Software Architecture for a Consensus Moderation System

Andrei TOMA
Bucharest University of Economics, Romania
andrei.toma@ie.ase.ro

Implementing a scalable consensus moderation system imposes certain restrictions on the choices in employed software technologies, as well as in the general architectural approach. Two approaches are necessary, due to the inclusion of a recommendation engine based on previous user behavior, which is computationally intensive. Apart from the recommendation engine, the system can be implemented with clarity of the model as a priority which will lead to better future maintainability.
Keywords: *Consensus Moderation, RIA, Java Enterprise*

1 Introduction

The principles behind the construction of a consensus moderation system have been presented in [4].
The present contains a review of the architectural choices that have been made in the construction of the system, as well the reasons for which certain technologies or approaches were selected.
In order to implement a flexible architecture for the system, certain considerations must be taken into account.
The first decision which must be taken into account is that of the type of application which is to be implemented. While the system is designed to facilitate the negotiation between a small numbers of actors, it could easily be adapted to cover decision in small communities as long as the designed architecture is flexible enough.
A special consideration in the design of the system is that some of the processing steps, especially those pertaining to the recommendation engine can be extremely time consuming when applied to a large amount of data.
The first important design decision is that of the type of application which the system is to be implemented as. While desktop applications used to be the standard for application solutions, today the most prevalent solution is that of web or mobile applications. This ensures that access to the system can be done from heterogeneous platforms. This degree of independence refers to independence of the operating system on the users' machines, from the browser they opted for and to a certain extent from the installation of any additional applications.
It would also be preferable that the technologies employed ensure a high degree of scalability in order to accommodate for future application growth. By application growth I refer to an increase in the number of users, but also to an increase of the complexity of interactions between the users.
Another important concern is that such an application must have a high degree of reliability. This must be ensured not only for the first version of the application, but also for future releases. As such, the technologies employed should, to such an extent as to not affect the performance of the application, be easily maintained and decrease the coupling between components.
The second important decision in establishing the architecture of the system is what kind of structures the application should have. Here we must decide between a two-tier design or an n-tier design.
A two tier design is similar to the classical client server architecture, with the mention that the server is usually a web server. This architecture is easier to implement, but usually leads to a system which is less adaptable if scalability issues arise.
An n-tier architecture adds additional components, which in our case refers to the addition of an application server. An application server adds to the flexibility of the system by supporting advanced features such as object lifecycle management and load balancing.

There is, however, a middle ground solution which allows for the application to be implemented as a two tier system initially, while allowing for an easy move to an n-tier architecture. The solution is to use a technological solution which is configurable in that the components are unaware of their container, be it an application server of a servlet container running in a web server.
A summary of the application architecture is presented in the figure below. Each component, as well as the actual implementation principles, will be explained further in this paper.

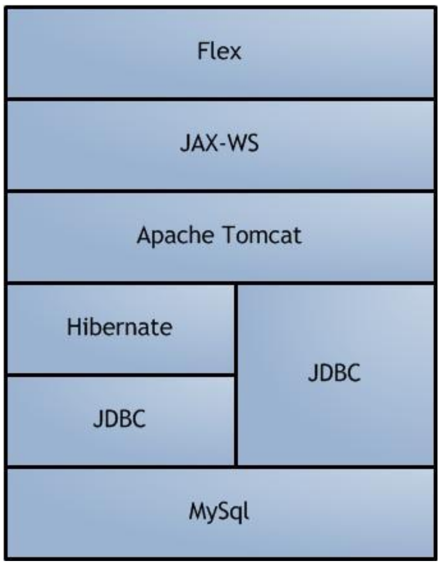

Fig. 1. Used technologies

As we can see, the application employs a Flex front-end and a JAX-WS back-end, running on Apache Tomcat. The JAX-WS web services run inside an Apache Tomcat container and access the data in two ways. If no optimizations are necessary, data is accessed through Hibernate which in turn uses JDBC to communicate with the database. If the Hibernate approach is not viable, JDBC is used directly. Considering that the priority is the execution of small transactions as fast as possible, the data is stored in MySql.

