A Simplified Database Pattern for the Microservice Architecture

Antonio Messina, Riccardo Rizzo, Pietro Storniolo, Alfonso Urso
ICAR - CNR
Palermo, Italy
Email: {messina, ricrizzo, storniolo, urso}@pa.icar.cnr.it

***Abstract*—Microservice architectures are used as alternative to monolithic applications because they are simpler to scale and more flexible. Microservices require a careful design because each service component should be simple and easy to develop. In this paper, a new microservice pattern is proposed: a database that can be considered a microservice by itself. We named this new pattern as *The Database is the Service*. The proposed simplified database pattern has been tested by adding ebXML registry capabilities to a noSQL database.**

Keywords–microservices, scalable applications, continuous delivery, microservices patterns, noSQL, database

I. INTRODUCTION

The microservice architectural style [1] is a recent approach to build applications as suite of services, independently deployable, implementable in different programming languages, scalable and manageable by different teams.

Microservices architectures are opposed to monolithic applications. Monolithic applications are simpler to build and to deploy, but their structure forces the developers to work in team. Working in team the developers tends to deploy large applications that are difficult to understand and to modify.

Moreover, if a required service is implemented by a single application, the transactions volume can be increased only by running multiple copies of the same application, that has to access to the same database.

On the other side, developing a system based on microservices requires a special attention because it is a distributed system. In this case, even the team of developers can be distributed, it requires a special effort in coordination and communication.

In microservices based systems, one of the biggest challenge is the partition into separated services, each of them should be simple enough to have a small set of responsibilities. Data management require a special attention, because it can be one of the bottleneck of the system. So that it is convenient that only one or few microservices access the data, but this can affect the responsiveness of the whole system.

When a careful project solves all these issues, microservices became an effective architectural pattern, in fact studies have shown how this architectural pattern can give benefits when enterprise applications are deployed in cloud environments [2] and in containers [3], e.g., Docker [4]. Microservices are also considered the natural fit for the *Machine-to-Machine* (IoT) development [5].

The microservices pattern implies several important auxiliary patterns, such as, for example, those which concern how clients access the services in a microservices architecture, how clients requests are routed to an available service instance, or how each service use a database.

In the new microservice pattern proposed in this paper, a database, under certain circumstances and thanks to the integration of some business logic, can be considered a microservice by itself. It will be labeled as *The database is the service* pattern.

The remainder of the paper is organized as follows: Section 2 presents a brief overview about the *old* monolithic style and its drawbacks. The microservices architectures and the related pattern are described in Section 3. Section 4 presents the proposed pattern. In Section 5, we show a proof of concept of the pattern and its improved performances. Finally, conclusions are reported.

II. BACKGROUND

To better understand the microservice style it is useful to compare it to the *monolithic style*: a monolithic application built as a single unit. Modern enterprise applications are typically built in three main parts: a client-side user interface, a server-side application, and a relational database. The server-side application handles the requests, executes domain logic, retrieves and updates data from the relational database, and selects and populates the views to be sent to the client-side. This server-side application can be defined as *monolith*, a single logical executable.

Essentially, a monolith application is the one where all its functionalities are packaged together as a single unit or application. This unit could be a JAR, WAR, EAR, or some other archive format, for which is all integrated in a single unit. This style of application is well known, because this is how applications have been built so far, it is easy to conceptualize and all the code is in one place. The most of existing tools, application servers, frameworks, scripts are able to deal with such kind of applications. In particular, IDEs are typically designed to easily develop, deploy, debug, and profile a single application. Stepping through the code base is easy because the codebase is all together.

