Refactoring Java Monoliths into Executable Microservice-Based Applications

Francisco Freitas
a81580@alunos.uminho.pt
University of Minho & INESC TEC
Portugal

André Ferreira
alferreira@di.uminho.pt
University of Minho & Bosch Car Multimedia S.A.
Portugal

Jácome Cunha
jacome@di.uminho.pt
University of Minho & INESC TEC
Portugal

Abstract

The way we develop software is constantly changing. In the last few years, we have seen a drastic change, where large-scale software projects are being assembled by a flexible composition of many (small) components. Thus, currently, most applications are developed by assembling, and many times reusing, several "small" components, possibly written in different programming languages and deployed anywhere in the cloud – the so-called microservice-based applications.

The dramatic growth in popularity of microservice-based applications has pushed several companies to apply major refactorings of their software systems. However, this is a challenging task that may take several months or even years.

In this paper we propose a methodology, supported by a tool termed MicroRefact, to automatically evolve a Java monolithic application into a microservice-based one. Our methodology receives as input the Java application and a proposition of microservices and refactors the original classes to make each microservice independent. Our methodology creates a REST API for each method call to classes that are in other services. The database entities are also refactored to be included in the corresponding service. The initial evaluation shows that our tool can successfully refactor 80% of the Java applications tested.

CCS Concepts: **• Software and its engineering → Software maintenance tools**; **• Computer systems organization → Cloud computing**.

Keywords: microservice architecture, microservice-based applications, monolithic decomposition, refactoring, Java

SBLP'21, September 27-October 1, 2021, Joinville, Brazil
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9062-0/21/09…$15.00
https://doi.org/10.1145/3475061.3475086

ACM Reference Format:
Francisco Freitas, André Ferreira, and Jácome Cunha. 2021. Refactoring Java Monoliths into Executable Microservice-Based Applications. In *25th Brazilian Symposium on Programming Languages (SBLP'21), September 27-October 1, 2021, Joinville, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3475061.3475086

1 Introduction

"The death of big software" has been announced in 2017 [1]. This was in part caused by the rise of the cloud and "small" software. Small software is built in simpler and more integrateable pieces which provide a flexibility big software does not. These pieces can be used to mix-and-match as to create new or even to evolve existing software [1]. Indeed, software is currently being developed as a set of loosely-coupled components, eventually deployed anywhere in the cloud, and communicating through the internet [16]. This has been motivated by the challenges associated with the development, maintenance, and evolution of large software systems, but also by the appearance of the cloud and the ease it brought in terms of horizontal scaling, reusability and flexibility in ownership and deployment.

An example of how to manage this new way of developing software is the popular architectural style termed microservices [16]. Microservice-based applications consist of (small) services that focus on a single functionality. One of the main motivations of a microservice architecture is that it has the potential to increase the flexibility and agility of software development as each service can be developed individually using different technologies. These services communicate with each other through some lightweight communication mechanism (e.g. HTTP REST requests) [16]. Although the advantages of the development of this kind of applications do surpass the disadvantages, the fact is that there are still many applications that were built as monoliths, that is, applications composed of all the core logic related to the domain of the problem contained in a single process [16]. The manual process of migrating them to this new paradigm is complex and, depending on the project's complexity, may take months or even years to complete [7, 15]. The decomposition of software systems is one of the main struggles, and as shown in the work of Fritzsch et al. [7], none of the participants in the study was aware of automated techniques that could assist the migration of a monolithic application to a microservice-based one. Thus, the research community has been working on techniques and methodologies to aid in this migration, i.e. in transforming a monolithic application into a microservice-based one, while preserving the semantics of the original application [4, 8–11, 14]. Moreover, several companies have also applied major refactorings of their backend systems to transform their applications [15]. More about related work can be found in Section 5.

Most of the previous works are focused on the identification of the services, but lack the step of actually refactoring the application to make it a microservice-based one. In this work we present a methodology, supported by a tool termed MicroRefact, that receives as input a list and composition of microservices for a Java monolithic application, and refactors that original application into a microservice-based one. Our methodology analyzes the source code and the services proposed and refactors the classes that have method calls to other classes that are part of other services. Each of these calls is replaced by a call to a new method we automatically generate, implementing a REST call to the original method which now is a different service. We also refactor the database classes as they need to be spread by the different services too. In Section 2 we present our methodology in detail and in Section 3 its implementation.

We performed an initial evaluation using 10 open-source projects extracted from GitHub. MicroRefact automatically refactored 8 of the 10 projects and MicroRefact can be further extended to include the remaining 2 projects. Section 4 presents in detail our evaluation. In Section 6 we draw our conclusions and describe some possible future work.

2 Refactoring Java Monoliths

In this section, we present the proposed methodology for refactoring Java monoliths. The proposed methodology is directed to the applications that use object-relational mapping (ORM) between database entities and Java classes, in particular applications that make the mapping between classes and entities through annotations in the source code. To demonstrate what transformations our methodology makes to monolithic projects, we use as an example a Spring Java application called *restaurantServer*[1], which is a backend application for restaurant management.