2 Front end

When choosing a possible implementation for the application front-end, one decision is paramount. Should the implementation be based on a so called “thin client” or a “thick client”.
A thin client is a client which uses only technologies which are supported by major web browsers “out of the box” without additional software extending the functionality of the browser. A thin client depends on the back end to run most of its processing. Thin clients come with the advantage of requiring very little maintenance on the client side since no additional software has to be installed. However there are serious difficulties in implementing a thin client which provides the same functionality as a thick client. This can negatively impact ease of use, since a full-fledged user interface, based on a flexible event model, is harder to design.
A thick client is a front end which does at least part of the processing necessary for the application on the client computer. Thick clients require the installation of additional components on the client computer and thus can lead to higher support costs if the users are untrained.
The typical thick client used to be desktop application connecting to a server. However, with the advent of rich internet applications, this is no longer the case. RIA’s provide similar functionality to traditional applications while running on a platform which is integrated with the internet browser the client already has.
An important question which must be answered is if the existence of the browser plug-in required to run the RIA can be assumed. FLEX, which is the selected technology for the front-end runs on the Flash plug-in, which has a remarkable adoption rate with users (http://www.adobe.com/products/player_census/flashplayer/version_penetration.html). It can thus be assumed that 90% of the users will already have the possibility to run the application without the installation of any additional software.
Taking into consideration these arguments, the best choice for the front-end is a RIA running on a browser plugin, which allows for a full-featured GUI without making extensive support necessary.

3 Backend

There are a series of considerations which must be taken into account when constructing the architecture of the back-end of the application.
The first matter which must be taken into account is that of the interface between the front-end and the back-end. Interfacing the two components must be as easy as possible while promoting flexible design. Flexible design is represented here by loose coupling of the two components. By this I understand that modifications in the front-end must not influence the back-end and vice versa.
An additional concern is that, while the system is designed as a web application, the possibility of a future mobile application must be taken into account. For this reason, the back-end, which contains the actual application logic, must remain usable if the front-end is changed, preferably without any modification.
This is doubly important since the degree of homogeneity which one can find in web applications is not found for mobile applications. Mobile applications are dependent on the platform on which they run and device producers have been known to limit the access to certain technologies. As such if the system is to be used on a wide variety of devices, the front-ends will be radically different and be necessarily implemented in different technologies.
The solution which grants to most flexibility is to use web services, since web services can be accessed by practically any type of front-end. Web services allow remote access to objects through specialized protocols. When using web services, there are two possible main technological variants, classic web services, communicating through SOAP or REST-ful services [6].
Both SOAP and REST operate by transmitting data over the HTTP protocol, but there are significant differences in the actual approach.
In a general sense, REST architectures are classical client server architectures. Clients send requests to servers while servers process those requests and return responses (wikipedia). Requests and responses consist of the transmission of representations of resources, where the meaning of a resource is anything that can be associated an address. The representation of a resource is a document reflecting a particular state of a resource, a state which changes subsequent to client requests.
The original implementation of a REST approach is the HTTP protocol itself, but the approach is not limited to HTTP. Indeed, a REST-ful approach can be applied further up the protocol stack in order to obtain more flexible application mechanisms.
While SOAP services are much more clearly defined, the definition for REST-ful services is more of a guideline. A REST-ful web service is a simple implementation of a web service using HTTP and built with REST principles in mind. Because REST-ful services are structured very closely to HTTP itself, a service definition is composed of the base URI of the service, the internet media type of the data which is to be transmitted by the service (also known as a MIME type) and the set of operations supported by the service [5].
It must be mentioned here that the operations supported by the service are subset of HTTP methods (GET, POST, PUT etc.) as no new actions (“verbs”) can be defined. In contrast, SOAP services can define an unlimited number of actions and are not constrained to the HTTP methods.
SOAP services allow remote access to objects through the Simple Object Access Protocol. While the communication with the remote objects is done through SOAP, the protocol in itself is not sufficient. An application accessing a SOAP web service must have access to the definition of the remote objects. This definition is contained by a special file which is written in the WSDL (Web Service Definition Language) language which is an extension of XML (as is SOAP itself).
The WSDL definitions for objects are extremely large and prohibitively hard to read and maintain. As such, they are usually generated through specialized tools. However, the fact that the WSDL must be downloaded by the client leads us to one of the weakness-

