RESTful Services for an Innovative E-Health Infrastructure: A Real Case Study

Fabio Vitali, Alessandro Amoroso and Marco Roccetti
Dept. of Computer Science and Engineering - Univ. of Bologna
Email: {amoroso, vitali, roccetti}@cs.unibo.it

Gustavo Marfia
Dept. for Life Quality Studies - Univ. of Bologna
Email: gustavo.marfia@unibo.it

***Abstract*—We present a REST approach to build Health services on the Web. Our proposal originates by the necessity of evolving the current regional health service to accommodate emerging needs that are too complex to accomplish with the current system. The original system extends on the regional scale of Emilia–Romagna, and it is one of the most advanced in Italy.**

Our architecture inherits some of the basic Web methodologies and techniques to implement an highly scalable and flexible system that is capable to satisfy the current needs and their planned evolutions. Moreover, our approach should allow for an effortlessly addressing of future requirements not foreseen at the moment.

Without revolutionizing the current IT infrastructure, our approach introduces a new paradigm that could be implemented by a sophisticated interface to access data and resources.

I. Introduction

The region *Emilia–Romagna*, in Italy, has a widespread health network, called *SOLE*, developed by the in-house company *Cup 2000*, that connects each other: all the local health authorities in its territory, the labs operating within the regional health service, the public medical specialists and pediatricians. This regional network is one of the two most advanced in Italy.

Based on a prescription from the family physician, the patients can make reservations for lab analysis trough the SOLE network. The physician and the patient will receive trough SOLE the medical report of the analysis. The documents and medical reports concerning the hospitalization of a patient are also exchanged by means of the SOLE network.

The Emilia–Romagna region is planning to extend the services and the base of users of SOLE, therefore new needs emerge; we can summarize them as: *scalability* and *flexibility*. These will be discussed in the next section.

We are proposing an innovative architecture to address the evolution of SOLE. Our main idea is to adopt the up-to-date methodologies and techniques that rules the World Wide Web. The Web exhibits extraordinary scalability and flexibility based on simple and powerful mechanisms. We propose a way to project them into the public health scenario that we are considering. Our aim is to preserve the existing IT infrastructure by adding a kind of Web interface to it in order to obtain the new functionalities. Moreover, due to the flexibility of the Web technologies, we suppose that our approach could accommodate future requests and constraints not currently foreseeable.

SOLE was designed at the beginning of this century, and the adopted technologies reflect that period. The data model is *document oriented* and the system uses the *Simple Object Access Protocol* (SOAP) messages to exchange data over the network. The payload of the SOAP messages are *Health Level 7* (HL7) documents that are generated according with the *Reference Information Model* (RIM) specifications.

Nowadays the *REpresentational State Transfer* (REST) approach offers a much more simple and lightweight architecture to *Web Applications*. This approach does not require the heavy overhead of SOAP, that requires both the client and the server to execute local computations to manage large and complex data messages [1]. Currently a huge portion of the Web services that offer both the approaches receives the great majority of request by the REST paradigm. As an example, *Amazon S3* offers the same Web services following both SOAP and REST approaches, more than the 85% of those services are accessed by means of the latter [2].

In the recent literature there are some early applications of the REST approach to e-health Web based applications. For instance, in [3] a semantic characterization of the content of health messages helps in the correct matching of services from two independent health-related databases. More closely, in [4] the HL7 *Fast Healthcare Interoperability Resources* (FHIR) standard is discussed, showing how the REST–based interchange protocol, an integral part of the FHIR standard, helps in generating faster and more responsive applications. In [5] the authors propose a modular design to implement an HTTP based system to storage, retrieval, and versioning of *Electronic Health Record* (EHR).

The remaining of the paper is structured as follows: the next section sketches the current SOLE systems and its main limitations. The subsequent section presents our proposal, that is based on a REST approach, highlighting its novelty and strengths. The §IV shows some crucial items in the design of the system based on our approach. The conclusions end the paper.