Finally, a monolith is easy to share, to test and to deploy. A single archive, with all the functionality, can be shared between teams and across different stages of deployment pipeline. Once the application is successfully deployed, all the services, or features, are up and available. This simplifies testing as

 ISBN: 978-1-61208-486-2

there are no additional dependencies to wait for in order to begin the test phase. Accessing or testing the application is simplified in either case. It is easy to deploy since, typically, a single archive needs to be copied to one directory. The deployment times could vary but the process is pretty straight forward. However, a monolithic application, no matter how modular, will eventually start to break down as the team grows, experienced developers leave and new ones join, application scope increases, new ways to access the applications are added, and so on. Moreover, it has a very limited agility, because every tiny change to the application means full redeployment of the archive. This means that developers will have to wait for the entire application to be deployed if they want to see the impact of quick change made in their workspace. Even if not intentional, but this may require tight coupling between different features of the application. This may not be possible all the time, especially if multiple developers are working on the application. This reduces agility of the team and the frequency by which new features can be delivered. If a single change to the application would require entire application to be redeployed, then this could become an obstacle to frequent deployments, and thus an important obstacle for continuous delivery.

Choice of technology for such applications are evaluated and decided before their development starts. Everybody in the team is required to use the same language, persistence stores, messaging system, and use similar tools to keep the team aligned. It is typically not possible to change technology stack mid stream without throwing away or rewriting significant part of existing application.

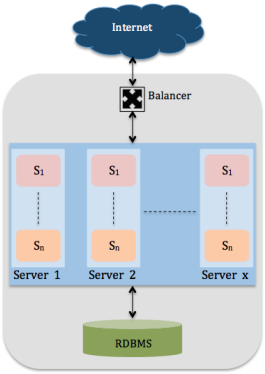

Figure 1. Services provided by horizontally scaled monolithic application

Monoliths can only scale in one dimension, i.e., they have to be entirely duplicated across a set of servers (see Figure 1). This way, each application instance will access all of the data. This makes caching less effective, increases memory consumption and i/o traffic.

Systems based on microservices present many advantages if compared to monolithic applications. Some of these advantages came from their distributed architecture and will be explained in the next section.

III. Microservices Architecture and Related Patterns

In the last years, several large Internet companies have used different mechanisms, strategies and technologies to address the limitations of the monolithic architecture: they can be referred as the *microservices architecture pattern*.

Microservices is a software architectural style that require functional decomposition of an application. A monolithic application is broken down into multiple smaller services, each deployed in its own archive, and then composed as a single application using standard lightweight communication, such as REST over HTTP (see Figure 2).

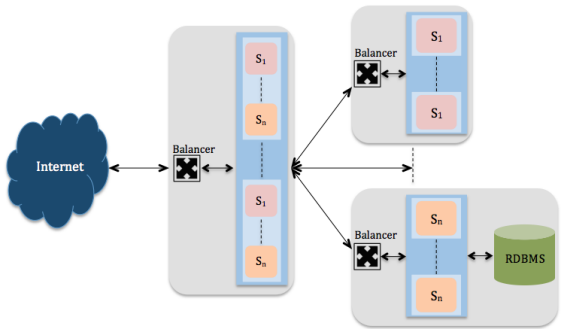

Figure 2. Tipical microservice-based application with lightweight frontend

The decomposition into a set of collaborating services is usually done applying the Y-axis scaling of the three dimension scalability model named the Scale Cube [6]:

- *X-axis scaling*: it is the simplest commonly used approach of scaling an application via horizontal duplication, namely running multiple cloned copies of an application behind a load balancer.
- *Y-axis scaling*: it represents an application's split by function, service or resource. Each service is responsible for one or more closely related functions. We can use a verb-based decomposition and define services that implement single use cases, or we can decompose the application by noun and create services responsible for all operations related to a particular entity.
- *Z-axis scaling*: it is commonly used to scale databases, because the data is partitioned across a set of servers. Each server runs an identical copy of the code and each service request is routed to the appropriate server according to a routing criteria.

Basically, the service design should be made by applying the *Single Responsibility Principle* [7], that defines a responsibility of a class as a reason to change, and states that a class should only have one reason to change.

There are several patterns [8] related to the microservices pattern. We mainly focus our attention on the following:

- The *API Gateway* pattern, that defines how clients access the services in a microservices architecture.
- The *Client-side Discovery* and *Server-side Discovery* patterns, used to route requests for a client to an available service instance in a microservices architecture.
- The *Service Registry* pattern, a critical component that tracks the instances and the locations of the services.
- The *Database per Service* pattern, that describes how each service has its own database.

