國立臺灣大學電機資訊學院資訊工程學系
碩士論文

Department of Computer Science & Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis

從單體服務到微服務：基於解構耦合的方法
From Monolithic to Microservice: A Dependency Decoupling Approach

陳力聖
Li-Sheng Chen

指導教授：李允中 博士
Advisor: Jonathan Lee, Ph.D.

中華民國 112 年 7 月
July, 2023

doi:10.6342/NTU202303287

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

從單體服務到微服務：基於解構耦合的方法

From Monolithic to Microservices: A Dependency Decoupling Approach

本論文係陳力聖君（學號 R10922120）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 112 年 7 月 27 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Computer Science and Information Engineering on 27 July 2023 have examined a Master's thesis entitled above presented by CHEN, LI-SHENG (student ID: R10922120) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

李允中	[illegible]	鄭有進
（指導教授 Advisor）		
徐國勛	陳錫民	

系主任/所長 Director: 洪士灝

誌謝

首先，我要感謝我的指導教授李允中教授，這兩年以來的教導我許多知識以及做研究的方法：尋找研究的主題、相關研究的探討、問題的發掘、以及解決問題的方法。再來，我要感謝臺灣大學軟體工程實驗室的同屆成員，林怡伶、林辰臻、許恆、梁峻瑞、黎光晏、張馨尹、劉仁軒，以及學弟妹、助理等，在一些共同的專案中一同努力、互相合作。由於許多人的幫助，讓我得以完成此篇論文。

doi:10.6342/NTU202303287

摘要

近年來，微服務架構漸趨流行，其有效改善大型系統的可擴展性和可維護性之特性，使得許多企業與機關將其資訊系統從單體式架構重構成微服務架構。

然而，如何將單體式架構進行切分才能獲得最好的切割結果爭論已久，由於其牽扯的層面廣且複雜因此目前尚未有定論，究其根本原因，皆是沒有清楚定義微服務。因此，本篇論文對微服務的架構進行了定義，根據此定義我們提出以解構耦合的方式，藉由分析開源程式中元素的依賴相關性對單體式架構進行切割，自動化地產出微服務。

另外我們也依據此架構產出了描述微服務的文件，藉由此文件我們將使用者的需求來做到微服務的需求匹配。

關鍵詞 — 服務元件、網路服務、網路應用程式重組、微服務、自動化程式碼生成、圖匹配

doi:10.6342/NTU202303287

Abstracts

In recent years, microservice architecture has gained popularity due to its ability to significantly improve the extensibility and maintainability of large-scale systems. This has led many enterprises and departments to refactor their information systems from monolithic architecture to microservice architecture.

However, achieving the best decomposition of monolithic architecture remains a topic of debate, as it involves various complex aspects. One of the root causes is the lack of a clear definition of Microservice and its internal structure. In this research, we aim to address this gap by defining the Microservice structure. Building upon this definition, we propose a decoupling methodology to separate the monolithic architecture into microservices based on the dependencies between program elements in open-source projects, and we automate the process of generating the microservices.

Furthermore, we introduce the Microservice description based on the proposed architecture. This description allows us to perform matchmaking between the microservices and the user's requirements, facilitating a more efficient and precise service integration process.

doi:10.6342/NTU202303287

Index terms — Service Component, Web Service, Repack Web Application, Microservice, Automatic code generation, Graph-based Matchmaking

doi:10.6342/NTU202303287

Contents

doi:10.6342/NTU202303287

doi:10.6342/NTU202303287

doi:10.6342/NTU202303287

doi:10.6342/NTU202303287

List of Figures

doi:10.6342/NTU202303287

doi:10.6342/NTU202303287

doi:10.6342/NTU202303287

List of Tables

doi:10.6342/NTU202303287

Chapter 1

Introduction

SOA (Service Oriented Architecture) is a commonly used software development pattern by enterprises, enabling the integration of service components through the reuse of services and loose coupling between them, achieved by unified interfaces and common communication standards. Microservice Architecture, as an extension of SOA, places greater emphasis on size and independence. In this research, we design a practical, automatic process for generating microservices. We start with open-source Java projects on Github and transform them into service components, considering each Java method. We then repack these service components into microservices based on their dependencies, ensuring the resulting microservices exhibit high cohesion and modularity.

In previous research [23], Huang utilized the Java Reflection API [5] to execute the generated service components by loading them into the Java ClassLoader. However, in many cases of open-source projects, they utilize software frameworks such

doi:10.6342/NTU202303287

as the Spring framework [13], making them incompatible with Huang's proposed process. To address this issue, Yu [32] proposed a new service component generation process that maintains compatibility with the Spring framework during the execution of service components. This was achieved by transforming the controller methods used in the Spring framework into service components and combining them with the frontend design process proposed by Hsieh [22], which stores the service binding information in the database for use by the frontend design system. However, it should be noted that this process has limitations and may not be applicable to projects written exclusively with the Spring Framework.

Therefore, in the research conducted by Shih [31] and Lu [29], they repack the service components into a microservice and utilize Spring Cloud for the deployment of microservice execution. However, their microservice generation process is associated with two implied issues:

1. In Shih's research, there is no clear definition provided for microservices, and the decision of which APIs belong to the same microservice is based solely on whether they are in the same class in the original project. This approach to decomposition may result in a reduction of cohesion within the generated microservices.

2. In Lu's research, there exist some high coupling repack logic in its repack process, which may result in the failure when it try to parse the project written with other framework.

To address the first issue, we propose a definition of the microservice. According

doi:10.6342/NTU202303287

to this definition, we extract the which APIs are suitable to be placed in the same microservice with the degree of the cohesion between each of them.

To address the second issue, we have made improvements by adopting a pure object-oriented approach in the repack process. We now handle plugins and dependencies individually using handlers, which can be easily added using the Chain of Responsibility pattern. With these enhancements, our aim is to achieve the decomposition of all Java projects into microservices successfully.

In the matchmaking process, previous research transformed the matchmaking request generated from the frontend into WSDL to perform method-level matchmaking with the service components. However, due to the large number of service components, this mechanism led to low matchmaking efficiency. To address this issue, we have made improvements to our matchmaking process:

1. Transform the requirement to the description of a microservice, perform the microservice-level matchmaking to filter out the candidate microservices

2. Using the result of the first step as the search scope, we gather the APIs belonging to the candidate microservices to create the matchmaking set for the method-level matchmaking process.

Due to this improvement, we can significantly decrease the input size of the matchmaking algorithm, leading to a notable enhancement in the efficiency of searching for appropriate services for frontend components.

This paper is organized as follows: Chapter 2 introduces the related work; Chapter 3 introduce a clear definition of microservices. Chapter 4 describe the genera-

doi:10.6342/NTU202303287

tion process of service components; Chapter 5 the process of composing service components to microservices. Chapter 6 explains the generation of microservice descriptions and the subsequent matchmaking process. Chapter 7 and Chapter 8 respectively summarizes the contribution of this work and the future work.

doi:10.6342/NTU202303287

Chapter 2

Related Work

In the domain of microservice research, notable findings have emerged. In this chapter, we will provide a comprehensive list of typical definitions and extraction methodologies.

2.1 Related Work

Nicola et al. [19] defines Microservices as *"cohesive, independent processes interacting via messages"* and Microservice Architecture as *"a distributed application where all its modules are microservices."*. They also highlight the differences between microservices and SOA, which include: 1. Bounded Context （first introduced by Domain-driven design [20]） 2. size 3. independency, numerous studies are built upon the extension of this concept; however, they do not specifically define the internal structure of microservices.

doi:10.6342/NTU202303287

Baresi et al. [17] utilized the Open API Specification [9] for microservice extraction. They conducted matchmakig between operation name defined in the specification and the concept defined in Schema.org [10]. The similarity between word segments was determined using the co-occurrence matrix provided by DISCO [25]. After computing the similarity matrix between the word segments, they applied the Hungarian algorithm to find the best match between the operation name and the concept. Finally, they used the type hierarchy to determine whether the APIs should be placed in the same microservice.

Chen et al. [18] utilized business requirements and data flow as input in their methodology, which effectively clusters APIs with similar output data or those handling the same type of data as the microservice. Gysel et al. [21] proposed a service decomposition method based on 16 coupling criteria extracted from literature and industry experience. This approach employs clustering algorithms to decompose an undirected, weighted graph transformed from SSA (Software System Artifacts, such as ER models, Use Case Specifications), and others.

The above-mentioned studies did not take into consideration the actual structure of the source code; instead, they relied solely on specifications and documents as the basis for analysis. This approach may result in microservices with low internal cohesion in their structure.

doi:10.6342/NTU202303287

2.2 Background Work

In our research, we decouple the open-source projects generate service components. We then extracted dependencies between these components and repackaged them into microservices. Additionally, we generated descriptions for the resulting microservices and deployed them to Spring Cloud. Throughout the entire process, we made use of various open-source tools and algorithms to facilitate our research. In the following sections, we will introduce these tools and algorithms one by one.

2.2.1 JavaParser

JavaParser [6] is an open-source project designed to analyze Java code. It has the capability to compile Java code into an abstract syntax tree (AST) and provides various APIs that allow users to operate on the AST.

In our research, both the Service Composition (see Chapter5)and Service Decomposition (see Chapter4) utilize JavaParser to extract relevant information from the code.

2.2.2 Soot

Soot is a static analysis tool for Java. In Service Decomposition (see Chapter4), we extract dependencies between program elements by analyzing the Jimple IR (Intermediate Representation) compiled by Soot.

doi:10.6342/NTU202303287

2.2.3 Reflection API

Java Reflection API [5] is a built-in tool in Java that allows loading Java classes using ClassLoader and examining the runtime behavior of program elements.

In Service Decomposition (see Chapter4), we utilize the Reflection API to obtain runtime information from open-source code.

2.2.4 JDK Compiler module

The JDK Compiler module includes Java Compiler API [2] and Java Compiler Tree API [3]. The Java Compiler API allows users to compile Java programs with code, while the Java Compiler Tree API offers the abstract syntax tree (AST) representation of the open-source code, from which we can extract Java classes. By iterating through the AST, we can access information that Java Reflection API cannot provide, such as method bodies.

In Service Decomposition (see Chapter4), we leverage the JDK Compiler module to analyze the AST of the open-source code.

2.2.5 Spring Cloud

Spring Cloud [11] is an open-source project that facilitates the construction of the microservice architecture and provides various implementations of common microservice patterns, including the API Gateway [12].

In Service Composition (see Chapter5), we utilize Spring Cloud to deploy the generated microservices and API Gateway.

doi:10.6342/NTU202303287

2.2.6 Hungarian Algorithm

Hungarian Algorithm [26] is a combinatorial optimization algorithm proposed by Harold William Kuhn in 1955. It efficiently solves the assignment problem in polynomial time by finding the maximum weighted match (or minimum weighted match) in a bipartite graph.

In Microservice Matchmaking (see Chapter6), we utilize the Hungarian Algorithm to compute the distance between microservices.

2.2.7 WSDL

WSDL (Web Service Description Language) [14] is a standard recommendation from W3C. It is based on XML and provides detailed descriptions of message formats and binding protocols used to invoke web services.

In Service Decomposition (see Chapter4), we utilize WSDL to describe the generated service components.

2.2.8 HDBSCAN

HDBSCAN（Hierarchical Density-Based Spatial Clustering of Applications with Noise） [30] is a density-based clustering algorithm. Unlike other density-based clustering algorithms that use an epsilon parameter to define the scope, HDBSCAN employs a hierarchy to replace epsilon, requiring only one parameter (minimum cluster size). This allows it to discover clusters with varying densities.

In Service Composition (see Chapter5), we use HDBSCAN to cluster APIs.

doi:10.6342/NTU202303287

Chapter 3

Microservice Definition

The microservices architecture is the decomposition of a system into a set of services, such that all services will have **minimal public interfaces**.

The threshold upon which the system can be decomposed is defined by the use cases of the system that the microservices are a part of.

This definition supports the known fact that each microservice should have **its own database**. That’s because in other the case, one of the services would have to expose its database as its public interface. And this huge public interface would make it a macro-service.

In our research, we have defined the internal structure of microservices. Building upon these definitions, we propose the description of microservices and an API clustering mechanism.

doi:10.6342/NTU202303287

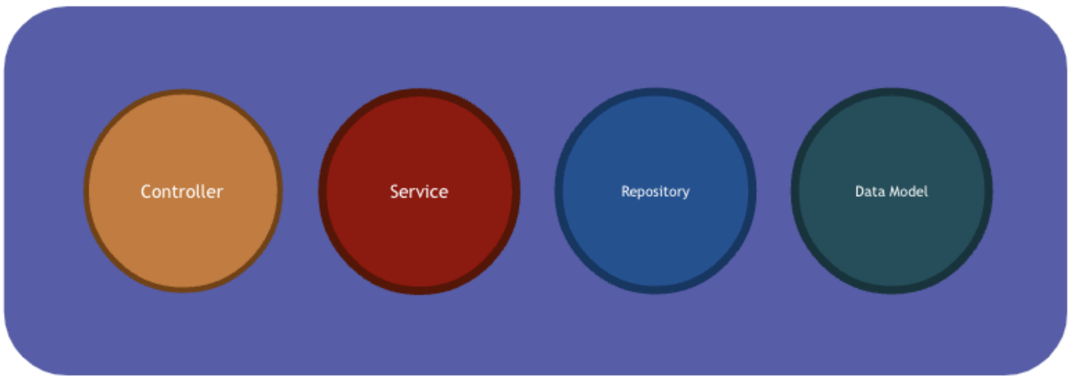


Figure 3.1: Microservice Structure

3.1 Microservice Structure

We define Microservice can be composed by four elements：

1. Controller: public interfaces
2. Service: business logic.
3. Repository: persistence logic.
4. Entity: database storage object.

Based on this structure, we can describe a microservice by combining the description of each part.

doi:10.6342/NTU202303287

3.2 APIs in a Microservice

We assume that APIs within the same microservice should exhibit high cohesion, implying the existence of dependencies between APIs. These dependencies can be seen as the following relationships between APIs:

1. APIs have the execution order
2. APIs have the invocation relationship
3. Composite API

Based on this definition, we can conduct cohesion analysis on the APIs in the project and cluster them based on the computed values. Each cluster formed would represent a microservice.

3.3 Example of a Microservice

Taking a user service Microservice as an example (see Figure 3.2), as shown in the diagram , this Microservice provides two API services (controllers), namely ”login” and ”register”. Within these two APIs, there are various business logics, such as checking user existence and generating login credentials. Additionally, to store and modify information, the Microservice must have an instance of the database to store data and access the database with persistent layer logic. Thus, the Microservice exhibits the four essential elements: controller, service, repository, and entity.

doi:10.6342/NTU202303287

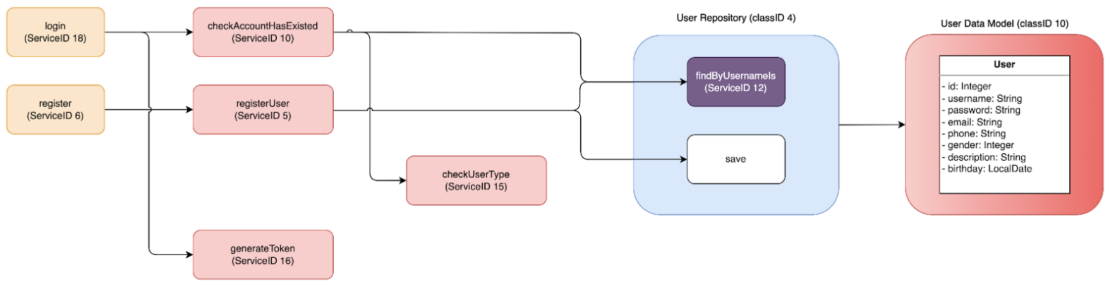

login (ServiceID 18)
register (ServiceID 6)
checkAccountHasExisted (ServiceID 10)
registerUser (ServiceID 5)
checkUserType (ServiceID 15)
generateToken (ServiceID 16)
User Repository (classID 4)
findByUsernameIs (ServiceID 12)
save
User Data Model (classID 10)
User
- id: Integer
- username: String
- password: String
- email: String
- phone: String
- gender: Integer
- description: String
- birthday: LocalDate

Figure 3.2: Microservice Example

doi:10.6342/NTU202303287

Chapter 4

Service Decomposition

Service Decomposition is the process of decomposing the Java program and generating service components with Java methods as units. A service component includes:

- **Data Model**: store the information of program elements
- **Dependency Model**: store the dependencies between program elements
- **Web Service Description Language**: used to describe the service component

The generation process of a service component includes following steps:

1. Service Decomposition
   (a) Source Code Parsing
   (b) Dependency Extraction

doi:10.6342/NTU202303287

   (c) Service Type Parsing

   (d) Database Object Creation

2. WSDL Generation

In this chapter, we will illustrate the implementation of each step and present the final output.

