

Database Access and Patterns in Erlang/OTP

LAURA M. CASTRO, VÍCTOR M. GULÍAS, CARLOS ABALDE, JAVIER PARÍS

University of A Coruña
Department of Computer Science
Campus de Elviña s/n, A Coruña
SPAIN
{lcastro,gulias,cabalde,javierparis}@udc.es

Abstract: Databases are an essential component in most software applications, whether they be critical or not, regardless of their scope and scale. Erlang/OTP is a very suitable framework, based on the functional programming paradigm, to develop robust highly available distributed systems. In its way to a place in the industry world, Erlang/OTP faces the challenge of getting on with traditional Database Management Systems (DBMS). To do so, there is no better way than using abstraction and high-level software engineering tools, i.e. design patterns.

Key-Words: Erlang/OTP, database access, software engineering, design patterns

1 Introduction

From its early days, one of the main purposes of computing has been data management. Any software application needs to pay special attention to management, storage and retrieval of the information needed to carry out the task it is designed for.

Database Management Systems (DBMS) are pieces of software which provide structured storage to data, and also a query language to inspect and get the information back. There are multiple well-known DBMSs, both in the proprietary and the free software worlds, such as Oracle [23], DB2 [14], Microsoft SQL Server [20], PostgreSQL [24], MySQL [22], Sybase [28]... All of them implement the relational database model [19] and have many years of market experience on their backs, and enough properties of stability and reliability to be trusted by developers and software companies.

Erlang/OTP is a functional language and a set of libraries that were created as a platform to develop robust applications meant to run over a net of computers. It was originated inside Ericsson Telecommunications and its initial aim was to be a tool to program telephone switches. But it turned out to be a programming environment that helped to speed up the development and to reduce the maintenance effort while generating high reliable robust pieces of software. This was the key for it to step out from the telephony world and start being used for other purposes. Nowadays, Erlang/OTP has proved that can be a perfectly valid environment to develop almost any kind of software application, especially when robustness, reliability, high-availability, maintenance

ease, and transparent distributability are essential requisites [18, 15, 30, 29, 7].

As it became more and more popular, the set of libraries and utilities included in the Erlang/OTP distribution increased as well. Now, it even provides its own DBMS, called *Mnesia* [21]. Amongst the most innovative Mnesia properties, which is actually a distributed DBMS with a hybrid relational/objectual data model, we find a very high level of fault-tolerance, and dynamic distribution and reconfiguration. But still, the influence and importance of traditional DBMS is too hard to overcome in many cases. Even more if data is already saved in such storage, i.e. when migrating an application, or building a new one that will work with already-modeled, pre-existing data.

For these reasons, it is important to know how to make Erlang/OTP and traditional DBMSs work together. Not only with regard to the technological solutions available, but also bearing in mind storage and retrieval strategies. For the former point, we have explored and compared two of the most important approaches. For the latter task, well-known database access patterns are of the most valuable help, and we will show how to implement those patterns, mainly identified with the object-oriented world, Erlang-style.

The structure of this paper is as follows: first, we will describe the different possibilities when talking to a DBMS, and which of those are available from Erlang/OTP. In section 3 we will describe and explain the basic database access patterns. Section 4 will show how to implement them in the Erlang language. Finally, we will present our learned lessons and conclusions.

2 External data sources

A database is a structured collection of pieces of information, stored in a computer system and accessible to be questioned, either by a human or a computer program, through a special query language.

The nature of the pieces of information stored by a database, which can be seen as its structural description, is called *database schema* [8]. Depending on how the database schema is organised or modelled, different *database models* are possible. Some of the most relevant database models are:

- *Hierarchical model*, where the database schema is a tree-like structure with one root and a number of branches or subdivisions.
- *Network model*, where data relationships are represented in a many-to-many way, allowing arbitrary graphs and not only trees (like in the hierarchical model), thus resembling a network.
- *Relational model*, where information and relationships amongst information items follow the set theory [13].
- *Objectual model*, where schema components are objects, comparable to those in Object-Oriented Programming [4].

From the above, the *relational model* was the first formal database model [27]. It has a strong mathematical background which includes set theory and two-value predicate logic. In the relational model, stored data is operated upon by means of relational algebra [6]. Thanks to this formalism and its rigorousness, relational databases became popular in the eighties and even though other database models were formalised afterwards, and many new formal models have appeared ever since, none has been able to take its place as the most commonly used database model. For this reason, we will focus on this type of databases.

2.1 Access types

In order to inspect the collection of data stored in a relational database (RDB), a person (or a computer program) must use, as previously said, a particular query language. The query language for relational databases is an ANSI [1] and ISO [16] standard called Structured Query Language (SQL [17]). To proceed, a human user would type some SQL-queries in an interactive environment. After being introduced, the query would be processed by the DBMS, which would retrieve the matching data, and the results would appear on the screen. If we need an external software application to query the database in order to obtain that same data (to later on be processed or dealt with), the interaction mechanism needs to be slightly different.

