A Methodology for Refactoring ORM-Based Monolithic Web Applications into Microservices

Francisco Freitas[a,d], André Ferreira[a,b], Jácome Cunha[c,d]

[a] *University of Minho*
[b] *Bosch Car Multimedia S.A.*
[c] *Faculty of Engineering, University of Porto*
[d] *HASLab/INESC TEC*

Abstract

In the last few years we have been seeing a drastic change in the way software is developed. Large-scale software projects are being assembled by a flexible composition of many (small) components possibly written in different programming languages and deployed anywhere in the cloud – the so-called microservices-based applications.

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

We propose a methodology to automatically evolve monolithic web applications that use object-relational mapping into microservices-based ones. Our methodology receives the source code and a microservices proposal and refactors the original code to create each microservice. Our methodology creates an 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 evaluation performed in 120 applications shows that our tool can successfully refactor about 72% of them. The execution of the unit tests in both versions of the applications yield exactly the same results.

Keywords: microservices, monolithic decomposition, refactoring, Java, software evolution, migration, ORM

1. Introduction

"The death of big software" has been announced in 2017 [1]. 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

Email addresses: a81580@alunos.uminho.pt (Francisco Freitas), alferreira@di.uminho.pt (André Ferreira), jacome@fe.up.pt (Jácome Cunha)

in ownership and deployment. Indeed, many software systems are currently being developed as a set of loosely-coupled components, possibly written in different programming languages, eventually deployed anywhere in the cloud, and communicating through the internet, creating an architectural style usually termed microservices [2]. These pieces can be used to mix-and-match as to create new or even to evolve existing software [1].

One of the main motivations of a microservices-based 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.

Although microservices are currently standard in industry, there are still many applications that were (and still are) built as monoliths, that is, applications composed of all the core logic related to the domain of the problem contained in a single process [2]. 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 [3, 4]. The decomposition of software systems is one of the main struggles, and as shown in the work of Fritzsch *et al.* [3], none of the participants in the study was aware of automated techniques that could assist the migration of a monolithic application to a microservices-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 microservices-based one, while preserving the semantics of the original application [5, 6, 7, 8, 9, 10]. Although some works already propose to refactor the code [11], most are focused on improving the migration process (e. g. deciding when to migrate) or in the identification of the microservices [12, 13]. Moreover, several companies have also applied major refactorings of their backend systems to transform their applications [4]. More about related work can be found in Section 5.

In this work we present a methodology, supported by a tool termed MicroRefact, that receives as input a microservices proposal for extracting microservices from a monolithic application, and refactors that original application into a microservices-based one. The microservices proposal is a set of sets where each of these sets contains components of the original application (e. g. a Java class) to be extracted into a microservices. For instance, if a monolithic Java application is composed of classes A, B, and C a possible microservices proposal could be {{A, B}, {C}}. Our methodology analyzes the source code and the microservices 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 MicroRefact automatically generate, implementing a REST[1] call to the original method which now is in a different service. We also refactor the database classes as they need to be spread by the different services. In Section 2 we present our methodology in detail and in Section 3 its implementation.

We performed a quantitative study to evaluate the applicability of our method-

[1] `https://www.redhat.com/en/topics/api/what-is-a-rest-api`

ology. From the 120 applications randomly selected from GitHub our tool was able to refactor 86, almost 72%. We also performed a qualitative study by running unit tests available in some of the applications showing the results in both the original and in the refactored applications are exactly the same. Section 4 presents in detail our evaluation. In Section 6 we draw our conclusions and describe some possible future work.

2. Refactoring ORM-Based Monoliths

In this section we present our methodology for refactoring monoliths based on object-relational mapping (ORM). To demonstrate what transformations our methodology makes to monolithic projects, we use as an example of an ORM framework Java Spring[2], which makes the mapping between classes and entities through annotations in the source code.

Our methodology receives the source code of the application under analysis and a microservices proposal. A microservices proposal is a set of sets where each of these sets contains components (e. g. Java classes) of the monolith that are to be extracted into a microservice. Each component must belong to just one microservice. Our methodology outputs a microservices-based application where the microservices descriptions are realized and the classes moved to the corresponding microservice. To have a running application after the refactoring, all the initial components (classes/interfaces) must exist in some microservice. If the user wants to refactor just part of the application, a possibility would be to have a single microservice description with the entities that one does not want to migrate. In this case, the dependencies within this microservice would not be part of the refactoring process, but the dependencies from and to this microservice would still be tackled by our approach.

Figure 1 depicts the steps of the refactoring process. In the Information Extraction phase (Section 2.2), we extract the structural information from the source code and combine it with the microservices proposal to identify the dependencies between microservices. In the next step, Database Refactoring (Section 2.3), we use the structural information and the dependencies between microservices to identify entities, the relationships between entities, and to identify which relationships need refactoring. After this initial identification, our approach proceeds to the refactoring of those relationships. Note that our approach assumes all the information about the database is contained in the classes (through annotations) and thus does not cope with other database information that may exist. Finally, in Code Refactoring (Section 2.4), 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] `https://spring.io`

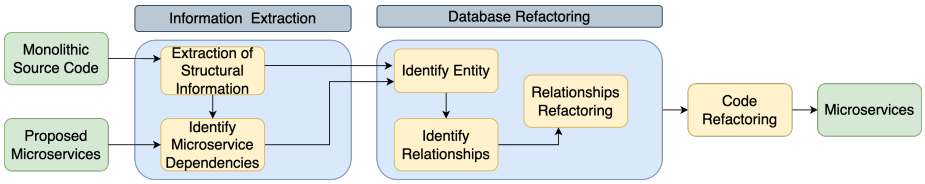

Figure 1: Overview of the proposed methodology

To demonstrate what transformations our methodology makes to monolithic projects, we use as an example a Java *Spring* application called *restaurantServer*[3].

In the following sections we present the *restaurantServer* application and explain in detail each of the phases of our methodology accompanied with examples of the transformations.

2.1. The Application restaurantServer

In this section we present the *restaurantServer*, a backend application for restaurant management. Although it is a small project, we chose to use this application as an example because it covers a large portion of the cases we consider. Our goal is to present extracted examples from this application to demonstrate what happens in each phase of our methodology, and to demonstrate the concrete modifications that each phase of our methodology makes to monolithic applications. To achieve this we present examples of interactions between classes in the original application and the same interactions in the generated microservices-based application.

To generate a microservices proposal for *restaurantServer*, to be used as input for our methodology, we used the tool developed by Brito *et al.* [14]. Although the tool allows the customization of the input, we use the default parameters. The microservices proposal generated by the tool is present in Table 1. The generated proposal consists of 7 microservices. Note that all classes in this project have the prefix *pl.edu.wat.wcy.pz.restaurantServer.* in their qualified name, which we remove for simplicity. The examples presented in this section are based on this microservices proposal.

2.2. Information Extraction

In the Information Extraction phase we extract structural information from the source code and we identify the dependencies between microservices. We next describe how to obtain this information.

2.2.1. 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

[3] `https://github.com/asledziewski/restaurantServer`

Table 1: Microservices proposal for restaurantServer.

#Microservice	Classes
1	security.WebSecurityConfiguration security.jwt.JwtAuthEntryPoint security.jwt.JwtAuthTokenFilter security.jwt.JwtProvider
2	security.service.UserDetailsServiceImpl repository.UserRepository entity.User security.service.UserPrinciple service.UserService controller.UserController
3	entity.Bill entity.BillPosition service.BillService controller.BillController service.BillPositionService controller.BillPositionController repository.BillPositionRepository repository.BillRepository
4	service.RTableService entity.RTable controller.RTableController repository.RTableRepository
5	entity.Dish RestaurantServerApplication repository.DishRepository service.DishService controller.DishController
6	repository.ReservationRepository entity.Reservation service.ReservationService controller.ReservationController
7	form.response.JwtResponse email.MailService controller.AuthController entity.Role repository.RoleRepository form.LoginForm form.SignUpForm

we identify the dependencies between the classes. We perform extraction of structural information in a structured version of the source code, in this case, the 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,
- its super class (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 the super class (if applicable). 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 super class, if it exists.

Algorithm 1 presents how the identification of classes which a class interacts with is performed. It receives as input a class (C) and returns the same class together with the list of its dependencies. The algorithm then iterates through the lists mentioned before (methods, interfaces, etc.) to extract the names of the classes that class C interacts with. Throughout the iteration, the classes' names are stored in Dep_C and finally the information contained in Dep_C is stored in C which results in the list of dependencies of C with other classes.

For example, from the AST of *restaurantServer* the information extracted for the *ReservationController* class is presented in Listing 1 in a JSON format[4].

```
1 {"name":"restaurantServer.controller.ReservationController",
2  "imports":["lombok.AllArgsConstructor",  "org.
       springframework.http.HttpStatus",  "org.springframework.
       web.bind.annotation",  "org.springframework.web.server.
       ResponseStatusException",  "restaurantServer.entity.
       Reservation",  "restaurantServer.service.
       ReservationService",  "java.text.SimpleDateFormat", "
       java.util.Collection",  "java.util.Date", "java.util.
       Optional"],
3  "extendedTypes":[],
4  "implementedTypes":[],
5  "annotations":["@AllArgsConstructor", "@RestController", "
       @CrossOrigin"],
```

[4]More about JSON can be found at `https://www.iso.org/standard/71616.html`.