Our methodology receives as input the source code of the application under analysis and a microservices proposal. The microservices proposal is a set of lists in which each list represents a microservice containing the name of the classes that form the microservice. Our methodology has as its output a microservices-based application that from the functional point of view is the same as the monolithic application under analysis.

[1]https://github.com/asledziewski/restaurantServer

Figure 1 depicts the steps of the refactoring process. In the Information Extraction phase, we extract the structural information from the source code of the project under analysis and combine it with the microservices proposal to identify the dependencies between microservices.

In the next step, Database Refactoring, we use the structural information and the dependencies between microservices to identify entities, relationships between entities, and to identify which relationships need refactoring, proceeding to the refactoring of those relationships.

Finally, in Code Refactoring, we use the structural information and the dependencies between microservices to analyze the class variables and the dependencies between classes, in order to identify and refactor the classes that have dependencies with classes that belong to different microservices.

2.1 Information Extraction

The building blocks of our methodology are the structural information and proposed microservices. In the Information Extraction phase structural information is extracted from the source code and dependencies between microservices are identified. We describe next how to obtain this information.

2.1.1 Extraction of Structural Information. To identify the dependencies between microservices, it is necessary to identify the dependencies between classes and relate the dependencies with the microservices proposal. Through the structural information of the source code we identify the dependencies between the classes. We perform extraction of structural information in a structured version of the source code: its underlying Abstract Syntax Tree (AST). For each class in the project we extract the following information from the AST: the list of imports, the list of implemented interfaces, the class from which it extends, if applicable, the list of annotations, the list of variables, the list of methods and the list of methods invoked from other classes. To identify the classes a class depends on and to create the dependency list for each class, we combine the list of invoked methods, the list of implemented interfaces and information about extends. The dependency list contains the name of the classes from which the class invokes methods, the name of the interfaces it implements and the name of the superclass, if applicable.

For example, from the *restaurantServer* class `MailService` we extract the following information :

Name *: MailService*

Imports*: [org.springframework.beans.factory.annotation.Autowired, org.springframework.mail.javamail.JavaMailSender, org.springframework.mail.javamail.MimeMessageHelper, org-.springframework.stereotype.Service, javax.mail.MessagingException, javax.mail.internet.MimeMessage]*

Implements *: []*

Extends *: []*

Annotations *: [@Service]*

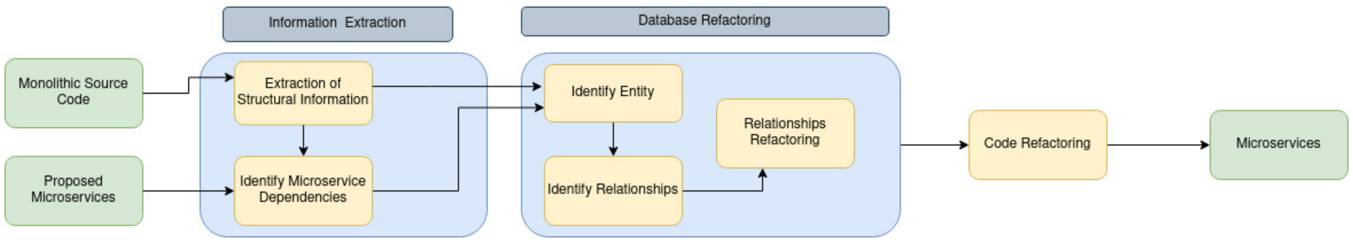

Figure 1. Overview of the proposed methodology

Variables *:[{ "annotations": [], "modifier": private , "identifier": [], "type": JavaMailSender, "variable": javaMailSender]}]*

Methods *: [{"name": sendEmail, "annotations": [], "returnDataType": [void], "identifier": [], "parametersDataType": ["type": String, "variable": destination, "type": String, "variable": subject, "type": String, "variable": content], "variables": ["type": MimeMessage, "variable": mail, "type": MimeMessageHelper, "variable": helper], "body": MimeMessage mail = javaMailSender-.createMimeMessage(); try MimeMessageHelper helper = new MimeMessageHelper(mail, true); helper.setTo(destination); helper.setReplyTo("restaurantprojectPZ@gmail.com"); helper.setFrom ("restaurantprojectPZ@gmail.com"); helper.setSubject(subject); helper.setText(content); catch (MessagingException e) e.print- StackTrace(); javaMailSender.send(mail);}]*

MethodsInvocation *: []*

Dependencies *: []*

2.1.2 Identify Microservice Dependencies. Since the composition of the microservices given as input may not have followed any microservice identification methodology, it is necessary to check if there are dependencies between microservices. We define dependencies between microservices as a reference to a certain non-primitive type that does not belong to the microservice. By comparing the list of dependencies of a class with the list of classes that form the microservice to which the class belongs, we observe that the classes that are not in both lists correspond to dependencies between microservices. Thus, a list is created for each class, containing the name of the classes that the class depends on and belong to different microservices, which represents the list of dependencies of the class with other microservices.

