Efficient Entities Management for Java Autonomous Web Applications

Patrick Bellot
Telecom Institute, Telecom ParisTech
CNRS LTCI-UMR 5141
Paris, France
bellot@telecom-paristech.fr

Loïc Baud
Telecom Institute, Telecom ParisTech
CNRS LTCI-UMR 5141
Paris, France
baud@telecom-paristech.fr

Abstract— **In [11], we presented the architecture of SAW, a Web server technology that can be efficiently applied to a wide range of Web applications and services. It is a framework for easy programming of Web applications and services. It is not intended to act as a full and open Web server but is application-dedicated. It has been implemented in Java and used by the authors to propose a fully functional Web application for online conference management. After a general description of the server, we give the design patterns concerning entities management used in its conception. The goal is to have efficient and easy-to-program entities in autonomous Web applications written within the SAW framework.**

Keywords: Web application, Web services, design, development, autonomous server, scalability, entities management, design pattern, Java.

I. INTRODUCTION

There are many ways to implement Web applications and services. We can use WAS (Web Application Servers) such as GlassFish, IBM WebSphere, JOnAS, BEA Weblogic, JBoss, etc. WAS are heavy and powerful tools implementing most of Web 2.0 and 3.0 advances. They are well suited for EIS (Enterprise Information Systems) because all EIS applications and services can be embedded in a single framework resulting in a corporate culture. We can also rely on LAMP architectures (Linux, Apache, MySql, PHP) but these Web applications are sometimes difficult to maintain, to modify and to extend. We can also use CMS (Content Management Systems) that allow elegant template-driven designs of Web sites and provide access to many community tools. However, CMS are not intended to build Web sites with significant business functions.

Our proposal, presented in [11], is to build dedicated autonomous Web applications based on our framework without using any external components (except the file system). SAW is the concatenation of many interesting paradigms that make it easier to program real size applications more efficient than equivalent LAMP applications. SAW stands for Stand Alone Web server and is written in Java. Many Web servers are available as Open Source software [2]. They are able to deliver static contents and some implement Servlet and JSP technologies. SAW tries to avoid WAS weight and complexity while providing the well known WAS triplet: Container, Session beans and Entity beans [7]. Session beans implement the business logic. Containers provides an execution context to the Session beans so that the programmer does not have to worry about technical aspects such as handling TLS/SSL, decoding the request, managing the GET and POST variables or managing the users sessions. Entity beans contain the data of the Web application. Because our Container, Session and Entity are not really Beans in the sense of Java EE6, we provide them under the similar name of Binz.

SAW is not intended to be as powerful as a WAS or LAMP architecture. A SAW server is not an open server where authorized users can deposit various contents. It must be dedicated to a single Web application. In [11], we emphasized that one of the reasons of its efficiency is that it is dedicated to a single application allowing automatic memory or file caching of Session binz outputs and Entity binz data. An application is a Java program where Entities binz are the global data and Session binz are entry points. Nowadays, there is a real need for high performance and scalable web servers [9]. A SAW application is intended to be executed by any reasonable computer and proxied by an Apache server if it does not have direct access to Internet. Apache servers are able to efficiently proxy many applications and thus can be a front-end for many back-end applications or services distributed on several back office computers, reducing the load of the font-end server. This ensures horizontal scalability with a low economical impact.

We have used SAW to implement a fully functional and very complete online conference management system [5]. It is operational and used for a conference with around 800 users, 150 submitted papers with 3 reviews per paper. The application covers every aspect of a conference or congress.

In this communication, we focus on Entities management in SAW autonomous Web applications. In section 2, we explain the different patterns we used to manage entities with the notions of Permanent entities, Ephemeral entities and Dispatched entities. In section 3, we briefly explain how entities are indexed using specialized hash tables. In section 4, we describe how we implemented Web client agent management of entities using a RESTful approach and how we implement the RESTful approach in a generic way using Java reflection classes. And, in section 4, we show how we use integrated RESTful services in our conference management application.

II. MANAGING ENTITIES

Web application data are usually stored in a relational database and represented by entities. An Entity class matches a table in a relational database and each instance is a row in that table. The mapping of entities objects to the relational database tables is done via hibernation mechanisms such as JPA (Java Persistence API) or another [10]. Access to entities is done via SQL commands and depends on the annotations provided in the class of the entity. For instance, the programmer of the entity class may declare that one attribute of the object corresponds to a primary index field in the corresponding relational database table. Programming this type of entities requires some knowledge and mastering the annotations mechanism but it is rather easy in the powerful IDE (Integrated Development Environment) associated to the WAS.