Algorithm 1 Identification of classes which a class interacts with.

Input: C
Output: C
$Dep_C = []$;
$methods = C.getInvoked\_methods()$;
$interfaces = C.getInterfaces()$;
$extends = C.getExtends()$;
for m *in* $methods$ **do**
  if ($m.targetClass$ *not in* Dep_C) **then**
    $Dep_C.add(m.targetClass)$;
  end if
end for
for i *in* $interfaces$ **do**
  if ($i.name$ *not in* Dep_C) **then**
    $Dep_C.add(i.name)$;
  end if
end for
for e *in* $extends$ **do**
  if ($e.name$ *not in* Dep_C) **then**
    $Dep_C.add(e.name)$;
  end if
end for
$C.setDependencies(Dep_C)$;


```
6  "instance_variables":[{"annotations":[],  "modifier":"
       private",  "identifier":[],  "type":"ReservationService"
       ,  "variable":"reservationService",  "lineBegin":20, "
       lineEnd":20}],
7  "myMethods":{"getReservationById":{...},  "addReservation":
       {...},  "updateReservation":{...},  "getReservations":{
       ...},  "deleteReservation":{...},  "
       getCurrentReservations":{...}},
8  "methodInvocations":[{"methodName":"getReservations",  "
       targetClassName":"restaurantServer.service.
       ReservationService"}, {"methodName":"
       getCurrentReservations", "targetClassName":"
       restaurantServer.service.ReservationService", {"
       methodName":"getReservationById", "targetClassName":"
       restaurantServer.service.ResrvationService"}, {"
       methodName":"getDate", "targetClassName":"
       restaurantServer.entity.Reservation"}, {"methodName": "
       setDateDays",  "targetClassName":"restaurantServer.
       entity.Reservation"}, { "methodName":"setDateTime",  "
       targetClassName":"restaurantServer.entity.Reservation"},
        {"methodName":"addReservaion",  "targetClassName":"
       restaurantServer.service.ReservationService"}, {"
```

```
    methodName":"updateReservation", "targetClassName":"
    restaurantServer.service.ReservationService"}, {"
    methodName":"deleteReservationById", "targetClassName":"
    restaurantServer.service.ReservationService"}]
}
```

Listing 1: Information extracted from the AST about *ReservationController* class.

After the identification of the classes which the class *ReservationController* interacts with, the list presented in Listing 2 is added to the *ReservationController*.

```
{"dependencies": [
"restaurantServer.service.ReservationService",  "
    restaurantServer.entity.Reservation"]}
```

Listing 2: List of classes that interact with *ReservationController* class

This process is repeated for all classes of the monolith under analysis, and at the end of this process all interactions between classes are known, which is fundamental to identify the interactions between classes of different microservices, which is done in the next step.

2.2.2. Identify Microservice Dependencies

Our algorithm needs to identify possible dependencies between microservices since they will impact the refactoring process as we will see. We define dependencies between microservices as a reference to a certain non-primitive type that does not belong to the microservice.

The process of identification of dependencies between microservices and the identification of dependencies between classes from different microservices is presented in Algorithm 2. To obtain the dependencies between microservices and the dependencies between classes of different microservices, we use the microservices proposal, *microservices_proposal*, iterating over each microservice *ms* to check if the classes that belong to the dependency list of the classes that belong to *ms* exist in the classes that belong to *ms*. Through this comparison, we observe that the classes that are in the list of dependencies of a class that belongs to *ms* and that do not belong to *ms* are classes that belong to another microservice resulting in a dependency between those microservices. When a dependency between microservices is detected, the algorithm adds a pair to *dep_microservice* to indicate which microservices are involved in the dependency and the class name that class *c* depends on is added to update the dependency list. At the end of the process, the dependency list of each class only stores the dependencies with classes of other microservices.

Using as an example the information obtained for the class `ReservationController` in the previous section, we can observe that the classes which the `ReservationController` class depends on belong to the same microservice as the `ReservationController` (microservice #6), so there is no dependency between microservices detected by this class. Thus, the `ReservationController`'

Algorithm 2 Identification of dependencies between microservices

Input: $microservices\_proposal = [ms_1, ms_2, ..., ms_n]$,$Classes = [c_1, c_2, ..., c_m]$
Output: *dep_microservice*

```
dep_microservice = set();
for (i = 0; i < size(microservices_proposal); i++) do
    for class_name in microservices_proposal[i] do
        c = Classes.getClass(class_name);
        dep_class_microservices = [];
        dependencies_list = c.getDependencies();
        for dep_name in dependencies_list do
            if (dep_name not in microservices_proposal[i] && dep_name not
in dep_class_microservices) then
                dep_class_microservices.add(dep_name);
                for (k = 0; k < size(microservices_proposal); k++) do
                    if dep_name in microservices_proposal[k] then
                        dep_microservice.add((i, k));
                    end if
                end for
            end if
        end for
        c.setDependencies(dep_class_microservices);
    end for
end for
```

dependencies list is empty. For the entire *restaurantServer* applications we found 16 dependencies between microservices, presented in Table 2.

Table 2: Dependencies between microservices.

Microservice	Depends on
1	2
2	6;7
3	4;5
4	3;6
5	2;4;7
6	2;4;7
7	1;2;6

With the list of dependencies between microservices generated for each class, in some cases we need to make adjustments to the microservice proposal given as input. If the microservices proposal indicates that an interface implemented by a class is in a different microservice than the class, we replicate the interface and place the copy in the microservice where the implementing class belongs. Regarding inheritance, our methodology also makes some adjustments to the

microservices proposal. If the proposal indicates that the super class that a class extends belongs to a different microservice, we replicate the super class and place the copy in the microservice where the sub class belongs. This process is recursive, and thus, if the super class is a sub class of another super class, this super class is also replicated to the microservice which is the sub class that triggered the replication process. In this way we ensure that the inheritance relationship between the classes in the microservices-based application is the same as the relationship that exists in the monolith.

2.3. 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 [15]. The database refactoring phase aims to identify entities, relationships between entities, and to refactor 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 3 shows the entities and relationships present in *restaurantServer*. The logical schema of the database is defined by 7 classes and 6 relationships.

Table 3: Relationship between entities.

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

2.3.1. Relationships Refactoring

With the breakdown of the monolith into microservices, we need to verify 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. To do so, we use several patterns from the literature:

- *Data Transfer Object (DTO)*

- *Move Foreign-Key Relationship to Code*
- *Database Wrapping Service*

Note that the distribution of the monolith among microservices requires the creation of mechanisms to allow data to flow between the microservices when dependencies exist, which may impact the application's performance. Microservices are not suitable for all kinds of applications and thus, this needs to be considered when applying this kind of migration.

In the next paragraphs, we present each pattern and in which scenarios they are applied.

Data Transfer Object

The data transfer object pattern (DTO) is a distribution pattern used to reduce the number of calls when working with remote interfaces [16]. When working with remote interfaces (e. g. web services), each service call is an expensive operation. Indeed the majority of the cost of each call is associated with the round-trip time between the client and the server. Therefore, the solution is to transfer more data within each call. This can be achieved by creating a data transfer object that can hold all the data for the call.

How do we use it in the refactoring process. When a relationship between entities that belong to different microservices is identified, it means that at least one of the classes that are mapped as entities has an instance variable with the same data type as the other entity/class involved in the relationship. As the entities/-classes involved in the relationship come to exist in different environments (since they are in different microservices) one of the classes has an instance variable with a data type unknown to its domain (recall that the entities are in different microservices). We apply the DTO pattern to create this unknown data type. In this way, the unknown data type comes to exist, avoiding changes in the classes that have references to this data type.

To demonstrate the transformations made by this pattern we use the relationship between the classes `User` and `Reservation`. Figure 2 shows the One-to-Many relationship between `User` and `Reservation` in the original application. We omit some attributes from `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 database entities and their instance variables are mapped as attributes. The `Reservation` table has a foreign key that corresponds to the `User`'s primary key. Moreover, both classes have their respective `Repository` class that allows the manipulation of the data present in the database.

In terms of code the One-to-Many relationship is represented by the `User` class having an instance variable of type list of `Reservation` with `One-to-Many` annotation and the `Reservation` class having an instance variable to store a primary key of `User`.

