Towards a Technique for Extracting Relational Actors from Monolithic Applications

Authors omitted

[1]Address omitted

Abstract. *Relational actors, or reactors for short, integrate the actor model with the relational data model, providing an abstraction for enabling actor-relational database systems. However, as a novel model of computation for databases, there is no extensive work on reasoning about reactor modeling. To close this gap, this paper aims to review methods for systems decomposition in order to analyze their completeness and propose as well as evaluate a technique for reactor modeling. Concretely, we put forward a technique to extract reactors from a monolithic system. For evaluation, we selected a REST-based open-source OLTP system in which a decomposition to microservices was conducted and applied our technique on its predecessor monolithic version. Our technique led to the same set of decisions, regarding table and behavior selection, taken by experts when decomposing the same system into microservices. The proposed technique can be seen as a first step towards supporting practitioners in decomposing OLTP systems into reactors.*

1. Evaluation Extended Version

An evaluation was was performed in order to assess the feasibility of the approach. This section describes the evaluation context, the application of the technique, and the results.

1.1. Evaluation context

Our technique is applied to the monolithic version of Petclinic[1], an OLTP open-source demonstration project of the Spring Framework.[2] The system adopts a three-tier layered architecture and has been in development since 2016. Petclinic allows the configuration of different types of relational DBMS, such as HSQLDB, MySQL and PostgreSQL.

1.2. Technique application

1.2.1. Dependency graph

In order to build the dependency graph of each interface of Petclinic application, it is important to assess the domain model, as shown on figure 1.

Based on source code, classes Owner, Pet, Vet, PetType, and Visit are annotated with Hibernate [3] framework annotations, a JPA implementation in Java platform. From annotation, we are able to retrieve how hibernate map classes to database tables.

Interfaces are identified as resources in Petclinic web application by the annotation *@RequestMapping*. Again, a request is an intent of a client, a communication being