 ISBN: 978-1-61208-486-2

A. The API Gateway Pattern

Microservices typically provide fine-grained APIs, which means that clients need to interact with multiple services. However, different clients need different data and network performance is different for different types of clients. Moreover, the number of service instances and their locations (host+port) changes dynamically and partitioning into services can change over time and should be hidden from clients.

An API gateway is the single entry point for all clients and handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services. Rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client. It might also implement security, e.g., verify that the client is authorized to perform the request. There is a couple of obvious drawbacks, at least:

- *Increased complexity*, due to another moving part that must be developed, deployed and managed.
- *Increased response time*, due to the additional network hop through the API gateway. However, for most applications the cost of an extra roundtrip is insignificant.

B. The Discovery Patterns

In a monolithic application, services invoke one another through language-level method or procedure calls. In a traditional distributed system deployment, services run at fixed, well known locations (hosts and ports) and so they can easily call each using HTTP/REST or some RPC mechanism. However, a modern microservice-based application typically runs in a virtualized or containerized environments where the number of instances of a service and their locations changes dynamically. Consequently, the service clients must be enabled to make requests to a dynamically changing set of transient service instances.

- *Client-side*: The clients obtain the location of a service instance by querying a *Service Registry*, which knows the locations of all service instances. This implies fewer moving parts and network hops compared to *Server-side Discovery*, but clients are coupled to the *Service Registry* and you need to implement a client-side service discovery logic for each programming language/framework used by the application (see Figure 3).

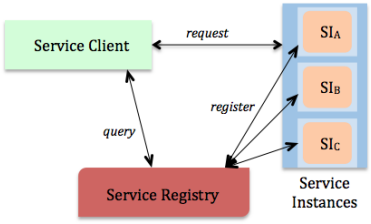

Figure 3. Client-side discovery pattern

- *Server-Side*: When making a request to a service, the client makes a request via a router (a.k.a. load balancer) that runs at a well known location. The router queries a service registry, which might be built into the router, and forwards the request to an available service instance. Compared to client-side discovery, the client code is simpler since it does not have to deal with discovery. Instead, a client simply makes a request to the router, but more network hops are required (see Figure 4).

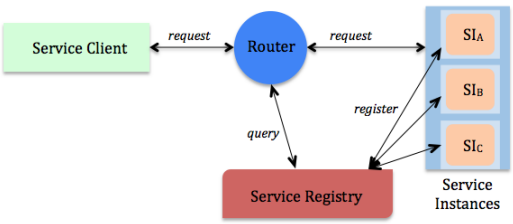

Figure 4. Server-side discovery pattern

C. The Service Registry Pattern

A service registry is a database of services, their instances and their locations. Service instances are registered with the service registry on startup and deregistered on shutdown. Client of the service and/or routers query the service registry to find the available instances of a service. Unless the service registry is built in to the infrastructure, it is yet another infrastructure component that must be setup, configured and managed. Moreover, the Service Registry is a critical system component. Although clients should cache data provided by the service registry, if the service registry fails that data will eventually become out of date. Consequently, the service registry must be highly available.

D. The Database per Service Pattern

According to this pattern, we should keep each microservice's persistent data private to that service and accessible only via its API. It means that the service's database is effectively part of the implementation of that service and it cannot be accessed directly by other services. There are a few different ways to keep a service's persistent data private:

- *Private-tables-per-service*: each service owns a set of tables that must only be accessed by that service.
- *Schema-per-service*: each service has a database schema that is private to that service
- *Database-server-per-service*: each service has its own database server. When the service has to be scaled, the database can be also scaled in a database cluster, no matter the service.