es of SOAP which is the fact that a lot of additional information is needed in order to use a remote object. Apart from the overhead generated by using a WSDL definition the SOAP envelope itself contains a lot of additional information. As such, SOAP services generally transmit a significant amount of additional data.

The advantage, however, is the use of implementation and flexibility, since the client does not have to have any prior knowledge of the implementation of the objects on the server. Especially since the system involves using different technologies from different families on the front-end and the back-end, this is a significant advantage which justifies the selection of SOAP services.

In order to implement web services, a dedicated API which is included in Java Enterprise edition will be used. As a language, Java has the advantage of flexibility and portability, while maintaining significant application speed (Java is similar to C++ in speed, despite running in a virtual machine). Since much of the language is open source, there are numerous options for any task. For example, while web services can be implemented with the standard Java API, which is JAX-WS [2], they can also be implemented with different open source technologies such as Apache Axis [3].

The second important decision when implementing the back end of the application relates to data representation. While any representation of the data will be exposed through SOAP web services, it is also important to establish what that data representation will be.

There are a number of approaches which could be employed. First of all, JSE (Java Standard Edition) offers an API for DBMS agnostic database access. JDBC (Java DataBase Connectivity) [1] [7] offers a way to abstract database connectivity through the use of JDBC drivers which intermediate communication with the actual database server. JDBC is a mature technology with a high degree of flexibility granted to the implementer. It allows the application to either abstract much of the communication with the database or to finely tune it through more database dependent mechanisms.

However, if implementing an application using JDBC, there is a lot of “boilerplate” code which must be included in the application. This code is difficult to abstract and will affect the clarity of the application logic and thus the ease with which the application can be maintained.

As such, JDBC is a good implementation solution only when fine control over database operations is needed. For all other situations, it is preferable to employ a database abstraction layer which allows the application objects to be written to the database without any code “clutter”.

The problem, which generates the additional code, is a fundamental mismatch between the way relational databases represent data and the way objectual programming languages do it. In order to avoid this problem, an ORM (Object Relational Mapping) API [8] can be used. Advantages of this approach are multiple, from the added clarity, since the application better reflects its model, to the fact that the way objects are stored in the database is not configurable and can easily be changed if the evolution of the application demands it.

In the implementation of the consensus moderation system however, there are two sets of tasks which have significantly different computational needs. As such, while an approach, which employs an ORM library and thus allows a better management of the complexity of the code, is preferable it is not possible to employ it for all tasks. The core classes of the application can thus be stored in the database through an ORM, while the recommendation engine will be implemented via classic JDBC in order to allow for finer control and thus make the needed optimization possible.

One additional decision is that of the DBMS on which the system will rely. Since there will be a significant amount of processing involving simple, but time consuming, queries a DBMS which handles low complexity queries quickly is preferable.

Due to the complexity of the architecture of the system, I will focus on the implementation details for the back-end components which are presented in the next section.

Hibernate

Most web applications involve the use of a database and the consensus moderation system is no exception. The main entities (proposals, issues etc.) involved in the functioning of the system are stored in the database and access to them must be provided through the web service.

Since these are core entities, it is preferable that their model is as clear and maintainable as possible. This is a problem when employing a traditional approach to database access.