[1]https://github.com/spring-petclinic/spring-framework-petclinic

[2]https://spring.io

[3]https://hibernate.org

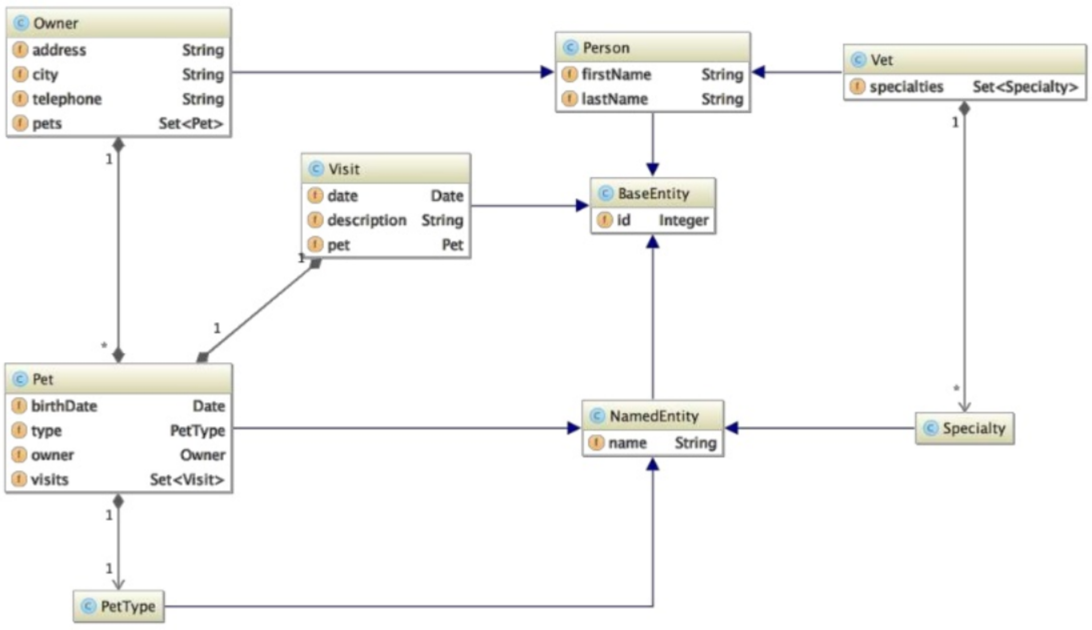


Figure 1. Petclinic domain model

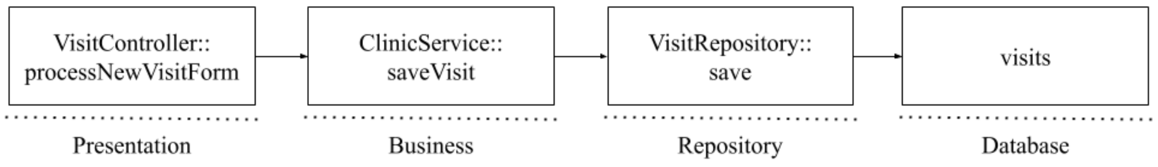


Figure 2. Petclinic dependency graph for interface /owners, GET operation

established to the system. This way, interfaces are the responsible for receiving a client request and handling the request to appropriate classes of the application (usually business classes).

Once the domain, interfaces and model classes are known, business functions and repository functions related to the set of interfaces must be identified. The dependency graph for each available resource in Petclinic is provided below.

1.2.2. Profile data collection

Once Petclinic is a demonstration project, this study relies on an artificial workload that aims at reproducing a real-world scenario for the Petclinic domain. The workload decisions were taken based on characteristics of Petclinic, e.g., the insertion rate into the *pets* table cannot be lower than of owners (a *pet* cannot exist without an *owner*) or table *visits* must incur the highest access frequency. Moreover, it is worthy to mention that the access frequency value of 100 was chosen to represent the maximum frequency the application can sustain for a given interface (minimum is 1).

Two different workloads were created and they consists of a typical transactional scenario for a web application such as Petclinic. Tables 1 and 2 represent the workload scenarios, also providing the respective HTTP operation, interface and the table where the

Operation	Interface	Table	Access frequency
GET	/owners/ownerId	owners	10
GET	/owners/ownerId/edit	owners	10
PUT	/owners/ownerId/edit	owners	10
GET	/owners	owners	60
POST	/owners/new	owners	20
GET	/owners/ownerId/pets/new	pets	25
POST	/owners/ownerId/pets/new	pets	25
GET	/owners/ownerId/pets/petId/edit	pets	10
PUT	/owners/ownerId/pets/petId/edit	pets	10
GET	/vets	vets	10
GET	/owners/ownerId/pets/petId/visits/new	visits	100
POST	/owners/ownerId/pets/petId/visits/new	visits	100

Table 1. Workload 1

Operation	Interface	Table	Access frequency
GET	/owners/ownerId	owners	10
GET	/owners/ownerId/edit	owners	60
PUT	/owners/ownerId/edit	owners	60
GET	/owners	owners	80
POST	/owners/new	owners	20
GET	/owners/ownerId/pets/new	pets	25
POST	/owners/ownerId/pets/new	pets	25
GET	/owners/ownerId/pets/petId/edit	pets	10
PUT	/owners/ownerId/pets/petId/edit	pets	10
GET	/vets	vets	10
GET	/owners/ownerId/pets/petId/visits/new	visits	100
POST	/owners/ownerId/pets/petId/visits/new	visits	100

Table 2. Workload 2

operation is accomplished.

1.2.3. Table coupling identification

The entity-relationship (ER) diagram for the Petclinic application is depicted in Figure 3. As can be seen in Table 3, the relationships with coupling equal to 1 are: types and pets, owners and pets, and visits and pets. Table 3 exhibits the degree of coupling between tables:

1.2.4. Reactor types identification

We aim to distribute workload among reactors, avoiding two data-intensive entry points to be allocated in the same reactor type. The objective is to take advantage of the trans-