II. Current architecture and data model

The current architecture of the SOLE system could be summarized in Fig.1. The ellipses represent four administrative actors: three health authorities, simply labelled *A*, *B* and *C*, and a fourth entity, the Emilia–Romagna Region Health Service, labelled ERRHS. The rounded boxes at each actor represent

978-1-4799-6644-8/14/$31.00 ©2014 IEEE

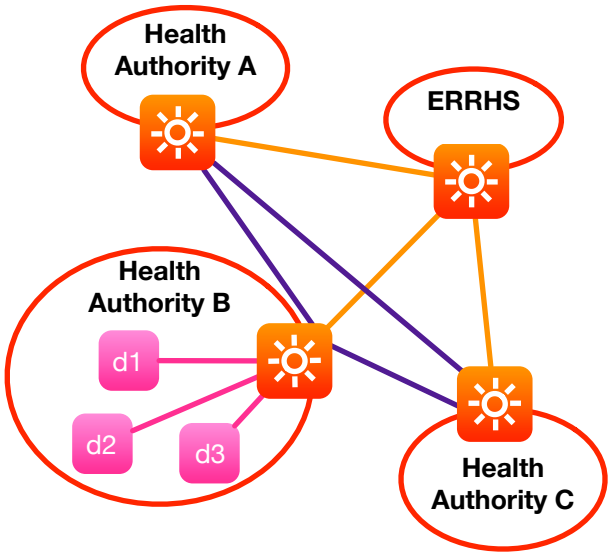

Fig. 1. Current architecture.

the SOLE services local to each one of those entities. The SOLE services exchange each other health data on behalf of the actors, and are connected to the centralized control system located at the regional health service. The solid lines connecting the SOLE services represent those communication channels.

The Health Authority B represents a large organization that includes several separate departments, depicted as small rounded boxes labelled *d1*, *d2* and *d3*. Each one of those departments has a large degree of autonomy that includes their IT infrastructures. For historical reasons, it is highly probable that each one of the departments has developed a custom information system to manage health, administrative and personal data.

The main role of SOLE is to allow all these different domains, both administrative and technical, to profitably exchange documents. This scenario leads to a "naturally" distributed system on regional scale, where the data and the relative computations are local to separate entities. It is worthwhile to mention that legal requirements concerning the privacy of patients impose that health and personal data could not be shared without an explicit authorization from the patient, and it is subject to restrictions.

A. Document vs. data model

The current SOLE implements a *document oriented* approach, *i.e.* the smallest addressable stored information are the documents, each one as a whole, regardless of the finest grain data that they contains. The main idea behind this approach was to preserve the integrity of the original documents.

Currently SOLE supports documents exchange between actors, those documents are expressed in XML following one or more document types of HL7. The documents contains data, that are frequently redundant, about entities and concepts that could be orthogonal to the purpose of the document. Those entities and concepts could be the focus of future searches and applications.

We are proposing a *data oriented* approach (see §III), that considers each single data directly addressable, while preserving the association with the original document.

B. Scalability and flexibility

The scalability has at least two dimensions: *physical*, *i.e.* the number of users, and *logical*, *i.e.* the number of possible services offered to them. The number of future users of SOLE 2.0 will extend to both all the physicians and all the residents in the region. That means increasing form 400 to 8000 the number of physicians and considering up to 4 millions of residents.

Following the data oriented approach, it is possible to implement straightforwardly a whole class of new services, that actually are too complex to design due to the current data model. An example of new services we could consider those that requires an "horizontal" access to data, that is in contrast with respect to the actual "vertical" approach that considers a "store and delivery" of HL7 messages.

One of those "horizontal" services could be the statistical analysis of a specific illness. A complete screening of prescriptions searching for correlations between specific data is hard to implement using the actual document oriented data model. The useful data are currently glued collectively with other data that are useless to the extent of the screening and they are not automatically addressable.