As stated in the introductory part of this chapter, there is a fundamental mismatch between the way object oriented programming represents real entities and the way they are stored in relational databases. Relations between entities are even harder to represent since there is no correspondent in the relational model for inheritance and composition. These relations can be represented through one-to-one and one-to-many relations which in turn involve the use of foreign keys in the database. While this approach accomplishes the task at hand, it leads to unnecessarily complex code and increased maintenance costs.

The solution is to somehow map the fields of objects to attributes in the relational database, a task which is accomplished through an ORM API [14]. In Java, there is significant standardization through the Java Persistence API (JPA) [15] which was launched with the release of EJB 3.0 [9][10].

The most well established persistence API is Hibernate, which is the industry standard for ORM. Hibernate is an implementation of the JPA standard, but offers significant added functionality, some of which can be employed while still keeping the application JPA compliant. This aspect is important because in the event that replacing the ORM API is considered necessary, Hibernate can be replaced with any other JPA compliant ORM .

While solutions such as Hibernate are not appropriate for any situation, they represent a good first choice since alternative approaches can be used simultaneously. This is especially important for the consensus moderation system, since some operations must be done using traditional JDBC [11].

Object Relational Mapping is a term defining direct persistence of traditional Java objects, which are called Plain Old Java Objects. The term POJO reflects the fact that these are indeed objects which do not contain any code specifying that they are to be stored in the database. Because of this, such objects can easily be used in a different application which does not involve storing them in a database. Hibernate is an API which allows persistence of POJO’s without significant constraints as to the type of the object which is to be persisted.

Hibernate allows considerable flexibility in mapping POJO’s and the relations between them. Objects can be mapped to one table or to multiple tables and several POJO’s can be mapped to a single table. There is also significant flexibility as to the conventions used to name database entities since the columns to which each field can be specified as well as the table names.

Hibernate supports a series of relations between objects which cannot be found in the relational model such as inheritance between classes.

Although there is some performance overhead when Hibernate starts up and its session factory is created and configured, Hibernate is a fast tool[11][12]. Finally, using Hibernate does not require using any special environment such as an application server or servlet container thus making the application using it much more flexible to change.

Since Hibernate uses POJO’s, there is very little dependence between the objects themselves and the persistence layer. As such, persistence could be taken out of the application without affecting the application logic. While this scenario is far-fetched, this leads to the fact that if optimization that is not possible through Hibernate is needed, it can easily be replaced with a different solution.

As such, there is very little that constrains the design of the application when using Hibernate, which remains a first option for almost

any application which requires that objects be saved in a database.
In order to communicate with the database, Hibernate employs a session factory, which is a heavyweight object (there should be only one session factory per application) from which the application can get database sessions. At a first glance, this might seem similar to getting JDBC connections from a DriverManager, but sessions have significantly more functionality than connections. For example, sessions support transactions out of the box and a session factory can easily be configured to support connection pooling.
Since the session factory involves significant processing, it is usually separated from the rest of the application and wrapped inside a singleton object [13], which is an object constructed on a design pattern which guarantees that only one instance of the object can be obtained. The code below shows a HibernateUtil class which plays this particular role, providing the same session factory to any object that might need it:

```
package util;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.SessionFactory;
public class HibernateUtil {
  private static final SessionFactory sessionFactory;
  static {
    try {
       sessionFactory = new AnnotationConfiguration().
                                        configure().buildSessionFactory();
        } catch (Throwable ex) {/*exception treating code*/}
    }
    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

What is not reflected by the code above is the fact that any session factory is constructed based on a configuration. This configuration is located by convention in an XML file called hibernate.cfg.xml located in the default package. This is, of course, a convention and as any convention, it can be overridden. The file contains session factory tag which specifies connection data (connection string, username, password) as well as what type of database is to be used and which JDBC driver. Since these settings are located in an external configuration file, they can easily be changed without modifying the code. Apart from these settings, the class also contains mapping tags for all the classes which must be persisted:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD
3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">
            org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">
            jdbc:mysql://localhost:3306/printing_house2</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">
            welcome123</property>
    <property name="hibernate.query.factory_class">
            org.hibernate.hql.classic.ClassicQueryTranslatorFactory
    </property>
    <mapping class="entity.Person"/>
   </session-factory>
</hibernate-configuration>
```