With the lists of dependencies between microservices generated for each class, it is sometimes necessary to make adjustments to the microservice proposal given as input. If the microservices proposal indicates that an interface implemented by a class is in different microservices, we replicate the interface and place the copy in the microservice where the implementing class belongs, since an interface only provides the signature of the methods that a class must implement, not having a significant impact on the microservice domain. On the other hand, regarding inheritance, our methodology does not allow the super class and sub classes to be in different microservices because they have an "is a" relationship and must belong to the same domain and therefore the same microservice.

2.2 Database Refactoring

One of the big challenges of migrating a monolithic system to microservices is database refactoring. It is necessary to consider issues of transactional integrity, referential integrity, joins, latency, and more [17]. The database refactoring phase aims to identify entities, relationships between entities, and refactoring the relationships between entities that belong to different microservices.

Using the structural information extracted in the previous phase we identify the classes that are mapped as entities and the relationships between the entities. We use the annotation list of each class to identify the classes that are mapped as entities and through the annotations of the instance variables we identify the relationships.

Table 1 shows the entities and relationships present in *restaurantServer*. The logical schema of the database is defined by 7 classes and 6 relationships.

Entity	Relationship	Entity
User	Many-to-Many	Role
User	One-to-Many	Reservation
Bill	One-to-Many	BillPosition
BillPosition	Many-to-One	Dish
RTable	One-to-Many	Reservation
RTable	One-to-Many	Bill

Table 1. Relationship between entities

2.2.1 Relationships Refactoring. With the breakdown of the monolith into microservices, it is necessary to check the integrity of the relationships between entities. As we are in the scope of applications that use annotations for mapping between classes and entities, when relationships between entities are identified, in terms of code this translates into a dependency between classes that needs to be handled. By refactoring the classes involved in the relationship we refactor the relationship of the database entities. Our methodology maintains the relationships between entities that belong to different microservices, using foreign keys to secure these relationships.

We use a relationship found in *restaurantServer* to demonstrate how our methodology refactors relationships. Figure 2 shows the one-to-many relationship between RTable and Reservation in the monolithic system. We omit some attributes from the RTable and Reservation because it has no impact on the refactoring of the relationship.

Both classes have the Entity annotation that indicates that they are mapped as entities and their instance variables are mapped as attributes. The Reservation table has a foreign key that corresponds to the RTable's primary key. In terms of code, the one-to-many relationship is represented by the RTable class having an instance variable of type list of Reservation with OneToMany annotation and the Reservation class having an instance variable to store a primary key of RTable.

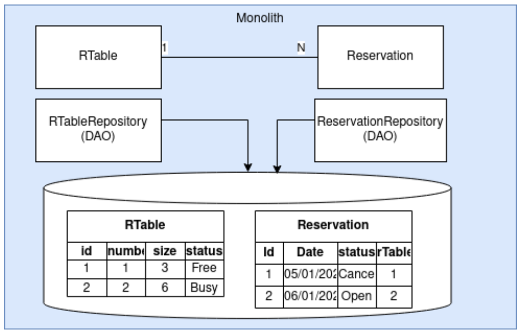

Figure 2. One-to-many relationship between RTable and Reservation in monolithic application

Let us assume that the microservice proposal given as input indicates that RTable and Reservation are in different microservices and therefore the classes are in different environments, leading to the fact that the Reservation type does not exist in the RTable microservice and therefore, to maintain the relationship at the code level and not be mapped to the database, it is necessary to remove the OneToMany annotation from the list of Reservation of the RTable class and create the Reservation type in the RTable microservice. We use *Data Transfer Object* (DTO) pattern [5] to create Reservation type in RTable microservice. Full replication of the Reservation class would create an entity in the database with the same name in the RTable microservice.

In the monolithic version, when information about RTable is retrieved, the information of the Reservation associated with the RTable is also retrieved because the database performs the join of information. In the microservices this is not possible because the information lives in different databases.

We apply the *Move Foreign-Key Relationship to Code* pattern [17], to move the join operation to code. By moving the join operation to the code the database calls are replaced by service calls and the primary key is used to filter the information that is retrieved. In the case of the relationship between RTable and Reservation, in microservices, when information from an RTable is retrieved, a call is made to the Reservation microservice with the primary key of the RTable to retrieve the reservations that have as foreign key the primary key of the RTable. To apply the *Move Foreign-Key Relationship to Code* pattern [17], in the relationship between RTable and Reservation, we use the structural information extracted from the AST to identify in the class RTable the methods that use the list of Reservation. Since this is a class that is mapped as an entity, typically it only has getters and setters (methods).

With the methods identified we create an interface termed Request with the methods signature and the class named RequestImpl that implements the interface which is responsible for making REST calls to the Reservation microservice. A new variable is added to the RTable class that has the type of the interface created and the body of the identified methods is changed to make calls to the Reservation microservice by invoking the methods of the Request class.