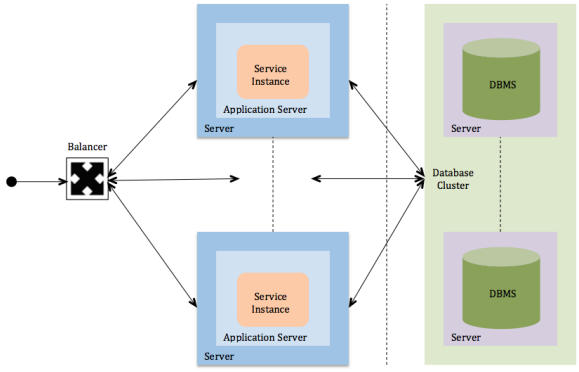

Figure 5. The Database per Service pattern applied to a scaled service using a database cluster

 ISBN: 978-1-61208-486-2

Figure 5 shows a typical architecture of a scaled service using its own database cluster.

IV. THE *Database is the Service* PATTERN

The granular nature of the microservice architectures may bring many benefits, but also comes with the cost of increased complexity.

Breaking a monolith into microservices simplifies each individual component, but the original complexity goes to surface when, at some point, someone has to put it all together [9]. Certainly, there is a sort of law of conservation of complexity in software and, if we break up big things into small pieces, then we push the complexity to their interactions [10].

Moreover, IT complexity in enterprise today continues to grow at a dizzying rate. Technology innovation, vendor heterogeneity, and business demands are major reasons why organizations are exposed to new risks, based on the gaps opened between the options and features of each IT element and product, and how they are implemented to support a well-defined policy and company strategy. The impact of such risks increases exponentially by failing to identify the handshakes and correlations of interrelated elements. Products, vendors, and IT layers must work together to prevent potential *black holes*: risks related to availability, resiliency and data loss.

It is not hard to understand how microservice architectures may amplify such risks, because their distributed nature. Moreover, in a monolithic application there was a method call acting as a subsystem boundary, in the microservice architecture we now introduce lots of remote procedure calls, REST APIs or messaging to glue components together across different processes and servers.

Once we have distributed a system, we have to consider a whole host of concerns that we didn't before. Network latency, fault tolerance, message serialisation, unreliable networks, asynchronicity, versioning, varying loads within our application tiers etc.

Starting from the *Database-Server per Service Pattern*, the addition of new behaviors and business logic at the database level may be a possible approach to reduce complexity, and thus the related risks, and also to gain improvements in terms of speed and scalability.

Problem: If each scalable service has its own database (cluster), as shown in Figure 5, is there any way to reduce the complexity of the architecture and the related risks, while also gaining more improvements in terms of speed and scalability?

Solution: Whenever the database has an open architecture and provides the necessary hooks to extend its capabilities, then it can embed the business logic that implements the desired service. The service is strictly coupled to the data, hence this pattern is even stronger than the *Database-Server per Service Pattern*, because the database itself acts as a business service. As shown in Figure 6, clients requests are routed via a load balancer, following the guidelines of the *Server-side Discovery Pattern*.

Some benefits of such approach are immediately clear at first sight:

a) the traditional service layer disappears, thanks to the whole removal of related hosts and application servers or containers;

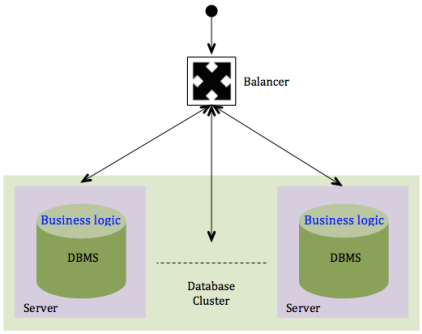

Figure 6. The Database is the Service pattern: DBMS with business logic

b) services deployed into the database have instant access to data, accessible at no cost (no third party libraries, no network issues, and so on);
c) less the involved components, less the complexity, the interactions and the potential risks.

If the database cluster layer is also available to clients, i.e., thanks to a specific library, we may achieve further simplification, because clients requests reach directly the service. Unlike *Client-side Discovery Pattern*, there's no need to implement a discovery logic into clients, there isn't any balancer, and the cluster layer supplies, at least, the same *Service Registry* capabilities. Figure 7 shows the way that super-simplified architecture looks.