By the microservice proposal in Table 1 and the class dependencies we extracted we verify that the `User` class depends on the `Reservation` class and

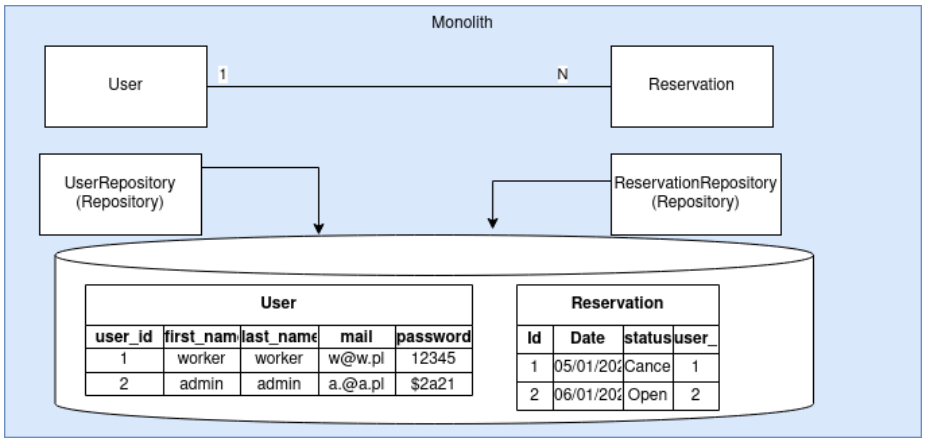

Figure 2: One-to-Many relationship between `User` and `Reservation` in monolithic application.

the classes are in different microservices. To create the `Reservation` data type in the microservice that the `User` class belongs to, our approach uses the DTO pattern. In this way, the `User` class remains unchanged and does not depend directly on the `Reservation` class that resides in another microservice. In the following pattern, we expand this example.

Move Foreign-Key Relationship to Code

This database decomposition pattern is used to move the foreign key from the database into the source code of the application. The relationships between entities are denoted by foreign-key relationships. Defining this relationship in the underlying database engine ensures data consistency and lets the database engine execute performance optimizations to ensure that the join operation is efficient. However, when one wants to split the database and the entities involved in the relationship in different schemas two problems emerge: *i)* the join of information cannot be performed via database join and *ii)* data inconsistency is now a possibility. The move foreign-key relationship to code pattern solves the first problem by moving the join of information to the 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. Note this pattern incurs some performance costs since we replace a local select in the database with a select in one database, plus a service call, plus a select in another database.

How do we use it in the refactoring process. We apply this pattern when we find relationships of the types One-to-One, Many-to-One, and One-to-Many between entities that will belong to different microservices. To do that the first step is to remove the annotation from the source code which creates the relationship. This annotation is typically present in one of the classes involved in the relationship.

With the removal of the annotation that created the relationship, the table that stored the foreign key loses the column for that purpose.

Next, we add an instance variable to the class that represents the other entity involved in the relationship. We add this new variable to create a column in the table that is responsible for storing the foreign key. In this way, the table has the same attributes that it had before the database was split. Furthermore, this new variable will be used as a filter to retrieve data from the database.

As we already have the tables involved in the relationship with the right attributes and each one in its schema, the next step is to identify the methods that manipulate the data that belongs to the other microservice. The algorithm then searches for methods that have a reference to that data. These methods will then have an internal service call to manipulate this data using the primary key as a parameter.

To create the service calls and to make the generated code loosely coupled we perform the following steps:

1. We create an interface where the signature of the identified methods is declared.
2. We create a class to implement the interface created that is responsible for making the service calls.
3. We create a variable of the type of the interface created and add it to the class that is mapped as an entity.

Finally, to respond to the service calls it is necessary to create an API in the other microservice involved in the relationship. Thus, we create two classes, one with the resource paths for the requests, and another to process the request.

```
1  @Entity
2  class User
3  {
4    @OneToMany
5    Reservation reservations;
6    int userId;
7
8    Reservation
         getReservations() {
9      return reservations;
10   }
11 }
```

Listing 3: Part of the original code related to the entity `User`.

```
1  @Entity
2  class Reservation { }
3
4  class ReservationRepository {
       }
```

Listing 4: Part of the original code related to the entity `Reservation`.

To demonstrate how this pattern is used in our example we again use the relationship between the classes `User` and `Reservation`. Listings 3 and 4 present a very minimalist version of the original code of these classes (note this code

does not compile, but the purpose is to have a minimal example that illustrates the transformations we designed).

Figure 3 gives an overview of the relationship between the `User` and `Reservation` after the refactoring while listing 5 presents a minimal example for part of the code for the `User`'s microservice and Listing 6 for the `Reservations`'.

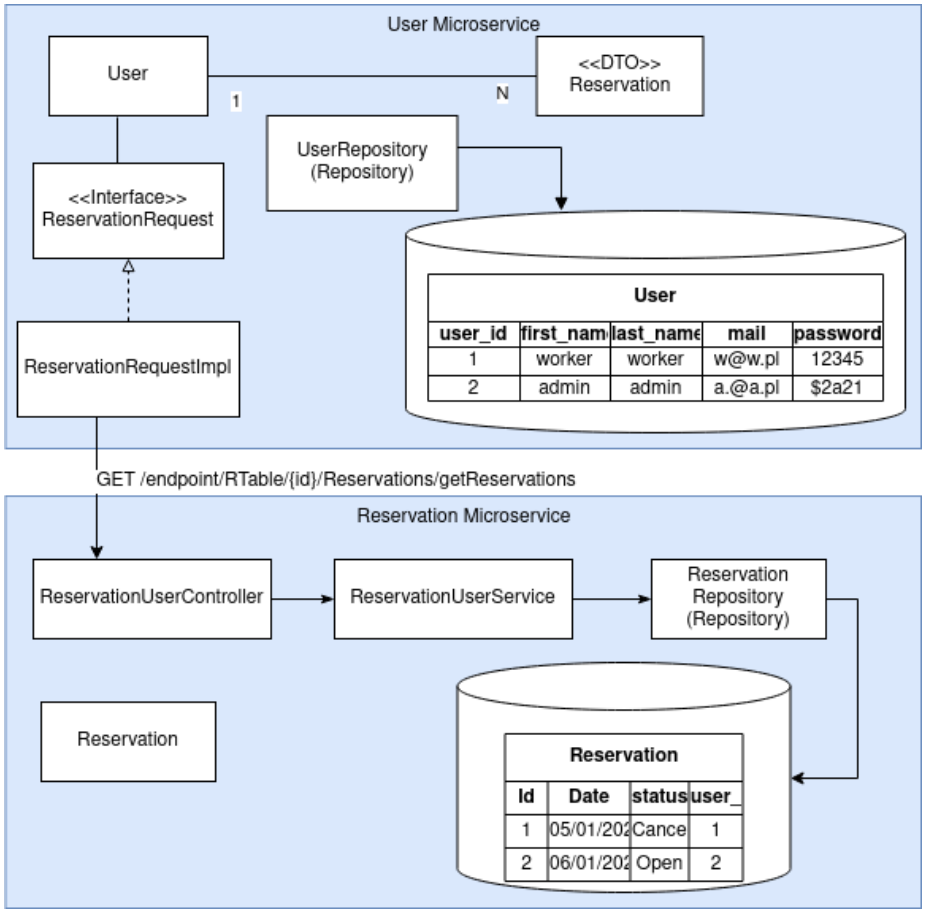

Figure 3: One-to-Many relationship between `User` and `Reservation` in microservice-based application.

```
1 @Entity
2 class User {
3   @Transient
4   Reservation reservations;
5   ReservationRequest rreq;
6   int userId;
7
8   Reservation getReservations() {
```

```
9       reservations = rreq.getReservations(userId);
10      return reservations;
11    }
12  }
13
14  class Reservation { // same attributes as in the original
        Reservation class  }
15
16  class ReservationRequest {
17    Reservation getReservations(int userId); }
18
19  class ReservationRequestImpl implements ReservationRequest {
20    Reservation getReservations(int userId) {
21      get("Reservation/getReservations", userId);
22    }
23  }
```

Listing 5: Part pf the code for the `User` microservice.

```
1   @Entity
2   class Reservation { }
3
4   class ReservationRepository {
5     Reservation getReservations(int userId); }
6
7   class ReservationUserController {
8     ReservationUserService service;
9
10    "Reservation/getReservations"
11    Reservation getReservations(int userId) {
12      service.getReservations(userId); }
13  }
14
15  class ReservationUserService {
16    ReservationRepository rep;
17
18    Reservation getReservations(int userId) {
19      rep.getReservations(userId); }
20  }
```

Listing 6: Part of the code for the `Reservation` microservice.

Our approach generates one interface and one class, `ReservationRequest` and `ReservationRequestImpl`, respectively, in the `User`'s microservice, which are responsible for making requests to the microservice where the `Reservation` information is. By applying the DTO pattern, our approach also creates a class called `Reservation` that has the same attributes as the original `Reservation` class.

The class `User` now has an instance variable of type `ReservationRequest`. The instance variable of `Reservation` that had the annotation `OneToMany` now

has the annotation `Transient`, which indicates that this information is not to be stored in the database. The `ReservationRequest` interface has the signature of the method `getReservations` that has as a parameter the user id. The `ReservationRequestImpl` class implements the `ReservationRequest` interface and is responsible for calling the reservation microservice.