This approach is interesting and appropriate if there is a large amount of data to manage. On the other side, current 2011 computers have high computational capabilities (e.g. 3 GHz and several cores, possibly hyper threaded), big memory cache in processors (up to several MB) and very big RAM memories (several GB is now a common memory size). Different entities may not have the same memory requirements. For instance, in an online shop, the entities corresponding to sold items may require a large amount of memory but a list of admitted customers countries is a small entity. We also make a distinction between entities often used and entities rarely used. In SAW, the programmer is free to manage entities using a relational database because she/he has access to the programming power of Java and can use the various databases drivers and hibernation mechanisms. However, SAW proposes mechanisms for managing entities: either to keep entities in memory or to store them in disk files. The class **`EntityBinz`** is the main tool to handle entities. It provides two main generic functions:

- The class method `static EntityBinz wakeup(Class eClass);` reads the entity of class **`eClass`** from the disk file and returns it.
- The instance method `void hibernate();` saves the calling entity to the disk.

All entities inherit **`EntityBinz`**. The functions use the class name of the entity to compute a name for the file where to save/load the entity. The entity is loaded or saved using Java serialization, thus entities must be serializable. Now, let us consider the three different patterns we used for entities management. The way an entity is managed depends on its usage and its size.

A. *Pattern 1 - Permanent entities*

In our conference management application, we have an entity that contains users internal identifiers, login name, rights, encrypted passwords and other users data. This entity is not very large (around 5 MB) because we have less than 800 users. This entity must be quickly accessed because it is used at each login on the web site and each operation requiring user rights control. Therefore, it can be kept in memory. Such a Permanent Entity can be declared with the pattern of Figure 1.

This entity has a unique instance within the application. This instance can be only accessed with the expression **`UsersBinz.getUnique()`**. It must be saved on disk at the server shutdown with a call to **`hibernate()`**. Because the entity is unique through the application, access to its internal data may require synchronization. A power failure is always possible. Thus, hibernation must be ordered at regular time intervals. That's why the entity registers to a backup manager.

B. *Pattern 2 – Ephemeral entities*

Our conference management application requires an entity containing all countries names. This entity is used to produce the code of a XHTML country selection widget at the start of the server and each time the list of countries is modified. It is also used each time a user modifies its account data to update her/his account data. These are rare usages. If needed, the entity is loaded. If the entity is modified, it must be saved using its **`hibernate()`** method. Several instances of the entity, loaded by different Session binz, may exist at a given time. Thus, if two authorized Session binz modify the entity, the one who hibernates it later wins.

```
public class UsersBinz extends EntityBinz
{
    private final static UsersBinz uniqueEntity ;

    static {

        UsersBinz entity = (UsersBinz)wakeUp(UsersBinz.class) ;

        if (entity == null)
            entity = new UsersBinz() ;

        BackupManager.registerEntity(entity) ;

        OnExitManager.registerEntity(entity) ;

        uniqueEntity = entity ;

    }

    public final UsersBinz getUnique() {
        return uniqueEntity ;
    }

    … // Entity implementation
}
```

Figure 1: Permanent Entity pattern

```
            return entity ;
        }

        … // Entity implementation
    }
```

Figure 2: Dispatched Entity pattern

C. *Pattern 3 – Dispatched entities*

Each submitted paper of the conference management application has data such as the paper unique identifier, title, authors, current version, reviewers, reviews and so on. The application must be able to retrieve papers identifiers associated to an author, to retrieve authors associated to a paper, to retrieve reviews associated to a paper, etc. These operations can be done on the numerical identifiers using entities of reasonable memory size (a few MB) such as **`PapersToAuthorsBinz`**, **`PapersToReviewsBinz`**, and so on. However, the data associated to a paper such as the reviews contents are large enough and used so rarely that they can be kept on the disk. But these data are not kept in a single collection entity. Instead, each review will have its own entity saved on disk. That means that the reviews entity is dispatched in many smaller entities, one for each review. Technically, we keep indexing tables in memory and dispatched data in disk files. That's why we introduced supplementary new wake-up functions with supplementary numeric parameters that are used only for computing more complex file names for the saved entity. And the reviews entities are stored as described in Figure 2. The entity can be loaded only when needed for display or edition. In our application, there will be no conflicting modification of the entity since only the reviewer can modify the review.

III. INDEXING ENTITIES COLLECTIONS

When we have entities collections such as the collection of all users of our conference management system, indexing these collections is mandatory. Many indexing mechanisms exist, for instance B-Tree was proposed by R. Bayer in 1971 [12]. Using hash table for indexing is known to be a memory waste [6,3]. However, because we do not have so much data, we decided to use oversized hash tables. Because Java hash tables only allow objects and no scalar values as keys and because they implement functions that we do not need (automatic size growing, many tests on values and keys), we wrote specialized hash tables classes: one for numerical keys and one for object keys. These hash tables can be used for primary or ordinary indexes. For instance, in our conference management application, we know that the number of users is around 800, probably never exceeding 1024. Thus, we pragmatically chose to index them by their numerical identifier and by their email using hash tables of size 2048. The cost of this table is not significant if compared to the computer memory size. With this scheme, the probability is very high that accessing an indexed user can be done in constant time. And if the number of users increases over our expectation, the only result would be to slow the access and the fix is evident: stop the server and double the size of the two hash tables.