4.1 Service Component Generation

4.1.1 Source code parsing

To acquire both compile-time and runtime information, we utilize the JDK compiler and the Reflection API to disassemble the source code (see Figure 4.2). We then convert the information provided by these APIs into the required models and merge the two sources. The model comprises two parts:

- **Data Model**: for program element information (e.g., class, field, method).
- **Input/Output Model**: for method parameters and return types.

4.1.2 Dependency extraction

Dependency Extraction involves extracting the dependencies between program elements and associating these relationships with the Data Model extracted in the previous step (see Figure 4.3). The process of Dependency Extraction consists of three main steps:

doi:10.6342/NTU202303287

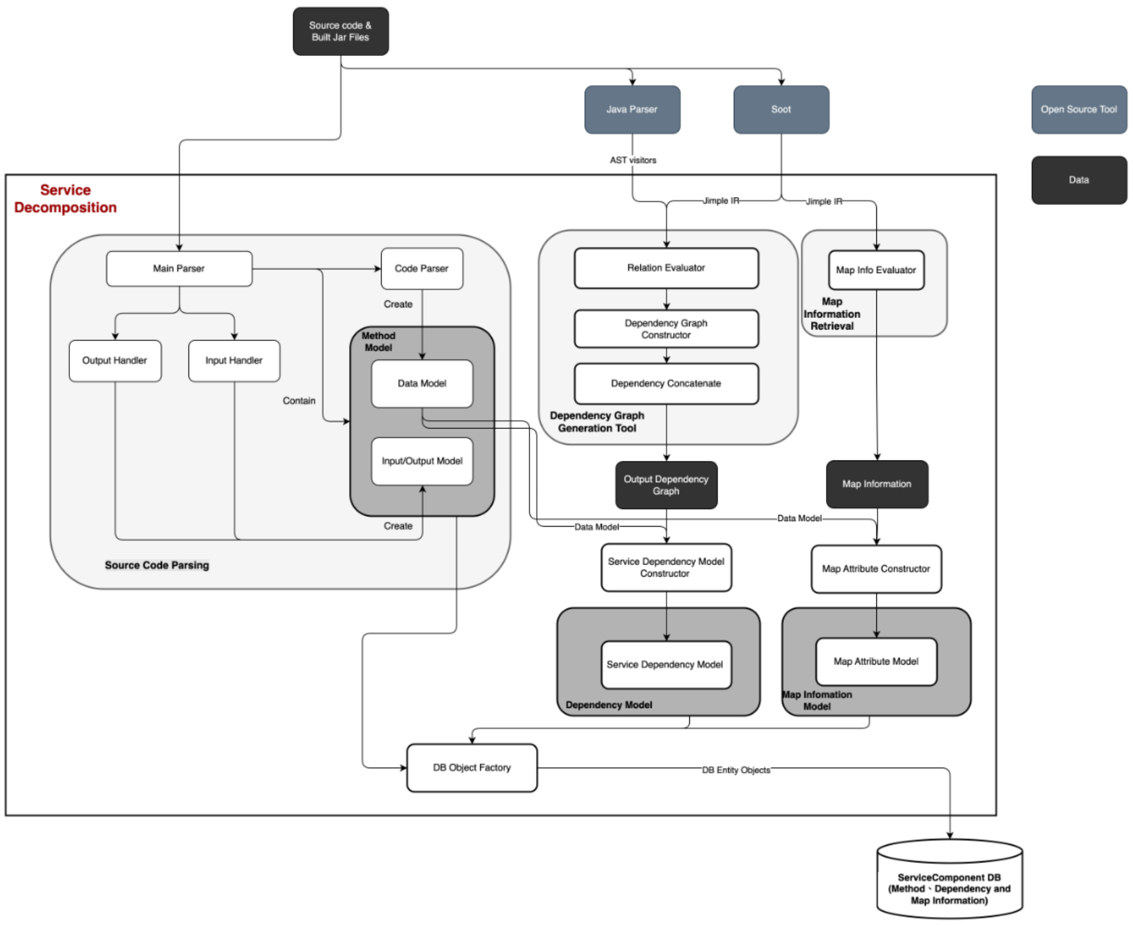


Figure 4.1: Source Code Parsing System Architecture

doi:10.6342/NTU202303287

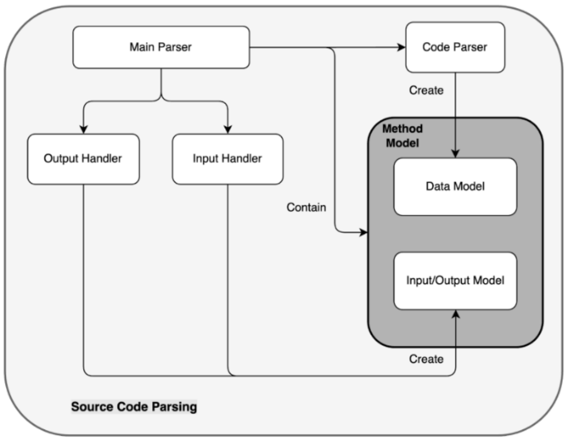


Figure 4.2: Source Code Parsing

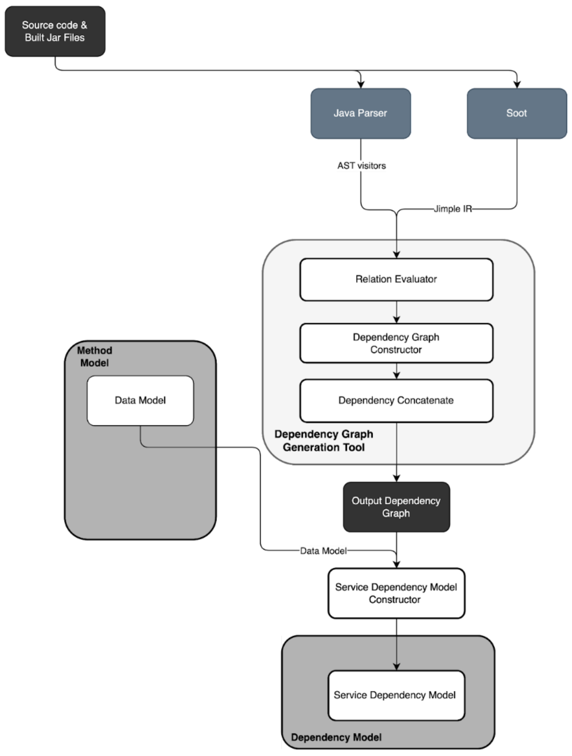
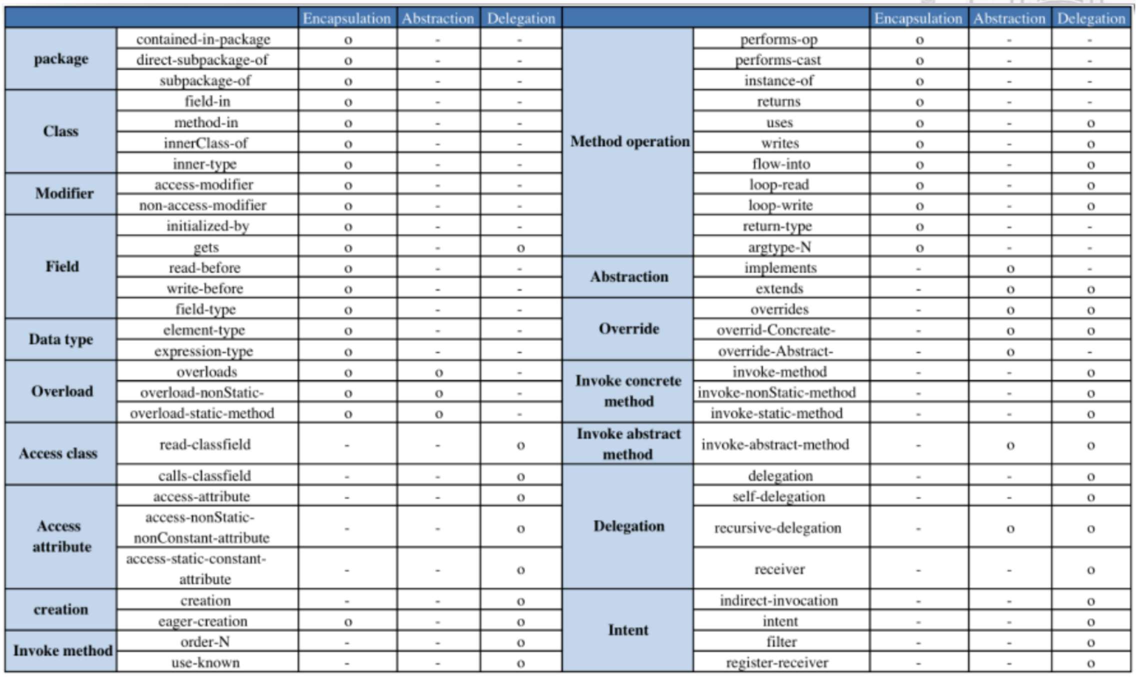


Figure 4.3: Dependency Extraction

doi:10.6342/NTU202303287



		Encapsulation	Abstraction	Delegation
package	contained-in-package	o	-	-
	direct-subpackage-of	o	-	-
	subpackage-of	o	-	-
Class	field-in	o	-	-
	method-in	o	-	-
	innerClass-of	o	-	-
	inner-type	o	-	-
Modifier	access-modifier	o	-	-
	non-access-modifier	o	-	-
Field	initialized-by	o	-	-
	gets	o	-	o
	read-before	o	-	-
	write-before	o	-	-
	field-type	o	-	-
Data type	element-type	o	-	-
	expression-type	o	-	-
Overload	overloads	o	o	-
	overload-nonStatic-	o	o	-
	overload-static-method	o	o	-
Access class	read-classfield	-	-	o
	calls-classfield	-	-	o
Access attribute	access-attribute	-	-	o
	access-nonStatic-nonConstant-attribute	-	-	o
	access-static-constant-attribute	-	-	o
creation	creation	-	-	o
	eager-creation	o	-	o
Invoke method	order-N	-	-	o
	use-known	-	-	o

		Encapsulation	Abstraction	Delegation
Method operation	performs-op	o	-	-
	performs-cast	o	-	-
	instance-of	o	-	-
	returns	o	-	-
	uses	o	-	o
	writes	o	-	o
	flow-into	o	-	o
	loop-read	o	-	o
	loop-write	o	-	o
	return-type	o	-	-
	argtype-N	o	-	-
Abstraction	implements	-	o	-
	extends	-	o	o
Override	overrides	-	o	o
	overrid-Concreate-	-	o	o
	override-Abstract-	-	o	-
Invoke concrete method	invoke-method	-	-	o
	invoke-nonStatic-method	-	-	o
	invoke-static-method	-	-	o
Invoke abstract method	invoke-abstract-method	-	o	o
Delegation	delegation	-	-	o
	self-delegation	-	-	o
	recursive-delegation	-	o	o
	receiver	-	-	o
Intent	indirect-invocation	-	-	o
	intent	-	-	o
	filter	-	-	o
	register-receiver	-	-	o

Figure 4.4: Program Relation Summary

1. Relation Evaluation
2. Dependency Concatenate
3. Generic Type Dependency

Relation evaluator

The Relation Evaluator is built upon previous research [24]. It analyzes Java bytecode using Soot and extracts 10 types of program elements and 56 types of dependencies (see Figure 4.4). Subsequently, it generates a dependency graph (see Figure 4.5) that describes the relationships between these program elements.

doi:10.6342/NTU202303287

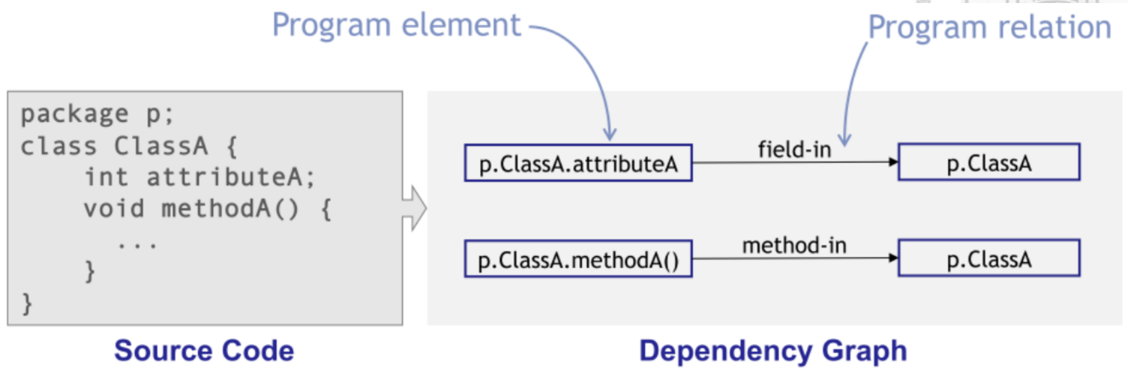


Figure 4.5: Dependency Graph Example

Dependency concatenate

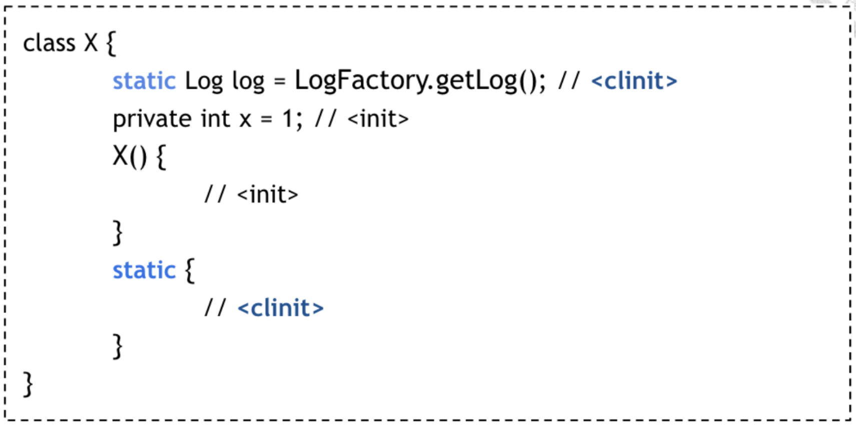
To bridge the gap between the model based on source code and the extracted dependencies based on bytecode, we need to establish connections for methods whose dependencies are linked by synthetic elements.

class level The <clinit>method is responsible for the static initialization of the Java class. Static elements, such as static blocks and static fields, are placed within this method to perform uniform initializations (see Figure 4.6).