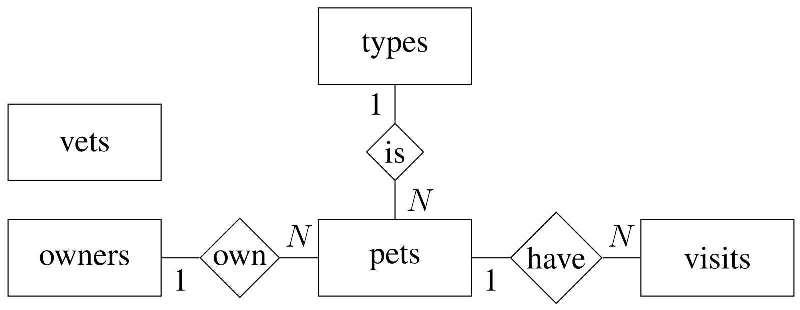


Figure 3. Petclinic ER diagram

Tables	Coupling
types and pets	1
owners and pets	1
types and owners	0
vets and pets	0
vets and owners	0
vets and types	0
vets and visits	0
visits and pets	1
visits and owners	0
visits and types	0

Table 3. Coupling between Petclinic tables

actional guarantees and serializability provided in order to distribute workload among reactors in a deployment.

Thus, in order to execute the model for the first workload, the parameter Q was set to 200, corresponding to the sum for the resource with the highest access frequencies (*visits*). On the other side, the parameter Q is set to 270 for the second workload. Figures 4 and 5 exhibit the result of the optimization allocating tables to clusters for workload 1 and 2, respectively.

The execution of the optimization problem for the two mentioned scenarios have provided two different distributed database design. In addition, it is important to depict the interfaces allocated to each cluster. The aggregation of interfaces and tables allow the definition of reactor types, as shown in tables 4, 5, and 6.

On the other side, the workload 2 provided the following reactor types, as shown in tables 7, 8, and 9:

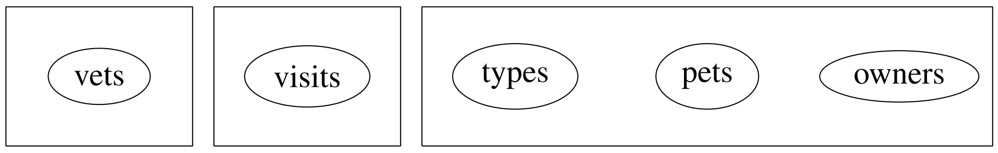


Figure 4. Optimization problem output for workload 1

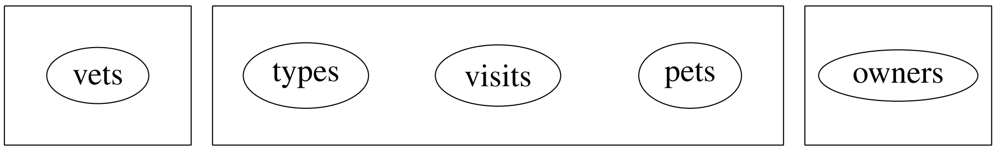


Figure 5. Optimization problem output for workload 2

Operation	Interface	Table
GET	/owners/ownerId	owners
GET	/owners/ownerId/edit	owners
PUT	/owners/ownerId/edit	owners
GET	/owners	owners
POST	/owners/new	owners
GET	/owners/ownerId/pets/new	pets
POST	/owners/ownerId/pets/new	pets
GET	/owners/ownerId/pets/petId/edit	pets
PUT	/owners/ownerId/pets/petId/edit	pets
		types

Table 4. Reactor type 1 for workload 1

Operation	Interface	Table
GET	/owners/ownerId/pets/petId/visits/new	visits
POST	/owners/ownerId/pets/petId/visits/new	visits

Table 5. Reactor type 2 for workload 1

Operation	Interface	Table
GET	/vets	vets

Table 6. Reactor type 3 for workload 1

Operation	Interface	Table
GET	/owners/ownerId	owners
GET	/owners/ownerId/edit	owners
PUT	/owners/ownerId/edit	owners
GET	/owners	owners
POST	/owners/new	owners

Table 7. Reactor type 1 for workload 2

Operation	Interface	Table
GET	/owners/ownerId/pets/petId/visits/new	visits
POST	/owners/ownerId/pets/petId/visits/new	visits
GET	/owners/ownerId/pets/new	pets
POST	/owners/ownerId/pets/new	pets
GET	/owners/ownerId/pets/petId/edit	pets
PUT	/owners/ownerId/pets/petId/edit	pets
		types

Table 8. Reactor type 2 for workload 2

Operation	Interface	Table
GET	/vets	vets

Table 9. Reactor type 3 for workload 2

1.2.5. Reactor methods identification

For this step, a manual verification was performed in Petclinic source code. Based on the strategy discussed in subsection 4, the metrics used for Petclinic were: CYCLE(M) $\leq$ 12 $\wedge$ NOAV(M) $\leq$ 5 $\wedge$ DDLOC(M) $\geq$ 2. Then, it was possible to identify a set of nine methods. The extracted methods are follows.