In `Reservation` microservice, the classes `ReservationUserController` and `ReservationUserService` are also created to receive the requests and to process the requests, respectively. Also, we add in the `ReservationRepository` class the method `getReservations`.

In the `Reservation` microservice the `ReservationUserController` class is also generated which is responsible for receiving the requests and sending them to the corresponding service (in this case `ReservationUserService`).

Finally, we create the class `ReservationUserService` in which the request is directed to the `Repository` class where the `getReservations` method is declared using the `User` id as a filter.

In Figure 4, we present the pseudo-code of two classes where the transformations just described can be applied. In this representation, `@entity` means the underlying class is an entity and thus its attributes are stored in a database; `@relationship` represents a relationship between the classes `C1` and `C2` (e. g. a one-to-many relationship); finally, `@transient` means the underlying attribute should not be stored in the database (as it is in the database of the other microservice.

```
@entity                          @entity
class C1                         class C2 { }
{
  @relationship                  class C2Repository { }
  C2 v;
  int c1Id;

  C2 get() {
    ...
  }

  set(C2 p) {
    ...
  }
}
```

Figure 4: Generic representation of classes that are refactored if each one is in a different microservice.

The refactoring process we just described will produce the pseudo-code presented in Figure 5.

```
@entity
class C1 {
  @transient
  C2 v;
  C2Request rreq;
  int c1Id;

  C2 get() { ... }

  set(C2 m) { ... }
}

class C2 {
  // same attributes as in the
  // original C2 class
}

class C2Request {
  C2 get(int c1Id);
  set(C2 p, int c1Id);
}

class C2RequestImpl implements
    C2Request {
  C2 get(int c1Id) {
    HTTPget("MS/get", c1Id);
  }

  setReservations(C2 v,
      int c1Id) {
    HTTPset("MS/set", c1Id);
  }
}
```

```
@entity
class C2 { }

class C2Repository {
  C2 get(int c1id);
  set(C2 p, int c1id);
}

class C2C1Controller {
  C2C1Service service;

  "MS/get"
  C2 get(int c1id) {
    service.get(c1id);
  }

  "MS/set"
  C2 set(C2 p, int c1id) {
    service.set(C2 p, c1id);
  }
}

class C2C1Service {
  C2Repository rep;

  C2 get(int c1id) {
    rep.get(c1id);
  }

  set(C2 p, int c1id) {
    rep.set(p, c1id);
  }
}
```

Figure 5: Pseudo-code after the refactoring of the pseudo-classes in Figure 4.

Database Wrapping Service

To break the original database into smaller databases for each microservice is a delicate process. The possible solution of having a shared database goes against the principles of microservices – each microservice has its own data and is loosely coupled. Thus, the database wrapping service pattern appears as a better solution and it is also seen as a stepping stone to more fundamental changes, giving developers time to break apart the schema underneath the API layer [15]. This pattern creates a new service to "hide" the database, providing

an API to access the data. In this way, the services replace access to the database with requests to the service that owns the database. This ensures the database schema is unchanged and the services exhibit low coupling.

How do we use it in the refactoring process. We apply this pattern when we find a Many-to-Many relationship between entities that will belong to different microservices. 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 move foreign key relationship to code pattern we would lose this table. Instead, we apply this pattern and the relationship is kept intact.

We apply this pattern by extracting the classes involved in the relationship that are mapped to the entities, and their respective DAO classes for a new service. This new service provides an API to access the data stored in the database and, therefore, the services that need to access that data, replace direct database calls with calls to the new service.

To demonstrate how this pattern is applied in our example we use the relationship between `User` and `Role`. By the microservices proposal of Table 1 we verify that the classes `Role` and `User` belong to different microservices and by Table 3 we verify that these classes have a Many-to-Many relationship. Our process creates an extra microservice to store the `User` and `Role` classes and their respective `Repository` classes to access the database. In this way the relationship between the entities is maintained and the refactoring process continues with this new microservice being added to the microservices proposal and the classes dependency lists are recalculated taking into account this new scenario.

2.4. Code Refactoring

Having already refactored the classes that create the database logical schema, our approach moves to analyze the other classes. The code refactoring phase aims to refactor the classes that have method calls to instances of other classes that are part of other services (e. g. the method `addBill` from class `BillService`, which is in microservice #3, calls method `findId` from class `RTableRepository`, which is in microservices #4). The regular method call will not work after the refactoring since the classes are now in different microservices. Thus, it is necessary to create a new mechanism for these calls to continue to work.

Figure 6 illustrates the steps that compose the code refactoring phase.

The code refactoring process starts with the analysis of the variables of the classes. The information about the variables is obtained from the structural information of the class. We analyze the data type of each instance variable and we check if the data type is in the list of dependencies that the class has with other microservices. If the data type of the instance variable under analysis is not in the list of dependencies, it is not necessary to continue the refactoring process for this variable, because it corresponds to a data type that exists in the microservice to which the class belongs or it is a primitive data type of the language. However, if the data type is among the dependencies with other microservices it is necessary to search for method invocations of the class that

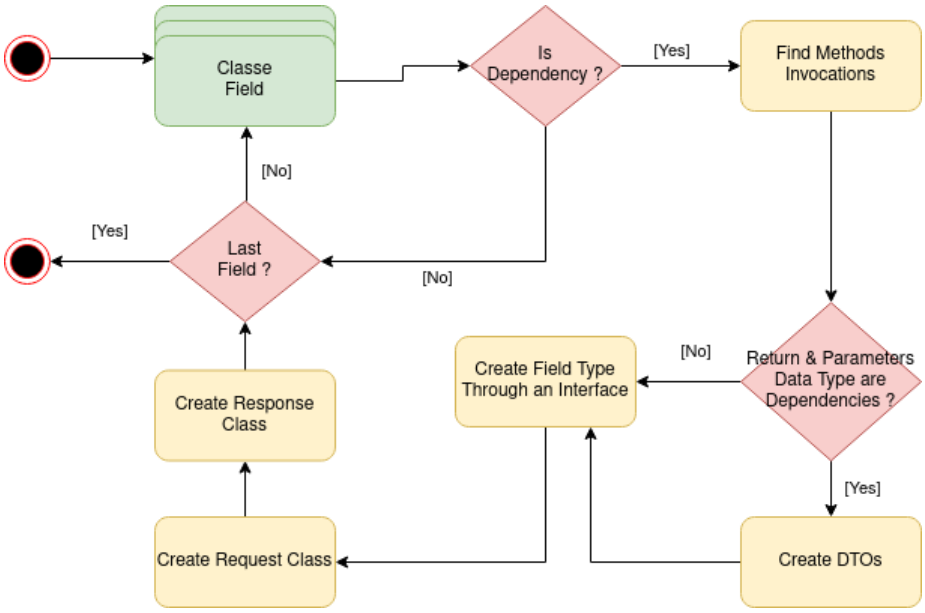

Figure 6: Overview of code refactoring.

correspond to the data type. This happens because these methods, after the refactoring, will no longer be invoked locally, but will be replaced by calls to the service that owns the data type and consequently the methods.

The identification of the invoked methods is also simple to obtain through the structural information. Since the identified methods may have as a parameter or return a data type that is present in the class's list of dependencies with other microservices, it is necessary to verify the data type of the parameters and the return. If in the parameters or in the return of the method we found data types that are in the list of dependencies of the class with other microservices we apply the DTO pattern to create these data types in the microservice. The reason for doing this is that the parameters and the method return will be sent in the service call so they are data transfer objects.

After having the invoked methods identified and their parameters and return types checked, we can create the service calls. We create the data type of the instance variable by creating an interface with the same name as the data type. In this way, the modifications made to the code will be transparent to the class under analysis. The interface created contains the signature of all the invoked methods identified that will become service calls. To define the service calls we create a class, termed `Request`, that implements the interface created. This new class is responsible for making the remote service calls and therefore we define the REST calls for each method in this class.

Finally, in the microservice that owns the class that corresponds to the data type that triggered the refactoring process, a REST API is created that allows

the invocation of the original methods.

In order to complete the code refactoring process our approach needs to verify the data types of the local variables (that is, the variables created within each method). We analyze if the data types of the local variables are in the list of dependencies with other microservices of the class under analysis. If the data type of the local variables is in the list of dependencies we create the data type by applying the DTO pattern and a search for invoked methods of the class that correspond to the data type. We apply the DTO pattern because these variables are typically the result of invoking a method of a class that has defined an instance variable. In fact, the refactoring of the instance variables already applies the DTO pattern to create the return data type of the invoked methods which corresponds, for the most part, to the local variables. To avoid creating classes that already exist when a data type is created by applying the DTO pattern, this data type is removed from the list of dependencies with other microservices of the class under analysis. The identified invoked methods of the local variables are defined as service calls in the class generated by the DTO pattern application. As in the analysis of the instance variables we create a REST API, in the microservice that owns the data type, that allows the invocation of the original method.