When the class is loaded by the ClassLoader, the <clinit>method is invoked immediately. Consequently, class-level methods and variables should have a class-level dependency with the class itself(see Figure 4.7).

final field During the optimizing compilation, the JDK compiler extracts the value of the final string field to the Constant Pool. As a result, there is no direct relationship between the original field and the program element that uses it in the

doi:10.6342/NTU202303287



```
class X {
        static Log log = LogFactory.getLog(); // <clinit>
        private int x = 1; // <init>
        X() {
                // <init>
        }
        static {
                // <clinit>
        }
}
```

Figure 4.6: example of the class level element presented in the bytecode

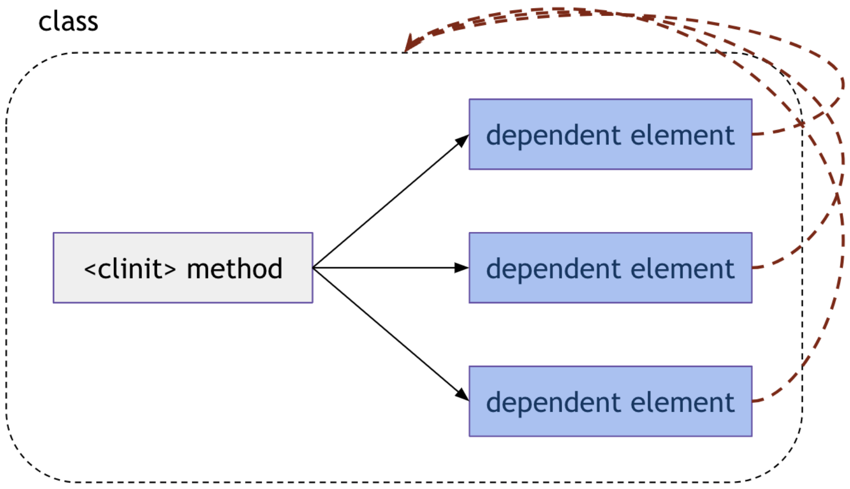
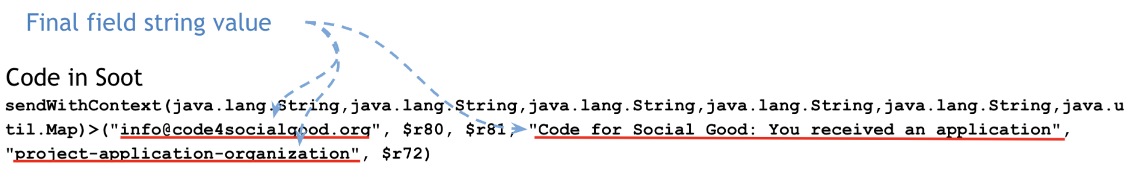


Figure 4.7: concate the dependency between the class level elements with the class

doi:10.6342/NTU202303287



```
sendWithContext(java.lang.String,java.lang.String,java.lang.String,java.lang.String,java.lang.String,java.u
til.Map)>("info@code4socialgood.org", $r80, $r81, "Code for Social Good: You received an application",
"project-application-organization", $r72)
```

Figure 4.8: final field invocation present in Jimple IR produced by Soot

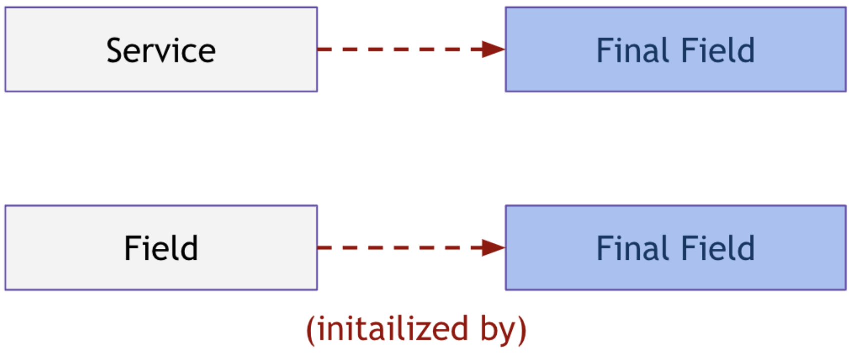


Figure 4.9: concate the dependency between the program elements with the final field

bytecode.

To address this, we use the SymbolSolver in JavaParser to obtain this type of dependency.

lambda expression, method reference and anonymous class After version 8, Java introduced support for lambda expressions and method references to replace the original anonymous class style. Java supports four basic function forms, which are:

1. **Consumer**: with parameters and without return value。
    (a) Its functional method is **accept**

doi:10.6342/NTU202303287

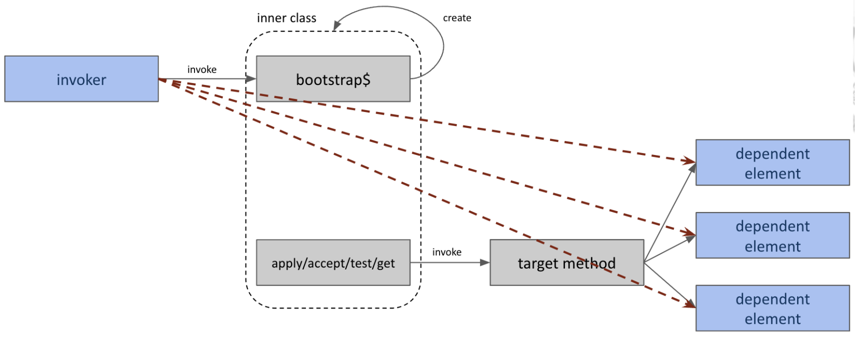


Figure 4.10: concate the dependency between the program elements with lambda expression dependent elements

2. **Function** : with parameters and with return value。
   (a) Its functional method is **apply**
3. **Supplier** : without parameters and with return value。
   (a) Its functional method is **get**
4. **Predicate** : with parameters and with the boolean type return value。
   (a) Its functional method is **test**

The JDK compiler generates inner classes for lambda expressions, method references, and anonymous classes that are not present in the original open-source code. These inner classes are used to invoke their internal logic through their functional method (or anonymous class method). Consequently, we have to concatenate the dependency between the invoker and the program elements dependent with the internal logic (see Figure 4.10、Figure 4.11、Figure 4.12).

doi:10.6342/NTU202303287

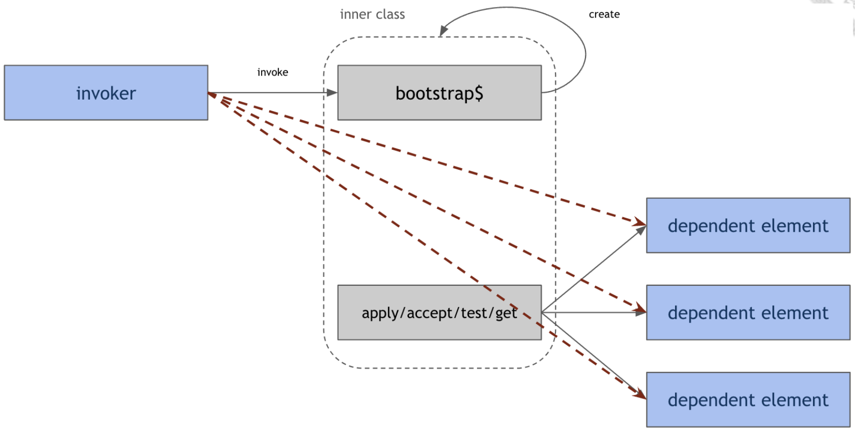


Figure 4.11: concate the dependency between the program elements with method reference dependent elements

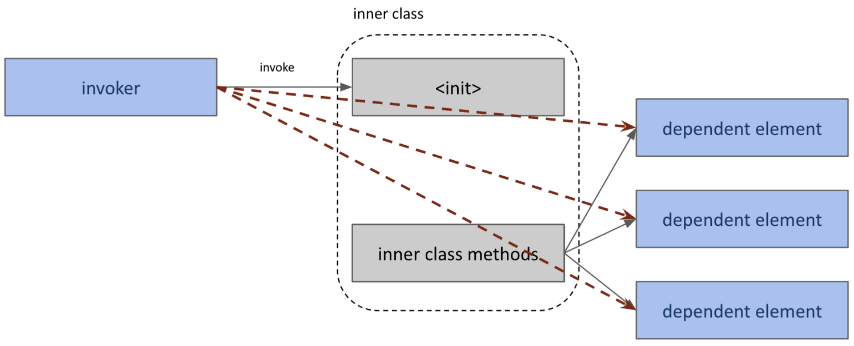
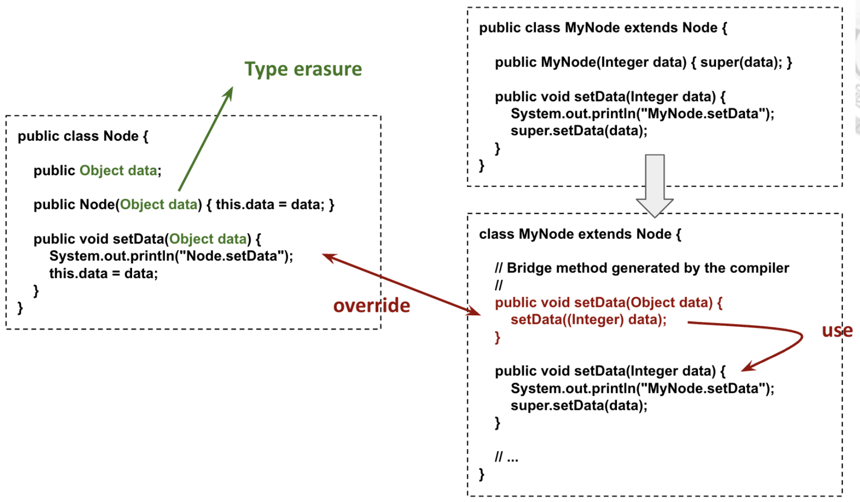


Figure 4.12: concate the dependency between the program elements with anonymous class dependent elements

doi:10.6342/NTU202303287



```
public class Node {

    public Object data;

    public Node(Object data) { this.data = data; }

    public void setData(Object data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}
```

```
public class MyNode extends Node {

    public MyNode(Integer data) { super(data); }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```

```
class MyNode extends Node {

    // Bridge method generated by the compiler
    //
    public void setData(Object data) {
        setData((Integer) data);
    }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }

    // ...
}
```

Figure 4.13: example of the type erasure and its corresponding bridge method

type erasure and bridge method Sometimes JDK compiler will automatically create some method that are no in the original source code, which were called ”Synthetic method”. When compiling a class or interface that extends a parameterized class or implements a parameterized interface, the compiler will create the synthetic method to cast the type of the parameterized type to avoid error occur in the JVM execution (see Figure 4.13). When the above situation occurs, we need to connect the dependency between the methods at both ends of the synthetic method.

Generic Type Dependency

Due to **Type Erasure**, there is a lack of information about generic types in bytecode, which prevents Soot from extracting dependencies with generic types. To address this issue, we use JavaParser to analyze the source code and obtain dependencies that involve generic types.

doi:10.6342/NTU202303287

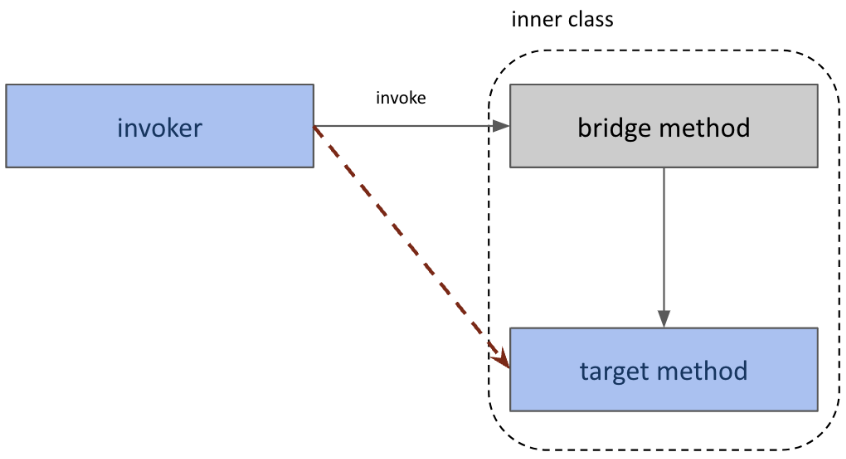


Figure 4.14: concate the dependency between the program elements with the target method

Map information evaluation

Map Information use Jimple IR compiled by Soot to evaluate the dynamic data stored in the Java Map (see Figure 4.15). From the register in the return statement, we use Chain of responsibility pattern to track the Map operation invocation in the method body. There are three chains: Map, Collection and Object creation, which can ensure that the dynamic data can be extracted under any kind of data structure.

4.1.3 Service Type Parsing

In this step, we determine the role of each program element in the project using the annotations provided by the Spring framework and JPA [4]. This information will be used for further matchmaking purposes.

The annotations used in this research are listed below, along with their descrip-

doi:10.6342/NTU202303287

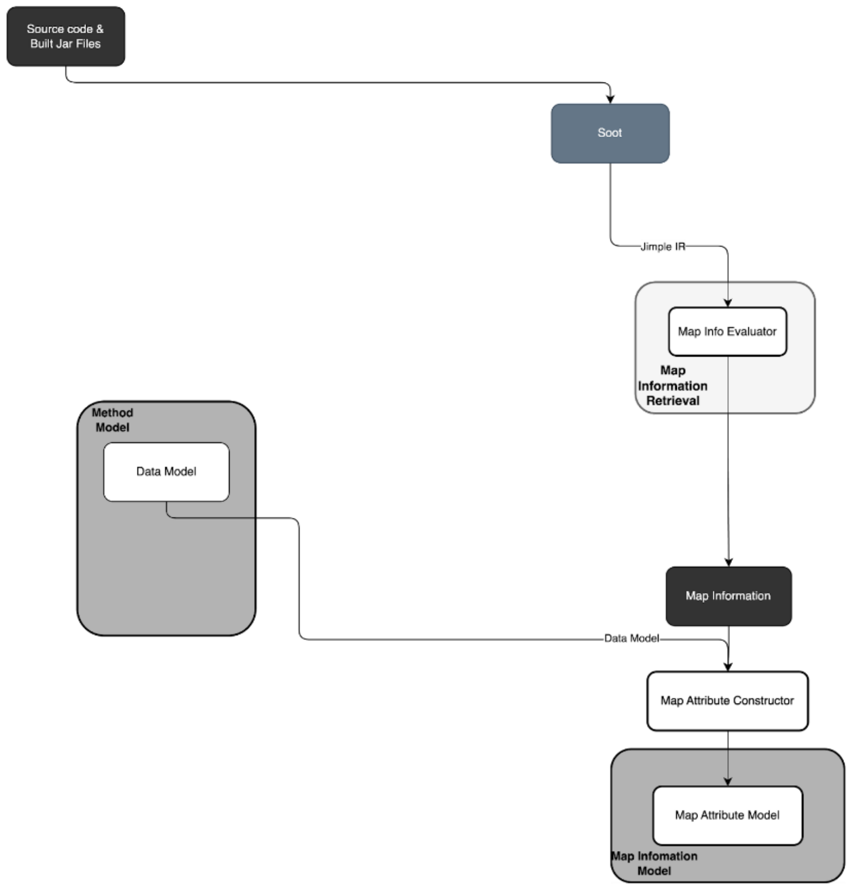


Figure 4.15: Map Information Evaluation

doi:10.6342/NTU202303287

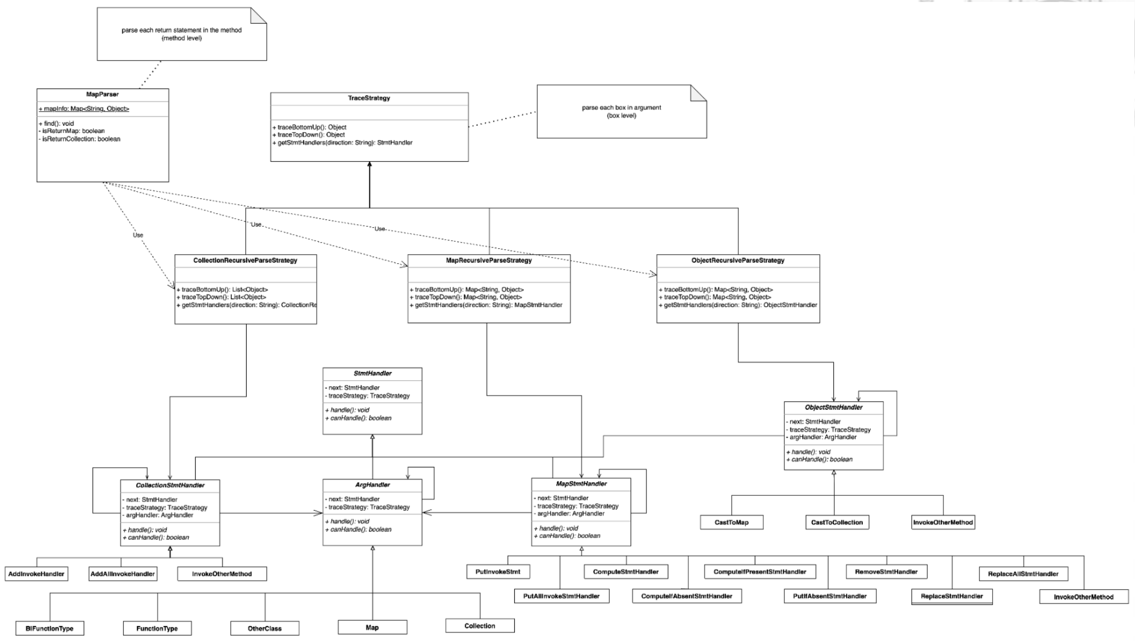

parse each return statement in the method
(method level)
MapParser
+ mapInfo: Map<String, Object>
+ find(): void
- isReturnMap: boolean
- isReturnCollection: boolean
TraceStrategy
+ traceBottomUp(): Object
+ traceTopDown(): Object
+ getStmtHandlers(direction: String): StmtHandler
parse each box in argument
(box level)
Use
Use
Use
CollectionRecursiveParseStrategy
+ traceBottomUp(): List<Object>
+ traceTopDown(): List<Object>
+ getStmtHandlers(direction: String): CollectionRe
MapRecursiveParseStrategy
+ traceBottomUp(): Map<String, Object>
+ traceTopDown(): Map<String, Object>
+ getStmtHandlers(direction: String): MapStmtHandler
ObjectRecursiveParseStrategy
+ traceBottomUp(): Map<String, Object>
+ traceTopDown(): Map<String, Object>
+ getStmtHandlers(direction: String): ObjectStmtHandler
StmtHandler
- next: StmtHandler
- traceStrategy: TraceStrategy
+ handle(): void
+ canHandle(): boolean
ObjectStmtHandler
- next: StmtHandler
- traceStrategy: TraceStrategy
- argHandler: ArgHandler
+ handle(): void
+ canHandle(): boolean
CollectionStmtHandler
- next: StmtHandler
- traceStrategy: TraceStrategy
- argHandler: ArgHandler
+ handle(): void
+ canHandle(): boolean
ArgHandler
- next: StmtHandler
- traceStrategy: TraceStrategy
+ handle(): void
+ canHandle(): boolean
MapStmtHandler
- next: StmtHandler
- traceStrategy: TraceStrategy
- argHandler: ArgHandler
+ handle(): void
+ canHandle(): boolean
CastToMap
CastToCollection
InvokeOtherMethod
AddInvokeHandler
AddAllInvokeHandler
InvokeOtherMethod
PutInvokeStmt
ComputeStmtHandler
ComputeIfPresentStmtHandler
RemoveStmtHandler
ReplaceAllStmtHandler
PutAllInvokeStmtHandler
ComputeIfAbsentStmtHandler
PutIfAbsentStmtHandler
ReplaceStmtHandler
InvokeOtherMethod
BiFunctionType
FunctionType
OtherClass
Map
Collection

Figure 4.16: Map Information Evaluation System Design

tions from the Javadoc:

1. **@Controller** : “*Indicates that as annotated class is a* ‘*Controller*’ *(e.g a web controller)*”

2. **@Service** : “*Indicates that an annotated class is a* ‘*Service*’ *, originally defined by Domain-Driven Design(Evans, 2003) as* ‘*an operation offered as an interface that stands alone in the model, with no encapsulated state.*”

3. **@Repository** : “*indicated that an annotated class is a* ‘*Repository*’ *, originally defined by Domain-Driven Design(Evans, 2003) as* ‘*a mechanism for encapsulating storage, retrieval, and search behavior which emulated a collection of objects*’ ”

doi:10.6342/NTU202303287

4. **@Entity** : “*Specifies that the class is an entity. This annotation is applied to the entity class*”

These annotations play a crucial role in defining the roles and functionalities of various program elements within the project.

4.1.4 DB Object Creation

For further usage, we transform the model into the database entity and store it in the database (see Figure 4.17). To handle the creation of different types of database entities, we employ the factory pattern, allowing for a flexible and organized approach to entity creation. We use MySQL to store the data.

service component DB （see Figure 4.18）store the service components. There are four kind of information in the storage:

1. core element (ex. class, method)
2. dependency between core elements
3. semantic annotation
4. other elements (ex. parameters, return value)

4.2 WSDL generation

To describe the services offered by each service component and enable requirement matchmaking, we generate a WSDL file for each service component.

doi:10.6342/NTU202303287

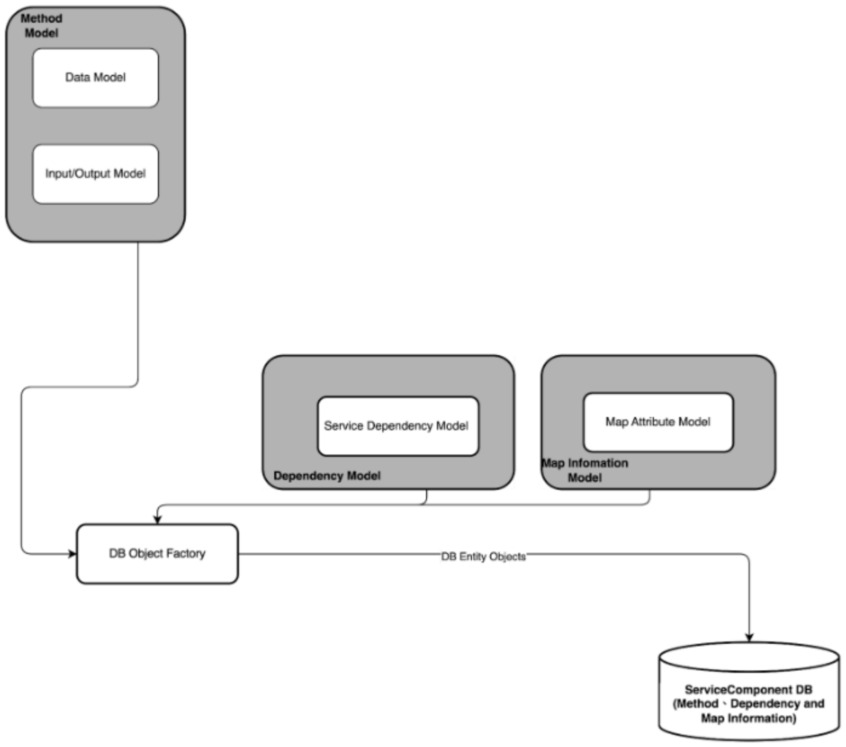


Figure 4.17: Model Creation

doi:10.6342/NTU202303287

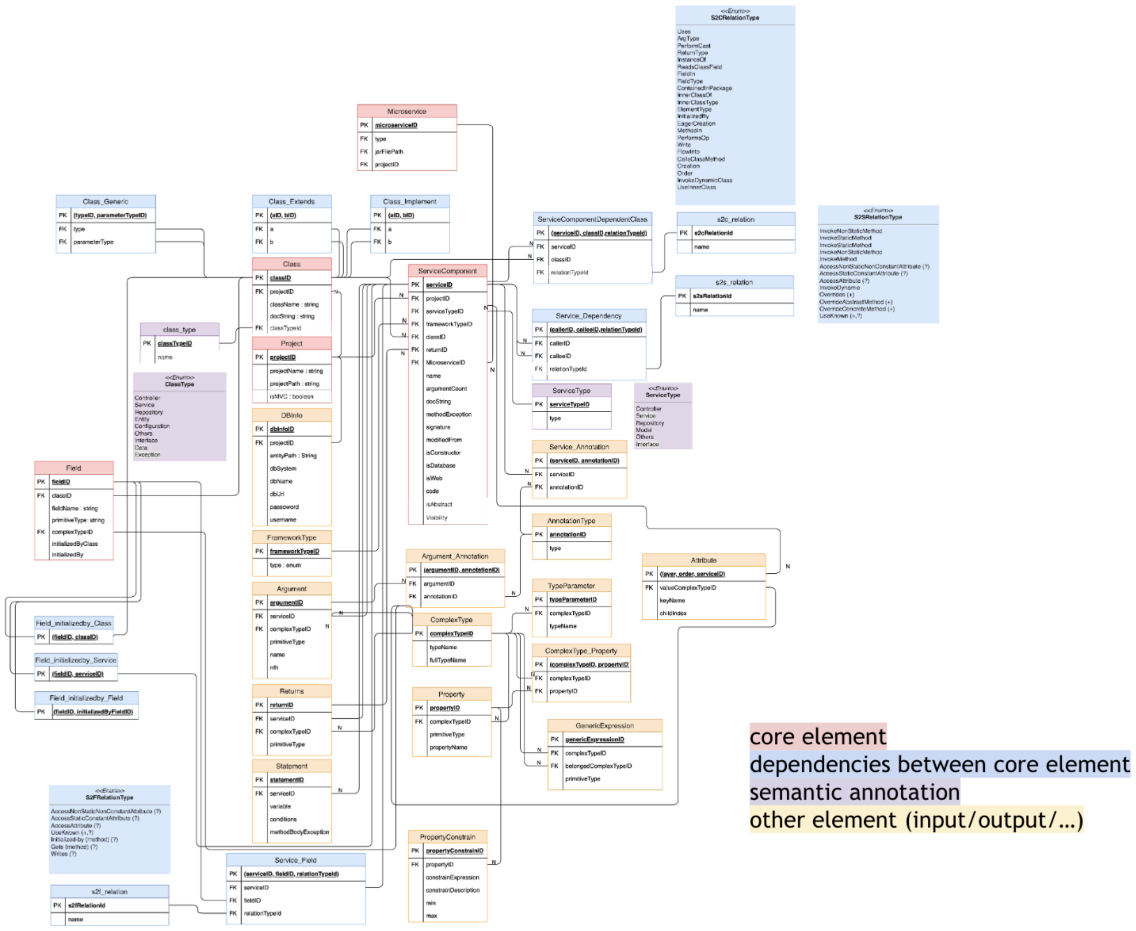
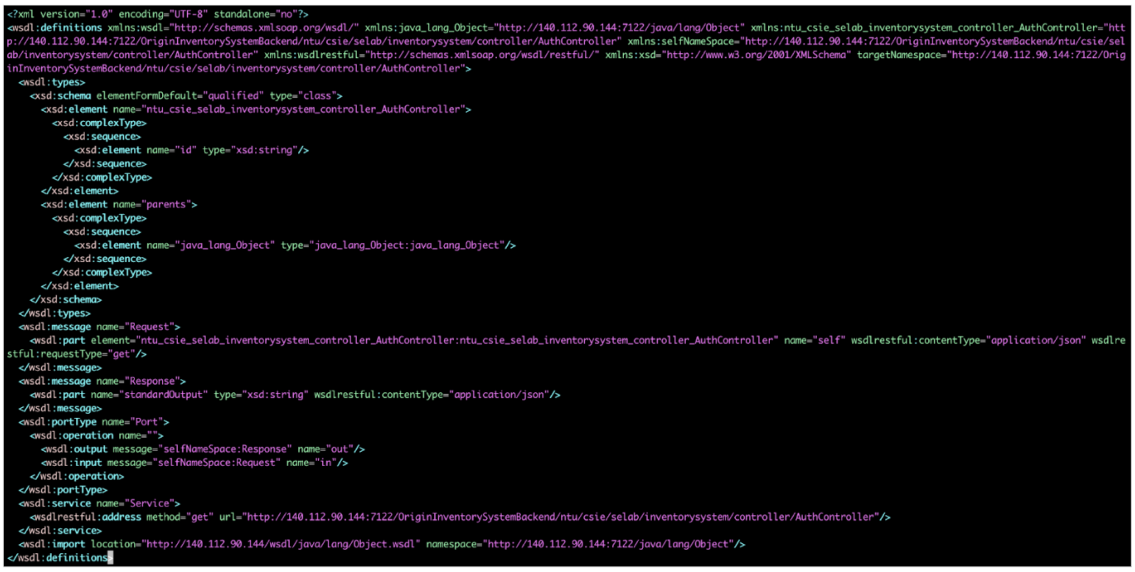


Figure 4.18: Service Component Database Schema

doi:10.6342/NTU202303287



