One solution of a RESTful API for a cloud based DTV content provider

Stefan Pijetlović, Nevena Jovanov, Violeta Vukobrat
RT-RK Institute for Computer Based Systems
Novi Sad, Serbia
{stefan.pijetlovic, nevena.jovanov, violeta.vukobrat}@rt-rk.com

Ilija Basicevic
Faculty of Technical Sciences
University of Novi Sad
Serbia
ilibas@uns.ac.rs

***Abstract*— In this paper we propose a solution of a RESTful application programming interface (API) for a cloud based digital television (DTV) content provider. The system described is responsible for the acquisition of data from the digital television transport stream, retrieving of the related content from the Internet and its exposure to the clients. The focus of the paper is on the REST architectural style and how it affects the overall design of the system.**

***Index Terms*— API, data acquisition, DVB, internet content, REST**

I. Introduction

Representational state transfer (REST) is an architectural style that focuses on roles of the components, which make up a particular system, and their interaction instead of details of the implementation and protocol syntax. The REST paradigm consists of several formal constraints which shape the system so that it evokes an image of how a well-designed web application behaves [1]. In a web application, a client navigates through it using links. Similarly, using the API defined in this paper, the client follows different hyperlinks in order to navigate through the virtual states of our application and retrieve the information from the server. This information consists of Digital Video Broadcasting (DVB) services, DTV events, teletext pages, electronic program guide (EPG) and various content from the Internet which is in some way related to the program being watched.

The paper is organized as follows: first, the system architecture is briefly described, then an insight in the REST paradigm is given, followed by the impact of the architectural style on the design of the system. In the end we have given our view on the solution along with several plans for the future.

II. System Architecture

The simplified system architecture is shown in figure 1. Our cloud based DTV content provider is an application server which communicates with two types of devices. The blocks on the left represent the client devices which are, in our current use-case, Android based set-top boxes. These devices contain the user interface and request the content from the server. On the other hand, the block on the right represents the acquisition devices which feed the server with the content from the DTV transport stream. The acquisition devices used in our use case are also Android based set-top boxes which scan the DVB terrestrial and satellite frequency span.

The term content represents several sorts of information which can be delivered (DTV events, web pages, movie trailers, etc.). All this content comes from two disjunctive sources: the stream and the Internet. The idea is to create a mix from both sources and represent it in such a way that the user does not know its origin, because this information is irrelevant to the average user. This way, we transform the role of the television receiver from a unilateral medium to an interactive device due to the fact that the input coming from the Internet is personalized and based on the client's profile. The user has control over his or her profile and can send feedback to the server so that the system can learn and improve in order to better pinpoint the users taste. The content is connected to the so called tags - words and concepts that represent some form of metadata and put certain events into logical groups such as movies, sports or events about cooking. Depending on the tag, different content will be displayed such as, for example, trailers or actors' biographies for the movie tag, or some new and interesting recipes for the cooking shows. Along with this, the system will recommend other events possibly interesting to the viewer, based on the viewer's profile generated by his or her habits and inputs [2].

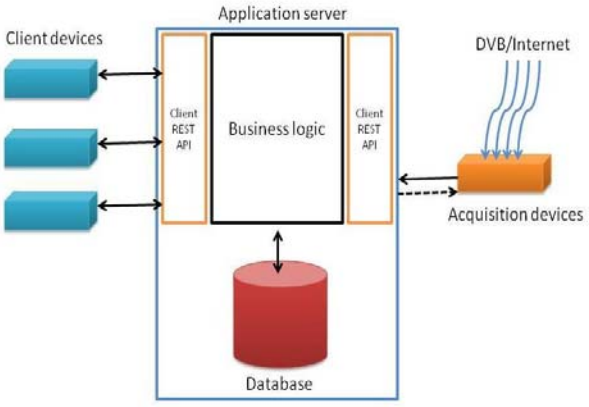

Figure 1. Simplified system architecture

III. REST

Like mentioned in the introduction, REST is a coordinated set of architectural constraints which apply to the connectors of the system. The goal of this kind of architecture is to minimize the latency and network communication while at the same time maximize the component independence and scalability. There are five formal REST constraints: client-server architecture, stateless communication, cacheable responses, layered system and uniform interface; and an optional code on demand constraint which was not applied. In order to characterize a web service as RESTful, it must not violate any of these constraints.