ORM with Hibernate can be accomplished in two ways. The principle of ORM is that some kind of mapping must exist between the objects which are to be stored and the structure of the database.
There are two possible approaches in accomplishing this with Hibernate, using XML mapping files and using annotations.
A third option is the use of a convention over configuration approaches, which consists in assuming table and attribute names from the English language. For example, a table storing objects of type User would be called Users with this convention (the table storing an entity is named as the plural of that particular entity) being used as long as it is not overridden. Hibernate has significant support for convention over configuration, especially when using annotations, but mapped objects have to contain a minimum amount of information about the way they are to be stored.
The first approach is to use mapping via XML files. This carries the advantage of having a highly configurable application since the mappings can be modified without having to recompile the application. However, the XML files can become extremely complex and, since they are separated from the objects they refer to, lead to a significant difficulty of keeping track of changes in the application.
The class which is mapped to a database table is an object without any additional information. An example of such a class can be seen in the code below:

```
package entity;
public    class    Employee    implements
java.io.Serializable {
     private int idemployee;
     private String name;
    public Employee() {
    }
    public Employee(int idemployees) {
        this.idemployee = idemployees;
    }
    public    Employee(int    idemployees,
String name) {
       this.idemployee = idemployees;
       this.name = name;
    }
    //omitted: setters and getters
}
```

In order to create the mappings, an external XML file has to be created. By convention, this file is called Employee.hbm.xml. The file specifies a mapping tag which contains a class tag. The class tag specifies which database is to be used (catalog), which object is to be mapped (name) and what name the table in which the objects are persisted should have (table). Inside the tag, the various fields of the object are mapped to table columns, with the possibility to specify restrictions on the field. Special attention should be given to the id column, in this case idemployees. This is one of the few restrictions hibernate imposes, that all mapped classes have the correspondent of a primary key. This is because while the objects can be accessed while in memory via a hash key, this key is lost through persistence, making it impossible to differentiate objects with the same content. The primary key is generated via one of the strategies the DBMS supports:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE  hibernate-mapping  PUBLIC  "-//Hibernate/Hibernate  Mapping  DTD  3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class catalog="hibtests" name="entity.Employee" table="employee">
    <id name="idemployee" type="int">
      <column name="idemployees"/>
      <generator class="assigned"/>
    </id>
    <property name="name" type="string">
      <column length="45" name="name"/>
    </property>
  </class>
</hibernate-mapping>
```

The second approach, and the one which is preferred in the implementation of the consensus moderation system, is to use annotations. Annotations are additional information inside a class which have no significance to the compiler, but are to be used by other tools. Using annotations, the mappings are no longer configurable from external files, but the connection between the object fields and their corresponding table columns is much clearer. Additionally, when using annotations, much less information has to be specified, since the application can rely heavily on conventions. As such, column names no longer need to be specified and will be extrapolated from the names of the attributes.
An annotated proponent class is presented below. Since conventions are used, there are very few things that must be added explicitly in the annotations. The @Entity annotation tells hibernate that this is to be a persistent class. The @Id annotations specifies which is the primary key while the @GeneratedValue annotation specifies that the primary key is to be generated with whatever default mechanism the DBMS uses.
Convention ensures that the table name is easily determined (Proponent) as well as the name of the attributes. The primary key will be idproponent, while the fields will maintain their names (username and password):