IV. GENERIC RESTFUL SERVICES FOR ENTITIES

Representational State Transfer (REST) is a style of Web services architecture dedicated to resources management [13]. The important concept in REST is the resource that is referenced with a global identifier, for instance a URI in HTTP. To manipulate these resources, client agents and servers communicate via HTTP(S) and exchange representations of resources. The REST server allows the four main CRUD operations: (C)reation of a resource, (R)ead a resource, (U)pdate of a resource and (D)eletion of a resource. The request type identifies the operation: POST for creation, GET for reading, PUT for update and DELETE for deletion. The parameters of the HTTP requests and the answers to the client are represented in a way independent from the resources themselves. Using XML or JSON are usual schemes for representing parameters.

Les us consider the News system of our conference management application. The more recent news is displayed when loading the web site. And the user has the ability to ask for the previous news. The site administrative users can add news or suppress them when required. Let us say that all the conference News are stored in a Permanent Entity of the class **`data.conference.NewsBinz`** that is a collection of objects of class **`data.conference.News`**. Each News has a random numerical identifier. It contains the identifiers of the preceding and following News. According to RESTful paradigm, the URI **`http://.../data.conference.NewsBinz/123`** identifies the news with number **`123`**. We chose JSON (JavaScript Object Notation) as the data interchange format [14] because it is lighter and supported by both Java on the server side and JavaScript on the client side. Moreover, we don't need the expressive power of XML. Our implementation of RESTful entities management system is based on a unique Session binz using the reflection classes of Java to propose a generic service and listening on a specific port. By extracting the name of the Entity class from the URI, it is able to get the corresponding class object and to recover the Entity object.

Then, by extracting the numerical identifier, it is able to recover the corresponding resource and to operate the client request. The reflection classes of Java make this generic RESTful implementation possible. A possibly more efficient solution would be to have a dedicated and automatically generated Session binz for each class of managed Entity.

V. RESTful client management of entities

Classically, Web application operates in transactional mode. They propose a page (the View) with a form reflecting the current state of the data (the Model). Local edition is done and the submit button sends the current state of edition to the server that replies with the new state of data.

We found that in some case, it is more interesting to forget about the usual transaction mode. The point is that the View on the client side must always reflect the Model on the server side. This is obtained using hidden light RESTful transactions during the client edition. The transaction is handled by some JavaScript code in the Web client agent. An action of the user on the edition interface results in a RESTful request. If the request fails, the interface must be restored to its previous state. We implemented some specialized pages using this technique when reloading a full page using form submission was really painful.

VI. Conclusions and futureworks

We presented three design patterns for efficient and rational management of entities in Web applications. We also explained how to efficiently manage entities edition using the concept of RESTful service.

Because some operations on entities require a lot of computations and produce peak of memory and CPU usage, for instance getting the list of the users sorted by their names, we now study the feasibility of having external servers dedicated to entities and accessed using Java RMI (Remote Method Invocation). The Web application server would ask entities operations to a entity server. This would reduce the load of the Web application server but it requires more backend communication. Moreover, entities could be shared by different Web applications. A first proof of concept has been implemented.

References

1. S. A. Walberg. “Tuning LAMP systems’. IBM DeveloperWorks, March 2007.
2. http://java-source.net/open-source/web-servers
3. B. Tindale. “Hash Tables in Java”. The Linux Gazette, n° 57, Sept. 2000.
4. K. Zorgdrager. “Tips and tricks for better Java performance”. IBM Developer Works, May 2001.
5. IEEE-RIVF Conference. http://www.rivf.org
6. H. Garcia-Molina, J. D. Ullman and J. D. Widom. Database Systems: The Complete Book. Prentice Hall, 2nd edition, ISBN-13: 978-0131873254, June 15, 2008.
7. E. Roman, R. P. Sriganesh and G. Brose. “Mastering Enterprise JavaBeans, 3rd edition”, Wiley, 2005.
8. C. Folini. “Apache attacked by a slow loris”. http://lwn.new/Articles/338407/
9. S. Souders. “High performance Web Sites”, O’Reilly Media, Sept. 2007.
10. W. Iverson. “ Hibernate: A J2EE™ Deveoper’s Guide”. Addison-Wesley, 2004.
11. P. Bellot, L. Baud. “A Dedicated Architecture for Efficient Web Server Technology”. The 3rd International Conference on Theories and Applications of Computer Science (ICTACS 2010), Can Tho (Vietnam), Sept. 2010.
12. R. Bayer, E.M. MacCreight. “Organization and Maintenance of Large Ordered Indexes”. Acta Informatica 1, pp. 173-189, 1972.
13. R. Fieldings. “Architectural Styles & the Design of Network-based Software Architecture”. PhD Dissertation. University of California, USA.
14. “JavaScript Programming Language, Standard ECMA-262 3rd Edition”, Dec. 1999.