Solving the Relational Data Access Problem with Data Access Services

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Vesion is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Solving the Relational Data Access Problem with Data Access Services

Ken Wang [0009-0002-3064-3421]

BackLogic LLC, Chino Hills, CA 91709, USA

Abstract. The object-relational impedance mismatch problem consists of two sub-issues: language impedance and data structure mismatch. Solution to the first issue is to abstract away the programming language from SQL; to the second issue is to abstract away the object-relational transformation from developer. This paper presents three standardized data access services, *query*, *command* and *repository,* collectively as a preferable solution to relational data access. These services are specified declaratively with SQL and JSON. The hard object-relational transformation is delegated to service engine. This paper also presents a reference implementation of the service engine for service execution. The complexity test proves that this reference service engine is capable of handling highly complex objects up to the *5x5* level on the complexity scale developed in this research.

Keywords: object, relational, database, data access, persistence

1 Introduction

Relational data access (RDA) is an old but not well-solved problem. ORM (object relational mapping) is intended to bridge the gap between the object and relational world, but has drawn as much criticism as praise over the years [1] [2]. Like raw SQL, it is incapable of gracefully handling even moderately complex objects. As a result, developers are forced to compromise on their object design, which takes a lot of flexibility and productivity away from them.

People have hoped that NoSQL database could provide a solution to the problem. However, as it stands now, relational database (RDBMS) is still the dominating database according to data from DB Engines Ranking [3] and will likely remain so solely because of the maturity and superiority of relational data model. This situation leaves us no choice but to find a better solution to relational data access.

In this research, we have reexamined the nature of object-relational impedance mismatch and realized that this impedance mismatch problem actually consists of two sub-issues: the *language impedance* between embedded SQL and the host language, and the *data structure mismatch* between the object model and the relational data set. We believe that the solution to the first issue is to abstract away the host language from SQL, or to separate SQL from the host language, and that the solution to the second issue is to abstract away the object-relational transformation from developer and delegate it to data access framework/library.

In this research, we have also reexamined the familiar DAO (data access object) and repository design patterns and realized that the whole RDA problem actually comprises only three basic data access patterns: *query*, *command* and *repository*.

These findings lead us to the idea of reducing the generic DAO and repository design patterns to three standardized data access services (DAS), namely *query*, *command* and *repository* services. These services will be specified declaratively with SQL and JSON, rather than being coded with an imperative programming language, and will be executed with a service engine residing on the data access tier, as opposed to a framework/library embedded in the data access layer. These DAS are conceptually simple and thus easy to learn and quick to develop. Most importantly, they delegate the hard object-relational transformation problem to service engine, while providing developers with the flexibility to design their object model according to application needs rather than framework constraints.

Due to space limitations, this paper provides only a brief description of the structure, composition, development process and test results of these DAS. The examples are available on GitHub [8] for public review.

To prove the concept of DAS, a reference service engine, along with a service builder tool, is implemented. Example services are developed with the service builder and tested with the reference service engine.

The success of the DAS concept is to a great extent determined by the capability of the service engine to handle complex objects. The complexity test shows that the reference engine is capable of handling objects at level *5 x 5* on the complexity scale developed in this research for measuring the complexity of an object structure. At this level, the object is a nested structure with 5 array structures in both the vertical and horizontal directions. It is unlikely that a real-world application would need data structures of this complexity, and thus this test virtually assures developers of the total flexibility in their object design.

2 Data Access Services

2.1 Query Service

Query service is for retrieving data from data source. It takes an input object of query parameters and returns an object or an array of objects as output.

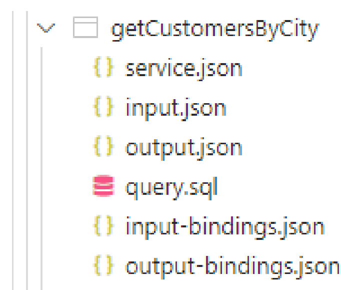

Fig. 1. Structure of query service

Fig.1 illustrates the structure of the example query service *getCustomersByCity*. Table 1 lists the JSON and SQL files composing the query service. The *service.json* file is not a component of the service but contains the meta data of the service.