The point of the client-server architecture is the separation of concerns - each side focuses on different tasks, making both sides easier to implement. For example, most of the complex application logic is located on the server and the graphic user interface is implemented on the client. This type of architecture gives the option to develop both sides relatively independent as soon as the basic infrastructure has been set up. Once the API is designed, the changes on the server have little or no effect on the clients because they will be using the same functions defined with the API with no idea what is underneath as long as the data model has not changed. And due to the fact that the biggest amount of processing is done on the server, the clients can even be installed on embedded systems. Also, by completely removing the UI from the server we achieve improved portability over multiple platforms because several types of clients can use the same API but have completely different implementations.

Stateless communication means that the entire session state is kept entirely on the client side. The server does not store the state of the clients, thus making the server implementation less complex and more scalable. Instead, each request is independent and every response has all the information needed to make a new request in order to progress to the next virtual states. This way, there is no need for an awareness of overall component topology.

Cacheable responses constraint is present in order to increase the efficiency of the entire system. Cache has the potential to partially or sometimes completely eliminate some interactions. If the requested resources have not changed between two consecutive requests, the client need not contact the server but can instead use a local copy of the representation. This not only relieves the pressure on the network and the server but also enhances the user experience on the client side, because the information needed for the user interface will be accessible faster.

A layered structure presumes the presence of intermediaries and every layer cannot see beyond its “neighbors”. This pipe-and-filter-style structure not only suits our system well, but also increases both the scalability (by providing load balancing, caching, etc.) and security of the system, thus improving the overall performance. Pipe-and-filter is a known architectural design pattern, published in [3].

The uniform interface constraint represents the backbone of the design of any RESTful service. This fundamental constraint has four principles: identification of resources, manipulation of resources, self descriptive messages and hypermedia as the engine of application state (also known as HATEOAS).

Identification of resources enforces that the resources are conceptually separate from their representation which is sent to the client on request. For instance, if we have a resource which is a rectangle, its representation could be two real numbers which are the numeric values of the length of its sides. Another representation of this exact same rectangle can be four pairs of real values which represent the vertices of the rectangle in some coordinate system. This representation is then sent to the client in a format which best suits the clients capabilities (XML, JSON, HTML, etc.). The manipulation of resources means that once the client gets hold of the representation of some particular resource, it has enough information to modify or delete that resource on the server, if it has permission to do so. The self descriptive messages principle means that every message should include information on how to be processed for example which parser should be used. As for HATEOAS, the client navigates through the application using the hyperlinks in a response which leads to the possible states. The client assumes that there are no actions available besides the ones described in the received representation [1].

REST relies on HTTP protocol and utilizes mainly the four request methods: GET, PUT, POST and DELETE. In addition, it uses many other features of HTTP like, for instance, layered proxy or caching.

An alternative to REST would have been the Web Services Description Language (WSDL) combined with Simple Object Access Protocol (SOAP). After defining the use-cases for our system, the WSDL and SOAP alternative was, however, dismissed due to REST’s lightweight infrastructure and relative ease of development [4]. The ease of development is measured with the number of architectural decisions which have to be made and is presented in the table below.

Architectural decision	REST	WSDL
Remote procedure call	X	X
Messaging		X
Resource interaction semantics – Lo-REST	X	
Resource interaction semantics – Hi-REST	X	
Resource relationships	X	
Request-response message exchange pattern	X	X
One way message exchange pattern		X
Service operations enumeration by functional domain		X
Service operations enumeration by non-functional properties and QoS		X
Service operations enumeration by organizational criterion		X

Total number of decisions	REST	WSDL
	5	7

Table 1. Number of architectural decisions which have to be made

We can confirm the results given by Pautasso [4].

IV. Impact of the REST constraints on the system design

Even though the uniform interface constraint is fundamental and affects the system design the most, it was not difficult to fulfill because the organization of data in the DVB standard is well structured and our data model replicates this structure. Every single service is uniquely identified with the so called DVB triplet which consists of the network ID, transponder ID and service ID. Every DVB event is identified with a combination of its own ID and the DVB triplet of the service it is broadcasted on. It is all the same when teletext pages are regarded. Thanks to REST's custom data modeling possibility, we have expanded this structure by connecting the additional content from the Internet to certain events. For performance issues and to increase the amount of data the recommendation engine has to work with, we have added the ability to access all events from a certain transponder or network, skipping the service and/or transponder identifier.