Based on the procedure defined in section **??**, the table below shows on which reactor type each method identified is allocated.

2. Evaluation results

The results suggest that the approach can be successfully applied to real world monolithic applications.

Based on workload 1, it is possible to correlate the distribution of relational actors and its respective tables and interfaces to the project Petclinic microservices [21], which provides the decomposition shown in figure 16.

```
reactor Vet {
  ...
  void upsert_owner(owner){
`    SELECT DISTINCT
        vet.firstName, vet.lastName, vet.name
    INTO v_vets
    FROM vets;

    return v_vets;
  }
}
```

Figure 6. Function that retrieve all tuples from vets table

```
reactor Owner {
  ...
  void upsert_owner(owner){
    if owner.id IS NULL then
        INSERT INTO owners VALUES
            (owner.address, owner.city, owner.telephone,
            owner.firstName, owner.lastName);
        return;
    end if;

    SELECT id FROM owners INTO o_id WHERE owner.id = id;

    if o_id IS NULL then abort; end if;

    UPDATE owners
    SET address = owner.address,
        city = owner.city,
        telephone = owner.telephone,
        firstName = owner.firstName,
        lastName = owner.lastName
    WHERE owner.id = id;
  }
}
```

Figure 7. Function that insert or update a tuple to owners table

Method	Reactor Type
1	3
2	1
3	1
4	1
5	1
6	1
7	1
8	2
9	2

Table 10. Method mapping to reactor types for workload 1

```
reactor Owner {
  ...
  void find_owner_by_id(owner_id){
    SELECT DISTINCT
        owners.id, owners.address, owners.city,
        owners.telephone, owners.firstName,
        owners.lastName, owners.name
    INTO owner
    FROM owners
    WHERE owners.id = owner_id;

    res := find_pet_by_owner_id(owner_id)
        on reactor pet;
    list<tuple> pets = res_pet.get();

    SELECT
        owners.id, owners.address, owners.city,
        owners.telephone, owners.firstName,
        owners.lastName, owners.name
        pets.birthDate, pets.name, pets.type_name
    INTO owner_info
    FROM owner
    LEFT JOIN pets ON pets.owner_id = owner.id;

    return owner_info;
  }
}
```

Figure 8. Function that retrieves all owner that matches the owner_id provided (in case both vets and owners are in different reactors)

Method	Reactor Type
1	3
2	1
3	1
4	1
5	1
6	2
7	2
8	2
9	2

Table 11. Method mapping to reactor types for workload 2

```
reactor Visit {
  ...
  tuple find_owner_by_id(owner_id){
    SELECT DISTINCT
        owners.address, owners.city, owners.telephone,
        owners.firstName, owners.lastName, owners.name,
        pets.birthDate, pets.name, types.name
    INTO owner_info
    FROM owners
    LEFT JOIN pets ON pets.owner_id = owners.id
    LEFT JOIN types ON pets.type_id = types.id
    WHERE owners.id = owner_id;

    return owner_info;
  }

  list<tuple> find_owner_by_last_name(last_name){
   if last_name IS NOT NULL then {
     SELECT DISTINCT
        owners.address, owners.city, owners.telephone,
        owners.firstName, owners.lastName, owners.name,
        pets.birthDate, pets.name, types.name
    INTO v_owners
    FROM owners
    LEFT JOIN pets ON pets.owner_id = owners.id
    LEFT JOIN types ON pets.type_id = types.id
    WHERE owners.lastName LIKE lastName
    return v_owners;
  }
   SELECT DISTINCT
        owners.address, owners.city, owners.telephone,
        owners.firstName, owners.lastName, owners.name,
        pets.birthDate, pets.name, types.name
    INTO v_owners
    FROM owners
    LEFT JOIN pets ON pets.owner_id = owners.id
    LEFT JOIN types ON pets.type_id = types.id
    return v_owners;
  }

}
```