In the Reservation microservice, it is necessary to create a REST API to respond the requests made by RTable microservice. To create a API, a class named Controller is created in which the resource paths for the requests are defined, and another class called Service is created to process the information from the requests and invoke the methods of the Reservation's Data Access Object (DAO). Finally, methods with the same name as the methods identified in the RTable class are added to the DAO class ReservationRepository. These methods take as argument the primary key of the RTable to be used as a filter. Figure 3 show the relationship between RTable and Reservation after refactoring.

In the example we use a one-to-many relationship, but the procedure also applies to other relationships, in particular to many-to-one and one-to-one relationships. The many-to-many relationship is a special case. In many-to-many relationships a "join table" is created, being formed by the two foreign keys (i.e. copies of the primary keys of the entities involved). If we apply the same procedure as we just explained we would lose this table. To keep the relationship intact we apply the *Database Wrapping Service* pattern [17]. Doing so, we create a new microservice to "hide" the database that contains the relationship. This new microservice provides an API to access the data stored in the database and, therefore, the microservices that need to access that data, replace direct database calls with calls to the new microservice.

2.3 Code Refactoring

In the code refactoring phase we use the variables and dependency list of each class to identify method invocation of classes that belong to other microservices. Our methodology aims to replace the local method invocation by a service call, respecting the data flow of the monolithic system. Figure 4 illustrates an overview of code refactoring process.

As in the previous phase, we use an example from *restaurantServer* to demonstrate the code refactoring process. Bill-

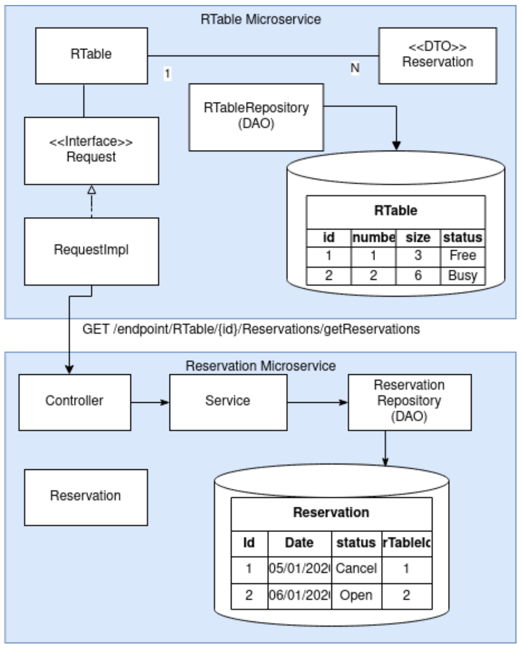

Figure 3. One-to-many relationship between RTable and Reservation in microservice-based application

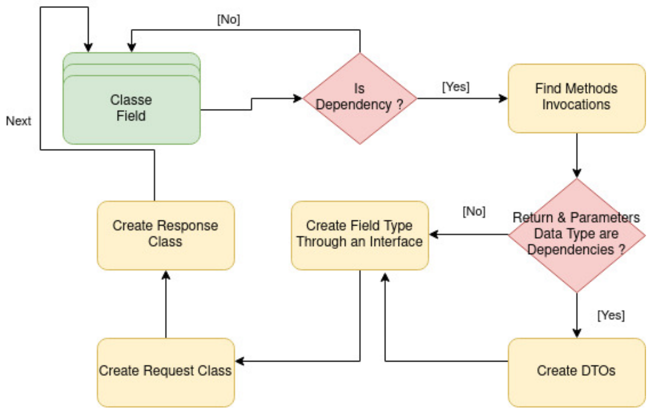

Figure 4. Overview of code refactoring

Service is a class that has two instance variables, one of type BillRepository and another of type RTableRepository, 6 methods: getBills, getBillById, addBill, updateBill, deleteBill and getBillPositions, and has two classes in the dependency list: RTable and RTableRepository.

The code refactoring process starts with the analysis of the data type of the instance variables to check if the types are among the dependencies between microservices of the class. The class BillService has an instance variable of type RTableRepository that belongs to another microservice.

Next, a search is made for method invocations that belong to the class of the instance variable type. For the case of the class BillService, inside of addBill method, an invocation of the findById method of the RTableRepository class was found.

For each identified invoked method it is verified if the return data type and the parameters data type are among the dependencies of the class with other microservices, and if so we apply the DTO pattern [5] to replicate the class it depends on. The findById method has as return data type Optional<RTable> and so, the RTable class is replicated to the microservice which the BillService class belongs to.

Next, an interface is created to represent the type of instance variable, in which the signatures of the identified invoked methods are declared, and a class is created that implements the interface that is responsible for making the remote service calls. Using the BillService example to demonstrate, an interface called RTableRepository, which contains the findById method signature, is created and added to the BillService microservice, and an RTableRepositoryImpl class is generated which declares the findById method as a call to the RTableRepository microservice. In this way the replacement of local method invocation with service method invocation is transparent to the BillService class. Finally, on the microservice side of the RTableRepository it is created a REST API that allows the invocation of methods.