There are several ways in which a computer program can submit SQL-queries to a RDB:

Using a standard API, such as Open Database Connectivity (ODBC [26]). An API (stands for Application Programming Interface) is a source code interface provided to support standard and independent requests for services. The use of standard APIs makes it possible for two independent parties (computer programs) to interact in a clean, pluggable and reliable way.

Using a database-native API Even though using a database-native API is generally more efficient than using just a generic standard API, it is also a solution that binds the program to a specific database and a specific implementation.

Using sockets, to directly connect to the database. This implies taking care of all low-level details of application-database communication, which usually is too much work to even take under consideration, unless there are very strong speed and efficiency requirements which can not be fulfilled by using an already-existing API.

2.2 Erlang and ODBC

As previously said, ODBC (Open DataBase Connectivity) is a standard API which allows us to query a relational database. Thus, it is a way of communicating applications and database servers using a standard set of functions to open a connection with the database, send a query, and get the results back.

Each ODBC-compatible DBMS provides an implementation of the ODBC API, which internally translates ODBC calls into its own native dialect. To be able to connect to a DBMS via ODBC, an ODBC driver is also needed, which is the piece of software the client application will link to, thus remaining independent from the DBMS. Actually, the fact that the client only performs standard function calls, makes it independent from the driver as well.

Erlang/OTP platform provides support for ODBC, hence allowing Erlang programmers to build applications that can communicate with a relational database via ODBC. And since the ODBC-support module is part of the standard set of libraries, OTP, such communication takes place in a transparent, Erlang-like style (see source fragment 1).

2.3 Erlang native driver

As happens on many other programming environments and development tools, there are also some native drivers available for Erlang/OTP. These Erlang drivers are database-specific, since what they do is

Listing 1: Erlang/ODBC code sample

```
Eshell V5.5.5 (abort with ^G)
% start ODBC Erlang application
1> application:start(odbc).
ok
% obtain connection reference to database
2> ok, Ref = odbc:connect("DSN=database-name;"
                        "UID=username;"
                        "PWD=apassword", []).

ok,<0.39.0>
% send query to database using a connection
3> odbc:sql_query(Ref, "SELECT * FROM atable").
selected,["column_a", "column_b", "column_c"],
        [ "Patricia", "GMT+02:00", 3 ,
          "Javier", "GMT+02:00", 3 ,
          "John", "GMT+01:00", 1 ]
% close database connection
4> odbc:disconnect(Ref).
ok
% close ODBC Erlang application
5> application:stop(odbc).
ok
```

Listing 2: Native Erlang driver code sample

```
Eshell V5.5.5 (abort with ^G)
1> application:start(sql).
ok
2> application:start(psql).
ok
3> sql:q("SELECT * FROM atable").
[ "Patricia", "GMT+02:00", 3 ,
  "Javier", "GMT+02:00", 3 ,
  "John", "GMT+01:00", 1 ]
4> application:stop(psql).
ok
5> application:stop(sql).
ok
```

to directly connect to a given DBMS (PostgreSQL, MySQL...) from an Erlang environment. Nevertheless, the actual communication is again transparent for Erlang client applications (see source code 2).

3 Database access patterns

From an application conception to its actual development, some initial work needs to be done. Apart from requirements elicitation and final functionalities determination, the broad analysis task, as far as persistence aspects are concerned, involves some (or all) of the following steps:

- *Identification of the persistent business objects.* Our system's persistent business objects will be those information or data elements which life can extend during the application's running time or even beyond, from one execution to the next. Once those relevant elements are listed, their persistent parts (usually referred to as "state")

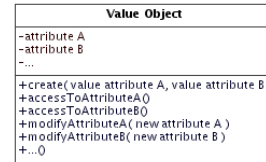


Figure 1: Value Object pattern structure

need to be identified as well (for the relational database to take care of them).

- *Identification of the persistence operations.* These will usually include creation, deletion, update, search and retrieval of persistent elements (or their states).
- *Control flow definition.* The control flow is the sequence of operations that will be needed to be executed in order to carry out an application's use case, i.e. a piece of its functionality, a service. Eventually, persistence operations will be involved in those use-case operations.
- *Functionality or service API implementation.* Once the control flow is clear and stated, related use-cases (services) should be grouped together, and one or more interaction APIs should be provided, to be used by other parts of the system (or even third-party clients).

These four stages help us approach and organise the process of building our application's storage and retrieval module/subsystem, where the actual implementation is the last task, for which other formal tools are available to help developers to perform it in an efficient and error-free manner. One of these conceptual tools are *software design patterns*.