```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:java_lang_Object="http://140.112.90.144:7122/java/lang/Object" xmlns:ntu_csie_selab_inventorysystem_controller_AuthController="htt
p://140.112.90.144:7122/OriginInventorySystemBackend/ntu/csie/selab/inventorysystem/controller/AuthController" xmlns:selfNameSpace="http://140.112.90.144:7122/OriginInventorySystemBackend/ntu/csie/sel
ab/inventorysystem/controller/AuthController" xmlns:wsdlrestful="http://schemas.xmlsoap.org/wsdl/restful/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="http://140.112.90.144:7122/Orig
inInventorySystemBackend/ntu/csie/selab/inventorysystem/controller/AuthController">
  <wsdl:types>
    <xsd:schema elementFormDefault="qualified" type="class">
      <xsd:element name="ntu_csie_selab_inventorysystem_controller_AuthController">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="id" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="parents">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="java_lang_Object" type="java_lang_Object:java_lang_Object"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="Request">
    <wsdl:part element="ntu_csie_selab_inventorysystem_controller_AuthController:ntu_csie_selab_inventorysystem_controller_AuthController" name="self" wsdlrestful:contentType="application/json" wsdlre
stful:requestType="get"/>
  </wsdl:message>
  <wsdl:message name="Response">
    <wsdl:part name="standardOutput" type="xsd:string" wsdlrestful:contentType="application/json"/>
  </wsdl:message>
  <wsdl:portType name="Port">
    <wsdl:operation name="">
      <wsdl:output message="selfNameSpace:Response" name="out"/>
      <wsdl:input message="selfNameSpace:Request" name="in"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:service name="Service">
    <wsdlrestful:address method="get" url="http://140.112.90.144:7122/OriginInventorySystemBackend/ntu/csie/selab/inventorysystem/controller/AuthController"/>
  </wsdl:service>
  <wsdl:import location="http://140.112.90.144/wsdl/java/lang/Object.wsdl" namespace="http://140.112.90.144:7122/java/lang/Object"/>
</wsdl:definitions>
```

Figure 4.19: Example of the generated WSDL

doi:10.6342/NTU202303287

Chapter 5

Service Composition

Service Composition is the process of composing service components to microservices (see Figure 5.1).

First of all, we evaluate which APIs are suitable to be placed in the same microservice based on the extracted dependencies.

Second, we compose the microservices with the service components those dependent with the APIs belong to them.

Finally, we deploy the generated microservices to the Spring Cloud.

5.1 Dependency Repack

We apply an object-oriented approach, beginning with the service component, which serves as the API in the original project. Through recursive repackaging along the dependency flow, a tree structure is formed (see Figure 5.2).

doi:10.6342/NTU202303287

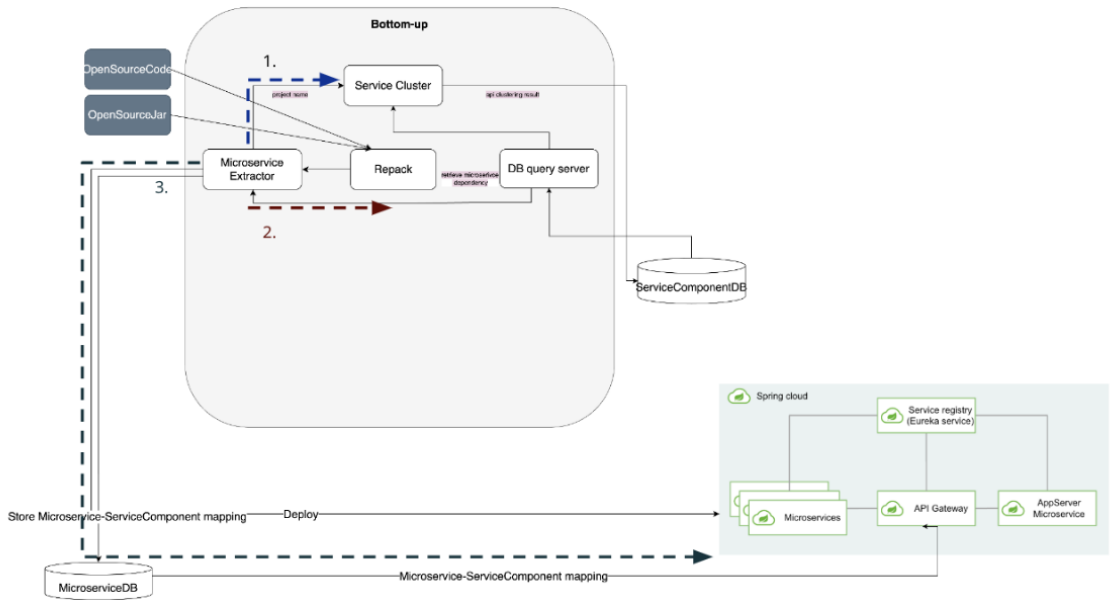


Figure 5.1: Service Composition

The nodes in this structure can be categorized into three types: **class**, **field**, and **service**.

The edge (Dependencies) in this structure can be further classified into three categories:

1. Service-to-class, service-to-service, and service-to-field dependencies extended from service nodes.

2. Field initialization dependency extended from field nodes.

3. Class-level dependency extended from class nodes.

doi:10.6342/NTU202303287

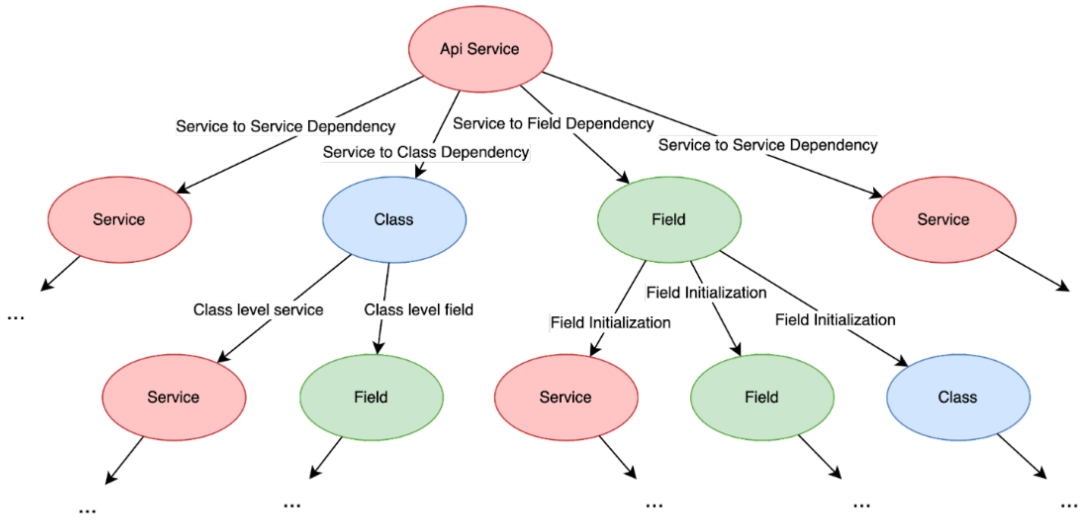


Figure 5.2: dependency repack visualization

5.2 API Clustering

Based on the definition of microservice, it encapsulates the business logic and exposes minimal public interfaces, which should exhibit high cohesion. Therefore, we need to determine which APIs should be placed in the same microservice according to this definition.

5.2.1 distance between APIs

To ensure high cohesion, we calculate the distance between APIs using an encapsulation metric. We begin by considering the service components, which represent the APIs in the original project, and then repack these service components based on their dependency flow, forming multiple dependency trees. We calculate the method usage overlap between pairs of APIs. This overlap serves as the clustering distance

doi:10.6342/NTU202303287

metric.

Next, we determine the distance relationship between two APIs based on the overlap of their dependency trees. The calculation method involves computing the intersection divided by the union.（see Equation5.1）。

$$D_{ij} = 1 - \frac{\#(M_i \cap M_j)}{\#(M_i \cup M_j)} \tag{5.1}$$

where:

M_i = method set under the API i dependency scope

M_j = method set under the API j dependency scope

D_{ij} = distance between API i and API j

5.2.2 cluster APIs

By calculating distances for all API pairs and transforming them into a distance matrix, we apply the density-based clustering algorithm, HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise), for clustering. We chose this clustering algorithm because, unlike other density-based algorithms that use an epsilon parameter to determine scope, HDBSCAN uses hierarchy to replace the parameter, which improves the robustness of the clustering result and allows for the identification of clusters with various densities. The resulting clusters are then stored in the Service Component DB, with each API cluster representing a microservice.

doi:10.6342/NTU202303287

5.3 Service Component Composition

The service component repack mechanism is based on previous research [29] and involves rewriting the original project. Once the decomposition of the Monolithic system is determined, we begin with the service components, which were the APIs in the original project, and repack all the program elements in the dependency tree.

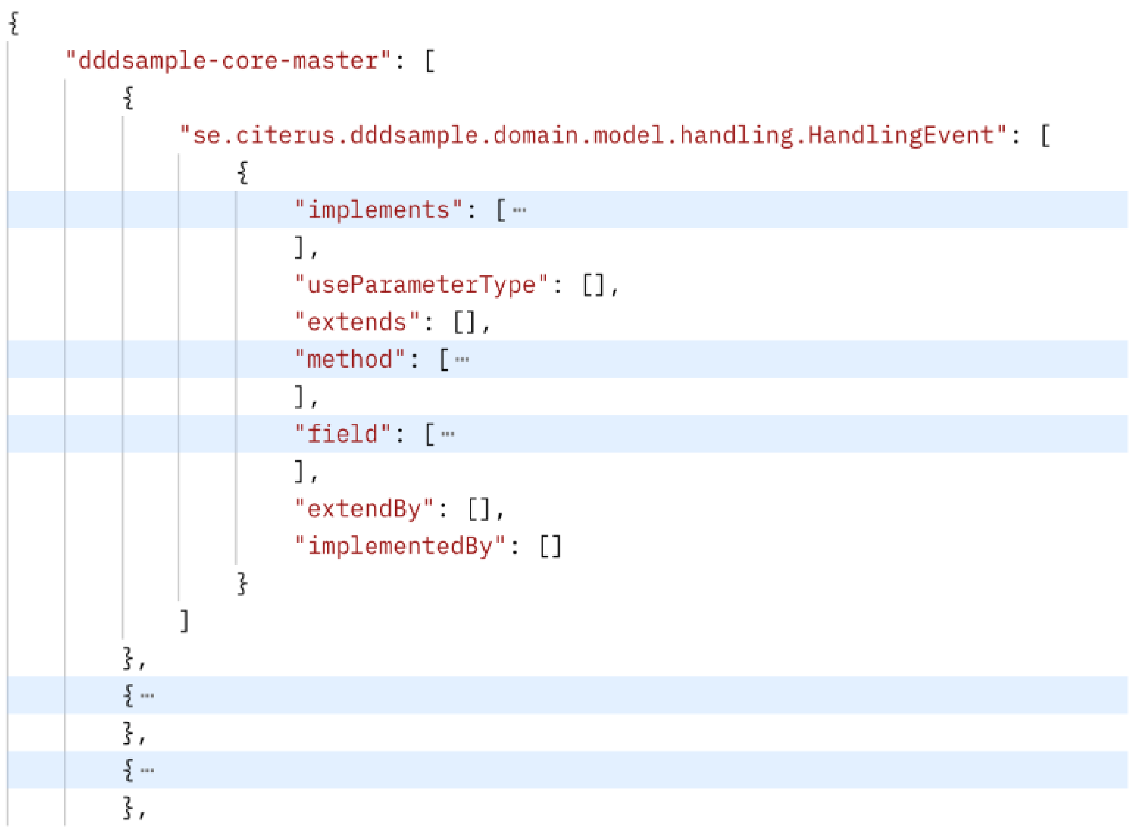
Next, we will provide a detailed explanation of the following processes:

1. Query dependencies
2. Model dependencies
3. Configure JavaParser
4. Attach model with AST
5. Clone the tree and remove the unnecessary node in the tree based on the information stored in model
6. Plugin Handler COR to collect the needed configuration AST
7. Write Project and Post Processing
8. Build project and zip

5.3.1 Query Dependency & dependency Modeling

Retrieve the microservice internal structure (see Figure 5.3) based on the API dependency tree from the database query server, and store this information in our self-defined model on the repack server (see Figure 5.4).

doi:10.6342/NTU202303287