Table 1. List of query files

File	Description
input.json	json file for service input
output.json	json file for service output
query.sql	sql file for service query
input-bindings.json	json file for input bindings that map query parameters to input data fields
output-bindings.json	json file for output bindings that map output data fields to query columns

The input and output define what the query service is for; the SQL query implements the data access logic of the service. To specify the query service, the user first

- Specifies the input and output for the service with JSON in the input and output files; then
- Composes the SQL query to retrieve data in the query file; and lastly
- Generates the input and output bindings from the service input, output and query.

The last step is to be assisted by service builder, a development tool for DAS, and be done with a click of a button. The output, query and output-bindings code for the *getCustomersByCity* service are listed below for illustration.

Output:

```
[{
    "customerNumber": 1,
    "customerName": "Land of Toys Inc.",
    "address": {
        "addr": "NYC",
        "city": "NYC"
    }
}]
```

Query:

```
select customerNumber, customerName, address, city
  from customers where city = :city
```

Output-Bindings:

```
[
   {"field": "..customerNumber", "column": "customerNumber"},
   {"field": "..customerName", "column": "customerName"},
   {"field": "..address.addr", "column": "address"},
   {"field": "..address.city", "column": "city"}
]
```

The query service provides developer with the flexibility to retrieve any ad hoc data structure per application needs.

2.2 Command Service

The command service is for writing and changing data in data source. The basic command service takes an input object of SQL parameters and returns nothing. The advanced version appends a query to the command. Fig.2 illustrates the structure of the basic command service *cloneProductLine*.

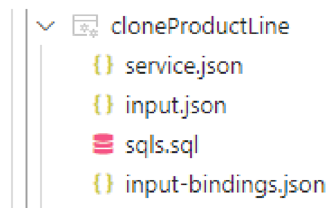

Fig. 2. Structure of command service

Table 2 lists the JSON and SQL files composing the command service.

Table 2. List of command files

File	Description
input.json	json file for service input
sqls.json	sql file for service DML statements
input-bindings.json	json file for input bindings that map sql parameters to input data fields

To specify the command service, the user first

- Specifies the input object for the service with JSON in the input file; then
- Composes a sequence of DML statements to be executed for the command service in the *sqls* file; and lastly
- Generates the input bindings from the input and SQL statements.

The JSON and SQL code for the *cloneProductLine* service are not included in this paper but can be found on GitHub [8], along with many other command examples.

The command service provides developer with the flexibility to perform any ad hoc data loading and any ad hoc data changes per the application needs.

2.3 Repository Service

Repository service is for CRUD operations of aggregate root object. It includes both read operations for querying the object and write operations for creating, updating and deleting the object. Fig.3 illustrates the structure of the repository service *Order*.

Fig. 3. Structure of repository service

Table 3 lists the JSON and SQL files composing the repository service.

Table 3. List of repository files

Component	File	Description
object	object.json	json file for service object
read	input.json	json file for read input
	query.sql	sql file for read query
	input-bindings.json	json file for input bindings that map query parameters to input data fields
	output-bindings.json	json file for output bindings that map object data fields to query columns
write	tables.json	json file for table bindings that map object structures to tables
	orders.columns.json	json file for column bindings that maps object data fields to table columns.
	orderdetails.columns.json	Same as above. One file per table.

The object component defines what the repository service is for; the read component specifies a query service for the object; the write component comprises a set of table and column bindings (aka mappings) for dynamic generation of insert, update and delete statements. To specify the repository service, the user first

- Specifies the object for the service with JSON in the object file; then
- Specifies the query service for the read component (the output of the query service is the object and needs not to be specified again); and lastly
- Generates the table and column bindings from the object, query and DB meta.

The last step is to be assisted by service builder and be done with a click of a button. The JSON and SQL code for the *Order* service are not included in this paper but can be found on GitHub [8], along with many other repository examples.

The read component of the repository service is designed to be a dynamic query, implying that the *WHERE* clause of the query will be dynamically modified at execution, based on the input parameters coming from the service call, to accommodate the different query needs for the object, such as *getOrderById*, *getOrdersByCustomer*, *getOrdersByStartAndEndDates*, etc.