A *software pattern* is a repeatable software solution to a particular kind of problem, which has proved to be both efficient and simple [10]. Applying software design patterns during a system's design and development process helps to reduce the development effort and prevents the appearance of common or known-errors.

In the next subsections some software patterns related to database access will be exposed.

3.1 Value Object pattern

The *Value Object* (VO) pattern represents the abstraction of the state or value of a persistent object in the domain [10]. The structure of the pattern is shown in figure 1, using UML modelling language[3].

The VO pattern abstracts the persistent properties of a business concept as elements which will be saved on a permanent storage (RDB). Access to this data

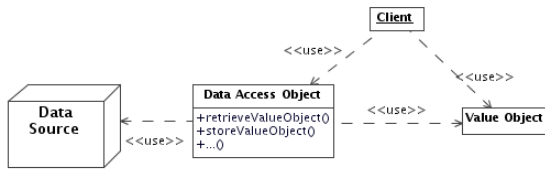


Figure 2: Data Access Object pattern structure

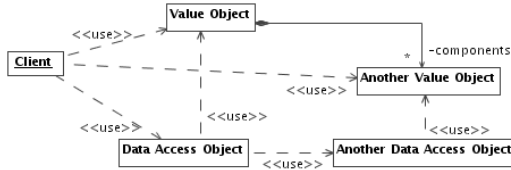


Figure 3: Data Access Object for a complex VO

is restricted and conducted through function calls, which can be read-only (“access” functions/methods) or read-write (“update” functions/methods).

3.2 Data Access Object pattern

The *Data Access Object* (DAO) pattern, which structure can be seen in figure 2, hides the interaction with the persistent storage (database) from the rest of the system or application, setting apart business logic from persistence logic [10].

A DAO is then an element in the system that will provide an internal API to recover/store VO data, abstracting the external storage source (typically, and in our examples, a relational database, but it could be any other storage media). It is the software piece of the system that will, for instance, invoke a standard API to access the database using ODBC, or else that will deal directly with it by means of a native driver or custom database access implementation.

The DAO pattern mixes functionalities commonly identified with two different software patterns:

- *Adaptor pattern* [10], since it provides an interface for persistence.
- *Factory pattern* [10], since it creates (recovers and also stores) VOs, making it datasource-independent for the rest of the system.

A DAO can even make use of other DAOs, if dealing with a complex VO composed by more than one basic VO, like in figure 3.

3.3 Facade pattern

The *Facade* pattern provides a single interface to a set of components in a system. By doing so, clients do not need to know all the details and complexities in a

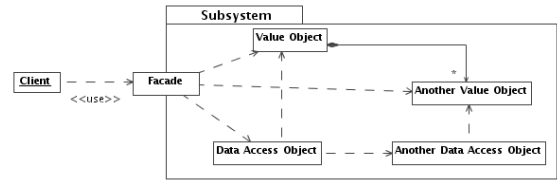


Figure 4: Facade pattern structure (and interaction with DAO and VO)

system: they just interact with the facade, which has convenient methods for fulfilling their needs.

The use of facades as interfaces for the functionality of a system (or part of it) decouples it from its clients, allowing modifications to be transparent (since only facade internals will be affected). Apart from offering a neat access point to a set of logically related use cases, a facade can represent work flow as well. This implies more flexibility (and simplicity) in developing the system, because different facades can be created (as many as needs), each of them presenting a single specific-purposed well-designed API.

4 Erlang implementation

Erlang is a distributed, concurrent functional programming language with no constructs inducing side effects (with the exception of thread communications –*processes*–). Values in Erlang (i.e., non-reducible expressions) range from numbers and *atoms* (symbolic lower-case constants) to complex data structures (lists, tuples –similar to C structures–) and functional values. In the absence of a static type system, lists and tuples can hold any valid value. Records are also provided as useful syntactic sugar for accessing tuples by name instead of by position.

An Erlang function is defined by a set of equations, each stating a different set of constraints based primarily on the structure of the arguments (*pattern-matching*). Iterative control flow is carried out by using recursion and expressions are evaluated eagerly. Functions are grouped into modules, and a subset of those functions can be exported, declaring both function name and arity, to be used in other modules.

So Erlang modules are the implementation unit elements in this language (like classes in Java), and thus, each value object in the system, representing application business objects, corresponds with a module (see code fragment 3) exporting suitable access and update functions.

As we previously outlined in section 3.2, for each class representing a business object, a DAO module is implemented. Source code section 4 shows an abstract and generic view of an Erlang DAO.

5 Conclusions

In this paper we have analysed, studied and presented the situation of accessing a relational database from an Erlang environment. Relational databases are the most commonly used kind of databases nowadays, and even though there are other, newer and more complex, database models in the market, relational DBMS are not likely to lose their predominance, at least in the next years to come.