```
{
    "dddsample-core-master": [
        {
            "se.citerus.dddsample.domain.model.handling.HandlingEvent": [
                {
                    "implements": [⋯
                    ],
                    "useParameterType": [],
                    "extends": [],
                    "method": [⋯
                    ],
                    "field": [⋯
                    ],
                    "extendBy": [],
                    "implementedBy": []
                }
            ]
        },
        {⋯
        },
        {⋯
        },
```

Figure 5.3: db query response format

doi:10.6342/NTU202303287

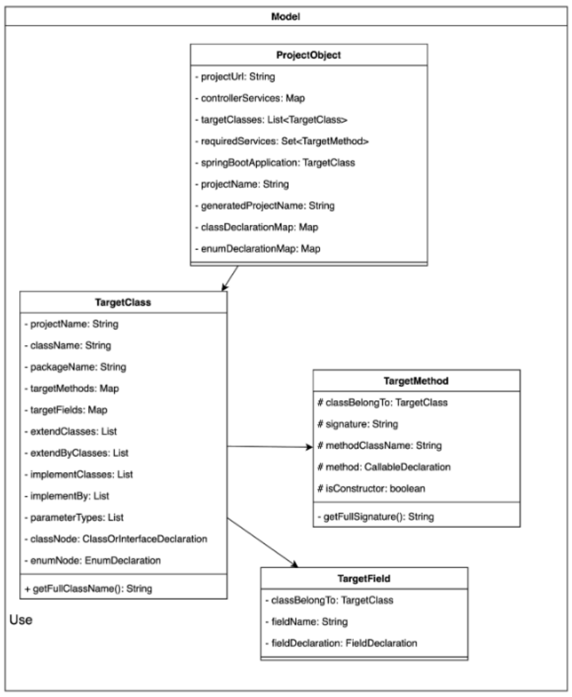

Model
ProjectObject
- projectUrl: String
- controllerServices: Map
- targetClasses: List<TargetClass>
- requiredServices: Set<TargetMethod>
- springBootApplication: TargetClass
- projectName: String
- generatedProjectName: String
- classDeclarationMap: Map
- enumDeclarationMap: Map
TargetClass
- projectName: String
- className: String
- packageName: String
- targetMethods: Map
- targetFields: Map
- extendClasses: List
- extendByClasses: List
- implementClasses: List
- implementBy: List
- parameterTypes: List
- classNode: ClassOrInterfaceDeclaration
- enumNode: EnumDeclaration
+ getFullClassName(): String
TargetMethod
# classBelongTo: TargetClass
# signature: String
# methodClassName: String
# method: CallableDeclaration
# isConstructor: boolean
- getFullSignature(): String
TargetField
- classBelongTo: TargetClass
- fieldName: String
- fieldDeclaration: FieldDeclaration
Use

Figure 5.4: repack model

doi:10.6342/NTU202303287

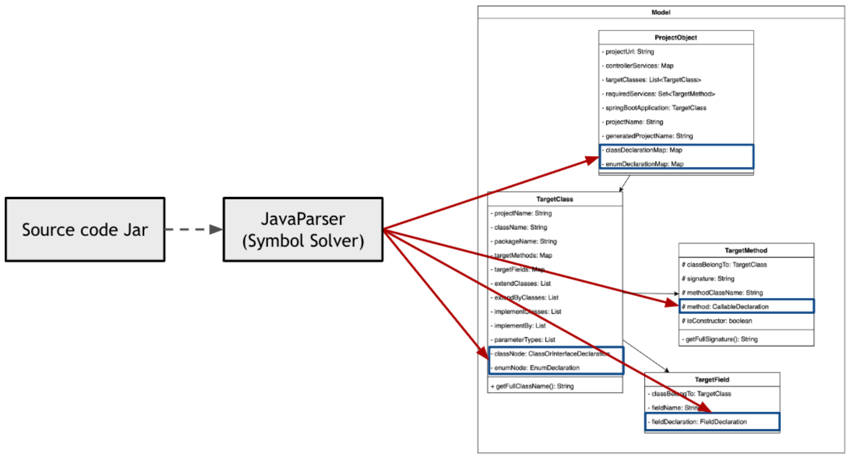

Source code Jar
JavaParser (Symbol Solver)
Model
ProjectObject
- projectUrl: String
- controllerServices: Map
- targetClasses: List<TargetClass>
- requiredServices: Set<TargetMethod>
- springBootApplication: TargetClass
- projectName: String
- generatedProjectName: String
- classDeclarationMap: Map
- enumDeclarationMap: Map
TargetClass
- projectName: String
- className: String
- packageName: String
- targetMethods: Map
- targetFields: Map
- extendClasses: List
- extendByClasses: List
- implementClasses: List
- implementBy: List
- parameterTypes: List
- classNode: ClassOrInterfaceDeclaration
- enumNode: EnumDeclaration
+ getFullClassName(): String
TargetMethod
# classBelongTo: TargetClass
# signature: String
# methodClassName: String
# method: CallableDeclaration
# isConstructor: boolean
- getFullSignature(): String
TargetField
- classBelongTo: TargetClass
- fieldName: String
- fieldDeclaration: FieldDeclaration

Figure 5.5: AST attaching

5.3.2 Configure JavaParser & attach AST to model

We use the visitors provided by JavaParser, overriding its logic to collect the AST nodes we need, and attach them to the model (see Figure 5.5).

5.3.3 Clone the element & remove the unnecessary node

The original AST tree is attached to the model, which retains all the information about the AST. Next, we identify the Java files required, clone them, and store them in a separate map. We then make modifications to this cloned AST node.

Using the model information, we perform subtraction on the cloned node, removing unnecessary elements such as methods, fields, inner classes, or import statements.

doi:10.6342/NTU202303287

5.3.4 Plugin Handler Chain

In the aforementioned repack process, we repack the service components with the dependency flow based on the object-oriented concept. However, many projects use frameworks and plugins that encapsulate the dependency flow into their libraries. As a result, the dependency flow starting from the API may not be able to repack some necessary configurations (e.g., Beans in Spring Framework).

To address this issue, we propose the Plugin Handler Chain (see Figure 5.6), which utilizes the Chain of Responsibility pattern. The input to this chain is the dependency tree compiled from the build tool, and the detailed execution process is as follows:

1. Model the dependencies as the PluginService.Dependency classes.
2. Create the Plugin Handler COR chain.
3. Iterate through each dependency using the COR chain.
4. Create the corresponding post-processor if needed (something that needs to be modified but is not in the AST, e.g., build script files).

5.3.5 Write Project and Post Processing

The construction of the microservice project is performed in the following detailed process:

1. Copying the original project and removing the source code directory.

doi:10.6342/NTU202303287

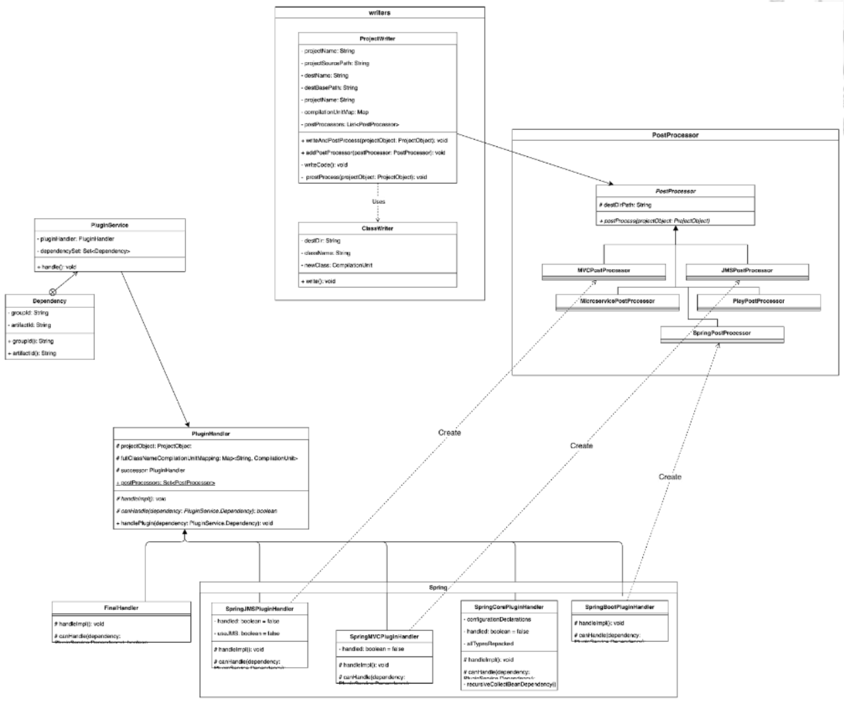

writers
ProjectWriter
- projectName: String
- projectSourcePath: String
- destName: String
- destBasePath: String
- projectName: String
- compilationUnitMap: Map
- postProcessors: List<PostProcessor>
+ writeAndPostProcess(projectObject: ProjectObject): void
+ addPostProcessor(postProcessor: PostProcessor): void
- writeCode(): void
- postProcess(projectObject: ProjectObject): void
Uses
ClassWriter
- destDir: String
- className: String
- newClass: CompilationUnit
+ write(): void
PostProcessor
PostProcessor
# destDirPath: String
+ postProcess(projectObject: ProjectObject)
MVCPostProcessor
JMSPostProcessor
MicroservicePostProcessor
PlayPostProcessor
SpringPostProcessor
PluginService
- pluginHandler: PluginHandler
- dependencySet: Set<Dependency>
+ handle(): void
Dependency
- groupId: String
- artifactId: String
+ groupId(): String
+ artifactId(): String
Create
Create
Create
PluginHandler
# projectObject: ProjectObject
# fullClassNameCompilationUnitMapping: Map<String, CompilationUnit>
# successor: PluginHandler
+ postProcessors: Set<PostProcessor>
# handleImpl(): void
# canHandle(dependency: PluginService.Dependency): boolean
+ handlePlugin(dependency: PluginService.Dependency): void
Spring
FinalHandler
# handleImpl(): void
# canHandle(dependency:
SpringJMSPluginHandler
- handled: boolean = false
- useJMS: boolean = false
# handleImpl(): void
# canHandle(dependency:
SpringMVCPluginHandler
- handled: boolean = false
# handleImpl(): void
# canHandle(dependency:
SpringCorePluginHandler
- configurationDeclarations
- handled: boolean = false
- allTypesReplaced
# handleImpl(): void
# canHandle(dependency:
- recursiveCollectBeanDependency()
SpringBootPluginHandler
# handleImpl(): void
# canHandle(dependency:

Figure 5.6: Plugin Handler Design

2. Writing AST into Java files one by one, according to the structure stored in the modification map.

3. Executing the post-processors to perform some rewrites that are not in the source code.

5.3.6 Build project and zip

After generating the microservice project, it is built using the build tool, and the resulting jar file is compressed for further usage.

doi:10.6342/NTU202303287

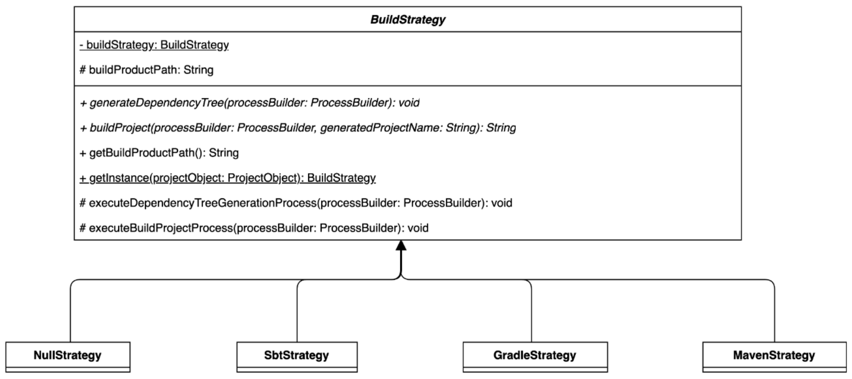

BuildStrategy
- buildStrategy: BuildStrategy
# buildProductPath: String
+ generateDependencyTree(processBuilder: ProcessBuilder): void
+ buildProject(processBuilder: ProcessBuilder, generatedProjectName: String): String
+ getBuildProductPath(): String
+ getInstance(projectObject: ProjectObject): BuildStrategy
# executeDependencyTreeGenerationProcess(processBuilder: ProcessBuilder): void
# executeBuildProjectProcess(processBuilder: ProcessBuilder): void
NullStrategy
SbtStrategy
GradleStrategy
MavenStrategy

Figure 5.7: Build Strategy

Build Strategy

During the repack process, we require the functionality of the build tool. However, due to variations in the implementation of different build tools, we utilize the strategy pattern to abstract the functionalities of the build tool (see Figure 5.7), making the entire system extensible. The functionality of the build tool:

1. build project

2. extract dependency tree

Currently supported build tools include gradle [1] and maven [8].

5.4 Deployment

Currently, we have repackaged the microservice projects to JAR files. Next, we will deploy them on Spring Cloud [12] and configure the API Gateway to handle access to the microservices uniformly.

doi:10.6342/NTU202303287

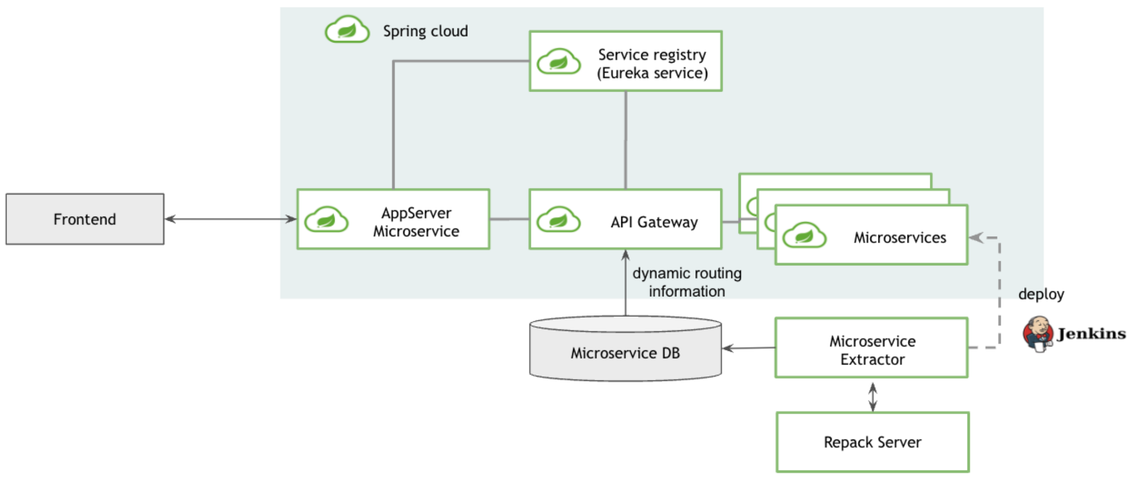


Figure 5.8: Service-Component-Based Microservice Architecture

5.4.1 Microservice Extractor

This step sets the necessary parameters of Spring Cloud for the jar file, mainly including the following steps（Figure 5.9）：

Allocate Port

Look for an idle port that can be used as the server port of the microservice.

Generate Startup Script

Generate a script that runs Microservice for subsequent Jenkins pipeline execution. In this script, three flags are mainly set for execution instructions:

1. server port: idle port allocate from the last step

2. application name

doi:10.6342/NTU202303287

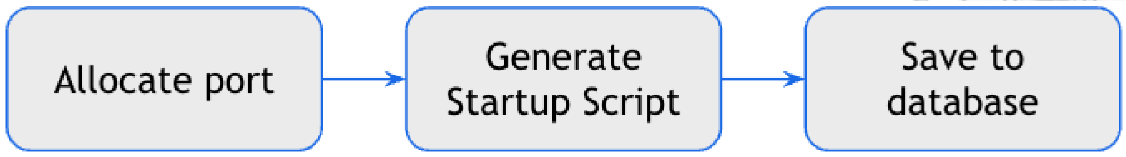


Figure 5.9: Microservice extraction process

3. eureka server URL

Example output of this step:

```
java -jar /Jar/File/Path/JarFileName.jar
--server.port=8764
--spring.application.name=microservice-application-name
--eureka.client.serviceUrl.defaultZone=http://{eureka-server-ip:port}/eureka
```

Save to database

Finally, we store the microservice information and the mapping relationship between the microservice and service components in the microservice database (see Figure 5.10), providing the necessary information for URL rewriting in the API Gateway.

5.4.2 Service Invocation

In Service Invocation (see Figure 5.13), the frontend communicates with the Application Server using WebSocket, while the Application Service communicates with the API Gateway using the HTTP protocol. When the frontend needs to

doi:10.6342/NTU202303287

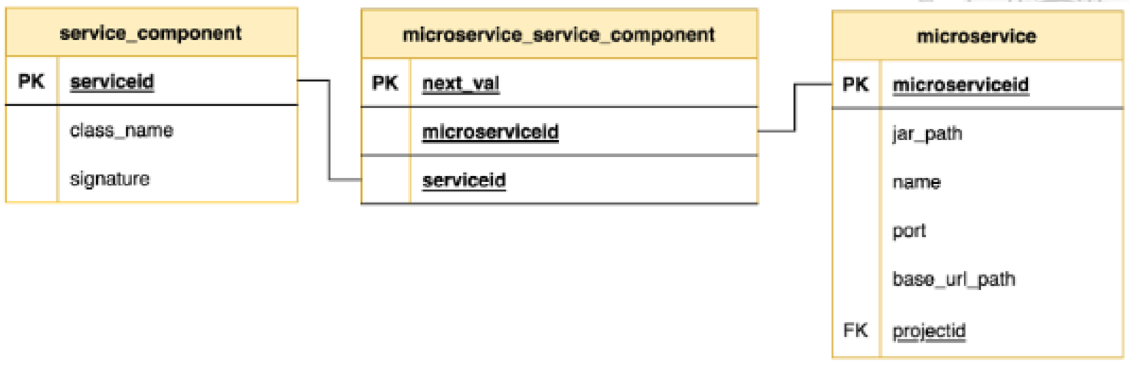


Figure 5.10: Microservice Database Schema

invoke backend services, it sends the target service ID and the parameters to the Application Server. The Application Server then transforms the request into an HTTP Request and sends it to the API Gateway. The API Gateway retrieves the URL rewrite information from the microservice database based on the target service ID for further processing. For more details on the URL rewriting process, please refer to Figure 5.11.

Based on the service invocation process described above, the API Gateway encapsulates microservices, so the Frontend server only needs to know the service's ID instead of the microservice it belongs to.

doi:10.6342/NTU202303287

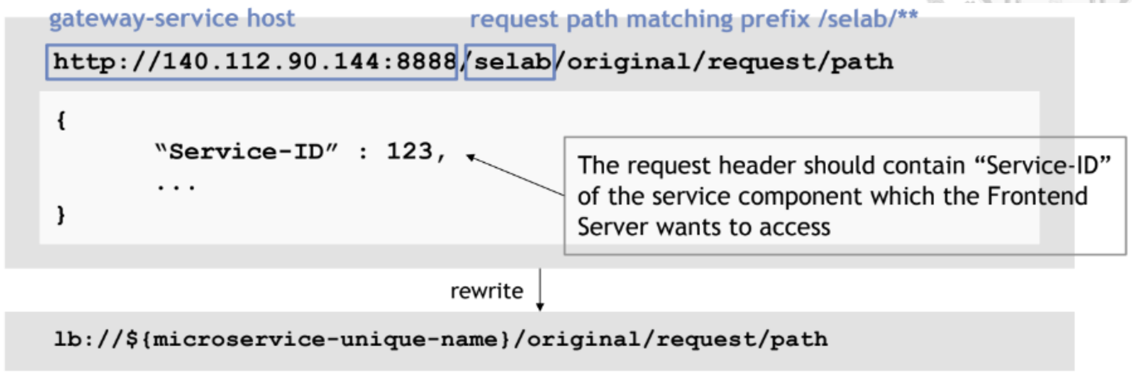

gateway-service host
request path matching prefix /selab/**
http://140.112.90.144:8888/selab/original/request/path
{
"Service-ID" : 123,
...
}
The request header should contain "Service-ID" of the service component which the Frontend Server wants to access
rewrite
lb://${microservice-unique-name}/original/request/path

Figure 5.11: API Gateway URL transform

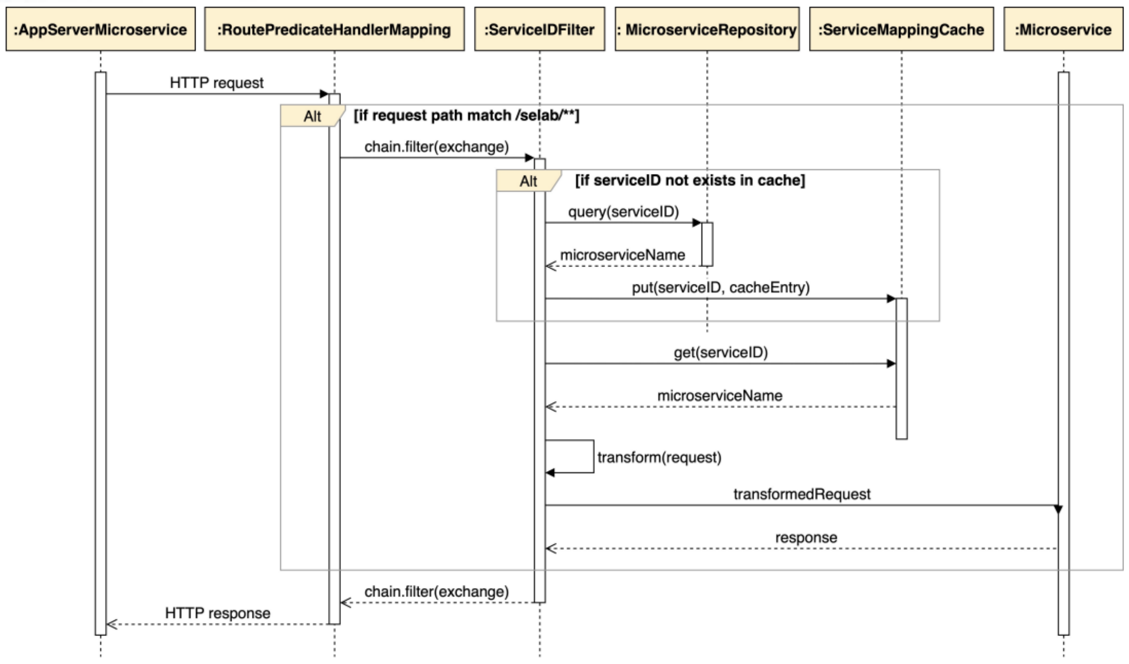


Figure 5.12: Sequence diagram of request URL transform in API Gateway

doi:10.6342/NTU202303287

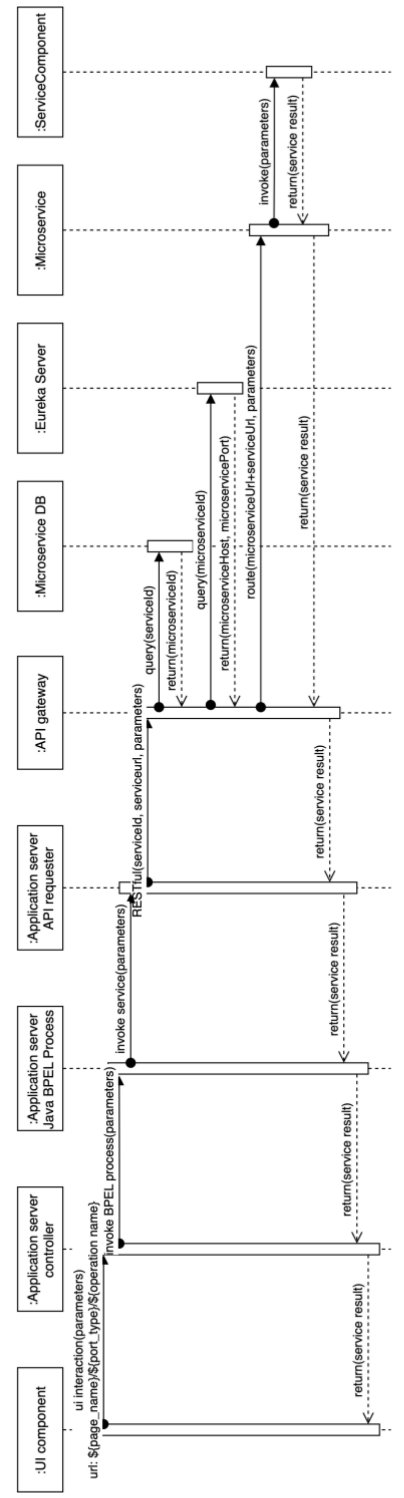

:UI component
:Application server controller
:Application server Java BPEL Process
:Application server API requester
:API gateway
:Microservice DB
:Eureka Server
:Microservice
:ServiceComponent
ui interaction(parameters)
url: ${page_name}/${port_type}/${operation name}
invoke BPEL process(parameters)
invoke service(parameters)
RESTful(serviceId, serviceurl, parameters)
query(serviceId)
return(microserviceId)
query(microserviceId)
return(microserviceHost, microservicePort)
route(microserviceUrl+serviceUrl, parameters)
invoke(parameters)
return(service result)
return(service result)
return(service result)
return(service result)
return(service result)
return(service result)
return(service result)

Figure 5.13: Sequence diagram of service invocation from UI Frontend server to service component

doi:10.6342/NTU202303287

Chapter 6

Microservice Matchmaking

To enable the system to identify candidate microservices based on user requirements [28], we need to describe the microservices and perform matchmaking with the requirements based on this description. This process allows the system to search for and select the most suitable microservices to fulfill the specific user needs.

6.1 Microservice Description Generation

In Microservice Definition (see Chapter 3), we define a microservice as composed of the following four elements: Controller, Service, Repository, and DB Model (see Figure 3.1). In Service Decomposition (see Chapter 4), we utilize the semantic annotations provided by Spring Framework and JPA to determine the roles of the program elements and store this information into the service component database.

To facilitate the matchmaking, we extract information that can be compared

doi:10.6342/NTU202303287

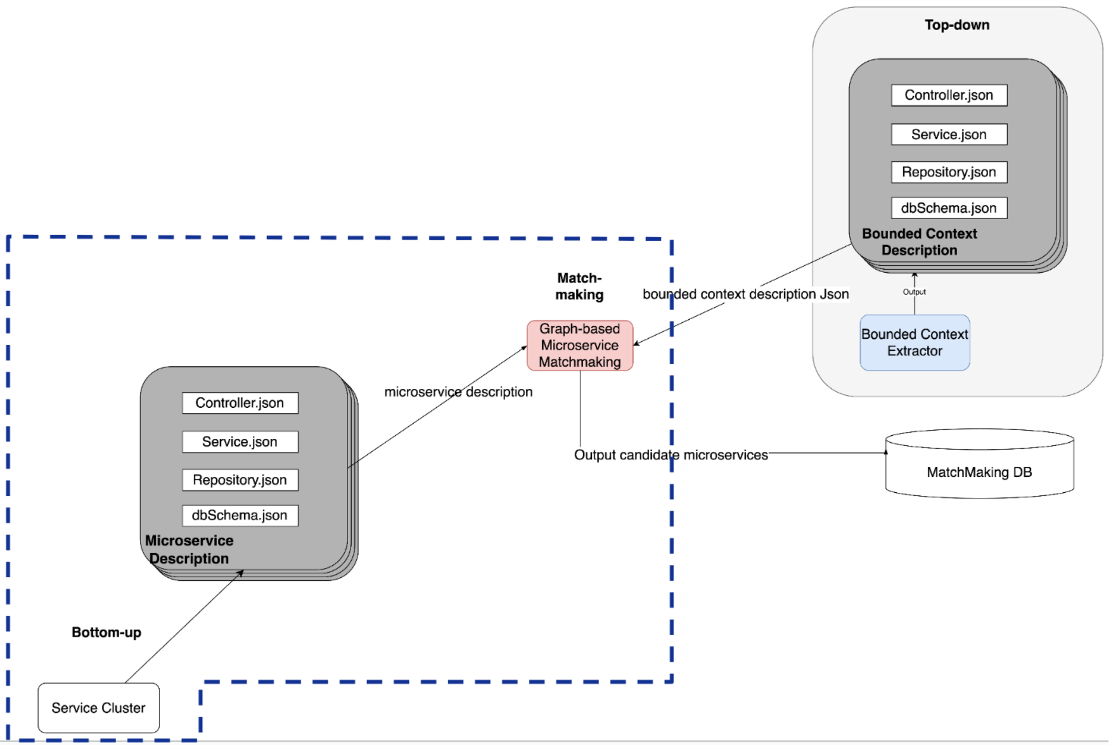


Figure 6.1: Microservice Matchmaking in SBmS

doi:10.6342/NTU202303287

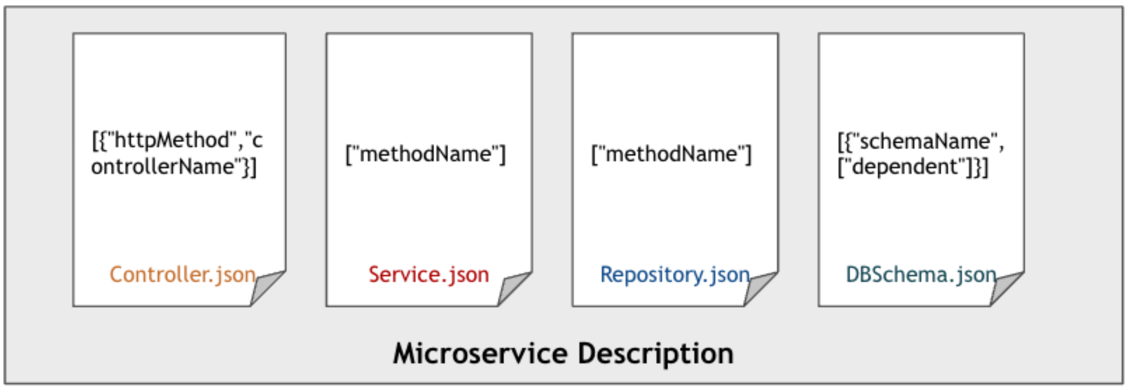


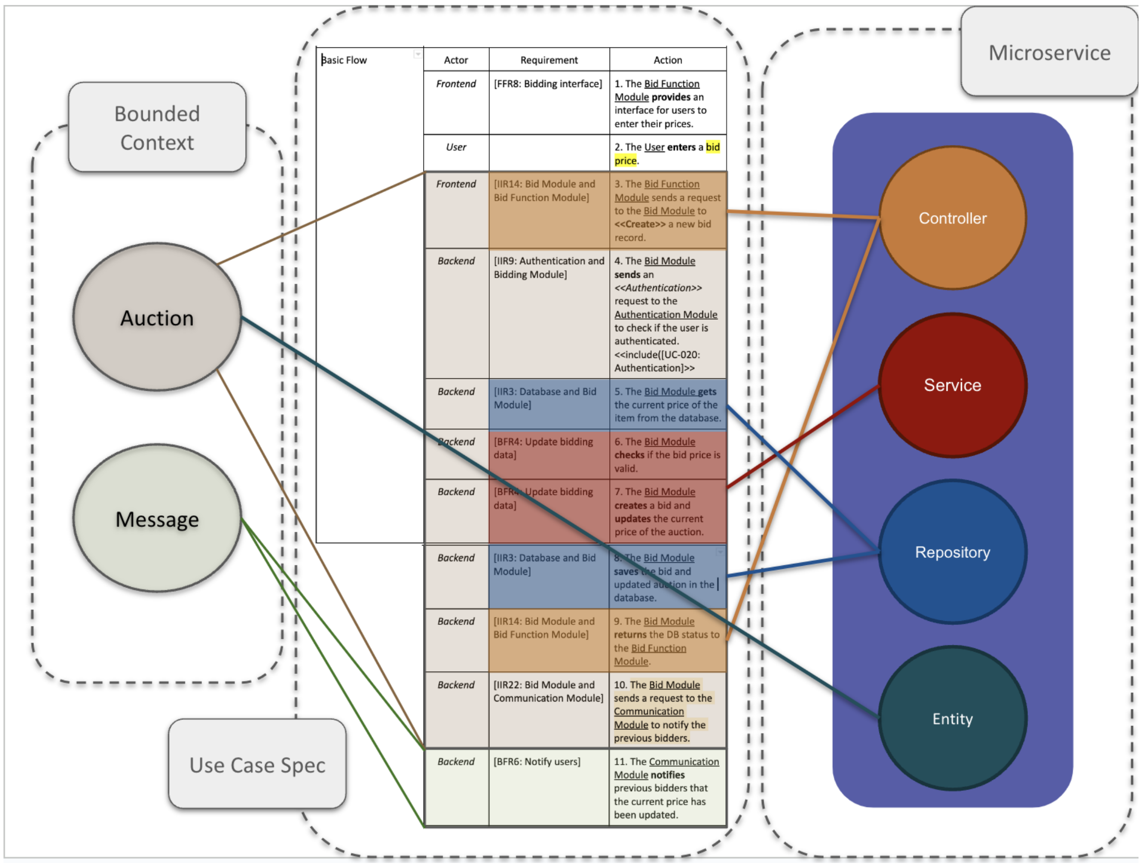
Figure 6.2: Microservice Description

from both sides (source code side and requirement side). These information then be arranged to the JSON [7] file for each part description. By combining the JSON file that describe each part of the microservice, we propose a integral microservice description (Figure 6.2).

6.2 Bounded Context to Microservice Description

In a separate study, Lin [28] proposed a method to derive bounded contexts from requirements. To facilitate matchmaking between bounded contexts and microservices, we let their descriptions share the same fields. This shared information enables us to align bounded contexts with the corresponding microservices. As described in Lin's work, relevant information is then extracted from the descriptions in the bounded contexts to fill in the fields in the description files.

doi:10.6342/NTU202303287



Basic Flow	Actor	Requirement	Action
	Frontend	[FFR8: Bidding interface]	1. The Bid Function Module **provides** an interface for users to enter their prices.
	User		2. The User **enters** a bid price.
	Frontend	[IIR14: Bid Module and Bid Function Module]	3. The Bid Function Module sends a request to the Bid Module to **<<Create>>** a new bid record.
	Backend	[IIR9: Authentication and Bidding Module]	4. The Bid Module **sends** an *<<Authentication>>* request to the Authentication Module to check if the user is authenticated. <<include([UC-020: Authentication]>>
	Backend	[IIR3: Database and Bid Module]	5. The Bid Module **gets** the current price of the item from the database.
	Backend	[BFR4: Update bidding data]	6. The Bid Module **checks** if the bid price is valid.
	Backend	[BFR4: Update bidding data]	7. The Bid Module **creates** a bid and **updates** the current price of the auction.
	Backend	[IIR3: Database and Bid Module]	8. The Bid Module **saves** the bid and updated auction in the database.
	Backend	[IIR14: Bid Module and Bid Function Module]	9. The Bid Module **returns** the DB status to the Bid Function Module.
	Backend	[IIR22: Bid Module and Communication Module]	10. The Bid Module sends a request to the Communication Module to notify the previous bidders.
	Backend	[BFR6: Notify users]	11. The Communication Module **notifies** previous bidders that the current price has been updated.

Figure 6.3: Mapping between Bounded Context, Use Case Specification and Microservice Description

doi:10.6342/NTU202303287

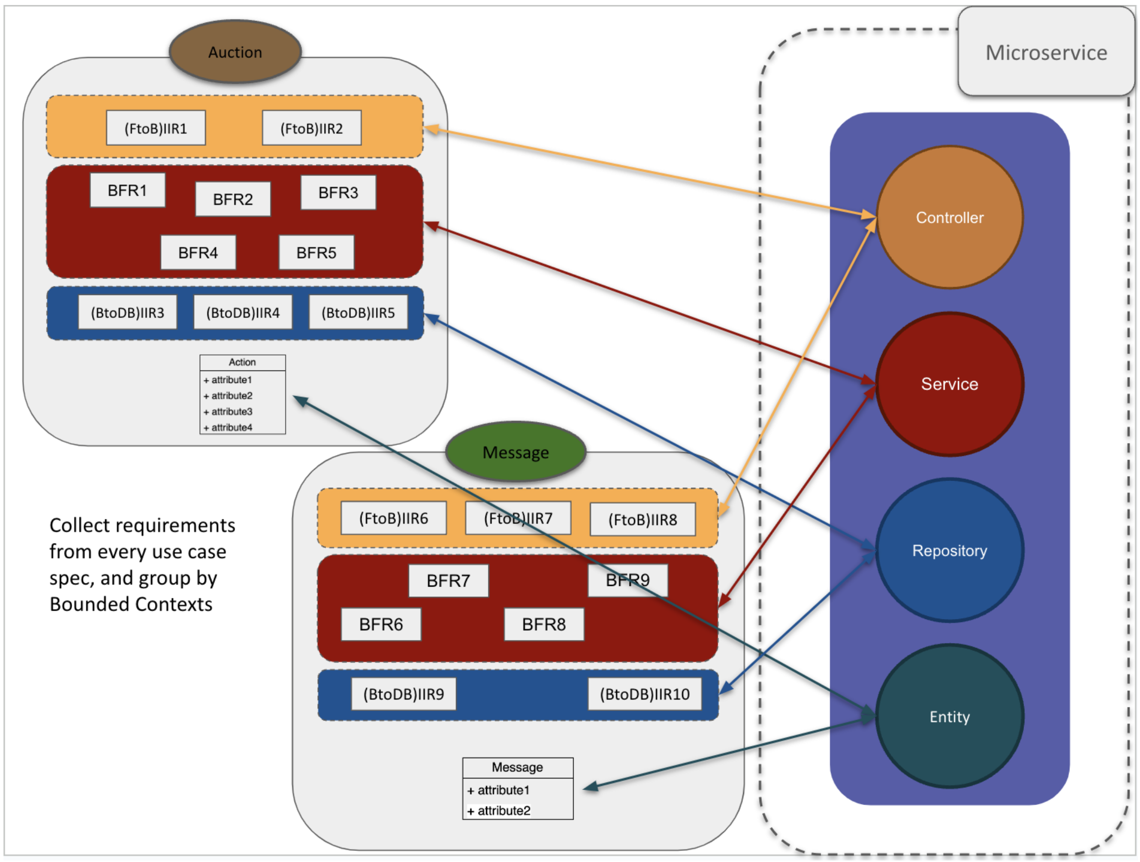


Figure 6.4: Bounded Context to Microservice Description

doi:10.6342/NTU202303287

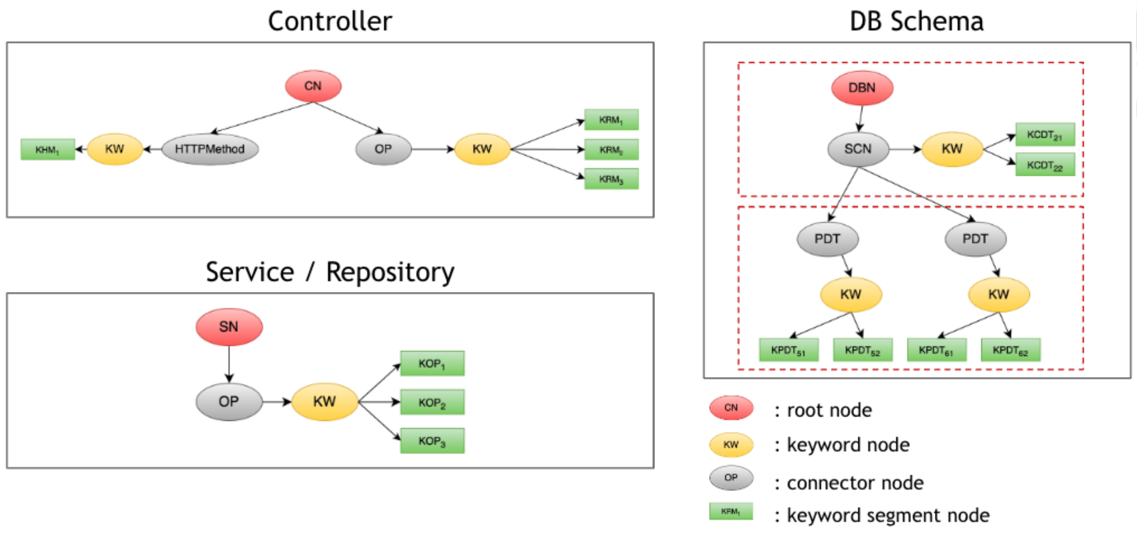


Figure 6.5: Graph Structure for Each part of description

6.3 Graph-based Microservice Matchmaking

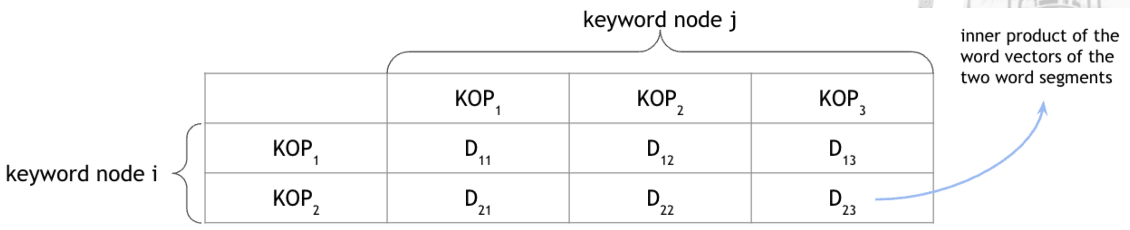
The matchmaking algorithm is based on previous research [27], but we have expanded its matchmaking level from method to microservice.

After transforming the microservice description into a graph, we proceed to calculate the distances between the keyword segments, keyword nodes, connector nodes, and the entire graph. The details of the distance calculation method will be described below.

6.3.1 Distance of keyword node

The distance calculation between two keyword segments involves the following steps:

doi:10.6342/NTU202303287


keyword node j
keyword node i
inner product of the word vectors of the two word segments

	KOP_1	KOP_2	KOP_3
KOP_1	D_{11}	D_{12}	D_{13}
KOP_2	D_{21}	D_{22}	D_{23}

Table 6.1: keyword segment distance matrix

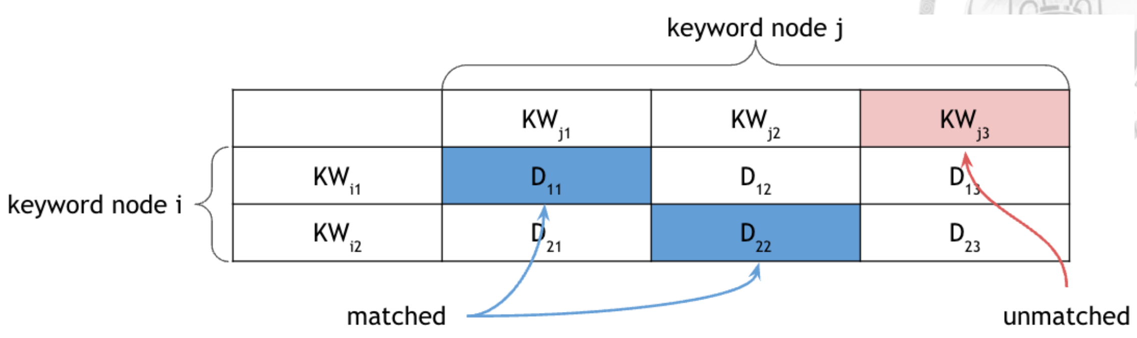
1. Utilizing a Python wordsegmentation [16] to segment the keyword into a list of individual keywords.
   - ex. getItem → [get, item]
2. Utilizing WordToVec [15] to convert each keyword segment into a corresponding vector representation.
   - Here we employ the pre-trained model "GoogleNews-vectors-negative300" provided by Google
3. Calculating the distance between keyword segment with the inner product of the two vectors (see Table 6.1)

After calculating the distance between each keyword segment, we use Hungarian algorithm to find best match, and define the distance between keyword nodes as the sum matched keyword segments distances and the average of unmatched distances.

Take the Table6.2 as example, assume KW_{i1} has best match with KW_{j1}, KW_{i2} has best match with KW_{j2}, and KW_{j3} does not matched with and keyword segment, then the distance between keyword node i and j would be: $D_{11}+D_{22}+(D_{13}+D_{23})/2$

.

doi:10.6342/NTU202303287


keyword node j
keyword node i
matched
unmatched

	KW_{j1}	KW_{j2}	KW_{j3}
KW_{i1}	D_{11}	D_{12}	D_{13}
KW_{i2}	D_{21}	D_{22}	D_{23}

Table 6.2: keyword segment matching

6.3.2 Distance of connector node

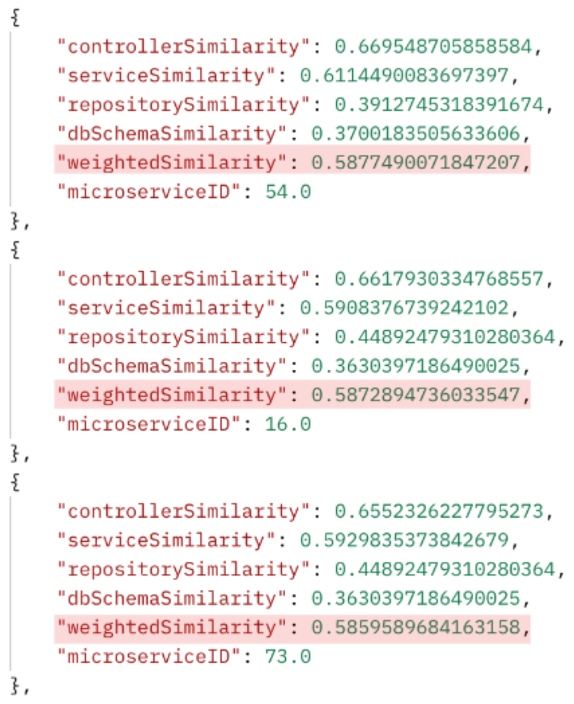
After calculating the distance between keyword nodes, we apply the same concept again to calculate the distance between connector nodes, using a second-pass Hungarian algorithm.

Among them, the DB Schema graph is composed of the schema part and the dependent fields part. By calculating the distance of each part, we can compute the distance between the graphs with a weighted average of both parts.

6.3.3 Graph Similarity calculation

Afterwards, we calculate the similarity for each element using the elastic-based distance transformation equation proposed in [27]. Finally, we determine the similarity between microservices by computing the weighted average of the similarity scores for the four elements.

doi:10.6342/NTU202303287