Following the data model approach, each data has its own identity and addressability. Therefore, almost all the data analysis could be made with respect to the *privacy* of the patients, since their personal data are not glued together with the data relevant to any research.

C. Standardization

Actually there are at least 17 different clients certified to access SOLE, each one offering different capabilities and user interfaces; each physician decides which one to use. The certification process for these clients is complex: it requires the clients to know the *internal data structure* of each remote system participating to SOLE and requires the system to check the correctness of the transactions between remote modules. Any change to the internal data structure would cause the modification of all the clients and a further run of their certification process.

Following the REST approach, SOLE will publish a set of API that could easily manage the standardization of the practitioners clients interface to the system. SOLE will not publish the internal data structure, it will publish, instead, the interface to invoke actions on the data.

III. Innovative approach

The driving idea of our proposal is to apply the models and technologies that have been developed for the World Wide Web to the new version of SOLE. Those instruments have been

developed during the last decades and share the characteristics of being highly robust and largely scalable.

A. URI vs. URL

Any resource available on the Web is univocally identified by means of a *Uniform Resource Identifier* (URI). A URI can represent a name, a location or both of them. A client can accesses a resource on the Web by means of the *Uniform Resource Locator* (URL) of the resource, that is the specific communication protocol to access the resource, *e.g.* HTTP and FTP, followed the URI of the resource.

As an example, consider this fictional URI of a document: *files.cs.unibo.it*. We could access this resource by means of different protocols, gaining different results. By means of the HTTP protocol, *i.e.* the URL: *http://files.cs.unibo.it*, we could render the content of the document in a browser window. By means of the FTP protocol, *i.e.* the URL: *ftp://files.cs.unibo.it*, we could download the document into our device.

B. HTTP commands

The main idea of the REST approach is that the functionalities and the protocol of HTTP are enough to implement complete Web applications. The *RESTful services* offer access to the resources, that are expressed in XML format, solely by means of the four main HTTP requests:

- GET: to download a document;
- PUT: to store a document in the server;
- POST: do add data to a document;
- DELETE: do delete a document from the server storage.

All of them generate a response from the server. Those requests admit also some specifications. The GET request could ask that the indicated document will be returned by the server only if it has been modified after a specific data, or if some of its metadata satisfy specific criteria. By means of the PUT request a client could ask to the server to store a specific document, with a defined name. The HTTP requests could be accepted by the server solely if they are sent by authorized clients. Moreover, the DELETE request could be accepted solely if it is sent by registered users that have specific permissions.

C. Resources–Oriented Architecture

The REST approach exposes resources and not processes. This is a complete departure from the *Remote Procedure Call* (RPC) paradigm that is shared by the SOA approach.

Exposing data is a very delicate issue, there are at least two options: the worst option is to directly show the content of the database, including its internal structure, the best option is to expose objects that wrap the data. As an example we could imagine the "prescription" object, that as a tree data structure that contains several nodes, each one individually addressable. Those nodes could include: the personal data of the patient, the id of the physician that issued the prescription, each single item, their cost, the state of each item, and so on.

The purpose of the following example is to highlight some topics of our proposal, not to exactly represent a real scenario. We refer to a likely scenario, keeping the example simple enough to be explained straightforward without being simplistic.

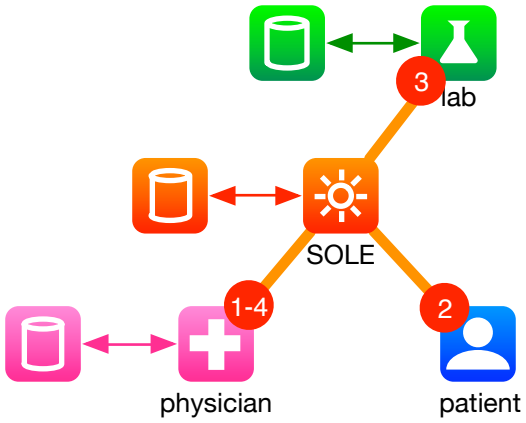

Fig. 2. Example: the doctor prescribes lab analysis to the patient by means of the Web services offered by SOLE.