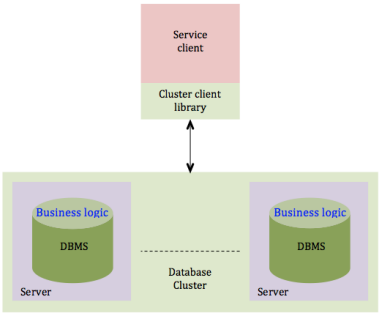

Figure 7. The Database is the Service pattern with client-side cluster support

Drawbacks are also obvious, first and foremost the dependency on the chosen database, because the service becomes integral to, and inseparable from, the database engine. Test and debug activities must also involve the database because of his primary role. For this reason, the database source code must be available, an open source product is the obvious choice.

V. PROOF OF CONCEPT

As a proof of concept for the proposed pattern, we have added ebXML registry capabilities to a noSQL database, starting from *EXPO* [19], a prototypal extension for OrientDB derived from *eRIC* [20], our previous SQL-based ebXML registry implemented as web service in Java.

An ebXML registry [15][16] is an open infrastructure based on XML that allows electronic exchange of business information in a consistent, secure and interoperable way.

The eXtensible Markup Language (ebXML) is a standard promoted by the Organization for the Advancement of Structured Information (OASIS) and was designed to create

a global electronic market place where enterprises of any size, anywhere, can find each other electronically and conduct business using exchange of XML messages according to standard business process sequences and mutually agreed trading partner protocol agreements.

Nowadays, ebXML concepts and specifications are reused by the Cross-Enterprise Document Sharing (XDS) architectural model [17], defined by *Integrating the Healthcare Enterprise* (IHE) initiative [18], which is promoted by healthcare professionals and industries to improve the way computer systems in healthcare share informations.

In the following subsections, we introduce the noSQL database engine we have chosen and we briefly illustrate what we have done and some interesting performances results.

A. Brief overview on OrientDB

OrientDB [11] is an open source NoSQL DBMS developed in Java by Orient Technologies LTD and distributed under the Apache 2 license [14]. It collects features of document databases and graph databases, including object orientation. In graph mode, referenced relationships are like edges, accessible as first-class objects with a start vertex, end vertex, and properties. This interesting feature let us represent a relational model in document-graph model maintaining the relationships.

OrientDB supports an extended version of SQL, to allow all sort of CRUD (Create, Read, Update and Delete) and query operations, and ACID (Atomicity, Consistency, Isolation, Durability) transactions, helpful to recover pending document at the time of crash. It is easily embeddable and customizable and it handles HTTP Requests, RESTful protocols and JSON without any 3rd party libraries or components. It is also fully compliant with TinkerPop Blueprints [12], the standard of graph databases. Finally, his feature can be easily customized and it supports a multi-master distributed architecture, a 2nd level shared cache and the other features offered by embedded Hazelcast [13].

B. A multi-model noSQL DBMS as an ebXML Registry

OrientDB is also a customizable platform to build powerful server component and applications: since it contains an integrated web server, it is possible to create server side applications without the need to have a J2EE and Servlet container. The customizations can be obtained by developing new *Handlers*, to build plugins that start when OrientDB starts, or implementing *Custom Commands*, the suggested best way to add custom behaviors or business logic at the server side.

The multi-model nature of the OrientDB engine allows it to support Object data model, too. This model has been inherited by Object Oriented programming and supports the inheritance between types (sub-types extends the super-types), the polymorphism when you refer to a base class, and the direct binding from/to objects used in programming languages.

The OrientDB Object Interface works on top of the Document-Database and works like an Object Database: manages Java objects directly. This makes things easier for the Java developer, since the binding between Objects to Records is transparent. In that context, the Objects are referred as *POJOs*: Plain Old Java Objects. OrientDB uses Java reflection and Javassist [21] to bound POJOs to Records directly. Those proxied instances take care about the synchronization between a POJO and its underlying record. Every time you invoke a setter method against the POJO, the value is early bound into the record. Every time you call a getter method, the value is retrieved from the record if the POJO's field value is null. Lazy loading works in this way too.