Figure 5 shows the code refactoring and the invocation of the findById method by BillService via a service call. In the RTable microservice the class RTableRepositoryController is created to define the resource paths for the requests and to invoke the original findById.

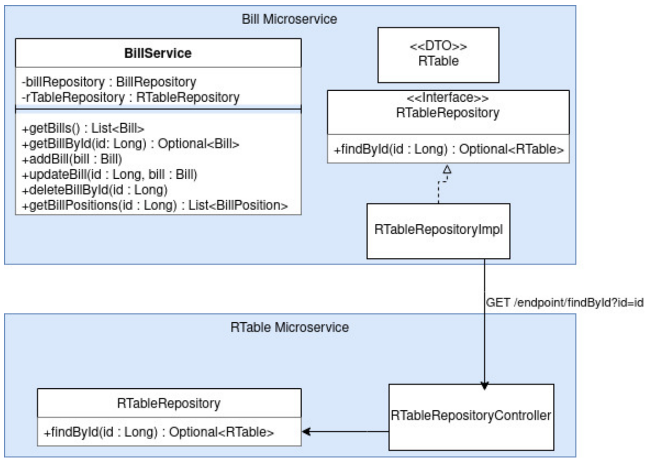
Bill Microservice
BillService
-billRepository : BillRepository
-rTableRepository : RTableRepository
+getBills() : List<Bill>
+getBillById(id: Long) : Optional<Bill>
+addBill(bill : Bill)
+updateBill(id : Long, bill : Bill)
+deleteBillById(id : Long)
+getBillPositions(id : Long) : List<BillPosition>
<<DTO>> RTable
<<Interface>> RTableRepository
+findById(id : Long) : Optional<RTable>
RTableRepositoryImpl
GET /endpoint/findById?id=id
RTable Microservice
RTableRepository
+findById(id : Long) : Optional<RTable>
RTableRepositoryController

Figure 5. BillService's invocation of the findById method of class RTableRepository through service call

For the refactoring process to be complete, we analyze whether there are dependencies in the local variables. If so, we apply the DTO pattern [5] to create the types and check for method invocations by these variables. If method invocations are found and the methods invoked are not declared in the DTO, we declare them. These methods make calls to the microservice that has the type of the local variable to respect the data flow.

3 MicroRefact

In this section we present the implementation of our tool, MicroRefact, which was built upon the methodology presented. MicroRefact serves as a proof of concept to validate the methodology. The MicroRefact is designed for Java applications, in particular for the Spring framework. Regarding the ORM, there are several ORM frameworks. MicroRefact supports the refactoring of applications that use the *Java Persistence API* [2] (JPA) to do the mapping. We decided to go with JPA since it describes a common interface to data persistence frameworks. The full implementation is available at https://github.com/FranciscoFreitas45/MicroRefact. Figure 6 represents an overall flow of MicroRefact.

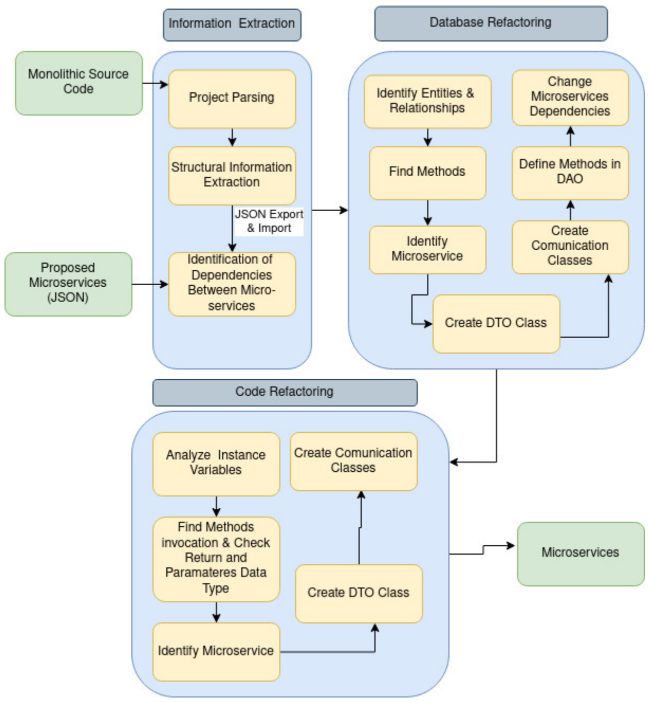
Information Extraction
Database Refactoring
Monolithic Source Code
Project Parsing
Structural Information Extraction
JSON Export & Import
Proposed Microservices (JSON)
Identification of Dependencies Between Micro-services
Identify Entities & Relationships
Change Microservices Dependencies
Find Methods
Define Methods in DAO
Identify Microservice
Create Comunication Classes
Create DTO Class
Code Refactoring
Analyze Instance Variables
Create Comunication Classes
Find Methods invocation & Check Return and Parameteres Data Type
Create DTO Class
Identify Microservice
Microservices

Figure 6. Overall flow of MicroRefact

3.1 Information Extraction

Our tool receives as input a JSON file with the composition of the microservices and the path to the source code of the monolith. The information extraction phase is responsible for processing the input provided by the user in order to identify the proposed microservices and extract information from the source code of the monolith.

