

# Final Report

## Interoperability in Social Networking Web Applications

Franco Sellitto

**Tutor:** Cédric Mesnage  
**Advisor:** Mehdi Jazayeri

June 29, 2007

### Abstract

Social networking web 2.0 applications are made of common semantics and functionalities. Interoperable web applications reuse machine readable information published by remote applications. We deliver a framework which provides an advanced starting point to develop interoperable social networking web applications.

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	What is the Web 2.0? . . . . .	4
1.2	What is Single sign-on? . . . . .	4
<b>2</b>	<b>Case Studies</b>	<b>5</b>
2.1	Photos: Flickr . . . . .	5
2.2	Social Bookmarking: Del.icio.us . . . . .	6
<b>3</b>	<b>Analysis</b>	<b>8</b>
3.1	Social networking . . . . .	9
3.2	Resource management . . . . .	10
3.3	Interoperability technologies in Web Applications . . . . .	11
3.3.1	OpenID . . . . .	11
3.3.2	SOAP . . . . .	12

3.3.3	Semantic Web - RDF and ActiveRDF . . . . .	12
3.3.4	REST - Representational State Transfer . . . . .	13
3.3.5	Mashup . . . . .	14
3.4	Analysis conclusions . . . . .	14
<b>4</b>	<b>Design</b>	<b>15</b>
4.1	Model-View-Controller . . . . .	15
4.2	Gems, Plugins and Engines . . . . .	15
4.3	Inheritance . . . . .	17
4.3.1	Polymorphism . . . . .	19
4.3.2	Object Prevalence . . . . .	20
4.4	Interoperability . . . . .	20
4.4.1	REST and ActiveResource . . . . .	20
4.5	Design conclusions . . . . .	22
<b>5</b>	<b>Implementation</b>	<b>23</b>
5.1	Webnode . . . . .	23
5.2	Database schema . . . . .	25
5.2.1	ID-Tuple . . . . .	25
5.3	SSO Single Sign-On . . . . .	25
5.4	Search Engines . . . . .	27
5.5	Rendering . . . . .	27
5.6	Caching . . . . .	28
5.7	Performance . . . . .	31
<b>6</b>	<b>Application Prototype</b>	<b>32</b>
6.1	How to install . . . . .	32
6.2	How to start . . . . .	34
6.2.1	Required Plugins . . . . .	34
6.3	Filtered and sorted lists . . . . .	34
6.4	How to extend the platform building custom websites . . . . .	35
6.5	How to customize the plugin views . . . . .	37
6.6	Issues . . . . .	38
6.6.1	Webnode not available . . . . .	38
6.6.2	Execution expired . . . . .	38
<b>7</b>	<b>Conclusions</b>	<b>39</b>

## 1 Introduction

In recent years, one of the biggest internet phenomenon has been the spreading and exploitation of social networking websites. Communities of people acting as content providers, sharing every kind of resource of interest through structured platforms. The ease of publishing convert each internet surfer into an active contributor, allowing them exhibit their freedom of speech and expression on the Net.

Today most of the web applications on the web are isolated and there is a big lack of interoperability between web applications. The output of data are rendered directly to the browser in human readable format. The trend is to improve the level of understandability of content by the machines, so that data can be interpreted and managed across applications without the intervention of human users.

We implement not just an isolated web application on the top of Ruby-OnRails framework (<http://www.rubyonrails.org>), but with an interoperability web approach, instead of just providing a set of isolated functionalities, our web application can be distributed among different domains and web servers, so that data from different resources can be processed exploiting their meaning.

In order to see the web not just as a set of segmented repositories, but as a big relational database as well, the data has to be meaningful to the machines too. Instead of providing just human readable data (basically HTML), the applications can exchange and interpret structured data as well, such as XML.

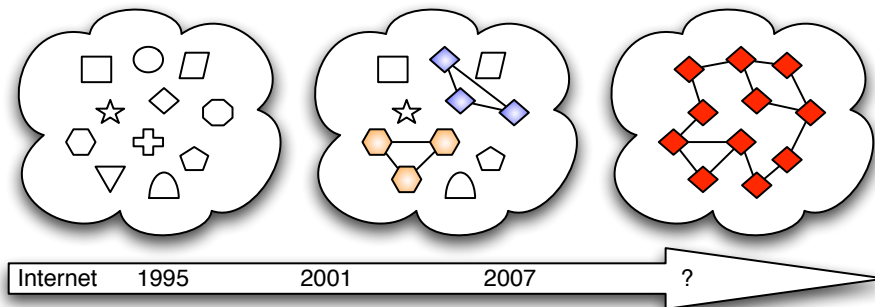


Figure 1: Internet evolution

## 1.1 What is the Web 2.0?

Web 2.0 refers to a perceived second generation of web-based services, such as social networking sites, wikis, communication tools, and folksonomies, that emphasize online collaboration and sharing among users. Web 2.0 is made up of contributions by ordinary people.

It also refers to the transition of Web sites from isolated information silos to sources of content and functionality, thus becoming computing platforms serving web applications to end-users.

In addition it can be thought of as a social phenomenon, embracing an approach to generating and distributing Web content itself, characterized by open communication, decentralization of authority, freedom to share and re-use, and “the market as a conversation”.

Summarized key principles of Web 2.0 applications:

- the web as a platform
- data as the driving force
- network effects created by an architecture of participation

The combination of social-networking systems with the development of tag-based folksonomies, delivered through blogs and wikis, sets up a basis for a semantic web environment.

## 1.2 What is Single sign-on?

In order to publish content, all web 2.0 applications require to have an account. The most annoying task is to create a new account for each web application, each of which comes with their own authentication needs and user repositories. At one time or another, everyone needs to remember multiple usernames and passwords to access different applications on a network, ending up having many disseminated accounts without any relation between them.

Authentication is a horizontal requirement across multiple applications, platforms, and infrastructures. Single sign-on (SSO) is a specialized form of software authentication that enables a user to authenticate once and gain access to the resources of multiple software systems. The approach of digital identity will shape a new world where a user could store the identity attributes within the framework of an online identity providers.

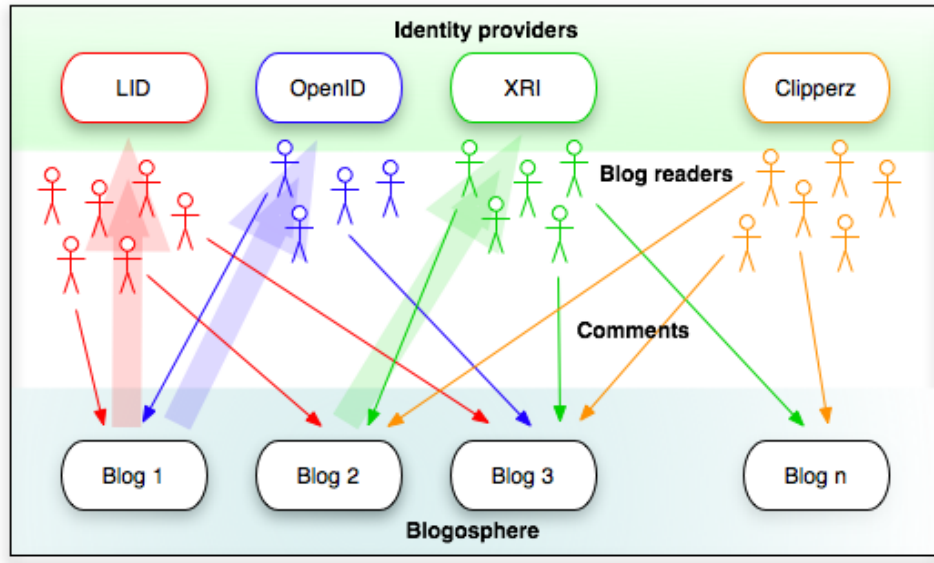


Figure 2: Single Sign-on scenario (image by <http://www.clipperz.com>)

## 2 Case Studies

In order to understand the success recipes of the most popular web 2.0 applications in use today, we list and observe the basic and common functionalities of some successful applications.

The most popular are YouTube, Wikipedia, MySpace, Facebook, LinkedIn, Badoo, Music.com, Last.fm, just to name a few. We examine in depth the navigation flow of a couple of them:

### 2.1 Photos: Flickr

Flickr is one of, if not the best, site for sharing photos online. With a nice, clean layout and easy-to-use tools, the users can upload massive amounts of pictures and label them. Using tags, they can search for photos of specific content. Flickr makes extensive use of AJAX-technology. Flickr also allows for a contacts list to keep track of friends and also provides an internal mail box to exchange messages.