The ebXML RIM objects are perfect POJOs, because they are serializable, have a no-argument constructor, and allow access to properties using getter and setter methods that follow a simple naming convention. They are also fully described by the standard set of Java XML annotation tags because they need to be transferred over the line properly encapsulated using the Java Architecture for XML Binding (JAXB) [22]. This means that we can add new custom properties preceded by the *@XMLTransient* tag without breaking things. We have used those new properties and the related getter and setter methods to add native OrientDB links between objects, which can be transparently serialized/deserialized by the OrientDB engine in their enriched form.

This approach has a great impact on the management of ebXML objects:

a) they are still used in the standard way within the client-server SOAP interactions;
b) the binding to the database records is transparent;
c) there is no need of extra data class object (DAO) hierarchy to manage the persistence;
d) we are able to make full use of the OrientDB capabilities.

An extension of the OrientdDB *OServerCommandAbstract* class has replaced the old Java servlet in the registry requests management. In particular, the *execute()* method is invoked at every HTTP request and let us to read the input HTTP stream and to write into the output HTTP stream. This is the place where we intercept, elaborate and reply to the incoming requests, by calling the real ebXML registry layer.

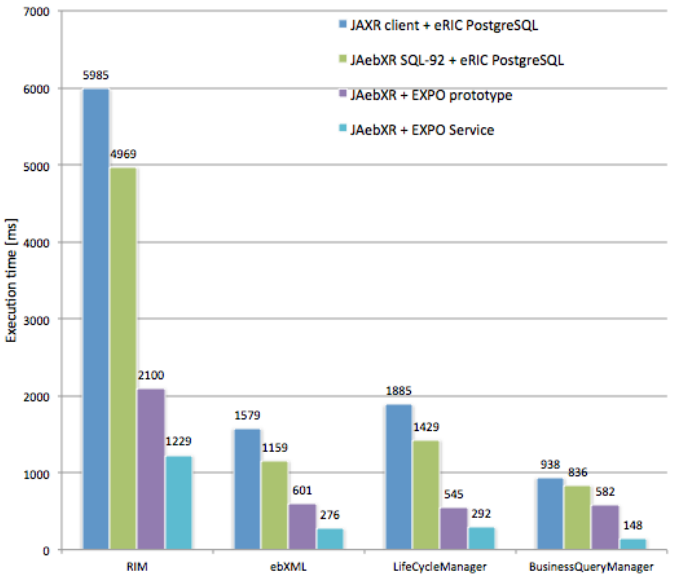

Figure 8. EXPO vs eRIC/PostgreSQL

To verify ebXML specifications compliance and to evaluate the performances, we have run the same ebXML test suite developed with the JAebXR API [23] and then we have compared the results in some different configurations. Figure 8

shows the interesting performance improvements obtained with EXPO service.

VI. CONCLUSIONS

While monoliths have been the norm for some time, microservices have emerged as an alternative to deal with certain limitation in monoliths. However, that doesn't mean that monoliths are completely obsolete. Just because others are gravitating to one more than the other, doesn't mean that it's going to be the best decision. Obviously, it's important to look at advantages and disadvantages of each and, as much information as possible, to make the aware decision. Keep also in mind that, due to the significant architectural differences, a direct comparative quantitative analysis is actually not easy to achieve, but we are working on it.

In this paper, we introduced a new microservice pattern where the database *is* the service. The proposed pattern has been tested adding ebXML registry capabilities to a noSQL database. Experimental tests have shown improved performances of the proposed simplified microservice architecture compared with SQL-based ebXML registry implemented as traditional Java web service.

REFERENCES

[1] M. Fowler, "Microservices, a definition of this new architectural term", URL: http://martinfowler.com/articles/microservices.html [accessed: 2016-02-12].

[2] M. Villamizar, et al., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud", "Computing Colombian Conference (10CCC), 2015 10th", 2015, pp. 583–590, DOI: 10.1109/ColumbianCC.2015.7333476.

[3] M. Amaral, et al., "Performance Evaluation of Microservices Architectures Using Containers", "2015 IEEE 14th International Symposium on Network Computing and Applications (NCA)", 2015, pp. 27–34, DOI: 10.1109/NCA.2015.49.