The client-server architecture was the natural choice from the start due to the two distinctly different tasks of the system: the data storage on one hand and the user interface on the other. The majority of application logic is located on the server side, so the clients are simple enough to be implemented on low performance embedded systems, such as set-top boxes, without the drop in the quality of service. Stateless communication and caching constraints apply with every request the clients send and also contribute to the overall performance of the system.

The "top" REST layer is implemented using the JAX-RS, a Java API for creating RESTful web services. JAX-RS uses annotations to map resource classes as web resources. The number of lines of code did not cross 1500 during the period when this paper was written. This is mainly due to the fact that Java offers a very high level of abstraction. The annotation mechanism also greatly reduces the amount of code needed to implement the desired functionality. The entire system is implemented using Java Enterprise Edition Beans. From the "top" REST layer, all the data is forwarded to a management layer which is the only intermediary which this layer sees. The caching and load balancing is managed by the EJBs in the layers underneath. This way the layered system constraint is fulfilled.

The client set-top boxes retrieve all the data they need using HTTP get methods. By aiming at different hyperlinks, they can retrieve collections of data entities such as events or teletext pages, or a specific data entity from a group. Likewise, the acquisition set-top boxes deliver all the relevant information from the DTV stream to the server. Here we can see the intuitiveness of REST. For example, if the client wishes to request all the events from a particular service, it would send a HTTP get request to the link:

http://domain:port/rest/networks/{onid}/transponders/{tsid}/services/{sid}/events

Onid, tsid and sid represent, respectively, the network, transponder and service ID, because our system is based on the DVB standard [5]). On the other hand, the acquisition set-top box can store the events using the exact same link but he would send a HTTP put request. The same goes for the request/submission of the services or teletext pages. If the client wants to see the additional content we provide, the hyperlink for the current event would look like the following link:

http://domain:port/rest/networks/{onid}/transponders/{tsid}/services/{sid}/events/{id}/content

This method does not have the acquisition version because the acquisition devices only feed the information from the DVB transport stream.

V. Conclusion

This paper presents our experience in designing a simple yet effective RESTful API. We have tried to fulfill all the constraints to the best of our ability in order to make the solution straightforward, flexible and easily expandable due to RESTs clever design.

In our proposed solution, we chose this alternative over SOAP and WSDL. By doing so, we had a fast start due to REST's intuitiveness and already well developed HTTP infrastructure. Furthermore, the style was rather easy to adopt. The custom data representation and modeling possibility of REST proved to be very handy and gave us the ability to further optimize our system.

As for future plans, the aim is to expand the system possibilities with advanced algorithms for text analysis in order to better categorize the DTV events. With better categorization the recommendation systems will be more precise when recommending the content to the clients. We also plan to improve the user profiling and to add new services.

Acknowledgment

This work was partially supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia under Grant TR32030.

References

[1] Fielding, Roy T.; Taylor, Richard N. (May 2002), "Principled Design of the Modern Web Architecture", ACM Transsactionss on Internet Technology (TOIT) (New York: Association for Computing Machinery) 2(2): 115-150

[2] S. Pijetlovic, N. Jovanovic, N. Jovanov, S. Ocovaj, "One solution of a system for data acquisition and storage from the digital television transport stream and its exposure to the clients", 21st Telecommunications Forum TELFOR, Belgrade, November 2013.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, M. Stal, "Pattern Oriented Software Architecture Volume 1: A System of Patterns", 1996.

[4] Pautasso, Cesare; Zimmermann, Olaf; Leymann, Frank (April 2008), "RESTful Web Services vs. Big Web Service: Making the Right Architectural Decision", 17th International World Wide Web Conference (WWW2008) (Beijing, China)

[5] ETSI Digital Video Broadcasting (DVB), Specification for service information (SI) in DVB systems: http://www.etsi.org/deliver/etsi_en/300400_300499/300468/01.11.01_60/en_300468v011101p.pdf loaded 25.02.2014.