Website: <http://www.flickr.com>



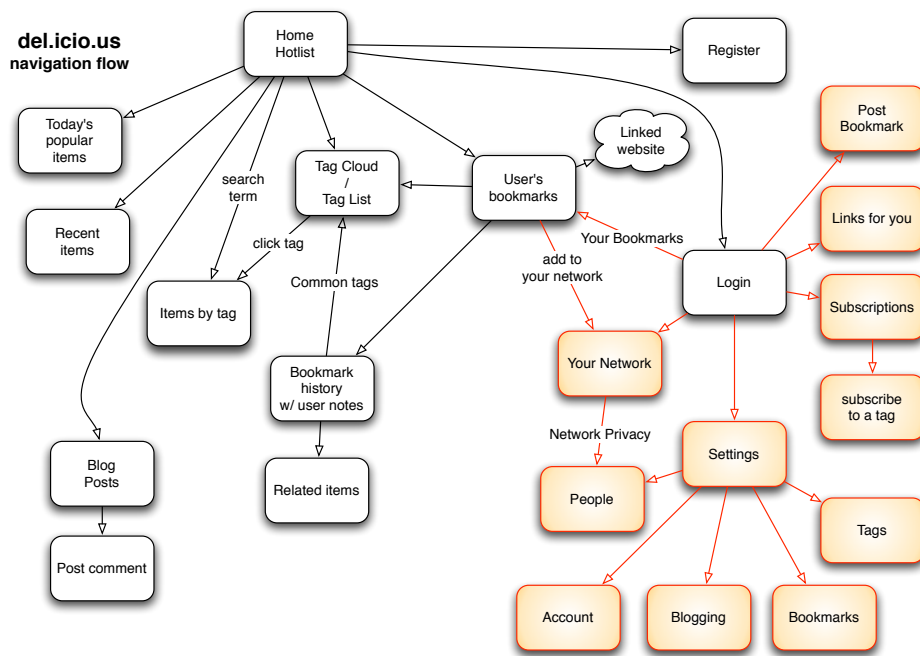


Figure 4: Del.icio.us navigation flow

### 3 Analysis

After a wide analysis of the most popular web 2.0 application, we can observe and extract the most useful and important functionalities:

- user authentication (sign-up and sign-in)
- social networking (contact list, messaging)
- resources management (upload and manage any kind of content type)
- relations (tags, ratings, comments)

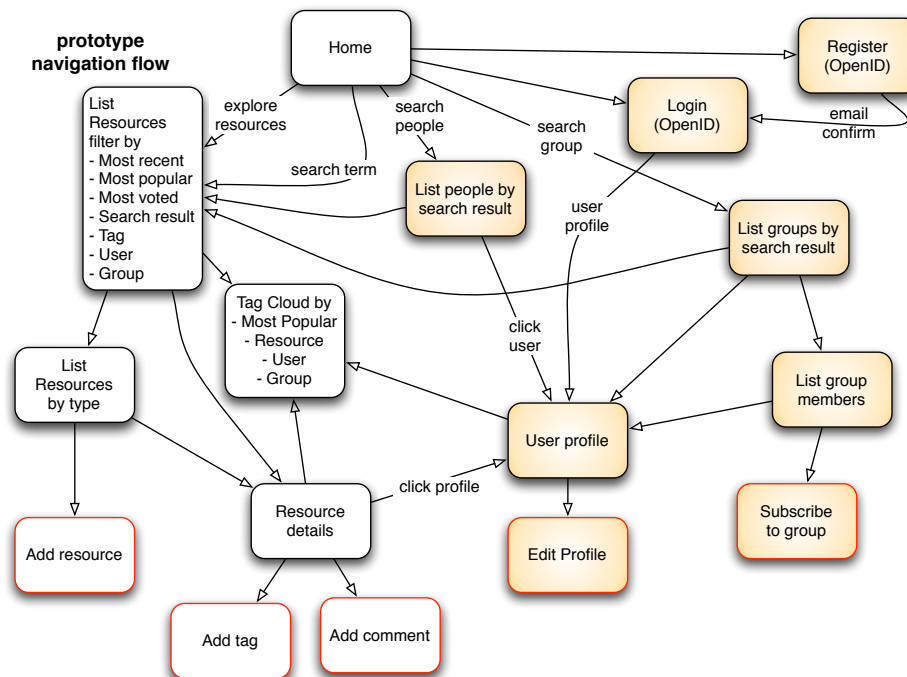


Figure 5: Prototype navigation flow

Collecting and resuming all the functionalities provided by web 2.0 applications, we can distinguish two main categories: social networking and resource management, which are respectively actors and data.

We focus on the fundamental interaction functionalities to provide a RubyOnRails web 2.0 application plugin generator, social\_platform, which

has the basic structure of the common account management and resource management of web 2.0 applications.

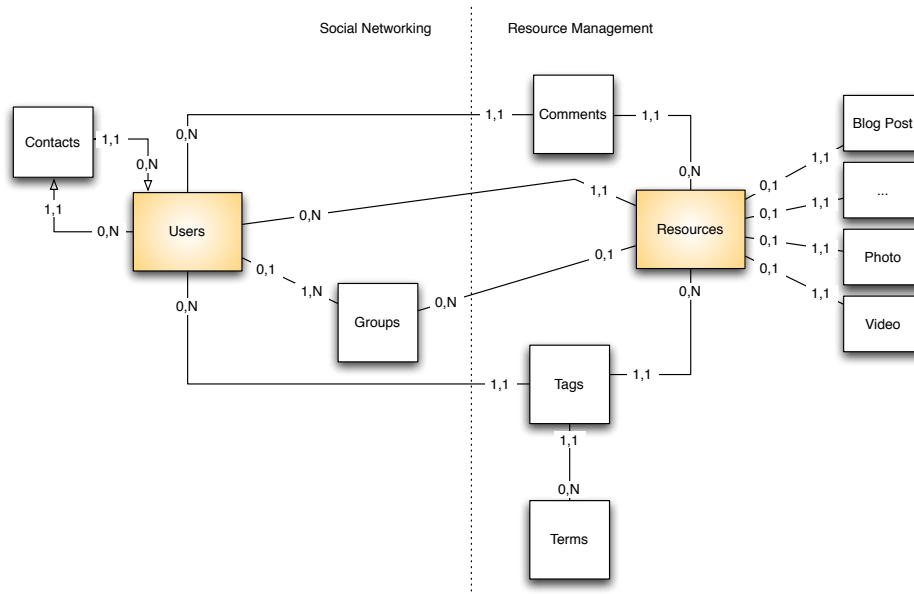


Figure 6: Entity Relationship Diagram

### 3.1 Social networking

Since the contribution of content is the crucial activity provided by users, web applications have to support account management, in order to transform passive visitors into active contributors.

By providing an authentication system, a normal visitor can signup and become a contributor of a given site. He can act as author and editor, producing his own content, managing it, tagging it and of course also tagging and commenting third persons' content (in private or public visibility).

In a social network, the common activities available are Register to Community, Meet new people, Connect with friends and Message Box. Moreover the users can organize themselves into groups, sharing data among the members and setting permissions.

Joining a community, a user can fill up his own profile, which summarizes among other things, personal and professional accomplishments. The profile helps to introduce the user to the community and allows for former

colleagues, friends, clients, and partners to find him.

### 3.2 Resource management

The users' contributions have to be organized into many different structured forms, depending on the kind of content. There are many types of data, from text posts to multimedia objects. Regardless of content details, all those types have common properties that can be summarized at a general level, such as an owner (author user), date of creation, tags and comments.

All kinds of resources have commonalities; these can be extracted and allow each kind of resource to be managed independently of content details.

The users define many different types of attributes to the resources, which form the basis for relations. The connecting edges between entities describe the relationship between them. Apart from having its own attributes, an entity also has a variety of associations with other entities. A resource cannot exist by itself, in fact it has to be created so that it belongs to an author that represents the first ownership relation. An infinite number of different kinds of relations can be established, but in every case there are commonalities that can be focused on, at least to identify the related entities.

