A formal pattern of information system design

A Y Unger
Institute of Information Technologies, MIREA – Russian Technological University,
Vernadskogo ave, 78, Moscow, 119454, Russian Federation

E-mail: unger@mirea.ru

Abstract. A new design pattern intended for distributed cloud-based information systems is proposed. Pattern is based on the traditional client-server architecture. The server side is divided into three principal components: data storage, application server and cache server. Each component can be used to deploy parts of several independent information systems, thus realizing shared-resource approach. A strategy of separation of competencies between the client and the server is proposed. The strategy assumes that the client side is responsible for application logic and the server side is responsible for data storage consistency and data access control. Data protection is ensured by means of two particular approaches: at the entity level and at the transaction level. The application programming interface to access data is presented at the level of identified transaction descriptors.

1. Introduction

Client-server architecture plays a crucial role in design of scalable information systems [1]. Backend side software is installed on some virtual or physical server. The aggregation of such coupled servers composes an information cluster that is the core of the whole system. The key features of the cluster, such as scalability and fault tolerance, allow to serve a continuous stream of client requests.

The main goals of any cluster as a core of information system are data processing and data storage. Essentially, the cluster’s primary part is the distributed database. Requests to the database are initiated by applications that are also installed within the cluster. Database server along with the application server and the guest operation system occupy almost all the computing potential of the cluster. To ensure fault tolerance the nodes for load balancing and system monitoring are required as well [2].

Usually the cluster is deployed in some data center along with clusters of other information systems. The problem of shared-resource separation is successfully solved by means of some kind of virtualization that is not the subject of this article. The major cloud infrastructure providers make it possible to deploy the whole information system or its parts almost instantly and to scale it accordingly to the growth of client requests number.

In particular, the trend of the last years is the migration of information systems in the cloud in whole or in part [3]. At the same time, it should be noted that almost all system tasks are implemented on server side. Such a system design lead to data storage consistency and durability as well as tolerance to hardware or software errors and balance of load distribution. The client only needs to send a request to the server and to display the result (or, in case of error, to display some error message).

Such a thin client, nowadays, appears as a multi-core terminal, capable to take on some of the information processing and decision-making burden. So, if we could resolve the problem of accurate
separation of competencies between client and server, we could significantly unburden the server side and, as a result, simplify the design of the whole information system.

This article proposes a new pattern for information system design that unifies some specific aspects of the client-server architecture.

2. Principal concepts
Modern capabilities of software and hardware virtualization [4] make it possible to consider the cloud as a computational environment having several processor cores, some memory and some capacity for permanent data storage. One of the main goals of software engineering in appliance to information system design is to minimize the computational resources per client.

Let’s consider the principal parts of almost all information systems (figure 1).

![Figure 1. Principal design model of information system.](image)

Here we can see three main components, which together form the core of information system:

1) data storage;
2) data access control;
3) data cache.

Data storage can be implemented as a relational or non-relational (NoSQL) database. The non-relational databases provide better compatibility with object-oriented programming model. Furthermore, NoSQL databases are simpler to scale [5].

Relational databases provide more capabilities for data manipulation and generally perform better than their non-relational counterparts [6]. Detailed comparison of advantages and disadvantages of these two technologies is out of the scope of the present article. It is important that for the pattern considered the needed functionality can be provided by both relational and non-relational database management systems. We will use the relational database (RDBMS) in the examples further.

Before we talk about the design principles of data storage, let’s introduce a unit of storage. The most popular object-oriented (OOP) methodology [7] for application design assumes the model of information system as a collection of objects related to each other by a certain set of associations. In the world of relational databases, the unique and the most natural analog of an OOP class is a table, where
the rows map to the instances of that class. Wherein the only way to link objects together is by using foreign keys. By means of a foreign key it is possible to realize the parent-child relation. So, it can be up to four types of relations:

1) “one-to-one”;
2) “one-to-many”;
3) “many-to-one”;
4) “many-to-many”.