```
{
    "controllerSimilarity": 0.669548705858584,
    "serviceSimilarity": 0.6114490083697397,
    "repositorySimilarity": 0.3912745318391674,
    "dbSchemaSimilarity": 0.3700183505633606,
    "weightedSimilarity": 0.5877490071847207,
    "microserviceID": 54.0
},
{
    "controllerSimilarity": 0.6617930334768557,
    "serviceSimilarity": 0.5908376739242102,
    "repositorySimilarity": 0.44892479310280364,
    "dbSchemaSimilarity": 0.3630397186490025,
    "weightedSimilarity": 0.5872894736033547,
    "microserviceID": 16.0
},
{
    "controllerSimilarity": 0.6552326227795273,
    "serviceSimilarity": 0.5929835373842679,
    "repositorySimilarity": 0.44892479310280364,
    "dbSchemaSimilarity": 0.3630397186490025,
    "weightedSimilarity": 0.5859589684163158,
    "microserviceID": 73.0
},
```

Figure 6.6: example of matchmaking response

doi:10.6342/NTU202303287

Chapter 7

Conclusion

In this work, we proposed a service generation process involving the following steps：

1. Parsing the source code to create a data model.
2. Extracting dependencies between project elements.
3. Determining the role of the program element in the project based on semantic annotation.
4. Generating WSDL for the service components' description.
5. Using the encapsulation-based metric to perform API clustering
6. Repacking the service components to a microservice based on Object Oriented concept

doi:10.6342/NTU202303287

7. Deploying the project, registering it on the Eureka server and exposing the server with the API gateway

Furthermore, we proposed the microservice matchmaking process involving the following steps:

1. Generating the microservice description
2. Transforming the microservice description into a graph representation.
3. Performing the matchmaking algorithm to calculate the similarity between the bounded context-extracted microservice and the microservices we have parsed.

doi:10.6342/NTU202303287

Chapter 8

Future Work

Currently, our approach has successfully accomplished microservice matchmaking, which narrows the search space when a UI component needs to bind to a service. We can compare the WSDL generated in the Service Decomposition (see Chapter 4) with the required service description generated by the UI component. By calculating the similarity between them, we can determine which service is the most suitable for the description and return the corresponding service ID to the frontend. The frontend component utilizes the service ID information to send HTTP requests to our API gateway, enabling it to invoke the corresponding microservice.

doi:10.6342/NTU202303287

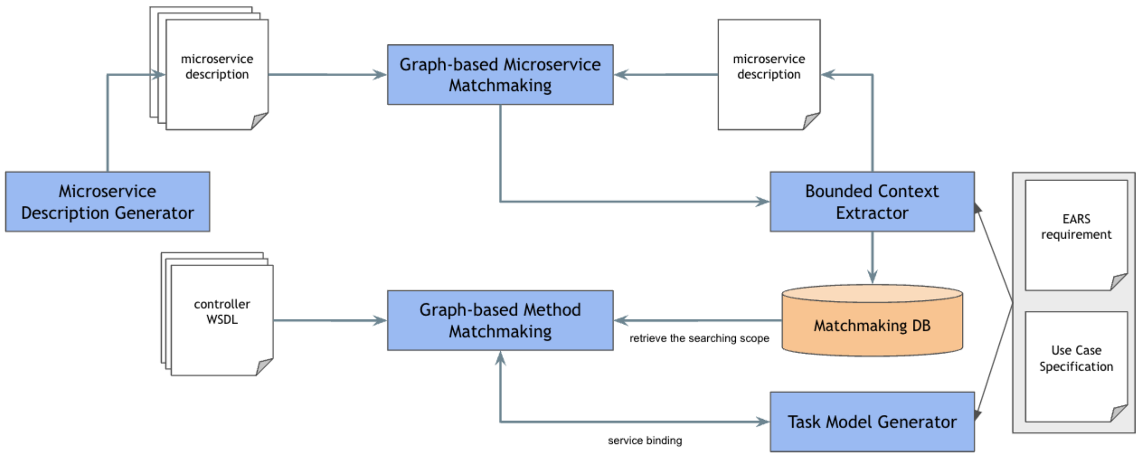

microservice description
Graph-based Microservice Matchmaking
microservice description
Microservice Description Generator
Bounded Context Extractor
EARS requirement
controller WSDL
Graph-based Method Matchmaking
retrieve the searching scope
Matchmaking DB
Use Case Specification
Task Model Generator
service binding

Figure 8.1: UI Component Service Binding

doi:10.6342/NTU202303287

Bibliography

[1] Gradle build tool. https://gradle.org/.

[2] Java compiler api. https://docs.oracle.com/javase/8/docs/api/javax/tools/JavaCompiler.html.

[3] Java compiler tree api. https://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/.

[4] Java persistence api. https://docs.oracle.com/javaee/7/tutorial/persistence-intro.htm.

[5] Java reflection api. https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html.

[6] Javaparser. https://javaparser.org/.

[7] Json. https://www.json.org/.

[8] Maven build tool. https://maven.apache.org/.

[9] Open api specification. https://swagger.io/specification/.

[10] schema.org. https://schema.org/.

[11] Spring cloud. https://spring.io/projects/spring-cloud.

[12] Spring cloud gateway.

[13] Spring framework. https://spring.io/.

[14] Web service description language. https://www.w3.org/TR/wsdl.html.

doi:10.6342/NTU202303287

[15] word2vec. `https://code.google.com/archive/p/word2vec/`.

[16] wordsegmentation. `https://pypi.org/project/wordsegmentation/`.

[17] L. Baresi, M. Garriga, and A. De Renzis. Microservices identification through interface analysis. In *Service-Oriented and Cloud Computing: 6th IFIP WG 2.14 European Conference, ESOCC 2017, Oslo, Norway, September 27-29, 2017, Proceedings 6*, pages 19–33. Springer, 2017.

[18] R. Chen, S. Li, and Z. Li. From monolith to microservices: A dataflow-driven approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475. IEEE, 2017.

[19] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216, 2017.

[20] E. Evans. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004.

[21] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann. Service cutter: A systematic approach to service decomposition. In *Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings 5*, pages 185–200. Springer, 2016.

[22] M.-H. Hsieh. Construct service components from java-based open source projects. Master's thesis, National Taiwan University, 2021.

[23] J.-W. Huang. Generate web application servers with bpel processes. Master's thesis, National Taiwan University, 2022.

[24] S.-W. Huang. Towards a solution to iot interoperability through reverse engineering. Master's thesis, National Taiwan University, 2017.

[25] P. Kolb. Disco: A multilingual database of distributionally similar words. 2008.

doi:10.6342/NTU202303287

[26] H. W. Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[27] J. Lee, K.-H. Hsu, S.-P. Ma, and C.-A. Lee. Service discovery through elasticity-based graph matching. 2018.

[28] Y.-L. Lin. From requirements to microservice: A domain driven approach with machine learning. Master's thesis, National Taiwan University, 2023.

[29] T.-C. Lu. Develop web applications through service components repacking. Master's thesis, National Taiwan University, 2022.

[30] C. Malzer and M. Baum. A hybrid approach to hierarchical density-based cluster selection. In *2020 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*. IEEE, sep 2020.

[31] W.-L. Shih. Construct service components from java-based open source projects. Master's thesis, National Taiwan University, 2022.

[32] J.-J. Yu. Construct service components from open source java projects. Master's thesis, National Taiwan University, 2021.

doi:10.6342/NTU202303287