For instance, as relation, the tagging system is a way to label content with short and precise terms, very useful to create collection of resources tagged with common terms. In the same way, any type of content can be commented by users. Tags and comments have different type of content, however in both cases they can be treated as relations.

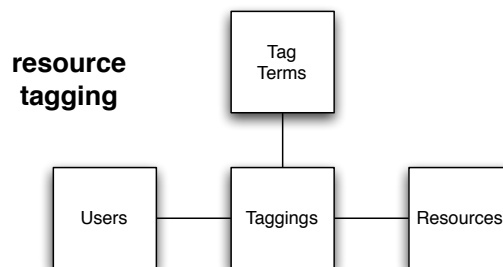


Figure 7: Resource Tagging

We focus on the common steps and functionalities related to any type of resource as Publishing, Sharing with different level of permissions, Rating (ranking), Tagging, Commenting and Hits statistics.

### 3.3 Interoperability technologies in Web Applications

As part of our analysis we take the crucial step of exploring existent available technologies and considering the possible benefits they can bring to our application.

#### 3.3.1 OpenID

A real life example of interoperability is the OpenID protocol of decentralized single sign-on, which provides to client applications the ability to exploit remote user profiles stored in external servers. We are looking to integrate the single sign-on functionalities into our plugin.

In order to gain experience in this field, we tried the integration of OpenID as a consumer into the prototype, figures 8 and 9 show how it looks like.

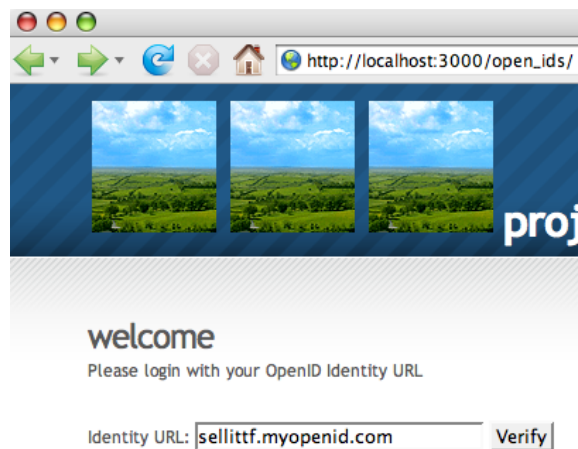


Figure 8: OpenID login form as embedded client plugin

The OpenID protocol is a very good solution, however it forces us to use external OpenID servers or integrate the server itself on our plugin framework. We decide to implement our own solution to provides single sign-on, exploiting the basic authentication system and extending it with the concept of ID-Tuple.

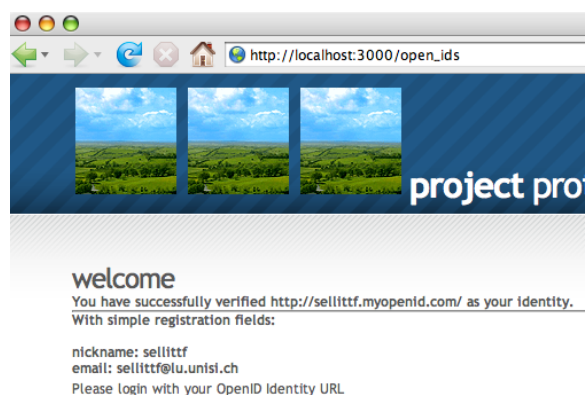


Figure 9: OpenID success response

### 3.3.2 SOAP

SOAP is a protocol for exchanging XML-based messages over computer networks, normally using HTTP/HTTPS. SOAP forms the foundation layer of the Web services stack, providing a basic messaging framework that more abstract layers can build on. There are several different types of messaging patterns in SOAP, but by far the most common is the Remote Procedure Call (RPC) pattern, in which one network node (the client) sends a request message to another node (the server), and the server immediately sends a response message to the client. SOAP is the successor of XML-RPC (<http://en.wikipedia.org/wiki/SOAP>).

### 3.3.3 Semantic Web - RDF and ActiveRDF

The Semantic Web = a Web with meaning.

The word semantic stands for the meaning of. The semantic of something is the meaning of something.

“If HTML and the Web made all the online documents look like one huge book, RDF, schema, and inference languages will make all the data in the world look like one huge database” (Tim Berners-Lee, *Weaving the Web*, 1999)

RDF, the Resource Description Framework, is designed to describe store graphs of interconnected information. Such a graph is made out of triples, and every triple has a subject, predicate and object. A triple is like a simple sentence in English and might for example express information such as: “Peter knows Paula” (<http://wiki.activerdf.org>).

With the purpose of exploiting the semantic web on Rails, the ActiveRDF library is available for accessing RDF data from Ruby programs. It can be used as data layer in RubyOnRails, similar to ActiveRecord (which provides an O/R mapping to relational databases) (<http://www.activerdf.org>).

Another library for RubyOnRails is RDFa. It provides a set of helper methods used to publish RDFa data. No modification of model's and controller's code is required, in fact RDFa is embedded directly on the views' code in order to provide machine semantic data together with human readable data. RDFa On Rails is developed and maintained by Cédric Mesnage (<http://rdfa.rubyforge.org>).

### 3.3.4 REST - Representational State Transfer

REST is more of an old philosophy than a new technology. Whereas SOAP looks to jump-start the next phase of Internet development with a host of new specifications, the REST philosophy espouses that the existing principles and protocols of the Web are enough to create robust Web services. This means that developers who understand HTTP and XML can start building Web services right away, without needing any toolkits beyond what they normally use for Internet application development (by Amit Asaravala -<http://www.devx.com/DevX/Article/8155>).

The Web is comprised of resources, which are any kind of item of interest. Other than being human readable, a resource can also be fetched by machines using server/client communication protocols. REST is an XML messaging format interface based on HTTP protocol, a valid alternative to the SOAP XML-based protocol. REST is not a standard, there are no W3C REST specification, but it is an architectural style for writing networked applications where the communications protocol consists of a very small set of verbs applied to a very large set of nouns. The response format can have many different representations (html, xml, etc...) of a required resource.

REST is the request and manipulation of web-accessible resources through the use of standard HTTP request methods (GET, POST, PUT, DELETE). REST is based on the notion that the individual HTTP methods represent the type of action to take (the verb) and the URI (Uniform Resource Identifier) is the location of the resource to act upon (the noun). Using this paradigm, the same URL can be used to perform different actions.

### 3.3.5 Mashup

A mashup is a website or application that combines content from more than one source into an integrated experience. Since more and more applications on the Net have interoperability functionalities, we are able to generate content as result of mixing and filtering other contents available on external servers.

## 3.4 Analysis conclusions

Summarizing all the information we collected, we can distinguish between two distinct fields of study among the interdisciplinary computer science branches of knowledge.

The first one is based on social networking structures, where we can extract the most important elements needed to construct a platform operating with people, resources and relations between them. Every social networking application has to consider this fundamental abstract aspect before implementing customized models on top of it. As the least common denominator, independently of the content, we take care of the container entities which must be present, and in the way they have to interact with each other.

The second field is focusing on the available network technologies that permit the communication and data exchange between different machines running different applications. More precisely we look at the techniques which exploit the available protocols and we give the opportunity to construct APIs over it. We pay particular attention to the REST architecture style, which takes advantage of the already consolidated HTTP protocol and XML data structure.

Moreover, the two aspects have to be combined in a way such that we can provide a solution to generate distributed and extendible web applications. The initial step is how to quickly generate a social networking web application, and then transform it into an interoperable application, working together with disseminated web applications.

## 4 Design

The platform has to be implemented in the RubyOnRails framework, extracting the basic structure in a plugin. The big motivation that drives this decision is to deliver to the Rails developers community a set of models and functions frequently used to implement social networking websites. In the DRY (Don't Repeat Yourself) code philosophy, a goal is to provide to the user developers a system that allows the reuse of our code as starting point of their projects and generate detailed resources and relations extending the generic code of the plugin.

### 4.1 Model-View-Controller

The RubyOnRails is a Model-view-controller framework. MVC is an architectural pattern used to separate data (model) and user interface (view) concerns, so that changes to the user interface do not affect the data handling, the data can then be reorganized without changing the user interface. The model-view-controller solves this problem by decoupling data access and business logic from data presentation and user interaction, by introducing an intermediate component: the controller.