Figure 9. Function that retrieves all owner that matches the owner_id provided (in case both vets and owners are in the same reactor)

```
reactor Pets {
  ...
  void upsert_pet(pet){
    if pet.id IS NULL then
        INSERT INTO pets VALUES
            (pet.name, pet.birthDate,
            pet.type_id, pet.owner_id);
        return;
    end if;

    SELECT id FROM pets INTO p_id WHERE pet.id = id;

    if p_id IS NULL then abort; end if;

    UPDATE pets
    SET date = pet.name,
        birthDate =  pet.birthDate,
        type_id =  pet.type_id,
        owner_id =  pet.owner_id
    WHERE pet.id = id;
  }
}
```

Figure 10. Function that insert or update a tuple to pets table

```
reactor Pets {
  ...
  void find_pet_types(){
    SELECT DISTINCT types.name FROM types INTO pet_types;

    return pet_types;
  }
}
```

Figure 11. Function that insert or update a tuple to pets table

```
reactor Pets {
  ...
  void find_pet_by_id(pet_id){
    SELECT DISTINCT
        pet.id, pets.birthDate,
        pets.name, types.name as type_name
    INTO v_pet
    FROM pets
    LEFT JOIN types ON types.id = pets.type_id
    WHERE pets.id = pet_id;

    return v_pet;
  }
}
```

Figure 12. Function that retrieve a tuple of pet based on id

```
reactor Visit {
  ...
  void find_visits_by_pet_id(pet_id){
    res_pet := find_pet_by_id(pet_id) on reactor pet;
    list<tuple> pets = res_pet.get();

    SELECT DISTINCT
        visits.date, visits.description, visits.pet_id,
        pets.birthDate, pets.name, pets.type_name
    INTO pet_visits
    FROM visits
    LEFT JOIN pets ON pets.id = visits.pet_id
    WHERE visits.pet_id = pet_id;

    return pet_visits;
  }
}
```

Figure 13. Function that retrieves all visits that matches the pet_id provided (in case vets, types, and visits are in different reactors)