The information extraction phase starts by parsing the source code to an AST. We use the *Java Parser*[3] library to do the parsing. The AST contains the following information for each Java file: name, list of imports, list of extends, list with the name of the interfaces it implements, list with the name of classes it depends on, list of annotations, list of instance variables, list of methods and list of invoked methods. The AST is extracted into a JSON file to be processed together with the proposed microservices by a Python program.

A data structure is created to store the information extracted from the AST and to represent the microservices in the program. We create a class called `Cluster` to represent a microservice and a class called `Class` to represent a class. The `Cluster` class is composed by a dictionary, where the key is the name of a class and the value is an object of type `Class`, by a list for adding new classes to the microservice and by the path to the folder where the microservice Java files are created. The structure created is composed of a list of `Cluster`. One object of type `Class` is created for each Java file. These objects contain the information extracted from the AST and are added to the dictionary of the `Cluster` that represents your microservice.

In the next step, to identify the dependencies between microservices for each `Class` object, it is checked if the name of the classes that the class depends on are keys in the dictionary that the `Class` object belongs to. If so, the name of these classes is removed from the list of dependencies of the `Class` object. Thus, the dependency list of a `Class` object represents dependencies with classes that belong to other microservices.

3.2 Database Refactoring

At database refactoring, entities are identified by searching for the word `Entity` among the annotation list of each `Class` object, and the relationships are identified by searching for one of the following words in the annotation list of each instance variable of the `Class` object under analysis: `OneToMany`, `ManyToOne`, `ManyToMany` and `OneToOne`.

If relationships are found, it is verified whether the type of the instance variable that has the annotation is in the object's dependency list and, if so, a search is performed on all methods in the method list of the `Class` object under review to check whether the type of the instance variable is used in the return or in the parameters or in the declaration of variables and a list is created with methods that use the type of the instance variable.

For the application of the DTO pattern [5], the type of the instance variable is used to identify the position (index) in the `Cluster` list, of the `Cluster` object that has in its dictionary a key with the same name as the type of the variable. Then, the index and the type of the variable are used to do a get

[2] https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html

[3] https://javaparser.org/

of the corresponding `Class` object to make a clone of it and add it to the `Cluster` dictionary of the class under analysis.

Next, an interface is created with the signature of the methods that are in the identified list and a new `Class` object is instantiated that represents the class that implements the interface and is responsible for making the service calls. This object is added to the list of new classes of `Cluster`.

Finally, for the communication between microservices to be possible, two objects of type `Class` are instantiated and added to list of new classes of `Cluster`. One uses the methods declared in the interface to define routes and the other processes the requests. For the retrieved data, the signatures of the methods declared in the interface are added to the DAO class. The type of instance variable has come to exist in the microservice through the DTO, so it is removed from all dependency lists of `Class` objects which belong to `Cluster`.

3.3 Code Refactoring

The code refactoring phase has some similarities with the previous phase. The `Class` objects that do not have the `Entity` annotation in the annotation list are analyzed to check if the type of their instance variables are in their dependency list and, if they are, the same procedure used in the previous phase with addition of, in identification of the methods, verification of the return type and the type of the parameters. If these types are in the object's dependency list, a DTO pattern [5] is applied as in the previous phase.

Finally, the information contained in each object `Class` is used to write Java files. The `Class` class has a method called `create`, which generates a Java file with the class information. To do this, Python's built-in `open` function is used to create a file object and a iteration over all the fields in the `Class` is performed to write them to the file using the `write` method of file object. A folder is created for each microservice where the files are written.

4 Evaluation

To quantitatively assess the applicability of our methodology, we collected 10 Java Spring applications from GitHub on which we run MicroRefact. Our main objective is, from a quantitative point of view, to understand the percentage of projects that MicroRefact can automatically refactor.

4.1 Project Collection

We used the GitHub search API for code to find repositories that contained the terms `org.springframework.data.jpa` and `org.springframework.data.jpa.repository.JpaRepository`, since these are very common terms in applications that use JPA annotations to do object-relational mapping and are terms exclusive to applications built with the Spring framework. The query used is as follows:

https://api.github.com/search/code?q=org.springframework.data.jpa+org.springframework.data.jpa.repository.JpaRepository+language:java

Since the GitHub search API limits each request to 1000 results and the query is to identify repositories through code there are repeated results. Executing this query and after removing duplicate repositories, we identified 686 repositories. We also use filters to exclude demo and test projects using the following stop words *{'release', 'framework', 'learn', 'source', 'spring', 'study', 'demonstration', 'test', 'practice', 'practice'}*. That reduced the 686 projects to 353. To ensure that only monolithic applications are used, we only consider projects with one 'src' folder. Within the identified projects, we selected 10 applications with different sizes at random.