To show the changes made by the code refactoring phase we use the dependency between the `BillService` class and the `RtableRepository` class, as an example. Figure 7 gives an overview of the interaction of the class `BillService` with the class `RTableRepository` in the monolithic application. The `BillService` class has two instance variables, one of type `BillRepository` and another of type `RTableRepository`, and six methods.

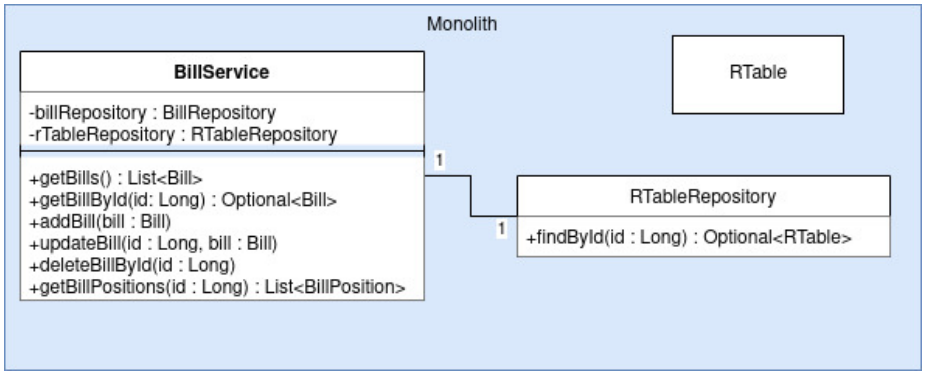

Figure 7: Interaction between `Billservice` and `RTableRepository` in monolithic application.

In the monolithic version, the relationship between the `BillService` class and the `RTableRepository` class is characterized by the `BillService` class having an instance variable of type `RTableRepository` and by invoking the `findbyId` method, which returns a `RTable`, from the `RtableRepository` class within the `addBill` method.

The relationship between the two classes and the invocation of the `findById`

method after the code refactoring phase is presented in Figure 8. In the code refactoring phase an interface called `RTableRepository` is created in the `BillService`'s microservice to represent this type in the microservice where the `findById` method signature was declared since it is only this method that is invoked from the original `RTableRepository` class. Besides that, we create a class called `RtableRepositoryImpl` that is responsible for making the REST call to invoke the original method and a class called `RTable` to represent the `finById` return type. In `RTable`'s microservice a class called `RtableRepositoryController` is created where the REST API is defined and that invokes the `findById` method in the original `RTableRepository` class.

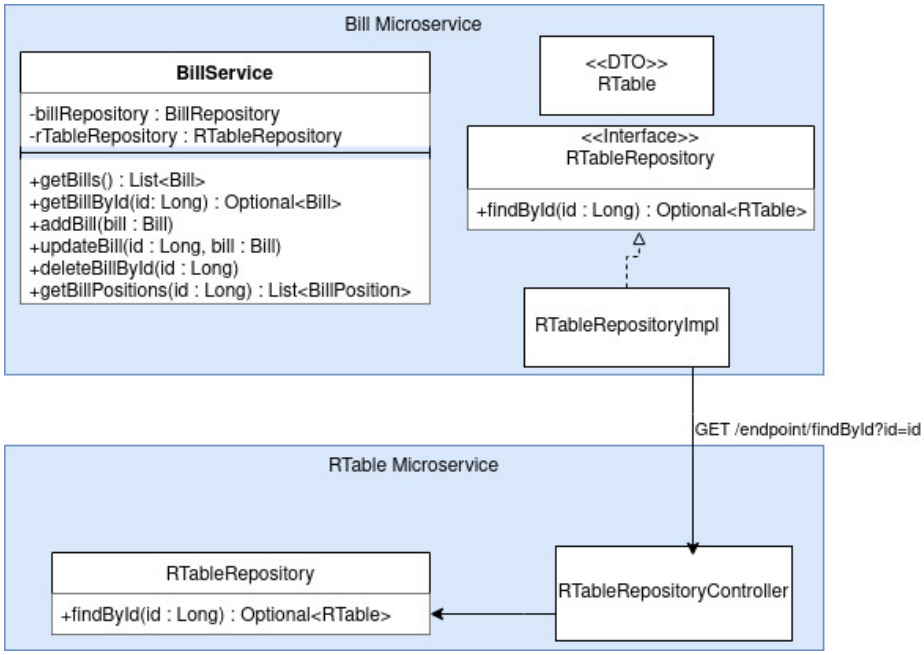

Figure 8: Interaction between `Billservice` and `RTableRepository` in microserices-based application.

All the transformations made by the code refactoring phase can be seen in `https://github.com/MicroRefact/restaurantServerMs` where we provide the source code of the refactored application.

2.5. On the Migration of Monoliths to Microservices

The refactoring of an application from a monolith to a microservices architecture has several issues to be considered.

Performance. As we have discussed before, when splitting the database between microservices, if one microservice needs to access data from another, that data must now be transferred using HTTP requests. This obviously causes some

degradation in the performance that should be considered when performing such a migration.

New Errors. The fact that the application needs to communicate with some of its parts using HTTP requests implies new types of errors need to be taken under consideration. If a request fails (e. g. because of a failure in the network), the requester service will block waiting for the result or eventually timeout. Our refactoring does not introduce any type of mechanism to handle such situations which means the application we produce does not handle errors that may occur from the communication. Thus, one may argue that the transformations introduced by our approach do not preserve behavior because an operation that was previously error-free in the original system may now produce errors. Nevertheless, apart from the kind of errors related to communication, our approach maintains the original behavior. Moreover, there are several ways of making microservices applications resilient [17], but in this work, we have not considered any of them.

3. MicroRefact

In this section we present the implementation of our tool, MicroRefact, which serves as a proof of concept to validate the methodology. MicroRefact is designed for Java applications, in particular for the Spring framework. MicroRefact supports the refactoring of applications that use the *Java Persistence API*[5] (JPA) to do the object-relational mapping. We decided to support JPA since it describes a common interface to data persistence frameworks. The implementation of MicroRefact is available at `https://github.com/FranciscoFreitas45/MicroRefact`. Figure 9 represents the overall flow of MicroRefact, which we describe in the following paragraphs.

3.1. Information Extraction

Our tool receives as input a JSON file with the microservices proposal to be extracted 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 into an AST. We use the *Java Parser*[6] 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.

[5] `https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html`
[6] `https://javaparser.org/`

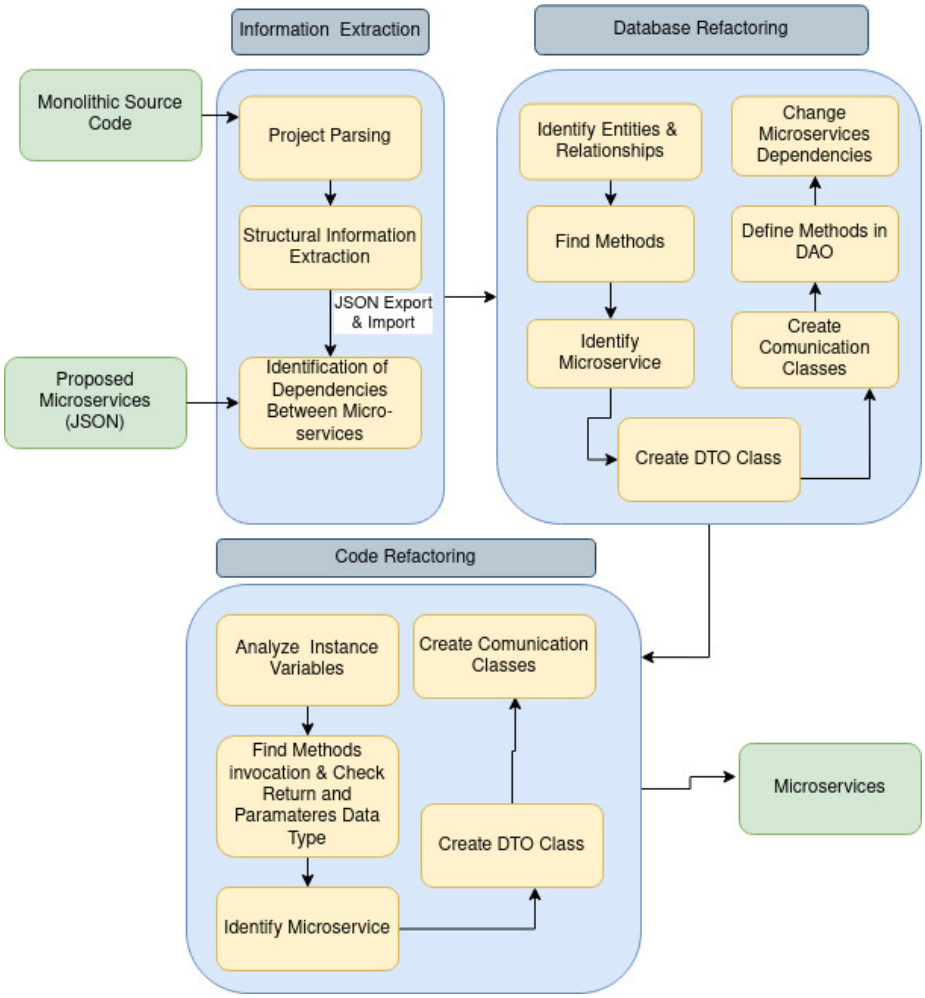

Figure 9: Overall flow of MicroRefact.