The example is depicted in Fig. 2 and shows a physician that prescribes to a patient same laboratory analysis by means of the services offered by SOLE. Each one of the actors accesses the services offered by SOLE by means of an interface that is implemented by a Web page accessible by a browser. Some of the actors, *i.e.* SOLE, the physician and the lab, can store locally a copy of relevant data. The shown actors interact with SOLE performing the steps described below, that are represented in the figure by means of numbered circles:

1) the physician prescribes some laboratory analysis to the patient issuing a prescription;
2) the patient books the analysis into a laboratory (this is not modeled in the example);
3) the laboratory exec the analysis and issues a medical report;
4) the physician can access the medical report.

In this example SOLE acts like a communication channel and records all the messages exchanged by the actors. It is not strictly required that the physician has to store locally all the issued prescriptions, she/he could access the storage facility of SOLE to manage them. The laboratory stores all the medical reports that it issues and could link each one of them to the corresponding prescription.

In terms of the REST approach, each one of the steps numbered above is a single HTTP invocation:

1) the physician requests a PUT of the prescription, generating a new URI;
2) the patient is not modeled;
3) the laboratory requests a PUT of the report generating a URI derived by the one of the prescription;
4) the physician can verify the completion of the analysis either by a GET of the specified prescription or by a lookup, again with a GET, of all the prescriptions

associated with a lab report.

The physician knows the URI of a prescription, *e.g.* *abcd1234*, either new or previously issued. The physician issuing a new prescription could add it to the system by the PUT *http://www.SOLE.it/prescriptions/abcd1234*. The lookup of the prescriptions issued by the physician John Smith could be obtained by the GET *http://www.SOLE.it/prescriptions/?physician=Smith*. To add data to a prescription, *e.g.* to record that it has been reimbursed by the national health system, the actor performs the PUT *http://www.SOLE.it/prescriptions/abcd1234/reimbursed*.

The databases involved in this example are managed by SOLE, while a copy of them, either partial or complete, could be locally stored by some of the actors:

- *Citizens's personal data* used by the physician to complete the prescription. Data do not change too frequently, the physician could store personal data of their own patients;
- *Medical services* that could be supplied. Data seldom change, the physicians could store a local copy of the whole database;
- *Diseases* classified following the ICD9CM. Data seldom change, the physicians could store a local copy of the whole database;
- *Prescriptions* issued under the public health system. Data could change on a daily basis, the physicians could store a local copy of their issues;

We realized a fast prototype of both the services offered by SOLE and of the interfaces for the actors. The main technologies server side that we adopted are *URL rewriting* [6] and PHP. The client side technologies were *Ajax*, *Ajax Framework* (ExtJS) and *HTML 5*. We implemented the above databases by means of a NoSQL database, that was not a technology constraint and we adopted it to test possible bottlenecks of performances.

IV. Design of the System

We present a possible *Application Programming Interface* (API) for the Web that is based on the REST paradigm. The aim of this API is to be a factual interface to documents, entities and concepts stored in the databases local to each instance of SOLE. The clients of that API are applications that can access entities and concepts to provides services and visualizations. This approach masks to the clients both the internal complexity of the software architecture and of the data structure and storage. Any possible future modifications of the system could be implemented without requiring modifications to the client side. The API represents a single and unified point of access to the resources that is independent from the model of storage of data, their site, and their retrieval techniques.

Our proposal is flexible enough to accommodate any entity and concept that will be defined by the authorities in charge. We present both the general architecture of the API and how to modify it to allow for new entity or concept to be accessed. This approach leads to a simple method to evolve the API accordingly with future requirements, that are currently unpredictable.