4.2 Setup

The microservices proposal is done by the tool available at https://github.com/miguelfbrito/microservice-identification. Although the tool allows the customization of the input, we use the default parameters. For each project the tool generates several proposals for decomposition of the monolith and we always choose the one that reveals the greatest value in the metrics that the tool uses to evaluate the proposed decomposition. Table 2 shows the projects with their number of classes in monolithic version, the number of microservices proposed, and the results after refactoring the projects . The composition of the proposed microservices and the refactored applications are publicly available at https://microrefact.github.io/.

4.3 Results

MicroRefact was able to refactor 8 out of 10 of the applications. In *restaurantServer* one more microservice was created than the number of microservices that the microservices proposal indicated because it has a many-to-many relationship between entities. Five of the eight refactored projects do not have classes generated by the database refactoring because they do not have relationships between entities expressed in the code.

4.4 Discussion

Regarding the results obtained, the two applications that are not refactored present cases that we will take into consideration in the future. The refactoring of the *segue-me* application fails because there are `Entity` classes that do not have a DAO, ending up being an entity that cannot retrieve information from the database or is dead code. In the case of *oa-system*, the refactoring fails because we identify the DAO with the annotation `Repository` which indicates that a class behaves as a repository of the database and this project does not use this annotation. We identified these classes because of the application of the *Move Foreign-Key Relationship to Code* pattern [17], which implies adding new methods on

Projects	#Classes Monolith	#Proposed Microservices	Refactored	#Classes Database Refactoring	#Classes Code Refactoring	#Total Classes	#Total Microservices
restaurantServer	38	7	Yes	20	38	96	8
ProyectoUNAM[4]	110	9	Yes	75	128	313	9
Hospital_Management_System [5]	127	8	Yes	57	71	255	8
segue_me [6]	133	11	No	-	-	133	11
oa_system [7]	175	15	No	-	-	175	15
SDRC-Collect-Web[8]	192	22	Yes	0	64	195	22
Athena[9]	195	16	Yes	0	132	327	16
HotelManageSystem[10]	219	17	Yes	0	131	350	17
coolweather(server)[11]	401	28	Yes	0	129	530	28
HrEsayWebApiPune[12]	603	28	Yes	0	372	975	28

Table 2. Results of refactoring the 10 applications

these classes for the join to be possible. The addition of only two classes in *SDRC-Collect Web* is explained with the fact that the tool that generated the microservices proposal did not consider all the classes of the project to generate the microservices proposal. In fact, only 131 of the 192 classes of the project were considered.

The presented results also show a large addition of classes in the projects. This happens because for each dependency between microservices we create a request and a response class, i.e, we can have several request and several response classes for the communication between two microservices. One possible optimization is to create only one response class for all the request classes of a given microservice.

5 Related Work

Several authors studied the migration process from different perspectives. Balalaie et al. [3], in order to improve the migration planning process and combat the *ad-hoc* aspect, carried out a survey of design patterns through the analysis of migration processes of industrial-caliber applications. In this work, the authors analyze the entire migration process, from the identification of the architecture to the process of deploy. In [13] the authors discuss the requirements for a model-driven approach for the migration. They propose a set of metrics that can be used to guide the process. In [2] the authors propose a framework to support the decision of migrating or not a monolithic application. The framework is based on facts and metrics collected by the entity that intends to do the migration. In our work we focus on the refactoring step assuming the user has already handled the remaining phases of the migration.

Fowler and Lewis [6] suggest an incremental migration, which consists of the gradual construction of a new application by extracting features from the monolith thus avoiding a "big bang" rewrite. The generated application consists of a set of microservices that interact with the monolithic application. Over time, the number of features implemented by the monolith tends to decrease, as these are migrated to microservices, until the monolith disappears and becomes a microservice-based application. Given we are proposing an automatic approach, our migration is done all at the same time. However, it could also be done partially too, if the set of microservices given as input is also partial.

One of the the challenges in these migrations is the identification of the services existing in monolithic applications. The techniques proposed can be divided into three categories: static, dynamic, and model-based approaches. Static analysis techniques are promising given the amount of information that can be extracted from the source code [11, 14]. Dynamic analysis techniques have emerged as an alternative to static analysis using program execution analysis (e.g., logs) in order to obtain extra information about the software in question [9, 10]. Model-based solutions allow the use of models to support migrations since models also represent a view over the interactions between system's components [4, 8]. Tyszberowicz et al. [18] propose a different approach based on the specification of use cases for the software requirements and a functional decomposition of those requirements. Using text analysis tools the nouns and verbs are extracted from the specifications of the use cases in order to obtain information about the operations of the system, as well as state variables. Using this information they identify clusters of components, and consequently the candidates for microservices. Previous work is mostly focused on the identification of microservices and no tool has been proposed that can identify and specially refactor a whole system into a working version of a microservices application. Our work receives as input the results of microservices identification and proceeds with the refactoring of the code and database in order to achieve a real microservice-based application.