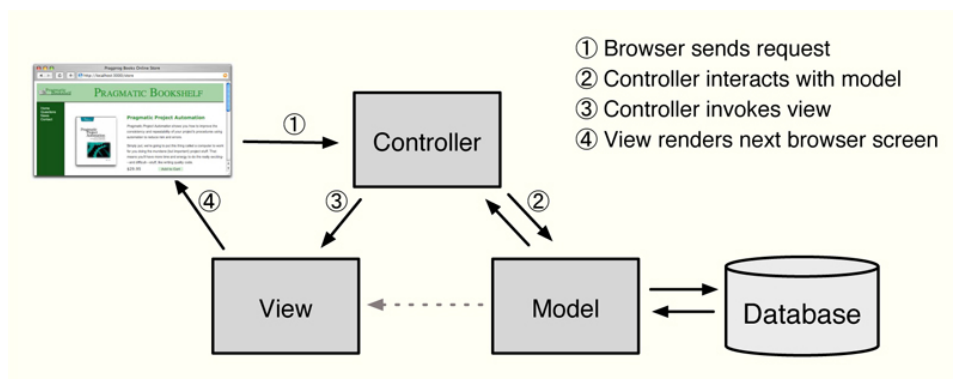


Figure 10: The Model-View-Controller Architecture

Source Book: Agile Web Development with Rails

### 4.2 Gems, Plugins and Engines

RubyGems is a package manager for the Ruby programming language that provides a standard format for distributing Ruby programs and libraries (in a self-contained format called “gem”), a tool designed to easily manage the

installation of gems, and a server for distributing them. The packages are available to all rails project running in the same machine.

To use a library in a dedicated rails project, and in order to step forward the development of a new web 2.0 application, we intend to deliver a Ruby-OnRails plugin able to provide all the common and generic functionalities of a social networking and resource management system.

The Rails plugins can run together with the application implementing library code that can be invoked from the application. Another use of a plugin is as generator, which provides a scaffold code that will be copied directly into the application folder. This second case is useful to have quickly generated code that can be modified by the user developer, however in case of upgrades of the plugin, it should not be possible to update the already generated code.

There is a special Engines plugin that give the possibility to implement a Model View Controller pattern application inside a plugin. We intend to use this possibility to work on a Social\_Platform plugin, in order to provide a basic functionalities platform that the user developer can take advantage of and eventually extend.

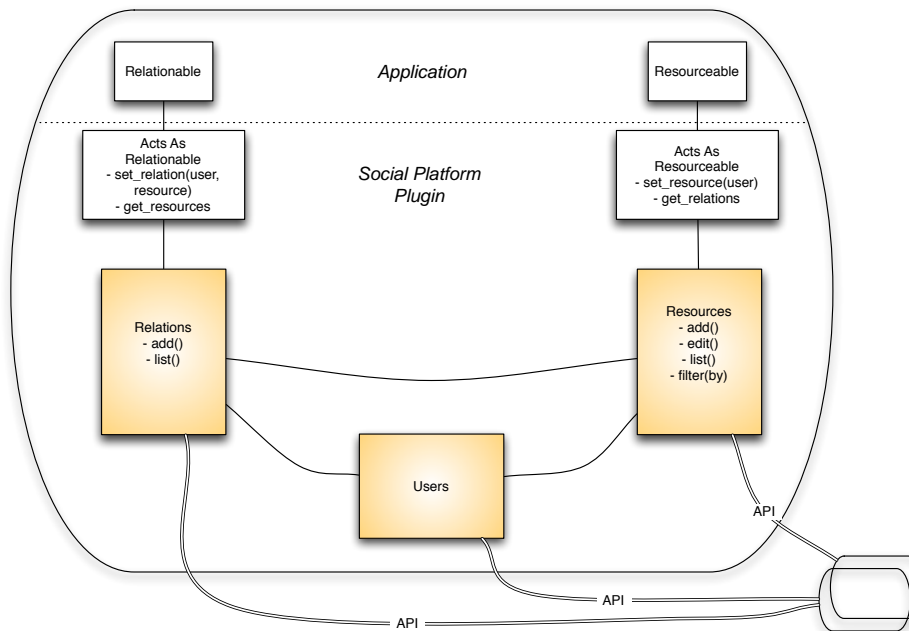


Figure 11: Social Platform Plugin

### 4.3 Inheritance

The core feature of the RubyOnRails framework is the ActiveRecord class, which provides the very powerful ORM (object relational mapping) between the Ruby Models and the SQL-records on the database. Although Ruby is an Object Oriented programming language, the ActiveRecord gives some constrain and exhibits a big lack in the way the models can actually inherit from each other. Apart from Single Table Inheritance, the main limit given by the mapping to the database for example, is the fact that there is no clean way to define a Cat object that inherits from an Animal object. The problem will exhibit when we want to store an instance of Cat including the attributes provided by the superclass Animal.

Relational databases don't support inheritance, so the approach suggested by the Rails framework is the STI (Single Table Inheritance), but it's not suited to our purpose because it forces us to map all fields of all classes of an inheritance structure into a single table, ending up with a unique big SQL table including all columns used for the different types of subclasses. We consider this solution a bad workaround, and since we don't know what kind of extensions the user developer will implement, we discard this approach because doesn't match the inheritance that we are looking for. Link: <http://wiki.rubyonrails.org/rails/pages/SingleTableInheritance>

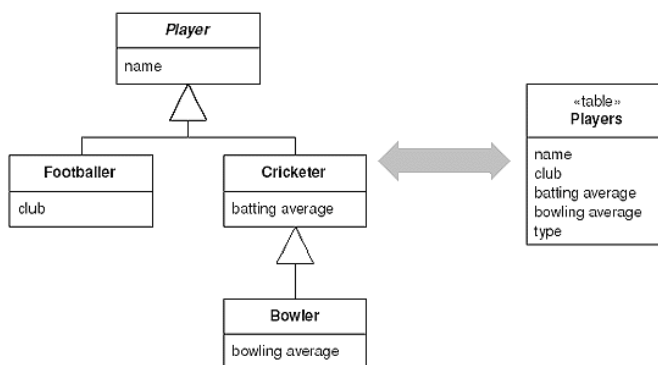


Figure 12: Single Table Inheritance

We were considering reviewing this aspect for some time because it is a crucial design decision. We were also focusing on using the patterns **Class Table Inheritance** and **Concrete Table Inheritance**, however, currently they are not supported by the RubyOnRails framework.

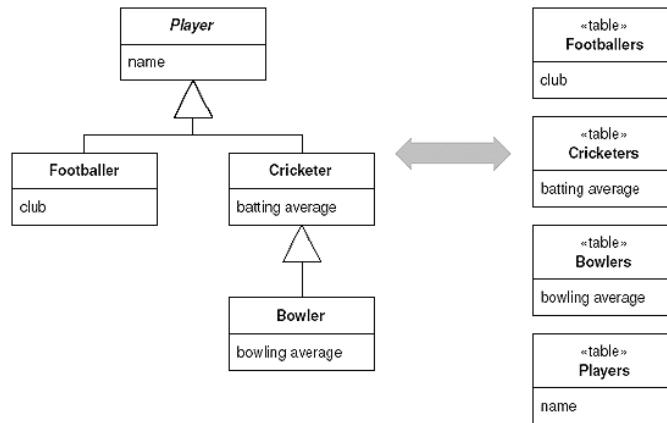


Figure 13: Class Table Inheritance

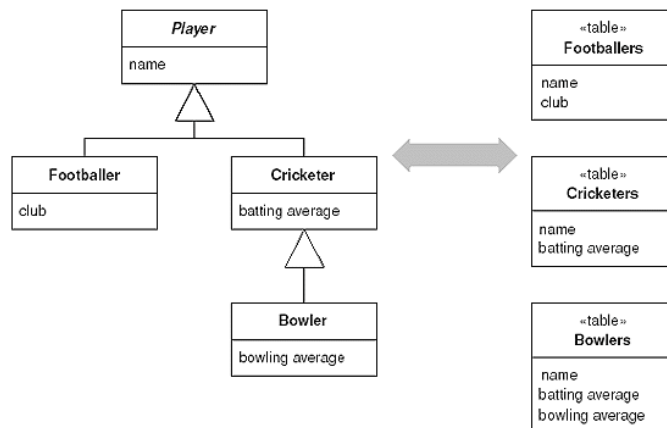


Figure 14: Concrete Table Inheritance

### 4.3.1 Polymorphism

A solution we found to workaround this issue is Polymorphism. It is a way to define a Model that has association with other Models of undefined type, so that both Cat and Dog have a common alias as AnimalType and associated to Animal (actually it is the Animal belonging to the AnimalType). Understanding Polymorphic Associations, link:

<http://wiki.rubyonrails.org/rails/pages/UnderstandingPolymorphicAssociations>

To implement our prototype we use polymorphic associations to define a Resource model that has association with more detailed models. This method solves the problem of defining the relationship between the User model and many different resource types, even those that are not yet present on the platform but may be provided in the future by the user developers.