Our code includes a data structure to store the information extracted from the AST and to represent the microservices in the program. We create a class called `Cluster` (as in a group classes) 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`,
- a list for adding new classes to the microservice, and
- 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 a microservice.

In the next step, to identify the dependencies between microservices, for each `Class` object, MicroRefact analyzes 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

During the database refactoring phase, entities are identified by searching for the word `Entity` among the annotation list of each `Class` object. Also, the tool identifies the relationships between entities by searching for one of the following words in the annotation list of each instance variable and in the annotation list of each method of the `Class` object under analysis: `OneToMany`, `ManyToOne`, `ManyToMany`, and `OneToOne`. These are the annotations used in JPA for denoting database relationships among Java classes.

If relationships are found, our tool verifies whether the type of the instance variable that has the annotation is in the object's dependency list. If it is, a search is performed on all methods in the method list of the `Class` object under analysis to check whether the type of the instance variable is used in the return, or in the parameters, or in the declaration of variables. If this is the case, the tool creates a list with the methods that use the type of the instance variable.

If the annotation is found in methods, typically in getters, the tool identifies to which instance variable the method corresponds and the process described in the previous paragraph is performed for the instance variable.

For the application of the DTO pattern, 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 find the corresponding `Class` object to make a clone of it and add it to the `Cluster` dictionary of the class under analysis.

Next, we create an interface with the signature of the methods that are in the identified list and a new `Class` object is instantiated which 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 the list of new classes of `Cluster`. The first one uses the methods declared in the interface to define the routes and the second one 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. If the type of their instance variables are in their dependency list, the same procedure used in the previous phase is applied 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 is applied as in the previous phase.

Finally, the information contained in each object `Class` is used to write Java files. A new file is created for each class and the tool iterates over all the fields in the `Class` objects to populate the files. A folder is created for each microservice where the files are written.

4. Evaluation

The evaluation of our work has two different purposes: *i)* to assess the applicability of the approach, that is, to understand how many software projects it can refactor, and *ii)* to assess if the changes to the code impact the functionality of the original software. For the first purpose, we have randomly collected 120 projects from GitHub (more details in Section 4.1) and run MicroRefact on them. The results can be found in Section 4.3. For the second purpose, we have used the unit tests available in the original projects and have used them to compare the results obtained by running some of the units of the original and refactored software (more in Section 4.3). Different kinds of evaluations could be done, from system test to performance to security since all are impacted by the paradigm change (from monolith to microservices). However, with the performed evaluation we try to show that the available tests proposed by the developers/testers in the original software did not have a different outcome after the refactoring.

4.1. Project Collection

To evaluate the proposed methodology we test it in several projects of different sizes and complexity. To do this, we extracted projects from GitHub since it is the most popular platform for hosting open-source software. We used the GitHub search API to find repositories that contained the three terms, namely: `org.springframework.data.jpa`, `org.springframework.data`, and `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 was:
`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 aims to identify repositories through code there are repeated results. After executing this query and removing duplicate repositories, we identified

686 repositories. To ensure that only monolithic applications are used, we only consider projects with one 'src' folder. We also discard projects with less than 25 classes because they may represent "toy" projects and we use filters to exclude demo and test projects using the following stop words: "release", "framework", "learn", "source","spring", "study", "demonstration", "test", "practice". That reduced the 686 projects to 353. From the identified projects we randomly selected 120 projects which is a considerable amount of applications. The list of all projects can be found in [18]. The histogram of the projects collected by the number of classes is presented in Figure 10. The biggest project has 1339 classes and the smallest 27 classes. We use the number of classes as an indicative metric of the projects' dimension as our approach needs to analyze classes and their dependencies.

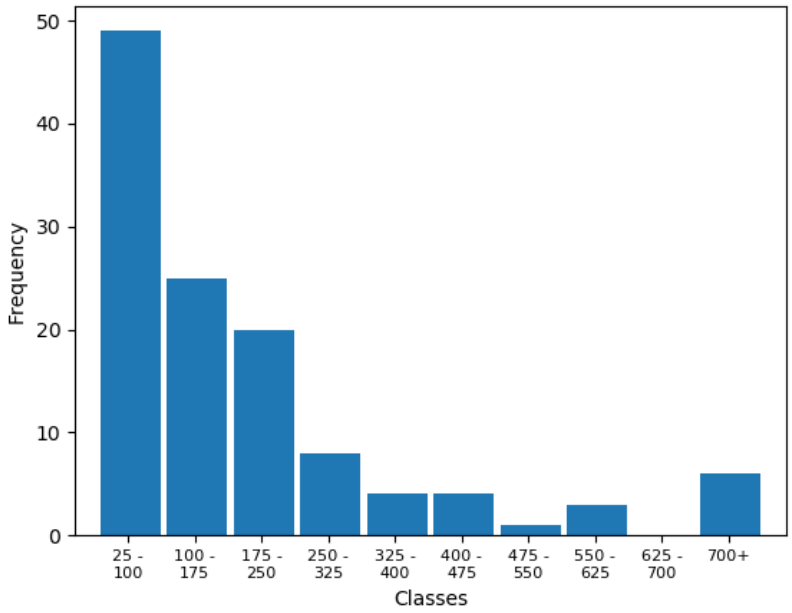

Figure 10: Histogram of collected projects by class count.

4.2. Setup

The setup of the evaluation was divided in two parts: in the first part it was necessary to obtain a microservices proposal for each of the collected projects and in the second part it was necessary to adapt the unit tests to run on the microservices-based version, since they were designed for the monolithic application.

To obtain a microservices proposal for each collected project, we used the tool developed by Brito *et al.* [14] with its 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, as suggested by the authors [14].

After running MicroRefact on each project, the result is analyzed. If successful, and if the project has unit tests, these are run on the generated microservices-based application. However, the testing process has not been automated because, although the code generation is automated, it is necessary to define *SQL* queries for the methods that were added to the `Repository` classes and because the unit tests need some adjustments that depend on the context (which we describe next).

4.2.1. Adaptation of Unit Tests

As the tests are not part of the refactoring process, it was necessary to identify to which microservices the test classes should be transferred. This process was manual and had a detailed analysis of the test classes as well as the domain of the generated microservices. In projects with a good design and developed with good practices the test class name served to identify to which microservice it should be transferred to. In more disorganized projects we did an analysis of the imports of the test class as well as the declared variables to identify to which microservice it should belong.

In addition, and given that the tests were designed for the monolith, it was necessary to adapt the tests for microservices. Among the modifications made to the test classes are imports because the test classes contained references to classes that do not belong to the microservice. These imports were replaced with imports that refer to the types created to replace the reference to classes that do not belong to the microservice, i. e. the DTO and interfaces created.

The other change made was in tests that manipulate the database. As unit tests are designed for the monolithic application, there are unit tests that test the interactions between classes that are mapped as database entities (we present examples of this in Section 4.2.2). In the unit tests where new records are inserted in the database tables, and as we are in the scope of applications that use annotations to define the database logical schema, these create objects and use the `Repository` classes to insert them. However, the objects created have as attributes objects that refer to other entities, so the objects that refer to other entities also have to be instantiated. In the monolithic version, when the test is run, insertions are made in all tables referring to the classes involved in the test. However, in microservices and with the database refactoring, the references to classes that are mapped as entities that do not belong to the microservice have been replaced by DTOs with the same name and therefore the unit tests create the objects through the instantiation of a DTO class, except for the entities that belong to the microservice that are created in the database. The objects created through the instantiation of a DTO class exist in memory in the microservice where the unit test is executed, but the record corresponding to the object does not exist in the microservice database where the entity that is represented by the DTO belongs, which causes methods like *setters* to fail. In addition there

are also unit tests where the creation of a record in the database is tested by instantiating an object and using *setters* to set the values of other entities to `null`. In this way the test is focused only on the entity that belongs to the microservice.

In these situations, since a test may depend on objects from other microservices, when necessary, we manually created them in the database so they would exist when the test would run.

4.2.2. On the Usage of Unit Tests

Unit tests may not be sufficient to fully guarantee the preservation of the behavior of the system after the refactoring. For instance, many unit tests do not exercise the interaction between the different components. This is particular relevant in our case where components may be in different services after the refactoring.

To illustrate that some of the applications we refactored have tests that actually force the interaction between different services after the refactoring we present an example from the projects of our evaluation: *Online-medicine-shopping-ecommerce*[7]. Listing 7 shows (part of) a unit test for creating a `User` in this application. The attribute `userAddress` is of type `Address`, which is a class that is mapped as an entity with a one-to-one relationship with `User`. The microservices proposal indicates that the classes `User` and `Address` belong to different microservices.