The authors of [12] propose a set of automated refactoring techniques, implemented in the IDE Eclipse, which facilitate the application transformation process to support services in the cloud. These techniques offer extraction of functionalities for services and remote access to them, treatment of failures and replacement of services accessed by the customer with services in the cloud equivalent. This work cannot refactor classes that use parameter passing, serialization, and local resources such as databases and disk files.

6 Conclusion

We present a methodology that, given the list and composition of target microservices as input, refactors the original monolithic Java application, which use annotations to map between entities and classes, into a microservices-based application. We built a tool, as a proof of concept, that supports Java Spring applications that use JPA annotations, and performed a quantitative evaluation against a collection of 10 open source java Spring applications from github. The results show that 8 of the 10 applications were automatically refactored and that the tool can be extended to support the remaining 2 applications.

As future work, we plan to address the issue of inheritance between classes that belong to different microservices and expand the capabilities of the tool to allow refactoring of an Java project into other languages.

Acknowledgments

This work is supported by the national funds through the Portuguese Funding Agency (FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020).

References

[1] Stephen J. Andriole. 2017. The Death of Big Software. *Commun. ACM* 60, 12 (Nov. 2017), 29–32. https://doi.org/10.1145/3152722

[2] Florian Auer, Valentina Lenarduzzi, Michael Felderer, and Davide Taibi. 2021. From monolithic systems to Microservices: An assessment framework. *Information and Software Technology* 137 (2021), 106600. https://doi.org/10.1016/j.infsof.2021.106600

[3] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian Tamburri, and Theodore Lynn. 2018. Microservices migration patterns. *Software: Practice and Experience* 48 (07 2018). https://doi.org/10.1002/spe.2608

[4] R. Chen, S. Li, and Z. Li. 2017. From Monolith to Microservices: A Dataflow-Driven Approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 466–475. https://doi.org/10.1109/APSEC.2017.53

[5] Martin Fowler. 2004. LocalDTO. https://martinfowler.com/bliki/LocalDTO.html. (Accessed on 10/02/2021).

[6] Martin Fowler. 2004. StranglerFigApplication. https://martinfowler.com/bliki/StranglerFigApplication.html. (Accessed on 11/20/2020).

[7] Jonas Fritzsch, Justus Bogner, Stefan Wagner, and Alfred Zimmermann. 2019. Microservices Migration in Industry: Intentions, Strategies, and Challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. https://doi.org/10.1109/ICSME.2019.00081

[8] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. 2016. Service Cutter: A Systematic Approach to Service Decomposition. In *Service-Oriented and Cloud Computing*, Marco Aiello, Einar Broch Johnsen, Schahram Dustdar, and Ilche Georgievski (Eds.). Springer International Publishing, Cham, 185–200.

[9] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng. 2021. Service Candidate Identification from Monolithic Systems based on Execution Traces. *IEEE Transactions on Software Engineering* 47, 5 (2021), 1–1. https://doi.org/10.1109/TSE.2019.2910531

[10] W. Jin, T. Liu, Q. Zheng, D. Cui, and Y. Cai. 2018. Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering. In *2018 IEEE International Conference on Web Services (ICWS)*. 211–218.

[11] M. Kamimura, K. Yano, T. Hatano, and A. Matsuo. 2018. Extracting Candidates of Microservices from Monolithic Application Code. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 571–580.

[12] Young-Woo Kwon and Eli Tilevich. 2013. Cloud Refactoring: Automated Transitioning to Cloud-Based Services. *Automated Software Engineering* 21 (09 2013). https://doi.org/10.1007/s10515-013-0136-9

[13] Robin Lichtenthäler, Mike Prechtl, Christoph Schwille, Tobias Schwartz, Pascal Cezanne, and Guido Wirtz. 2020. Requirements for a model-driven cloud-native migration of monolithic web-based applications. *SICS Software-Intensive Cyber-Physical Systems* 35, 1 (2020), 89–100. https://doi.org/10.1007/s00450-019-00414-9

[14] G. Mazlami, J. Cito, and P. Leitner. 2017. Extraction of Microservices from Monolithic Software Architectures. In *2017 IEEE International Conference on Web Services (ICWS)*. 524–531. https://doi.org/10.1109/ICWS.2017.61

[15] Manuel Mazzara, Nicola Dragoni, Antonio Bucchiarone, Alberto Giaretta, Stephan T. Larsen, and Schahram Dustdar. 2018. Microservices: Migration of a Mission Critical System. *IEEE Transactions on Services Computing* (2018), 1–1. https://doi.org/10.1109/TSC.2018.2889087

[16] S. Newman. 2015. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media. https://books.google.pt/books?id=jjl4BgAAQBAJ

[17] S. Newman. 2019. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Incorporated. https://books.google.pt/books?id=iul3wQEACAAJ

[18] Shmuel Tyszberowicz, Robert Heinrich, Bo Liu, and Zhiming Liu. 2018. Identifying Microservices Using Functional Decomposition. In *Dependable Software Engineering. Theories, Tools, and Applications*, Xinyu Feng, Markus Müller-Olm, and Zijiang Yang (Eds.). Springer International Publishing, Cham, 50–65.