Figure 15 shows how we can focus on the main entities such as People, Resources and Relations. All classes are extended though polymorphic association, labeling them with abstract names, respectively Resourceable and Relationable.

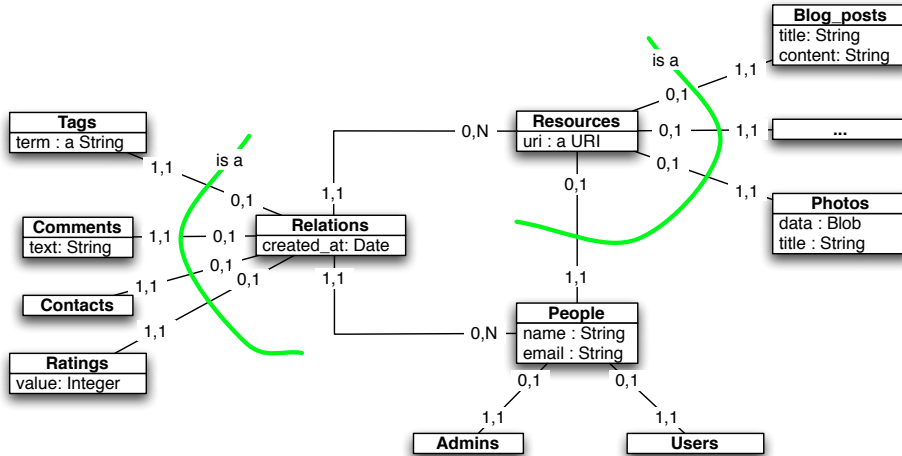


Figure 15: Resource, Relations and Polymorphic Association

In order to benefit from additional methods, commonly used by at least a couple of models, we use the **Mixins** in order to emulate the methods inheritance (more specifically the multiple inheritance). To reach this goal, we define modules to include into the desired models.

### 4.3.2 Object Prevalence

A prevalent system makes use of serialization, and is useful only when an in-memory data set is feasible, in fact everything is kept in RAM. If the application data fits in memory, we could use object prevalence to save snapshots of our object model on disk. A serialized snapshot of a working system can be taken at regular intervals as a first-line storage mechanism. Serializable command objects manage all the necessary changes required for persistence. Fault-tolerance and data consistency are provided by the use of command objects to perform all transactions that change the state of the data. All commands are stored using serialization.

Madeleine is a Ruby implementation of Object Prevalence (available as gem).

## 4.4 Interoperability

The interoperability between servers (where each server also acts as client) is fundamental to connecting and managing a large quantity of resources. In order to separate concerns and define a structured way to exchange data, the API and data format are very relevant to cope with interoperability.

We are looking to manage the interoperability of content resources, so that objects stored in domain A can be fetched from domain B from a user registered in domain C.

### 4.4.1 REST and ActiveRecord

Starting from version 1.2.2, Rails is providing a REST generator using the `scaffold_resource` command (similar to the CRUD generator with the `scaffold` command). The Rails routing system give support to interface with the RESTful invocations. There are 7 basic invocations using 4 different HTTP methods (GET, POST, PUT and DELETE). For example if we want have the details of an object, we call with the GET method `"/objects/:id"`, that will return the HTML format of the method `show` of the `objects controller`. In the same way we can just ask for the xml format calling `"/objects/:id.xml"`. The same kind of style is used to obtain list of objects, update a given object or delete it.

Moreover we can define nested RESTful resources, so that if an object contains a collection of other resources (for example items), we can look up the items of the a given object with the path `"/objects/:id/items"`.

ActiveResource is promoted and developed by Rail's creator, it provides a large piece of the REST puzzle by basically implementing the client side of

a RESTful system the parts of a decoupled system that consume RESTful services. At its essence, ActiveRecord provides a way to utilize model objects as REST-based client proxies to remote services. It merely uses the full power of REST and Rails ability to serve different representations of the same resource according to the mime type specified in the accept header to perform some basic proxying over HTTP. Using it we can model objects that are stored in an external repository and we can run an application as client to fetch, create, update and delete resource remotely. We can exploit it to implement interoperable web application as server/client (and vice-versa).

In the current version Rails 1.2.3 ActiveRecord is not yet an official part of the framework, but we can download the trunk version from the Rails developers repository in order to start working with it. This are some of the features offered by ActiveRecord:

- Model objects become REST-based client proxies to remote services
- Maps CRUD to REST using HTTP commands GET, POST, PUT, and DELETE

Figures 16 and 17 illustrate the difference between querying directly through ActiveRecord and querying through ActiveRecord client.

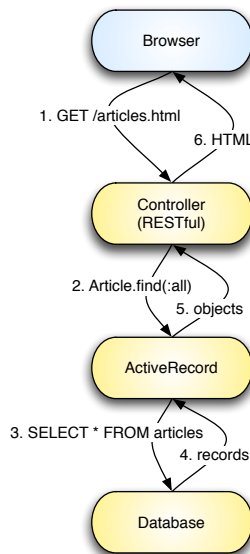


Figure 16: Querying through ActiveRecord

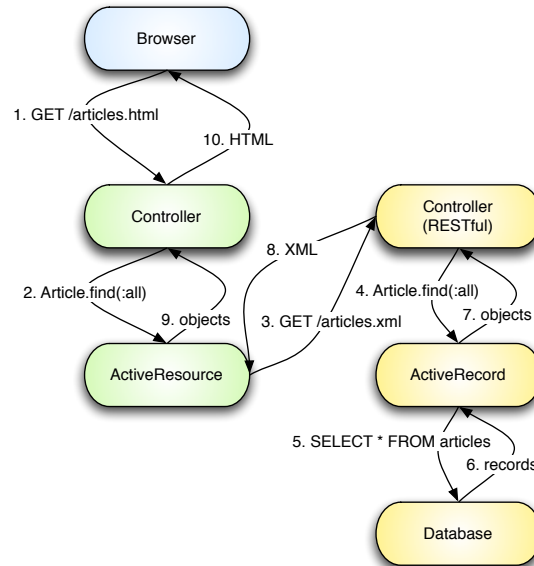


Figure 17: Querying through ActiveResource

## 4.5 Design conclusions

Starting with the prototype implementation we had to cope with many issues related to RubyOnRails, in fact we discovered many advantages and disadvantages of the framework. The first big problem to solve is related to the object oriented inheritance, since the ActiveRecord class doesn't provide a transparent mapping to database of inherited models, we worked a lot to find a valid alternative and currently we emulate an object oriented inheritance implementing library extensions together with polymorphic association.

A second aspect is related with the plugin implementation, that has to act as a wrapped and reusable model-view-controller application. We reached our goal using the Engines plugin.

We discovered a lot of useful features to include to our platform such as the ActiveResource to act as REST client, but it is not yet included in the stable Rails release, so we are trying to work on the Rails Edge to anticipate the evolution. However this choice exhibits drawbacks because there are many important changes to the release and we have to change a lot of code already implemented in order to make it work on edge.

## 5 Implementation

As agile development process, the plugin prototype was initially implemented as a plugin using the additional Engines plugin functionalities, that give a smart way to design a MVC structure application in a separate plugin; after that we restructured the plugin in a lighter solution, avoiding the use of the additional Engines plugin.

The plugin provides the Resource and Relation models, which with polymorphism are associated with Resourceable and Relationable models, which are implemented outside the plugin by the user developer. In order to reuse common code and since there is no real inheritance, we provide in the plugin two libraries: `acts_as_resourceable` and `acts_as_relationable` that are included by the outside models. So, finally the only thing that the user developer has to do is to implement his own type of resource and to set the model as `acts_as_resourceable`.