[4] Docker Inc., "Docker, An open platform for distributed applications for developers and sysadmins", URL: https://www.docker.com [accessed: 2016-02-12].

[5] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture", "International Journal of Open Information Technologies", 2014, vol. 2, no. 9, pp. 24–27, ISSN: 2307-8162.

[6] M. Abbott, T. Keeven, and M. Fisher, "Splitting Applications or Services for Scale", URL: http://akfpartners.com/techblog/2008/05/08/splitting-applications-or-services-for-scale/ [accessed: 2016-02-16].

[7] R. C. Martin, "Agile Software Development: Principles, Patterns, and Practices", Pearson Education, Nov. 2009, ISBN: 978-0-13597-444-5

[8] C. Richardson, "Microservice architecture patterns and best practices", URL: http://microservices.io/index.html [accessed: 2016-02-12].

[9] H. Hammer, "The Fallacy of Tiny Modules", URL: http://hueniverse.com/2014/05/30/the-fallacy-of-tiny-modules/ [accessed: 2016-02-28].

[10] M. Feathers, "Microservices Until Macro Complexity", URL: https://michaelfeathers.silvrback.com/microservices-until-macro-complexity [accessed: 2016-02-28].

[11] Orient Technologies LTD, "OrientDB", URL: http://orientdb.com [accessed: 2016-02-10].

[12] Apache Software Foundation, "Apache TinkerPop", URL: http://tinkerpop.incubator.apache.org [accessed: 2016-02-11].

[13] Hazelcast Inc., "Hazelcast, the Operational In-Memory Computing Platform", URL: http://hazelcast.com [accessed: 2016-02-10].

[14] Apache Software Foundation, "Apache License v2.0", Jan. 2004, URL: http://www.apache.org/licenses/LICENSE-2.0 [accessed: 2016-02-11].

[15] OASIS ebXML Registry Technical Committee, "Registry Information Model (RIM) v3.0", 2005, URL: http://docs.oasis-open.org/regrep/regrep-rim/v3.0/regrep-rim-3.0-os.pdf [accessed: 2016-02-10].

[16] OASIS ebXML Registry Technical Committee, "Registry Services and Protocols v3.0", 2005, URL: http://docs.oasis-open.org/regrep/regrep-rs/v3.0/regrep-rs-3.0-os.pdf [accessed: 2016-02-10].

[17] R. Noumeir, "Sharing Medical Records: The XDS Architecture and Communication Infrastructure", "IT Professional", Sep. 2010, Volume: 13, Issue: 4, ISSN: 1520-9202, DOI: 10.1109/MITP.2010.123.

[18] Integrating the Healthcare Enterprise (IHE), 2010, URL: http://ihe.net [accessed: 2016-02-17].

[19] A. Messina, P. Storniolo, and A. Urso, "Keep it simple, fast and scalable: a Multi-Model NoSQL DBMS as an (eb)XML-over-SOAP service", "The 30th IEEE International Conference on Advanced Information Networking and Applications (AINA-2016)", IEEE, in press.

[20] A. Messina and P. Storniolo, "eRIC v3.2: ebXML Registry by ICAR CNR", Technical Report: RT-ICAR-PA-13-03, Dec. 2013, DOI: 10.13140/RG.2.1.2108.9124.

[21] JBoss Javassist, "Javassist (Java Programming Assistant)", 2015, URL: http://jboss-javassist.github.io/javassist/ [accessed: 2016-02-29].

[22] Java Community Process, "JSR 222: Java Architecture for XML Binding (JAXB) 2.0", 2009, URL: https://jcp.org/en/jsr/detail?id=222 [accessed: 2016-02-29].

[23] A. Messina, P. Storniolo, and A. Urso, "JAebXR: a Java API for ebXML Registries for Federated Health Information Systems", "DBKDA 2015: The Seventh International Conference on Advances in Databases, Knowledge, and Data Applications", Rome, Italy, May 2015, pp. 33–39, ISBN: 978-1-61208-408-4.

 ISBN: 978-1-61208-486-2