The first step towards optimization the system design is to exclude the redundant relations. Those relations are “one-to-one” and “many-to-many”. As can be seen, these two relations are symmetric, but the table can hold the foreign key on one side only. So, we have to choose which one of the two sides of the relation is an owning side (holding the foreign key) and which one is an inverse side. We can exclude the relation “one-to-one”, because from a database point of view this relation is absolutely identical to the relation “many-to-one”. The database schema for these two relations is following:

```
CREATE TABLE parent_table (id INT NOT NULL, PRIMARY KEY (id));
CREATE TABLE child_table (id INT NOT NULL, parent_id INT, PRIMARY KEY (id), FOREIGN KEY (parent_id) REFERENCES parent_table (id));
```

The relation “many-to-many” requires the auxiliary table, so can be replaced by two relations “many-to-one” and “one-to-many”. Here we consider the auxiliary table as an auxiliary class.

For the purpose of linking these two concepts together – table and class – let’s consider the entity as a unit of data storage. The concept of entity can be found in almost all object-relational mappers (ORM). In terms of ORM [8] an entity is described by means of certain OOP class. The instance of such an entity is saved in and retrieved from the database by means of certain adapter (mapper). ORM are widely used in practice of information systems design, however they do not use the capabilities of RDBMS very effectively, leading to poor performance impact.

Our main goal is to minimize system resources used per client. First, let’s reject complex JOIN requests at all. Obviously, this will increase the number of requests to database server. For example, in place of one request like this

```
SELECT a.*, b.* FROM parent_table AS a LEFT JOIN child_table AS b ON b.parent_id = a.id
WHERE <some_condition> ORDER BY a.id;
```

we should use two requests like these

```
SELECT * FROM parent_table WHERE <some_condition>;
SELECT * FROM child_table WHERE parent_id IN (<parent_id_set>) [WHERE
<some_condition>];
```

As can be seen, both variants select the parent entity along with the array of child entities. However, the first variant suggests that the server knows about associations between entities. Those associations are important when defining the database schema, but they are redundant at runtime. So, the next step in optimizing the information system is to delegate the logic of working with entities to the client side. The database server only acts as an entity store.

We can now postulate the following statement about fetching entities from a relational database: a single query can return a single object or a collection of objects of the same entity, selected according to certain criteria. Here we are not talking about unloading the system as a whole, but about the distribution of powers between the client and the server. Tracing entities and associations between entities is not a backend task, but a frontend task.

There are two important notes here. First, by rejecting complex JOIN queries, we make it easier to parallelize the data storage nodes. Indeed, parent entity and child entity can be stored on different physical nodes, so providing parallel and independent access. Second, by selecting only individual objects or collections of objects belonging to the same entity, we greatly simplify the caching component
[9]. Indeed, without loss of generality let’s collection be composed of a number of attributes – data fields and associations. When a collection is fetched by certain criteria, it is stated that all objects from this collection have the same values of all the attributes mentioned in the WHERE clause of the SELECT statement. Thus, by concatenating the entity name and field-value pairs from WHERE clause, we get the collection’s signature – a tag like this

\[
tag = MD5(EntityName.field_1=value_1.field_2=value_2...field_n=value_n)\]

Here dot is operator of concatenation.

A collection is stored in cache by its tag. If an object has changed, a check is performed to find any cached collections that have the same tag as that object's tag. All found collections are removed from the cache.

3. Security issues
The shift in the logic of working with entities to the client side raises problems with data protection and data access control [10]. The target problem is that we cannot trust the information received from the client. The target solution is to limit the client's authority as much as possible. We can do this by two ways:

1) At the level of entities. This way the access to entities is controlled by means of three-level hash table (access control list or ACL), where each entry looks like this

\[
ACL[EntityName][EntityAction][EntityFingerprint] = GroupList
\]

Here, EntityName – an entity’s unique identifier, EntityAction – an action allowed for this entity, EntityFingerprint – an entity’s signature, GroupList – a list of user groups with access granted. First of all, any user can be a member of several groups, just as group can contain several users. There is an example of “many-to-many” relation. But this relation is forbidden. So we have to inject a supplementary entity – Role, which is linked to the User entity and the Group entity through “many-to-one” relation.

EntityAction is limited to the set of allowed data manipulation language (DML) operators. The set is composed of: create, retrieve, update and delete (CRUD). These actions form the complete lifecycle of any object in the database.

EntityFingerprint plays the same role for data access control as a tag for caching. It is nothing more than a unique string that uniquely identifies the belonging of an object to a given subset of entities (1).