To make each resource a searchable content, we use Ferret as indexer on creation and update of any resourceable object. Querying for a term, the system search on the index for the relevant documents. As opposed to exact matching, this technique is about information retrieval based on relevance.

The plugin is delivered already with some common relational models, such as Rating, Tag and Comment.

### 5.1 Webnode

We introduce the entity of a Webnode to identify a webhost serving the distributed application. Basically the application can work as a single host (like normal websites work), in addition we improve the visibility of data by extending the local database to a distributed database through webnodes.

The webnodes can be added automatically to each single application, but can also be discovered. The discovering occurs when a resource fetched from a known webnode belongs to a user stored in an unknown webnode (that now became known).

The distributed system is implemented by extensively exploiting the remote relation (through ID-Tuple).

We consider a scenario where Request A is submitted by the end user to perform a remote data collection. As shown in figure 18, Webnode A is directly invoked by the human user, then in order to collect all the resources on the network, Webnode A invokes all known webnodes through the `ActiveResource` support. Each webnode receives a REST request, performs the query on the local database and returns the results as XML format

to Webnode A. At this point the objects are reconstructed on the fly and merged with the objects retrieved in the local database. Finally, human readable output in HTML format is generated treating the objects as they are, independent of whether they are stored locally or remotely.

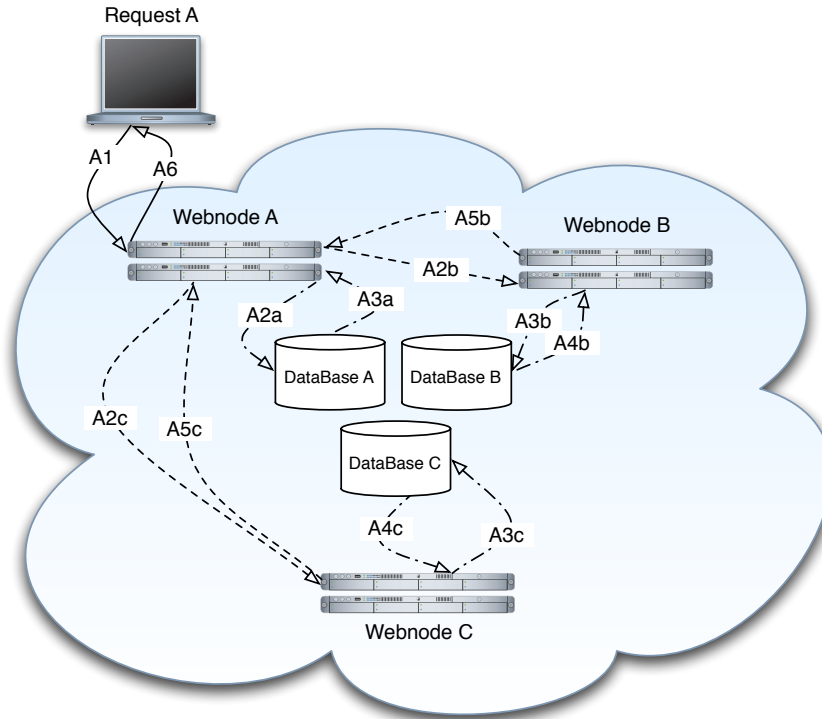


Figure 18: Remote data collection

As a second scenario we consider Request B, shown in figure 19. The user invokes an object from Webnode B, which is stored in the local database, not a big deal. However, the resource's author is stored on another webnode, Webnode B then launched a REST request to Webnode D, which in turn retrieves the author details from its database and returns the user object in XML format to Webnode B. As before, the data is reconstructed and associated to finally render it to the user.

All the operations of interoperability are done behind the scenes, as a result, the human users perceive the webnode that they are navigating as a huge repository, although there may be no real object stored on it.

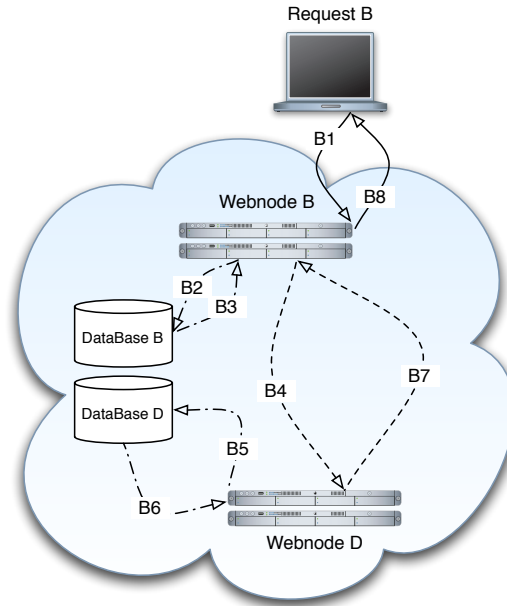


Figure 19: Remote data collection

## 5.2 Database schema

After a long analysis on database design patterns to meet our purposes of polymorphism and distributed resources, we structured the database as figure 20 shows.

### 5.2.1 ID-Tuple

We introduce the use of ID-Tuple, a combination of webnode-ID and reference-ID, in order to manage data on distributed storage. An ordinary relational database management system keeps records in relation through the record ID. As illustrated in figure 21, we experimented with the use of ID-Tuple, which allows us to refer to data relations stored on remote records. Assuming reliability, there is no data replication among the distributed database, in fact the records are remotely related on the fly.

## 5.3 SSO Single Sign-On

A useful and important feature of the system is to permit a user to create and manage resources and data from external webnodes than where

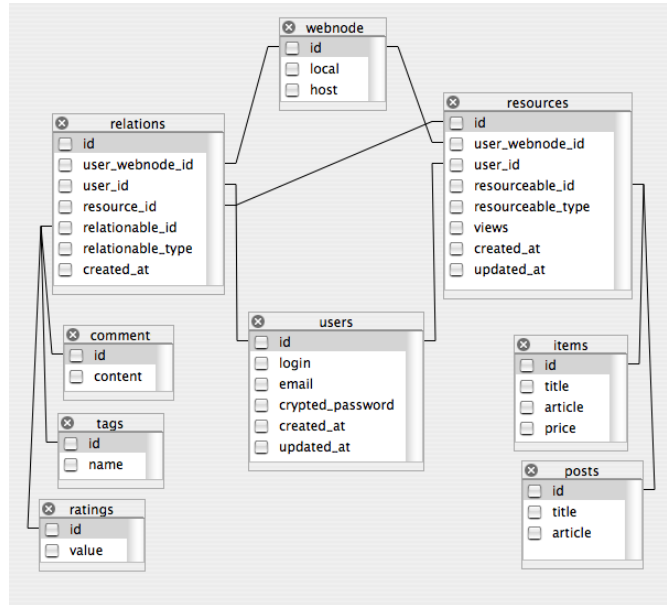


Figure 20: Database schema

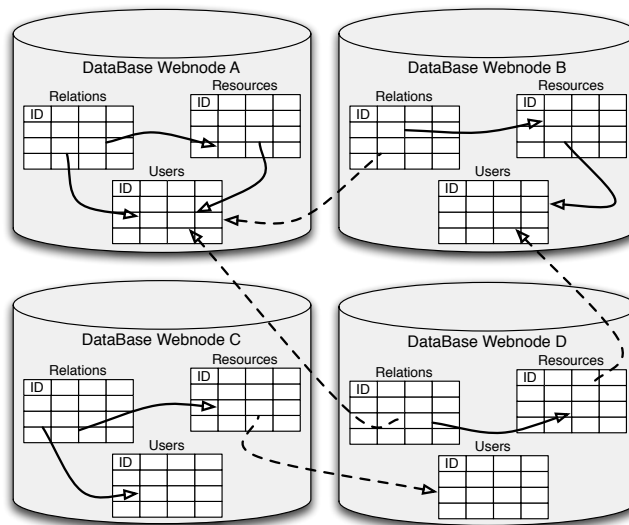


Figure 21: Distributed Database using ID-Tuple

he is registered. The account management is implemented adapting the `acts_as_authenticated` plugin, integrating it completely into the our `social_platform` framework plugin. With our authentication system, the user creates an account on a webnode and is able to operate on data on remote webnode.

In the first scenario the user logs in on the webnode where is registered, then he can create new resources and create relations on the same local webnode.

As second scenario he can create relations on remote resources, that are stored on remote webnodes keeping track of the author.