The write component of the repository service is to be derived by design from the read component. The tables to write are derived from the query; the column data are both derived from the query and retrieved from database meta data; and the table and column bindings are helped by the existing output bindings. For this purpose, a set of rules is imposed on the read query. For example, the table join must be in ANSI format; the tables must be aliased; the columns must be prefixed with table alias; the alias for reference table (aka read-only table) must start with "_", so that it can be excluded from the table bindings, etc. However, most of these rules are also best practices.

The repository service is for persistence of domain objects. Because of its capability of handling complex aggregate root object, it is expected to be a good fit for domain-driven design (DDD) that is driving microservices development these days.

3 Reference Engine

Just like SQL is executed by SQL engine, DAS is executed by service engine. The concept of DAS hinges on the successful implementation of the service engine. Additionally, the concept of DAS is bundled with a builder tool for automatic generation of input, output, table and column bindings. Therefore, to prove the concept of DAS, we have implemented a reference service engine, along with a service builder.

3.1 Service Builder

The service builder is implemented as a VS Code extension [7], so that the SQL and JSON can be developed in specialized SQL and JSON editors, respectively. The SQL editor is to be connected to the data source, so that the SQL development can be done in a database-centric environment.

An important feature of this service builder is its capability of generating input and output bindings for query and command services and its capability of generating table and column bindings for repository service. For this purpose, a SQL parser and a fuzzy data field matcher have been installed with the service builder.

A slew of example query, command and repository services [8] have been developed with this service builder in a streamlined process, as described in the previous section.

3.2 Service Engine

The service engine is implemented as a library component and embedded in the service builder [7], so that the service can be tested as it is developed. In this research, the service engine is tested with the various example query, command and repository services developed using the service builder, and it has worked as designed for all.

However, the real success of the service engine is measured by its capability of handling complex objects, which is what distinguishes DAS from other approaches. Hence, we have developed a complexity scale for measuring object complexity and carried out a complexity test on the service engine, as discussed in the following sections.

4 Complexity Scale

The difficulty of object-relational transformation increases with the complexity of the target object structure. To gauge this difficulty, a five-level complexity scale is developed in this research for measuring the complexity of the object structure, as described in the following.

At level ***0 x 0***, the object is the same as the relational data set.

At level ***1 x 1***, the object is different from the relational data set but does not contain any array structure. The object may include nested object structure vertically and horizontally. Nonetheless, a skilled developer can easily handle this type of objects.

At level ***1 x n***, the object includes nested array structure only vertically, as illustrated below:

```
{
    "id1": 1,
    "a1": [{
        "id2": 2,
        "a2": [{
            "id3": 3
                ...
        }]
    }]
}
```

A skilled developer can comfortably handle objects at level *1 x 2*. Beyond this level, it would be a hard stretch and may require the infamous N+1 technique.

At level ***n x 1***, the object includes nested array structure only horizontally, as shown in the following:

```
{
  "id1": 1,
  "a2": [{"id2": 1}],
  "a3": [{"id3": 1}]
    ...
}
```

Horizontally nested array structure is more difficult to handle than vertically nested array. Even a *1 x 2* structure is quite hard to a skilled developer.

At level ***n x n***, the object includes both vertically and horizontally nested array structures. This is the hardest level to deal with.

This 5-level complexity scale could be used as a standard for measuring the complexity of an object and the capacity of an RDA solution. Note that, at all levels, the difficulty of object-relational transformation is measured by transforming a relational data set into an *array* of the objects.

5 Complexity Test

5.1 Test Service

The capability of service engine to handle complex objects is measured by its capability to *query* complex objects. Therefore, a set of specially designed query services are used for the test. The test service does not take any input but returns an array of objects of different complexity per the test level.

For example, the test output for a *3 x 3* test would look like:

```
[{
  "id11": 1,
  "a2": [{
    "id22": 1,
    "b2": [{
      "id23": 1
     }]
  }],
  "a3": [{
    "id32": 1,
    "b2": [{
      "id33": 1
     }]
  }]
}]
```

It contains 3 array structures in both the vertical and horizontal directions. To provide data for the test output, a test query is prepared for the test service to produce a data set of 5 data fields and 8 rows for the *3 x 3* test, as shown in table 4.

5.2 Test Result

Once executed, the *3 x 3* test service is expected to return a data structure the same as the test output defined for the test service, but with two records in each of the 5 array structures. The test result has come out exactly as expected.