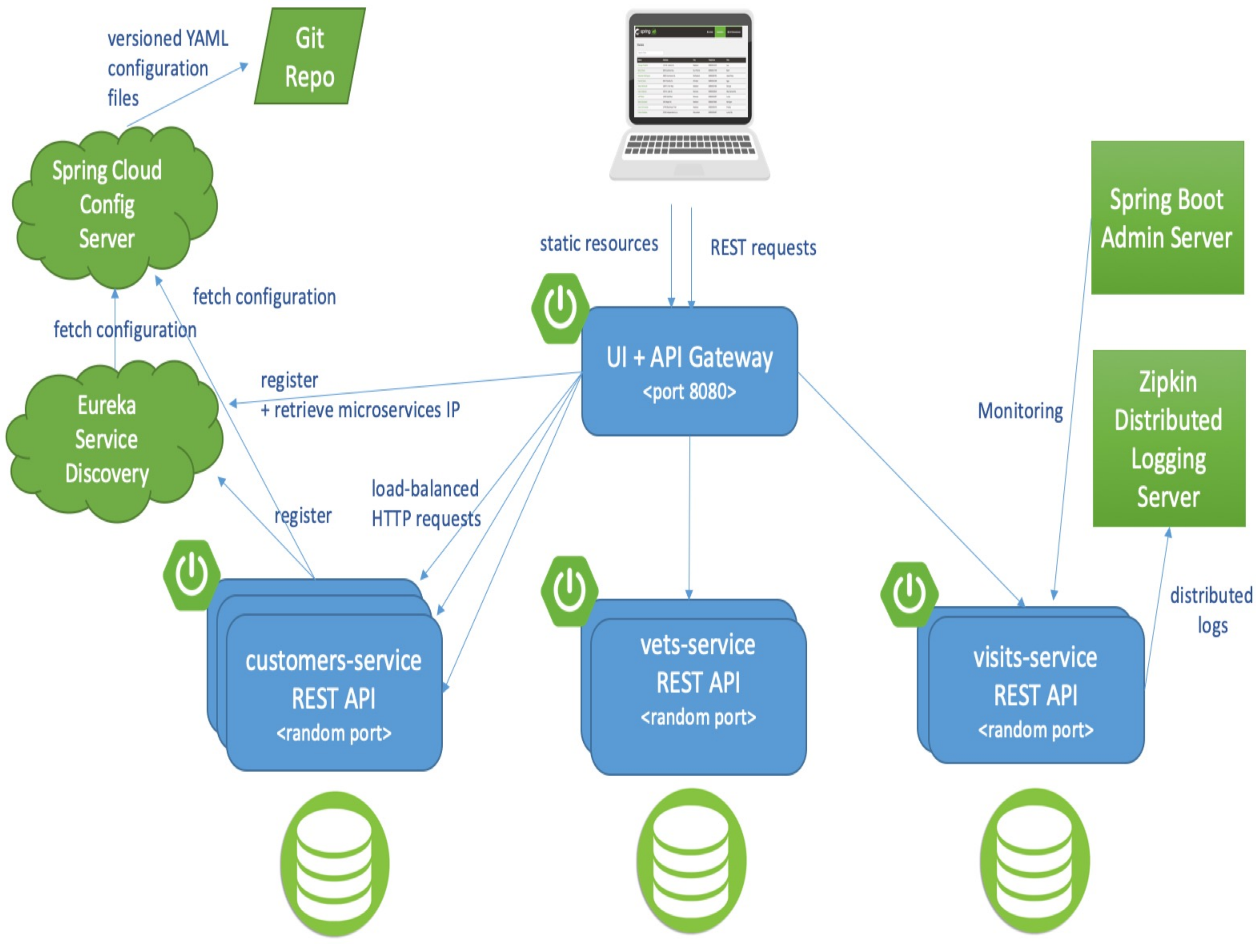
```
reactor Visit {
  ...
  void find_visits_by_pet_id(pet_id){
    SELECT DISTINCT
        visits.date, visits.description, visits.pet_id,
        pets.birthDate, pets.name, types.name
    INTO pet_visits
    FROM visits
    LEFT JOIN pets ON pets.id = visits.pet_id
    LEFT JOIN types ON pets.type_id = types.id
    WHERE visits.pet_id = pet_id;

    return pet_visits;
  }
}
```

Figure 14. Function that retrieves all visits that matches the pet_id provided (in case both vets and visits are in the same reactor)

```
reactor Visit {
  ...
  void upsert_visit(visit){
    if visit.id IS NULL then
        INSERT INTO visits VALUES
            (visit.date, visit.description, visit.pet_id);
        return;
    end if;

    SELECT id FROM visits INTO v_id WHERE visit.id = id;

    if v_id IS NULL then abort; end if;

    UPDATE visits
    SET date = visit.date,
        description = visit.description,
        pet_id = visit.pet_id
    WHERE visit.id = id;
  }
}
```

Figure 15. Function that insert or update a tuple to visits table

Figure 16. Architecture diagram of the Spring Petclinic Microservices (extracted from [21]


versioned YAML configuration files
Git Repo
Spring Cloud Config Server
fetch configuration
fetch configuration
Eureka Service Discovery
register + retrieve microservices IP
register
load-balanced HTTP requests
static resources
REST requests
UI + API Gateway <port 8080>
Spring Boot Admin Server
Zipkin Distributed Logging Server
Monitoring
distributed logs
customers-service REST API <random port>
vets-service REST API <random port>
visits-service REST API <random port>

Microservice	Tables
customers-service	owners, types, and pets
vets-service	vets
visits-service	visits

Table 12. Example of resources definition

Petclinic microservices provides the following microservices: customers-service, vets-service, and visits-service. Table 12 depicts the tables presented in each microservice in Petclinic microservices application.

It is possible to observe that the decomposition provided by Petclinic microservices is exact the same as the decomposition provided by the optimization problem on workload 1, in terms of the tables selected for each service.

It means that the formulation for defining reactor types proposed in this work is able to employ a correlated set of decisions that architects from Petclinic application have made in order to decompose Petclinic into microservices archictecture style.

References