As third scenario he leaves the local webnode and enters directly on a remote webnode, then he can log into the remote webnode using the same original authentication. To do that, he has just to chose the original webnode on the special form, without having to submit any password, the system performs a quick double redirect to check and collect the authenticated user. At this point the user is able to create new resource and store it completely on thirds part webnodes.

The figure 22 illustrates how the user authenticates on remote webnodes. The step 4 and 5 are done without the interaction of the user, so that it seems that are done behind the scenes as direct communication between the webnodes, however through the user browser to keep track of the user session.

## 5.4 Search Engines

The searching system works as a distributed information retrieval system, in fact it is able to retrieve relevant resources from a distributed collection of data. Every webnode keeps the local resources indexed by implementing a helpful `acts_as_ferret` plugin, the client then submits a search query to every webnode and collects all responses.

## 5.5 Rendering

At the eyes of regular users, each website distinguishes itself from others first of all on the graphic layout. A resource is a structured information set, which can be illustrated in a wide variety of designs. Figure 23 shows how the same resource is exposed to human users in different dresses, depending from which webnode the request is submitted.

Since our framework give precedence responsibility to the frameworks controller, avoiding the invocation to the customized controllers, also the customized views have to be called through the framework. To achieve this

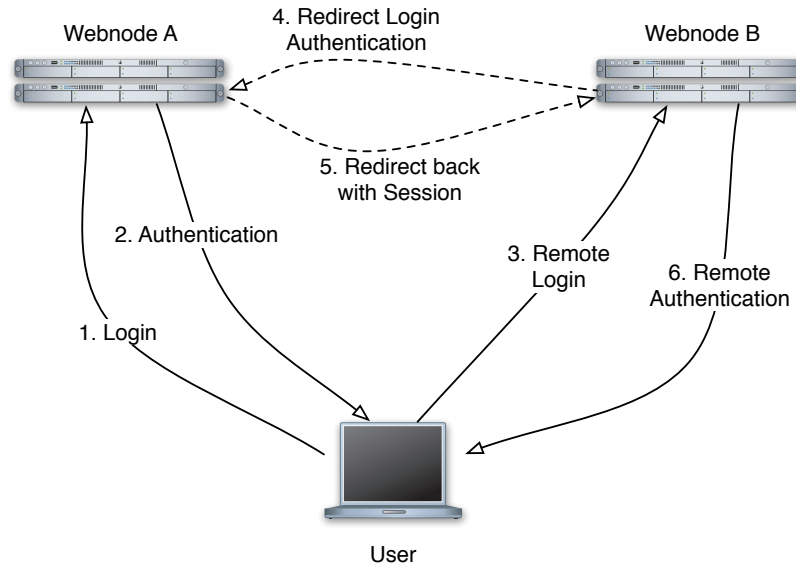


Figure 22: Social Platform Single Sign-On

purpose, we forward dynamically the requests to the extending controllers, in order to generate the human readable views. This task makes extensive use of the partial rendering system.

Looking at the details of a resource (figure 24), we can forward the invoked show method to the Resourceable type, in order to obtain the partial view with the detail content.

## 5.6 Caching

In order to improve the speed of fetching data, and assuming that data is not frequently updated in realtime, we introduce a cache system that stores in local sessions the data fetched from external webnodes. The cached data is automatically invalidated when an object is locally edited. To keep the data consistent and updated, the cached result should be periodically invalidated in order to fetch it again. In addition, the cache system can be also switched off.

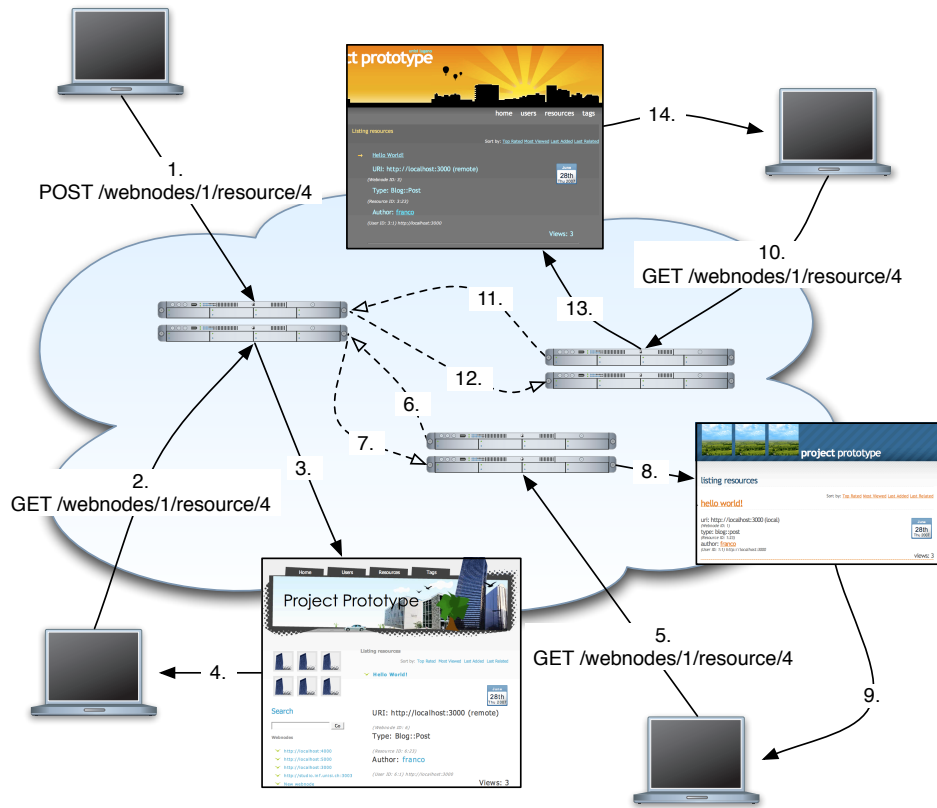
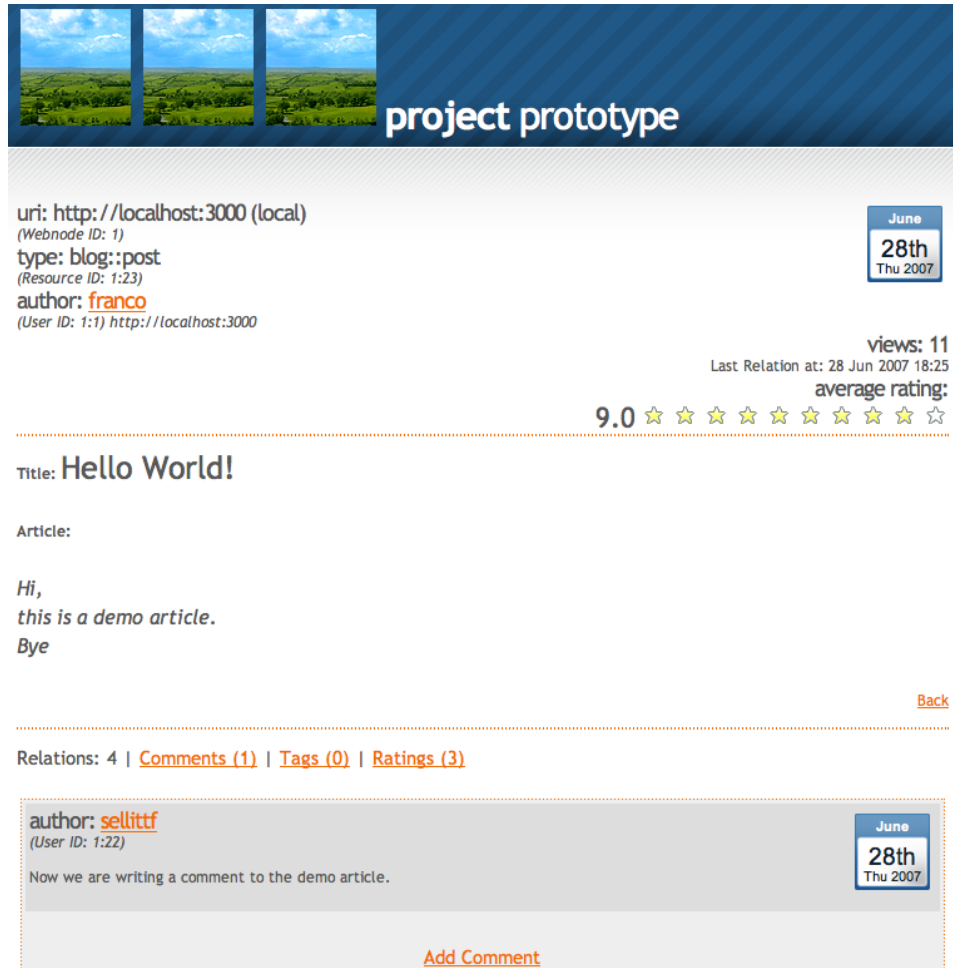