Table 4. Data records from test query.

Id11	Id22	Id23	Id32	Id33
1	1	1	1	1
1	1	2	1	2
1	2	3	2	3
1	2	4	2	4
2	3	5	3	5
2	3	6	3	6
2	4	7	4	7
2	4	8	4	8

The *3 x 3* test is a scaled down version for illustration. For the real test, we have tested the service engine with objects at various levels of complexity, including:

- 1 x 5
- 5 x 1, and
- 5 x 5

The results all come out as expected and are saved in GitHub [9] for public access. Therefore, we may conclude that the reference implementation of service engine is capable of handling complex objects up to the 5 x 5 level, and that the concept of DAS is feasible for handling complex objects.

6 Advantages

Table 5 compares DAS with Spring JDBC and JPA, which represent the two mainstream approaches at the moment: raw SQL and ORM. Compared to JDBC and JPA, DAS offers some unique advantages:

- DAS is scalable, meaning that as the query and, especially, the object gets complex, the level of effort required do not increase dramatically.
- DAS is platform independent. It works not only with Java but also with other applications.
- DAS requires only SQL and JSON skills and thus presents an opportunity to delegate the data access development to a SQL developer, just like the presentation layer is delegated to front-end developers.
- DAS development is highly efficient. The SQL is developed in a database-centric environment; the data mapping is largely automated; and the developer needs not to concern about database session and transaction, etc.
- DAS is safe from SQL injection. The SQLs are fully parameterized and, by default, are never altered through string appending or substitution.

Architecturally, DAS separates the data access layer from the application and makes data access a backing service to the application. To serverless or microservices applications, DAS may be deployed to provide not only data access but also connection polling services.

Table 5. Comparison of DAS with JPA and Spring JDBC.

	Spring JDBC	JPA	DAS
Architecture			
Application architecture	3-tier	3-tier	4-tier
object-data model coupling	loose	tight	loose
platform dependency	Java	Java	None
Scalability			
with SQL complexity	Fully scalable	Semi-scalable	Fully scalable
with object complexity	Not scalable	Not scalable	Highly scalable
Development			
paradigm/environment	Java	Java	SQL
skill requirements	Java, SQL	Java, QL, SQL	SQL, JSON
coding style	Imperative	Imperative	Declarative
mapping automation	By convention	By convention	Fussy matching
Security			
SQL injection	Up to developer	Generally safe	Safe

7 Summary

A small set of standardized data access services, namely query, command and repository services are presented as a preferable solution for the hard relational data access problem. Different from all traditional approaches, these services are specified declaratively with SQL and JSON, and are capable of handling highly complex object structures, as the hard object-relational transformation problem is abstracted away from developer and delegated to service engine. A reference service engine, along with a service builder tool, is implemented and tested with a slew of example query, command and repository services. A complexity scale is developed for measuring the complexity of object structure, as well as the capability of RDA solutions. The reference service engine is capable of handling objects at level *5x5* on above complexity scale, potentially covering all realistic application needs.

References

1. Fowler, M.: ORM Hate, https:// www.martinfowler.com/bliki/OrmHate.html, last accessed 2023/11/09.
2. Neward, T.: The Vietnam of Computer Science, https://www.semanticscholar.org/paper/The-Vietnam-of-Computer-Science-Neward/331e490c55ee72d6011bbceb323c03f0572a5235 (2006), last accessed 2023/11/09
3. DB-Engines DB-Engines Ranking, https://db-engines.com/en/ranking (November 2023), last accessed 2023/11/09

4. Bauer, C., King G.: Java Persistence with Hibernate. Manning Publications Co., 209 Bruce Park Ave. Greenwich, CT 06830 (2007).
5. Ireland, J.: Object-relational impedance mismatch: a framework based approach, PhD thesis The Open University (2011).
6. Oracle Data Access Object, https://www.oracle.com/java/technologies/data-access-object.html, last accessed 2023/11/09
7. GitHub Service Builder, https://github.com/bklogic/ServiceBuilder, last accessed 2023/11/09
8. GitHub Data Access Service Example, https://github.com/bklogic/data-access-service-example, last accessed 2023/11/09
9. GitHub Complexity Test, https://github.com/bklogic/complexity-test, last accessed 2023/11/13