In this test a `User` object is instantiated through setters. In particular, the setter `setUserAdress` receives `null` as input instead of an object. In fact, as it is, this test fails in the microservices-based application because the relationship between `User` and `Address` has been refactored. In the refactored application the setter `setUserAddress` now calls the respective service to update the data. This call receives as parameter the primary key of the address, so when making a service call with `null` it results in a *not found error* because the passage of the primary key in the request is made through the URL.

```
1 @Test
2 void testAddUser() {
3     User user = new User();
4     user.setEmailId("vino@gmail.com");
5     user.setFirstName("vino");
6     user.setUserAddress(null):
7     ResponseEntity<User> postResponse = restTemplate.
          postForEntity(getRootUrl() + "/User/newUser", user,
          User.class);
8 }
```

Listing 7: Original unit test for creating a user in Online-medicine-shopping-ecommerce.

A possible solution is to instantiate an object of type `Address` and set it in the `User` as shown in Listing 8. However, this would be instantiated through a

[7] `https://github.com/ariv98/Online-medicine-shopping-ecommerce`

DTO class which would cause the information stored in the object not to exist in the `Address`'s microservices database, causing the `setUserAddress` method to try to update records that do not exist. Thus, to avoid having an object instantiated in a microservice and no corresponding record in the database of the microservice that owns the entity, before running the unit test we manually inserted a record in the database that corresponds to the object instantiated during the test run.

```
@Test
void testAddUser() {
    User user = new User();
    Address address = new Address();
    address.setAddressId(1);
    user.setEmailId("vino@gmail.com");
    user.setFirstName("vino");
    user.setUserAddress(address):
    ResponseEntity<User> postResponse = restTemplate.
        postForEntity(getRootUrl() + "/User/newUser", user,
        User.class);
}
```

Listing 8: Refactored unit test for creating a user in the microservices-based version of Online-medicine-shopping-ecommerce.

This example illustrates a possible yet small interaction between different microservices through unit tests.

We now present (part of) a second example taken from another project, *inTeams*[8], in Listing 9.

```
public class ProjectServiceTests {
    private final ProjectService projectService;
    private final TeamRepository teamRepository;

    @Test
    void canGetAllProjectsOfTeam() throws InvalidOperation {
        Team mainTeam = teamRepository.findByName("Test␣Team
            ␣001").orElseThrow();
        Assertions.assertEquals(1L, projectService.
            getAllProjectsOfTeam(mainTeam.getId()).size());
        Team subTeam = teamRepository.findByName("Test␣Team␣
            007").orElseThrow();
        Assertions.assertEquals(0L, projectService.
            getAllProjectsOfTeam(subTeam.getId()).size());
    }
}
```

Listing 9: Original unit test of inTeams project.

[8] https://github.com/BarCzerw/inTeams

As we can see , the test uses objects from classes `ProjectService` and `TeamRepository` which are in difference microservices. In this case the interaction between the different objects/services is more evident as in line 8 the test uses the result of the method `getId` of the class `TeamRepository` as input (through the variable `mainTeam`) to the method `getAllProjectsOfTeam` from the `ProjectService` class. Since `ProjectService` and `TeamRepository` are in different microservices after the refactoring, this test in another example of how unit test can in fact exercise the communication between microservices.

Although these two examples do not demonstrate the preservation of the behavior after the refactoring, they show that there are indeed unit tests that go beyond the basic and test the integration between different components/microservices.

4.3. Results and Analysis

From the 120 applications MicroRefact was able to refactor 86, approximately 71.7%. Within the universe of the refactored projects 33% have unit tests. All the unit tests executed had the same output in both versions of the application, which shows the refactoring was successful, at least in terms of the unit tests' results.

4.3.1. Degree of Complexity and Size of the Applications

Since the dataset used for the study consists of projects with a wide range of classes and different domains, we try to understand the impact of the application size and complexity on the percentage of refactored applications. A large percentage of the projects used in the evaluation, approximately 41%, are projects where the range of classes is between 25 and 100. One might think that these projects are small and or have a low degree of complexity. As the class range in which the most projects were refactored was this range, one could think that the tool is not prepared for projects of great magnitude and high complexity. However, we must understand the context of the study. Since we are using applications that use annotations to apply the ORM technique, the number of classes in these projects tends to be smaller than projects that do not use this technique because they do not need to declare classes to define DAOs, being these projects complex, but with a low number of classes. Furthermore, as shown in Figure 11, the tool was able to refactor projects in all ranges of class numbers, which shows that the tool is prepared to support all degrees of complexity present in these projects. Although for the range of projects over 175 classes there seems to be a decrease in the number of refactored projects, there is no particular reason for this to happen. All the cases where our approach and tool do not work are discussed in Section 4.3.2 and the size is not one of the reasons.

4.3.2. Unrefactored Projects

Regarding the unrefactored projects, there are different reasons why refactoring did not happen. Table 4 shows the reasons why projects were not refactored (first column) and the number of projects that fit into it (second column).

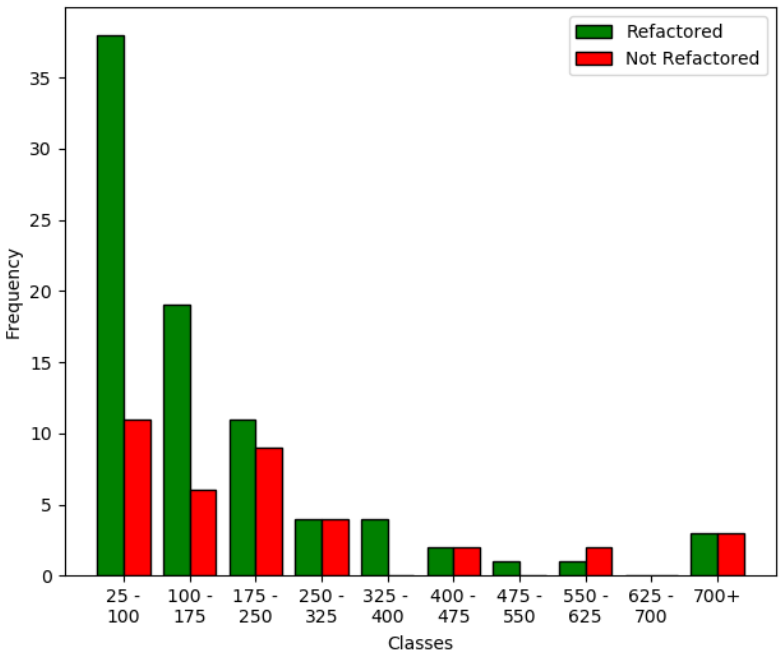

Figure 11: Histogram of number of refactored projects by class count.

Table 4: Reasons for unrefactored.

Reason	#Projects
Missing a repository class	17
Use DAOs	8
Missing @id annotation	6
Use other ORM frameworks	2
@Id annotation in method instead of variable	1

About 50% of the projects that were not refactored are projects where the class `Repository` is missing (17 projects). We can divide the reason why refactoring fails due to the lack of the `Repository` class into two types: *i)* the `Repository` class exists but in the microservices proposal it is in a different microservice than the `Entity` class that the `Repository` class refers to; *ii)* an `Entity` class exists that does not have a corresponding `Repository` class. For the first case a different microservice proposal where the `Entity` class and the corresponding `Repository` class are in the same microservice would solve the problem. In the second case refactoring is not possible since there is an `Entity` class, but not a corresponding `Repository` class that allows the retrieval of data

from the database, which may be an error in the code or dead code. We identify the `Repository` classes to apply the *move foreign-key relationship to code pattern*, because it implies adding methods to these classes so that the join is possible.

Another reason why refactoring was not possible is the lack of the `Id` annotation in classes that are mapped to database entities (6 cases). In the database refactoring phase it is necessary to identify the primary key of the entities involved in the relationship that is under analysis because in the application of the *move foreign-key relationship to code pattern* the primary key of one of the entities is used as a parameter in the invocation of the REST API generated to filter the information retrieved during the join. The identification of the primary key is accomplished through the `Id` annotation in the class instance variables. We also cover the cases in which the class `Entity` under analysis is a sub class and in which the `Id` annotation is in the super class, which means, if the `Id` annotation in the class `Entity` under analysis is not found in one of the instance variables our approach analyzes if the super class contains the `Id` annotation in one of the instance variables. Without the identification of the primary key, the refactoring is not possible, because the calls to the service to make the join of the data would not have a filter.

The use of DAOs by some projects together with `Repository` classes led to the refactoring in these projects to fail, since some `Entity` classes have a corresponding `Repository` class and others have a DAO class (8 projects). MicroRefact can only refactor `Entity` classes that have a corresponding `Repository` class.

The other reasons why refactoring is not possible are also implementation reasons. MicroRefact only supports JPA annotations (2 cases) and only searches for the primary key in instance variable annotations (1 case).