```
package entities;
import java.io.Serializable;
import javax.persistence.*;
@Entity
public    class    Proponent    implements
Serializable {
    private String username;
    private String password;
    @Id
    @GeneratedValue
    private Long id;
    public Proponent() {
    }
    //ommitted  constructor,  setters  and
getters for fields
}
```

One of the most important features Hibernate offers is the ability to reflect connections between objects in a relational database. As I said before, there is a fundamental mismatch in the way that the objectual model and the relational model see entities. Connections between objects which seem trivial lead nevertheless to complicated persistence code.
In the code below we have a composition relation between Proposal and Issue, in that a proposal contains a list of issues. It is necessary to save proposals and issues in the database without allowing any anomalies, such as "orphan" issues. Also, each proposal is associated a proponent. This is done by specifying the fact that there should be a one to one relation between Proponent and Proposal and a one to many relations between Proposal and Issue.
At the database level, one to many relations are implemented by default via linking tables, which are tables which map primary keys in a table to primary keys in another table. One to one relations are specified via the @OneToOne annotation. The attribute cascade specifies in what manner should modifications be propagated. If CascadeType.ALL is specified, any modification to a proposal is also propagated to the respective proponent (if, for example, a proposal is saved, the respective proponent is also saved).
One to many relations are specified with the @OneToMany annotation. Aside from specifying the cascade attribute, it is also necessary to specify how elements on the "many" part of the relation are to be retrieved. By default, Hibernate has "lazy" fetching, which means that if a proposal is read from the database, the issues will not be read unless expressly specified. By setting fetch type to eager this behaviour is changed and the issues are loaded and available as soon as their respective proposal has been loaded:

```
package entities;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.persistence.*;
@Entity
public class Proposal implements Serializable {
    @OneToOne(cascade= CascadeType.ALL)
    private Proponent proponent;
    @OneToMany(cascade = CascadeType.ALL, fetch=FetchType.EAGER)
    private List<Issue> issues = new ArrayList<Issue>();
    private String content;
    private String title;
    @Id
    @GeneratedValue
    private Long id;
    public Proposal() {
    }
    public Proposal(Proponent proponent, String title, String content){
        this.proponent = proponent;
        this.content = content;
        this.title = title;
    }
    public void addIssue(Issue issue){
        issues.add(issue);
    }
    public List<Issue> getIssues() {
        return issues;
    }
    //omitted: setters and getters for remaining fields
}
```

Using Hibernate it is possible to reflect one to one and one to many relations which are characteristic for the relational model into the object model of the application.

JAX WS

Apart from a way of persisting objects in the database there is another important matter which the back-end must cover. Since the objects are to be accessed remotely from a Flex interface, a technology must be employed to provide technology agnostic remote access.

As discussed above, web services represent a way of accessing remote object which presents the added advantage of maintaining a clear communication standard. The main concepts relating to web services and their implementation are presented below.

Since web services are a way of accessing objects remotely, web service definitions are represented in Java as classes. JAX-WS adopts a streamlined approach to defining web services in the sense that web services are simple classes with few other restrictions. The web service class itself does not have to do anything special and the web service specific parameters are added through annotation. This approach guarantees that the class can be used in a different context, such as a desktop application running outside a servlet container.

Specifying the fact that the class is to be exposed as a web service through SOAP is done with a simple @WebService annotation with the minimal requirement of specifying the service name. The service name is needed in order to construct the URL at which the service definition can be found.

For example, the (incomplete) web service below could be found at http://localhost:8084/consensusWeb/consensusWeb.

This URL is needed in order to access the service's methods. The service definition, which is an XML file written in the WSDL language can be found at http://localhost:8084/consensusWeb/consensusWeb?wsdl.

The basic definition of a web service can be seen in the code below; the service does not need to specify anything other than the annotations and does not have to implement any interfaces:

```
package services;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;
@WebService(serviceName =
"consensusWeb")
public class consensusWeb {/*web service
code*/}
```

In order for the web service to specify actual behaviour, web methods have to be added to the definition of the service

Although not obvious from the code (since the service does not implement Serializable) the methods of the service class should return serializable types and receive serializable parameters. This condition applies to any method exposed through the public interface of the class (web methods have to be public).

This is important since the result of the execution of the methods have to be used remotely. The types must be reconstructed when received by the method invoker which would not be possible if they contained resources which were fundamentally local (such as database connections or file descriptors).

There are two main types of methods which are employed by the data access service and examples for both will be presented below. Both types of methods employ Hibernate in order to attain access to the data.

The first method which is necessary is a method which adds an entity to the database. Such a method is addSolution which adds a Solution entity to the database. A Solution object is a POJO annotated for Hibernate persistence as seen in the code below. Of note is the fact that a Solution implements a custom comparator which is involved in the detection of compromises:

```
package entities;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.persistence.*;
@Entity
public class Solution implements Serializable, Comparable<Solution> {
    @OneToOne(cascade = CascadeType.ALL)
    private Proponent proponent;
    private String content;
    @OneToMany(cascade = CascadeType.ALL)
    private List<Endorsement> endorsements = new ArrayList<Endorsement>();
    @OneToMany(cascade = CascadeType.ALL)
    private List<Comment> comments = new ArrayList<Comment>();
    @Id
    @GeneratedValue
    private Long id;
    //omitted: contructor, setters and getters for the fields
    @Override
    public int compareTo(Solution s) {
        if(endorsements.containsAll(s.endorsements)) return 1;
        if(s.endorsements.containsAll(endorsements)) return -1;
        return 0;
    }
}
```

Solutions are, however, contained by issues so a solution will be added to the corresponding issue. The relevant parts of the definition of the issue are presented below. An issue contains a list of solutions, which are to be cascaded. By setting CascadeType to ALL we ensure that when an issue is modified, such as by adding a solution to it, if the issue is saved all operations cascade to the dependent entities (the solution is also saved).

The addSolution method itself is annotated with JAX-WS annotations. The @WebMethod annotation specifies that the method must be exposed by the web service. All parameters to the method are annotated with @WebParam and must be serializable types. In order to add a Solution to the database, the method receives as parameters the proponent, issue and the actual content of the solution.

Both the Proponent and the Issue objects must be loaded in order to construct a Solution object (since it contains reference to its proponent and is referenced by an issue). The load method is not generic [13] so it returns an Object type object which must be cast to the respective type.

After obtaining a Proponent and an Issue, a Solution object can be created based on the proponent and the content. The solution is the added to the issue to which it refers.
After this step, the issue is saved to the database. Since the issue contains a list of solutions which is set to be cascaded, it automatically saves the solution to the database. After saving the issue all that is left is to commit the changes and close the session:

```
@WebMethod(operationName = "addSolution")
public String addSolution(@WebParam(name = "issue") Long issue,
  @WebParam(name = "proponent") Long proponent, @WebParam(name =
  "content") String content) {
  Session session = HibernateUtil.getSessionFactory().openSession();
  try {
    session.beginTransaction();
    Issue i = (Issue) session.load(Issue.class, issue);
    Proponent prop = (Proponent) session.load(Proponent.class, proponent);
    Solution s = new Solution(prop, content);
    i.addSolution(s);
    session.save(i);
    session.getTransaction().commit();
    return "done";
  } catch (HibernateException he) {/*handle exception*/}
  finally {session.close();}
    return "done";
}
```

The second important type of method involved in the construction of the data service is a method which returns a list of entities based on some sort of criteria, such as returning all solutions proposed for a particular issue. This is accomplished via the listSolutionsByIssue web method which is presented below.
The issues are returned remotely as a generic list of List<Solution> type. Since the Solution type is serializable, the generic list is also serializable.
The method is annotated with the @WebMethod annotation and receives the issue as a parameter annotated with the @WebParam annotation.
After the initial steps of obtaining a session and opening a transaction, a query must be created via the createQuery method of the Session class. Queries created with the createQuery method are constructed with based on a character string written in the Hibernate Query Language (HQL). The Session class also supports traditional SQL queries defined with the method createSqlQuery, but using HQL queries is preferable due to the fact that they work with the actual entities and thus maintain a higher level of clarity.
Since we need to select all solutions proposed for an issue, we will select the Issue objects with the specified id. From the issue, using the special operator elements, a list of the solutions can be obtained [16].
HQL queries are parametrizable through the use of placeholders such as ":issue" [13][16] in the example below. The actual value of the parameter is then set via a setType method, in this case setLong. After setting the actual parameters, the query can be executed via the list method of the Query class. This method returns a List object which can then be returned by the web method (there are both generic and non-generic versions of list, but in the example the generic version is used):