As far as Erlang is concerned, we consider it as a very interesting development platform, most of all if certain requirements are present (robustness, fault tolerance, high availability, reliability, distributability, concurrency, ease of maintenance. . .). Thus, we have explored the different solutions available when combining Erlang/OTP development environment with relational DBMS storage.

The design methodology that has been explained here can be applied regardless of the implementation stage details. Nevertheless, only careful analysis of application workload, business use cases, and nature of both data operations and data nature, together with technology solution maturity, would provide enough judgement references, for each case, to choose between database access alternatives.

Acknowledgements: This work has been partially supported by MEyC TIN2005-08986 and XUGA PGDIT06PXIC105164PN.

References:

- [1] ANSI. American National Standards Institute. <http://www.ansi.org>.
- [2] ARMISTICE. Advanced Risk Management Information System: Tracking Insurances, Claims and Exposures. <http://www.madsgroup.org/armistice/>, 2002.
- [3] Booch, Grady, Jacobson, Ivar, and Rumbaugh, James. *The Unified Modeling Language. UML*. Addison Wesley, 1998.
- [4] Budd, Timothy. *An Introduction to Object-Oriented Programming*. Addison Wesley, 3rd edition, 2001.
- [5] Cabrero, David, Abalde, Carlos, Varela, Carlos, and Castro, Laura M. ARMISTICE: An experience developing management software with Erlang. In *Principles, Logics, and Implementations of High-Level Programming Languages*, Upsala, Sweden, aug 2003.
- [6] Date, C.J. *An Introduction to Database Systems*. Addison Wesley, 8 edition, 2003.
- [7] EJabberd. Jabber/XMPP instant messaging server. <http://www.ejabberd.im/>.
- [8] Elmasri, Ramez and Navathe, Shamkant B. *Fundamentals of Database Systems*. Addison Wesley, 5 edition, 2006.
- [9] Erlang-Consulting. PostgreSQL erlang native driver. <http://www.erlang-consulting.com/aboutus/opensource.html>.
- [10] Gamma, Eric, Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, 1996.
- [11] Gulas, Vctor M., Abalde, Carlos, Castro, Laura M., and Varela, Carlos. A new risk management approach deployed over a client/server distributed functional architecture. In *18th International Conference on Systems Engineering*, pages 370–375, University of Nevada, Las Vegas (USA), aug 2005. IEEE Computer Society. <http://www.icseng.info>.
- [12] Gulas, Vctor M., Abalde, Carlos, Castro, Laura M., and Varela, Carlos. Formalisation of a functional risk management system. In *8th International Conference on Enterprise Information Systems*, pages 516–519, Paphos (Cyprus), may 2006. INSTICC Press. <http://www.iceis.org>.
- [13] Hrbacek, Karel and Jech, Thomas. *Introduction to Set Theory*. CRC, 3rd edition, 1999.
- [14] IBM. DB2. <http://www.ibm.com/db2>.
- [15] Igalia. SERVAL: Internet software VLAN switch developed in Erlang. <http://serval.igalia.com>.
- [16] ISO. International Standards Organization. <http://www.iso.org>.
- [17] Kline, Kevin, Kline, Daniel, and Hunt, Brand. *SQL In A Nutshell*. Media. O Reilly, 2 edition, 2004.
- [18] LambdaStream. Video on Demand Kernel Architecture (VoDKA). <http://www.lambdastream.com/lambda-products/VoDKA?l=EN>.
- [19] David Maier. *Theory of Relational Databases*. Computer Science Pr., 1983.
- [20] Microsoft. SQL server. <http://www.microsoft.com/sql>.
- [21] Mnesia. Distributed functional object-relational Erlang/OTP database. <http://www.erlang.org/doc/apps/mnesia/index.html>.
- [22] MySQL. MySQL. <http://www.mysql.com>.
- [23] Oracle. Oracle Database Management System. <http://www.oracle.com>.
- [24] PostgreSQL. PostgreSQL. <http://www.postgresql.org>.
- [25] Rmond, Mickal. MySQL native erlang driver. <http://support.process-one.net/doc/display/CONTRIBS/Yxa>.
- [26] Sanders, Roger E. *ODBC 3.5 Developer s Guide*. McGraw-Hill, 1998.
- [27] Silberschatz, Abraham, Korth, Henry F., and Sudarshan, S. *Database Systems Concepts*. McGraw-Hill, 5 edition, 2005.
- [28] Sybase. Adaptive server enterprise. <http://www.sybase.com>.
- [29] Tsung. Multi-protocol distributed load testing tool. <http://tsung.erlang-projects.org/>.
- [30] Yaws. High performance webserver. <http://yaws.hyber.org/>.