Overall, MicroRefact can be extended to support some of the cases where it failed, namely the use of DAOs, other ORM frameworks and annotations on methods. The remaining cases cannot be included in MicroRefact because they go against the proposed methodology because the lack of `Id` annotation on the classes mapped as entities and the lack of `Repository` classes for some entities may compromise the refactoring of the application.

4.4. Threats to Validity

In this section we discuss some of the threats to the validity of the evaluation presented.

A possible threat is related to the use of only open source applications in the evaluation. However, it is common to find companies that make their code available in GitHub. Nevertheless, it is possible that the results may vary for proprietary software.

Another threat is the quality of the microservice proposals generated for the projects. The tool used to generate microservice proposals does not guarantee that the decomposition it proposes is the best possible. Furthermore, its authors did not perform an extensive study of how the number of topics and

the resolution directly affect the microservice proposal. Instead, they defined arbitrary ranges for each parameter. Nevertheless, they evaluated the quality of the proposed solution by using the microservice architecture metrics proposed by [8] and the results were positive.

Another threat is the unit tests available by the projects. Some projects present no tests and others present just a few tests. For the projects that do not present tests it was not possible to evaluate if from the functional point of view the generated application is equal to the original. For the applications that present few tests, these can be class-focused, not testing the interactions between classes, leaving several interactions between classes of different microservices to be tested. Nevertheless, for the 28 projects which had tests, the results were exactly the same in the original and refactored application.

5. Related Work

Several authors studied the migration process from different perspectives.

In previous work [19] we have proposed an initial version of this work. In this current work we have extended our work by including the algorithms used in the migration methodology, we have added support for inheritance to the methodology, which was not possible before, and we have detailed the different refactoring patterns used and how they are applied (Section 2). We have also added an extensive quantitative and qualitative evaluation using 120 software projects (Section 4).

Balalaie *et al.* [20], 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 [21] 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 [12] 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.

Fowler and Lewis [13] 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 [5, 6]. 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 [7, 8]. Model-based solutions allow the use of models to support migrations since models also represent a view over the interactions between system's components [9, 10]. Tyszberowicz *et al.* [22] 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 [11] 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 a microservices proposal as input refactors the original monolithic application. We built a tool termed MicroRefact, as a proof of concept, that supports Java Spring applications that use JPA annotations. Using this tool we performed a quantitative evaluation against a collection of 120 open-source Java Spring applications from GitHub. The results show that more than 70% of the applications were automatically refactored and that part of the remaining projects could not be refactored because of potential flaws or dead code contained in them. Moreover, we executed all the unit tests contained in the projects achieving the same results in the original and refactored application. This shows the refactoring did not change the functional behavior of the applications (evaluated in the unit tests).

As future work, we envision the design and implementation of capabilities to migrate the code into other programming 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] S. J. Andriole, The death of big software, Commun. ACM 60 (12) (2017) 29–32. doi:10.1145/3152722.
URL https://doi.org/10.1145/3152722

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

[3] J. Fritzsch, J. Bogner, S. Wagner, A. Zimmermann, Microservices migration in industry: Intentions, strategies, and challenges, in: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2019, pp. 481–490. doi:10.1109/ICSME.2019.00081.

[4] M. Mazzara, N. Dragoni, A. Bucchiarone, A. Giaretta, S. T. Larsen, S. Dustdar, Microservices: Migration of a mission critical system, IEEE Transactions on Services Computing (2018) 1–1doi:10.1109/TSC.2018.2889087.

[5] G. Mazlami, J. Cito, P. Leitner, Extraction of microservices from monolithic software architectures, in: 2017 IEEE International Conference on Web Services (ICWS), 2017, pp. 524–531. doi:10.1109/ICWS.2017.61.

[6] M. Kamimura, K. Yano, T. Hatano, A. Matsuo, Extracting candidates of microservices from monolithic application code, in: 2018 25th Asia-Pacific Software Engineering Conference (APSEC), 2018, pp. 571–580.

[7] W. Jin, T. Liu, Q. Zheng, D. Cui, Y. Cai, Functionality-oriented microservice extraction based on execution trace clustering, in: 2018 IEEE International Conference on Web Services (ICWS), 2018, pp. 211–218.

[8] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, Q. Zheng, Service candidate identification from monolithic systems based on execution traces, IEEE Transactions on Software Engineering 47 (5) (2021) 1–1. doi:10.1109/TSE.2019.2910531.

[9] R. Chen, S. Li, Z. Li, From monolith to microservices: A dataflow-driven approach, in: 2017 24th Asia-Pacific Software Engineering Conference (APSEC), 2017, pp. 466–475. doi:10.1109/APSEC.2017.53.

[10] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann, Service cutter: A systematic approach to service decomposition, in: M. Aiello, E. B. Johnsen, S. Dustdar, I. Georgievski (Eds.), Service-Oriented and Cloud Computing, Springer International Publishing, Cham, 2016, pp. 185–200.

[11] Y.-W. Kwon, E. Tilevich, Cloud refactoring: Automated transitioning to cloud-based services, Automated Software Engineering 21. doi:10.1007/s10515-013-0136-9.

[12] F. Auer, V. Lenarduzzi, M. Felderer, D. Taibi, From monolithic systems to microservices: An assessment framework, Information and Software Technology 137 (2021) 106600. doi:https://doi.org/10.1016/j.infsof.2021.106600.
URL https://www.sciencedirect.com/science/article/pii/S0950584921000793

[13] M. Fowler, Stranglerfigapplication, https://martinfowler.com/bliki/StranglerFigApplication.html, (Accessed on 11/20/2020) (June 2004).

[14] M. Brito, J. Cunha, J. Saraiva, Identification of microservices from monolithic applications through topic modelling, in: Proceedings of the 36th Annual ACM Symposium on Applied Computing, Association for Computing Machinery, New York, NY, USA, 2021, p. 1409–1418.
URL https://doi.org/10.1145/3412841.3442016

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

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

[17] M. Silva, Improving the resilience of microservices-based applications, Master's thesis, University of Minho (2021).
URL https://hdl.handle.net/1822/81099

[18] F. Freitas, Refactoring Java monoliths into executable microservice-based applications, Master's thesis, University of Minho (2022).
URL https://hdl.handle.net/1822/79920

[19] F. Freitas, A. Ferreira, J. Cunha, Refactoring java monoliths into executable microservice-based applications, in: 25th Brazilian Symposium on Programming Languages, Association for Computing Machinery, New York, NY, USA, 2021, p. 100–107.
URL https://doi.org/10.1145/3475061.3475086

[20] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. Tamburri, T. Lynn, Microservices migration patterns, Software: Practice and Experience 48. doi:10.1002/spe.2608.

[21] R. Lichtenthäler, M. Prechtl, C. Schwille, T. Schwartz, P. Cezanne, G. Wirtz, Requirements for a model-driven cloud-native migration of monolithic web-based applications, SICS Software-Intensive Cyber-Physical Systems 35 (1) (2020) 89–100. doi:10.1007/s00450-019-00414-9.
URL https://doi.org/10.1007/s00450-019-00414-9

[22] S. Tyszberowicz, R. Heinrich, B. Liu, Z. Liu, Identifying microservices using functional decomposition, in: X. Feng, M. Müller-Olm, Z. Yang (Eds.), Dependable Software Engineering. Theories, Tools, and Applications, Springer International Publishing, Cham, 2018, pp. 50–65.

Appendix A. Expanded code for the example show in Section 2

```
class ReservationRequestImpl implements ReservationRequest{

  private RestTemplate restTemplate;

  public void setReservations(List<Reservation> reservations
      , Long userId){
    restTemplate.put("http://6/User/{id}/Reservation/
        setReservations", reservations, userId);
  }

  public List<Reservation> getReservations(Long userId){
    List<Reservation> aux = restTemplate.getForObject("http
        ://6/User/{id}/Reservation/getReservations", List<
        Reservation>.class, userId);
    return aux;
  }
}
```

Listing 10: Generated class that is responsible for the calls to the `Reservation` microservice.

```
@RestController
@CrossOrigin
class ReservationUserController {

  private ReservationUserService reservationuserservice;

  @PutMapping
  ("/User/{id}/Reservation/setReservations")
  public void setReservations(@PathVariable(name="id") Long
      userId, @RequestBody List<Reservation> reservations){
    reservationuserservice.setReservations(userId,
        reservations);
  }

  @GetMapping
  ("/User/{id}/Reservation/getReservations")
  public List<Reservation> getReservations(@PathVariable(
      name="id") Long userId){
    return reservationuserservice.getReservations(userId);
  }
}
```

Listing 11: Generated class that exposes the API to handle `Reservations`.

```
@Service
class ReservationUserService {

  private ReservationRepository reservationrepository;
```

```
5
6    public void setReservations(Long userId, List<Reservation>
         reservations){
7      reservationrepository.setReservations(userId,
           reservations);
8    }
9
10   public List<Reservation> getReservations(Long userId){
11     return reservationrepository.getReservations(userId);
12   }
13 }
```

Listing 12: Generated class that processes and directs the request to the repository class.