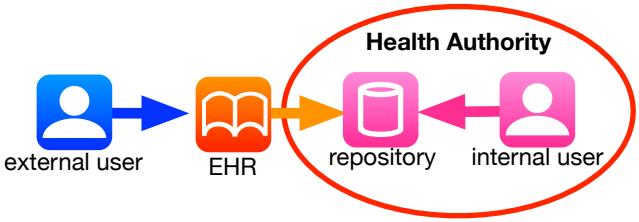

Fig. 3. Different users accessing the data.

We can summarize the main topics to design the API as follows:

1) identification of a *meta-syntax for the URIs* to systematically access, booth in reading and writing, the HL7 documents and the concepts and entities there described, *i.e.* the meta-syntax of the requests to the system;
2) identification of a *meta-format for data output* common to any request, that is linearly equivalent in XML, JSON and EDF, *i.e.* the meta-syntax of the responses form the system;
3) proposal for a *pre-elaboration of the requests* to parse the request received by the API before any access to the data and any activation of application logic of the system;
4) proposal for a *reorganization of the storage* introducing additional *indexes* to fast access of the original documents.

We consider that there are at least two different modality to access the data: internal and external access. Figure 3 shows an example of this scenario: the data are stored in a repository under the administrative control of an health authority, and they can be accessed both by an internal user or an external user.

The internal users belong to the administrative domain of the health authority and can access the data either by means of the interface directly exposed by the repository or by means of the interface implemented by a middleware or an *Enterprise Service Bus* (EBU). The external users could be either the citizens or the internal users belonging to different departments or health authorities. These external users can access the data under different rules with respect to the internal users. The *Electronic Health Record* (EHR) defines such an interface, that masks the repository structure. The EHR will belong to the administrative and technical domain of SOLE 2.0. Some of the main topics emerging from this depicted accessing methods are sketched below.

Access policies mechanism: needs to consider an increasing number of both users and their typology. It has to be easily extendable to face the needs that will surface in the future.

Data structure: the number of data and documents will significantly increase in the future. They are to be defined both the principles and the best practices to structure the data.

Selectivity of data access: must be established based on the role of each actor. They could be implemented by means of permissions lists associated to the actors.

A. Meta-syntax for the requests

The REST approach is based on the design and publishing of methods, based on URI, to access data. These URI could refer existing data, and both entity and concepts that could be either implicitly or explicitly present in the database.

The request to the system needs to be independent from the physical site of the data storage. A single point of access provides the abstraction of a "centralized" storage of the data. As an example, we can suppose that this single point of access has the following URL: *http://www.sole2.it/data/*

To define the purposes of our API, we need to mention the following basic concepts:

Instance: either of a concept or of an entity, is the set of data associated with a single individual belonging to a class; *e.g.* let "patients" be a class of entity, then the set of personal data associated with Mr. Mario Rossi from Bologna is an instance of that class. An instance could contain one or more *sub–instances*, *e.g.* the street address of the patient could be "via Indipendenza n. 5" (*i.e.* "5 Independence street").

Collection: is the set of data associated with one or more instances of that class; *i.e.* it is a set of instances. As an example, a collection could be the set of data associated with all the patients living in a specific city; that collection could be an ordered list containing solely some selected data about the patients, *e.g.* name, family name, address and tax code. Note that in Italy the tax code could be a unique identifier of each citizen.

Key: is a unique identifier associated with an instance of a class. A class could have multiple keys with non intersecting values.

In the following examples of URI we suppose the existence of the classes *physicians* and *patients*. The URI *www.sole2.it/data/patients/MRSFBA72H20A9432Z* identifies the patient whose tax code is MRSFBA72H20A9432Z. An URI identifying a collection ends with a slash "/", *e.g.* he URI *www.sole2.it/data/patients/* identifies the collection of patients. The collection of all the patients of the physician whose key is 123456 is *www.sole2.it/data/physicians/123456/patients/*.