2) At the level of transactions. Above we have claimed entity as a unit of data storage and data processing. Transaction is a natural unit of data exchange between a client and a server. A transaction is a sequence of database requests that ensure data integrity and consistency. This definition of transaction almost completely repeats the definition of transaction from the point of view of the RDBMS. So our design pattern transaction is based on database management system transaction.

Data access control at the level of transaction assumes that the information system provides an application programming interface (API). This API is based on a set of descriptors, each having a unique identifier. Descriptor is a set of requests to the database. Descriptor can be written in some language, that accepts primitive control flow statements. For performance reasons descriptor should be compiled to some underlying programming language – language of the information system being designed. An example of a descriptor written in pseudo code is

\[
\text{let A = select(<entity>, <criteria>, <sortBy>, <offset>, <limit>);}
\]

\[
\text{for a in A}
\]

\[
\text{update(a, <attributes>);}
\]

\[
\text{end;}
\]

\[
\text{insert(<entity>, <values>);}
\]

\[
\text{delete(<object>);}
\]

...
Here all tokens in angle brackets are the *context* of the descriptor. Context is an object that the client sends to the server. As can be seen, context contains not only client’s data, but also data fetched from the database during transaction execution. The code above is compiled into a set of SQL statements that start with `START TRANSACTION` and end with `COMMIT` or `ROLLBACK`, respectively.

The context and the descriptor identifier is all that the client need to send to the server to fulfill the request.

Comparing the two approaches, one can notice that the first approach is simpler, because there is no need to develop an auxiliary language for transaction descriptors. At the same time, the first approach suffers from redundant database queries. Indeed, according to (2), in order to ensure that some object can be accessed by some user, it is necessary to first fetch that object, find the corresponding group list, and then allow or deny the action, even if that action is an update or delete.

The second approach is much more powerful. In this case, access can be controlled not only at the level of the entire transaction, but at the level of individual context properties. This is possible thanks to the control flow expressions available in the descriptor language.

Conceptually, the two described approaches differ in that in the first case we send to the server the entire transaction scheme, i.e. data and actions on them, and in the second case – only the data itself. The transaction scheme is stored on the server side.

4. Conclusion
The proposed information system design pattern is about the separation of competencies between the client and the server. The pattern has been implemented in the cloud-based cluster, consisting of four computational nodes: load balancing node, data storage node, application node and caching node. The load balancing node is based on Nginx – web and reverse proxy server. The caching node is based on Redis – in-memory data structure store. The data storage node is based on MySQL – relational database management system. All mentioned components are open-source. An application node plays the role of a controller that orchestrates the operation of the entire information system.

The source code of the project is available in the public repository on GitHub https://github.com/AntonUnger/apitech-iii.

References
[1] Hatton L 2007 Empirical test observations in client-server systems *Computer* **40**(5) 24-9
[2] Lenhardt J, Chen K, Schiffmann W 2015 Energy-efficient web server load balancing *IEEE Systems Journal* **11**(2) 878-88
[3] Linthicum D S 2016 A guide to cloud-enabling your software *IEEE Cloud Computing* **3**(2) 20-3
[4] Silva L M, Alonso J, Torres J 2009 Using virtualization to improve software rejuvenation *IEEE Transactions on Computers* **58**(11) 1525-38
[5] Mitko R 2017 A Comparison between Characteristics of NoSQL Databases and Traditional Databases *Computer Science and Information Technology* **5** 149-53
[6] Han W, Whang K, Moon Y 2005 A formal framework for prefetching based on the type-level access pattern in object-relational DBMSs *IEEE Transactions on Knowledge and Data Engineering* **17**(10) 1436-48
[7] Bolognesi T 2000 Toward constraint-object-oriented development *IEEE Transactions on Software Engineering* **26**(7) 594-616
[8] Chillion A H, Ruiz D S, Molina J G, Morales S F 2019 A model-driven approach to generate schemas for object-document mappers *IEEE Access* **7** 59126-42
[9] Ermakov N V, Molodyakov S A 2020 A caching model for a quick file access system *Journal of Physics: Conference Series* **1864**(1) 012095
[10] Lesko S A 2020 Models and scenarios of implementation of threats for internet resources *Russian Technological Journal* **8**(6) 9-33