction styles. It could be interesting to try different integration methods based on new criteria. It could also be meaningful to develop new approaches to handle complicated events in which the services are invoked in loops or recursions.
2. To develop approaches of selecting an appropriate microservices deployment platform. In our case study, we choose to adopt Amazon EC2 as our deployment environment. There are various platforms ranging from IaaS model to PaaS model. We could test more platforms and summarize the experiences to choose the right deployment platform to reduce the complexity of microservices deployment.

References

[1] James Lewis, and Martin Fowler, “Microservices: a definition of this new architectural term”, https://martinfowler.com/articles/microservices.html, March 2014.

[2] “Gartner Says Hyperconverged Integrated Systems Will Be Mainstream in Five Years”, https://www.gartner.com/newsroom/id/3308017, May 2016.

[3] Enterprise service bus. https://en.wikipedia.org/wiki/Enterprise_service_bus

[4] Cerny, Tomas, Michael J. Donahoo, and Jiri Pechanec. "Disambiguation and comparison of soa, microservices and self-contained systems." In Proceedings of the International Conference on Research in Adaptive and Convergent Systems, pp. 228-235. ACM, 2017.

[5] C. L. Merkord, Y. Liu, A. Mihretie, T. Gebrehiwot, W. Awoke, E. Bayabil, G. M. Henebry, G. T. Kassa, M. Lake and M. C. Wimberly. 2017. Integrating malaria surveillance with climate data for outbreak detection and forecasting: the EPIDEMIA system. Malaria journal 16, no. 1 (2017): 89. DOI: 10.1186/s12936-017-1735-x.

[6] Environmental monitoring incorporating disease and environmental monitoring for integrated assessment (EPIDEMIA) Project Website. https://epidemia.sdstate.edu/.

[7] Y. Liu, M. D. Devos, M. Abdul-Rahim, J. Hu, and M. C. Wimberly. 2016. EASTWeb framework - a plug-in framework for constructing geospatial health applications. In 2016 IEEE International Conference on Electro-Information Technology (EIT).

[8] Y. Liu, J. Hu, I. Snell-Feikema, M. S. VanBemmel, A. Lamsal, M. C. Wimberly. 2015. Software to Facilitate Remote Sensing Data Access for Disease Early Warning Systems. Environmental Modeling and Software. Vol. 74, p. 247-257.

[9] J. S. Racine. RStudio: A Platform-Independent IDE for R and Sweave. *Journal of Applied Econometrics* 27, no. 1 (2012): 167-172.

[10] Y. Xie. Dynamic Documents with R and knitr. Vol. 29. CRC Press, 2015.

[11] Mell, Peter, and Tim Grance. "The NIST definition of cloud computing." (2011).

[12] Shao, Liangshan, Jinguang Sun, and Xiaowei Hui, eds. Fuzzy Systems, Knowledge Discovery and Natural Computation Symposium: FSKDNC 2013. DEStech Publications, Inc, 2013.

[13] David Clinton. “AWS just announced a move from Xen towards KVM. So what is KVM?” https://medium.com/@dbclin/aws-just-announced-a-move-from-xen-towards-kvm-so-what-is-kvm-2091f123991

[14] “Amazon EC2”, https://aws.amazon.com/ec2/?nc2=h_m1

[15] Mike Wasson Lead Content Developer, AzureCAT patterns & practices. “Design patterns for microservices”, https://azure.microsoft.com/en-us/blog/design-patterns-for-microservices/

[16] Masashi Narumoto, Mike Wasson. “Ambassador pattern”, https://docs.microsoft.com/en-us/azure/architecture/patterns/ambassador

[17] Masashi Narumoto, Nate Loftsgard, Mike Wasson. “Anti-Corruption Layer pattern”, https://docs.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer

[18] Masashi Narumoto, Sam Ferree, Mike Wasson. “Bulkhead pattern”, https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead

[19] Masashi Narumoto, Mike Wasson. “Gateway Aggregation pattern”, https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-aggregation

[20] Masashi Narumoto, Mike Wasson. “Gateway Offloading pattern”, https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-offloading

[21] Masashi Narumoto, Mike Wasson. “Sidecar pattern”, https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar

[22] Arun Gupta. “Microservice Design Patterns.” http://blog.arungupta.me/microservice-design-patterns/

[23] Chris Richardson, and Floyd Smith. “Microservices from design to deployment.” © NGINX, Inc. 2016.

[24] “Spring Boot”. http://spring.io/projects/spring-boot

[25] “Spring”. https://spring.io/

[26] “Amazon EC2 Instance Types”. https://aws.amazon.com/ec2/instance-types/