The requests could be quite complex, and could use predefined parameters. As an example, let us consider the parameters *q, s, start, length, accept, debug* in the following URL:

http://www.sole2.it/data/patients/?q=city%20eq%20modena%20and%20born%20gt%201999/12/31&start=301&length=100&s=familyname+name&accept=text/xml&debug=true

This returns the collection of 100 records starting from the number 301 (*start=301&length=100*) from all the patients living in the city of Modena that were born after the 31 December 1999 (*q=city eq modena and born gt 1999/12/31*). That list is ordered by both family name and given name (*s=familyname+name*) and it is formatted in XML (*accept=text/xml*). Moreover, the list will contain details, whenever it is possible (*debug=true*).

	sub-istance	istance	collection
GET	-	**read** (record)	**read** (list)
PUT	**update** (if PATCH is not used)	**create** (know key)	-
POST	-	-	**create** (unknown key)
DELETE	-	**delete**	-
PATCH	-	**update**	-

Fig. 4. Relationships between HTTP methods and CRUD operations.

The four main operations of REST, as described in §III-B, could be associated with the CRUD (Create, Read, Update, Delete) functions of persistent storages. The Fig. 4 shows that relationship.

The PATCH request has been added for the sake of simplicity, it allows to specify within the request what are the fields that will be modified ad what are the updated values. There are tree possible PATCH operations: the *substitution* of existing values, the *additions* of new values to a populated field, the *removal* of a value from a populated filed.

B. Meta-format for the responses

We consider that the systems encodes both the request and the responses as UTF-8. Any response could be described by the following MIME types: *i) XML:MIME type text/xml* or *application/xml*, *ii) JSON:MIME type application/json* and *iii) Turtle RDF:MIME type:text/turtle*. The type of the response could be contained in the request, as shown above.

Any response from the system could contain either entities or concepts that could be formatted as records or list.

Record format: contains all the known data about the considered instance. It could contain also linked data whenever those are appropriate with respect to the size of the record. As an example consider the record of a patient that could contain a section about the physician, and the record of a prescription could contain the records of the prescribed drugs. The whole record could be accessible solely by authorized actors. It could be the case that same actors are authorized to access solely a portion of the complete record.

List format: could be either a JSON array or an XML subtree, whose elements are a short representations of the selected instances. The response could also contain paging information, such as the number of all the records that satisfied the request and the number of returned records.

C. Logic–functional description of the system

Any public API accepts requests from unknown actors, we could generically call them *clients*, that request services. The services provided by our API, following the REST approach, are both reading and writing of the data that are actually managed by SOLE. Our API should be flexible enough to accommodate future evolutions that are not predictable now.

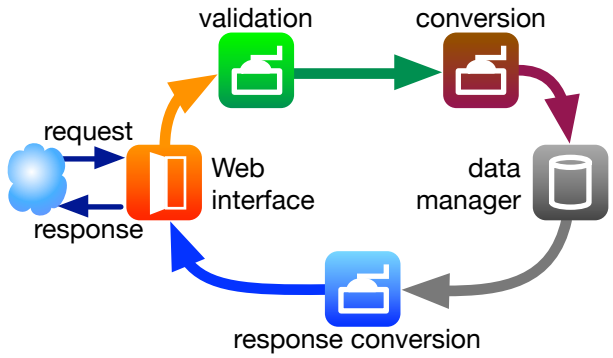

Fig. 5. Internal data flow for the proposed architecture and its modules

Moreover, our architecture should be able to manage in a seamlessly way the existing data, without disruption of the existing services.