```
@WebMethod(operationName = "listSolutionsByIssue")
public List<Solution> listSolutionsByIssue(@WebParam(name = "issue") Long issue) {
  Session session = HibernateUtil.getSessionFactory().openSession();
  List<Solution> list = new ArrayList<Solution>();
  try {
    session.beginTransaction();
    Query q = session.createQuery("select elements(i.solutions)
      from Issue i where i.id = :issue");
    q.setLong("issue", issue);
    list = q.list();
    session.getTransaction().commit();
        } catch (HibernateException he) {/*handle exception*/
        } finally {session.close();}
        return list;
    }
```

While the service defines accessor methods for all entities, their definition is similar to the ones for solutions. A slightly simplified version of the methods is defined if the respective entity does not depend on any other entity.

Conclusions

A consensus moderation system is best implemented as a web application in order to ensure easy access to all participants without requiring additional infrastructure. Also, flexible design is preferable in order to ensure high maintainability of the system.

Constructing a consensus moderation system capable of taking into account previous user behavior poses additional challenges, as recommendation related operations are computationally intensive. As such, components involved in the recommendation process must be designed with efficiency in mind, even if some of the design clarity is lost (with the effect of lower maintainability).

References

[1] M. Fisher, J. Ellis and J. Bruce, JDBC API Tutorial and Reference (3rd Edition). *Prentice Hall*, 2003.
[2] M. Kalin, Java Web Services: Up and Running. *O'Reilly Media*, 2009
[3] T. Kent, Developing Web Services with Apache Axis. *LULU* , 2006
[4] A. Toma, Consensus Moderation System**,** *Informatica Economica*,2011
[5] http://en.wikipedia.org/wiki/Representational_state_transfer
[6] C. Pautasso, O. Zimmermann and F. Leymann, RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision. (Beijing 2008).
[7] P. Mahmoud. JDBC Recipes: A Problem-Solution Approach, *Apress*, 2005.
[8] T. Halpin and T. Morgan, Information Modeling and Relational Databases, Second Edition, *Morgan Kaufmann*, 2008.
[9] R. Monson-Haefel and B. Burke, Enterprise JavaBeans 3.0 (5th Edition), *O'Reilly Media*, 2006.
[10] A. L. Rubinger and B. Burke, Enterprise JavaBeans 3.1, *O'Reilly Media*, 2010.
[11] J. Linwood and D. Minter, Beginning Hibernate, *Apress* 2010.
[12] S. M. Thampi, Performance Comparison of Persistence Frameworks, *ArXiv*, 2007.
[13] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, *Addison-Wesley Professional*, 1994.
[14] K. Roebuck, Object-relational mapping (Orm): High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors, *Tebbo* , 2011.
[15] M. Keith and M. Schincariol, Pro JPA 2: Mastering the Java(TM) Persistence API (Expert's Voice in Java Technology), *Apress*, 2009.
[16] G. Mak, Hibernate Recipes: A Problem-Solution Approach, *Apress*, 2010.

Andrei TOMA has a background in both computer science and law and is interested in an interdisciplinary approach to IT Law related issues. He has graduated the Faculty of Cybernetics of the Academy of Economic Studies in Bucharest and the Faculty of Law of the University of Bucharest. He holds a Ph.D. in Economic Cybernetics and Statistics at the Academy of Economic Studies. His current fields of interest include IT Law related issues, as well as flexible enterprise application architectures.