The proposed architecture is sketched in Fig. 5. The system could be decomposed in 5 modules, each one performs a step in the elaboration of the request and is connected in a pipe with the others, *i.e.* the output of one module is the input for the next one. The module receiving a request is the *Web interface*, that is connected to the Web, depicted as a cloud, that module is available at a predefined HTTP address, and represents the centralized access point to the system. It receives the requests and sends back the responses. The *validation* module analyzes and transforms the requests according to the capabilities of the client to avoid that she/he could access unauthorized services. The *conversion* module, translates the request into a sequence of actions that can be executed by the internal system, *i.e.* it allows the API to be independent from the actual service. The *data manager* module is the actual service. The *response conversion* module translates the format of the data from the internal representation of the actual service to the format requested by the client. The output of the *response conversion* module goes to the first module that therefore can respond to the request.

The communication channels between the modules are secured by HTTPS and implement an adequate level of cryptography with respect to the current state of the art.

The users of the system could play different *roles* depending on their current activity, *i.e.* we could say that the user is "acting as" a specific role. As examples consider a physician that could be also a patient, or a physician that is an internal user in a specific health department while she/he is an external user for all the remaining departments. The *validation* module avoids the return of restricted data to unauthorized actors.

Depending on the role on the requester, the system could return different levels of detailed data. *E.g.* a physician could access all the details about the data of her/his patients, while could access solely a digest of the patients of other physicians; a patient could access all of her/his data, while none of the data of the others.

Our system has to be robust with respect to possible misuse and attacks, such as the *SQL injection* technique [7]. More generally, a robust system could be obtained applying one or more of these techniques:

- *input validation*: by parsing the query to identify suspicious code;
- *temporary sandboxing*: executing the query into secure and controlled environments;
- *virtual tables*: accessing the databases solely by means of VIEWs;

A critical point when migrating to the REST approach is the architecture of the storage system maintained by SOLE. It is not feasible to force any department or health authority to revolutionize their current storage systems, that are mostly document oriented, to satisfy the new data oriented paradigm. Our main idea is that the local instances of SOLE will maintain their own "version" of the local data according with the new paradigm. Considering the huge quantity of data, the DBMS to adopt has to be carefully chosen between: XML native, JSON native and SQL. We estimated that the "atomic" data, *i.e.* not further divisible, contained in the XML documents produced solely by the emergency departments of the regional public hospitals in the 2013 is about 5×10^8.

V. Conclusions

We presented a novel approach to build the regional health services of the Emilia–Romagna region in Italy. Our approach is based on the REST paradigm and has been described in this paper mainly from a syntactic point of view.

We believe that our proposal is flexible enough to accommodate the requirements and the planned evolutions of the current system.

There are remaining open issues, such as the kind of DBMS to store the data maintained by SOLE, that will be analyzed in the close future.

Acknowledgment

We are indebted towards *CUP 2000 s.p.a.* for the financial and technical support.

References

[1] A. Rodriguez, "Restful web services: The basics," IBM, Tech. Rep., 2008. [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-restful/

[2] T. O'Reilly, "Rest vs. soap at amazon." [Online]. Available: http://www.oreillynet.com/pub/wlg/3005

[3] N. Miyoshi, A. Ferreira, and J. C. Felipe, "Ontology-based approach to achieve semantic interoperability on exchanging and integrating information about the patient clinical evolution," in *22nd IEEE International Symposium on Computer-Based Medical Systems*. IEEE, August 2009, pp. 1–6.

[4] D. Bender and K. Sartipi, "Hl7 fhir: An agile and restful approach to healthcare information exchange," in *26th IEEE International Symposium on Computer-Based Medical Systems (CBMS 2013*. IEEE, June 2013, pp. 326–31.

[5] E. Sundvall, M. Nyström, D. Karlsson, M. Eneling, R. Chen, and H. Örman, "Applying representational state transfer (rest) architecture to archetype–based electronic health record systems," in *BMC Med Inform Decis Mak*, May 2013, pp. 13–57.

[6] "Apache mod_rewrite," The Apache Software Foundation, Tech. Rep., 2014. [Online]. Available: http://httpd.apache.org/docs/current/rewrite/

[7] [Online]. Available: http://en.wikipedia.org/wiki/SQL_injection