Figure 23: Data rendering



The screenshot shows a web interface for a resource titled "project prototype". At the top, there are three landscape images and the text "project prototype". Below this, the resource details are listed: "uri: http://localhost:3000 (local)", "type: blog::post", and "author: franco". A calendar widget shows "June 28th Thu 2007". Statistics include "views: 11", "Last Relation at: 28 Jun 2007 18:25", and an "average rating: 9.0" with ten star icons. The article title is "Hello World!". The article content reads: "Hi, this is a demo article. Bye". A "Back" link is present. Below the article, it says "Relations: 4 | Comments (1) | Tags (0) | Ratings (3)". A comment form is shown with the author "sellittf" and the text "Now we are writing a comment to the demo article.", with an "Add Comment" button.

uri: <http://localhost:3000> (local)  
(Webnode ID: 1)  
type: blog::post  
(Resource ID: 1:23)  
author: [franco](#)  
(User ID: 1:1) <http://localhost:3000>

views: 11  
Last Relation at: 28 Jun 2007 18:25  
average rating:  
9.0 ★★★★★★★★★★

Title: **Hello World!**

Article:

*Hi,  
this is a demo article.  
Bye*

[Back](#)

Relations: 4 | [Comments \(1\)](#) | [Tags \(0\)](#) | [Ratings \(3\)](#)

author: [sellittf](#)  
(User ID: 1:22)  
Now we are writing a comment to the demo article.

[Add Comment](#)

Figure 24: Resource show with details partial

### **5.7 Performance**

All `ActiveResource` invocations use the `Net::HTTP` class. The mean delay time of a request is 1 second, depending on the workload of the webservers. Launching the requests in synchronous mode, every query to a webnode has to wait for the relative answer before going on and querying the next webnode. In order to optimize the performance of requests on several servers, a suggested improvement is to query the webnodes in asynchronous mode, so that the requests are launched in a multithreaded fashion, letting every webnode perform answers concurrently.

## 6 Application Prototype

The goal of the prototype is to show the benefits of developing a new social web application using the plugin as base platform and interoperability interface. The integrated system acts as a distributed web application able to collect and manage resource and relational data present in remote servers. Like a peer-to-peer network, the webnodes interoperate by exchanging data in machine understandable formats.

### 6.1 How to install

1. Create a new Rails application (`rails myapp`)
2. Install the plugin **social\_platform** in `vendor/plugins`
3. Install the plugin **acts\_as\_ferret** in `vendor/plugins`

4. Run Rails version  $\geq 1.2.3$

Link the project to the Rails repository:

```
$ svn propset svn:externals  
"rails http://dev.rubyonrails.org/svn/rails/trunk" vendor  
$ svn up vendor
```

or checkout the last version on edge:

```
$ svn co http://dev.rubyonrails.org/svn/rails/trunk vendor/rails
```

or set the last Rails gem version to run among the installed gems, define it in `config/environment.rb`: `RAILS_GEM_VERSION = '1.2.3'`

5. The prototype requires that all webnodes run with the same set and version of model resource types.
6. Define the resource types in `config/environment.rb`

```
Available_resourceable_types = ["Blog::Post", "Shop::Car"]
```

7. Define the relation types in `config/environment.rb`

```
Available_relationable_types = ["Comment", "Rating", "Tag"]
```

8. Optional: if running many environments on the same machine, define the multiple session names in `config/environment.rb`

```

config.action_controller.session = {
  :session_key => "_PrototypeDistributed_session_#{ENV['RAILS_ENV']}",
  ...
}

```

9. The new Rails Edge now saves the entire session in a hashed cookie. The cookie-based sessions seem to be faster at retrieving and processing than hitting the file-system on every request. The data is stored in cookies of the client browser, preventing the cumulation of data on the server. However we noticed that using cookies, the session data gets huge and cookie overflow errors occur. We then decided to use the old Rails data storing default: PStore. Set into the initializer block in `config/environment.rb`:

```

config.action_controller.session_store = :PStore
config.action_controller.session = {
  :tmpdir      => "#{RAILS_ROOT}/tmp/sessions/"
  ...
}

```

This solution also uses cookies, but just as reference key of session data stored at server side instead of at client side. The drawback of storing sessions on the filesystem is the increasing number of sessions day over day. This is a common problem, solvable by implementing a garbage collector and running it as a cron job.

10. Copy all migrations files from `vendor/plugins/social_platform/db/migrate/` directory to `db/migrate/` directory.
11. Check the settings of `config/database.yml`
12. run `rake db:migrate` in order to create all tables in the database (the system has been tested with MySQL and SQLite3)
13. Copy the routing settings from `vendor/plugins/social_platform/config/routes.rb` directory to `config/routes.rb` directory.
14. In the main layout, define a div tag as `main_body_content`, which is the container of the content pages:

```

<div id="main_body_content">
  <%= yield %>
</div>

```

## 6.2 How to start

1. Launch the webserver on each webnode: `$ ./script/server`
2. Optional: launch the application on the same machine with different environments, and relative console:

```
$ ./script/server -p 3000 --environment=development3000
$ ./script/server -p 4000 --environment=development4000
$ ./script/server -p 5000 --environment=development5000
$ ./script/console development3000
```

3. For each webnode, define the desired webnodes to connect to as peer-to-peer by opening the browser at URL `http://localhost:3000/webnodes/new`

### 6.2.1 Required Plugins

`acts_as_ferret` In order to keep the system content indexed, we use the functionalities of the `acts_as_ferret` plugin

## 6.3 Filtered and sorted lists

The list of resources can be filtered according to many criteria:

- by Webnode all the resources on the given webnode are collected, the request can be done on the local webnode or on a remote webnode.
- by Tag a query is performed on all webnodes by collecting resources with the given tag. It is very useful to list all resources that have similar topic of content.
- by User the system is able to recollect all the resources that a given user has disseminated on the network. It appears to the users that they have a distributed repository.
- by Search result using the search box, the resources can be searched by a given term. Exploiting the `acts_as_ferret` plugin, all the resources are kept in an index file in order to perform a high quality information retrieval based on document relevance.

Except of the filter by the Webnode, the queries are performed on all webnodes connected to the network, all remote data is then collected together

and shown to the client webnode.

Once a list is shown to the user, it can be sorted based on some predefined criteria. In order to sort the list based on the same actual scope, the resource listing is kept in session to prevent the overhead of repeating the same query remotely.

- Top Rated Sort in descending order on the average rating. Every resource has a particular attribute that computes on the fly the average rating through the associated rating resources.
- Most Viewed Sort in descending order based on the number of hits. Every resource has a counter stored on the record that increments at every visualization of the resource details.
- Last Added Sort the resources by date of creation and show the newest at the top.
- Last Related Sort the resources based on the date of the last relation associated.

#### 6.4 How to extend the platform building custom websites

All the plugins and the project itself has given birth to a package that provides all the basic and common functionalities as a base platform to all social websites to build on top of it. After the plugin installation, the user developer just wants to design and build the custom resources and relations taking advantage of our complete platform.

In order to provide new kind of content, the user developer just has to generate a new model, a new controller, a minimal set of views and the migration file. Surprising, instead of generating a common scaffold with repetitive methods and functionalities, the requirements are now very minimal, for instance if the target project is to open a Blog website publishing daily articles, the content model to generate is Article, it is structured as the following steps show, to generate a Resource extension:

- Model Inherits as usual from `ActiveRecord::Base` but with a module extension provided by our plugin in order to have polymorphic association with the main Resource model. In addition, the method `content_resume` which is already provided in the module, can be overwritten in order to retrieve the resuming information of the resource.

```
class Article < ActiveRecord::Base
  acts_as_resourceable
```

```

    def content_resume
      self.title
    end
  end
end

```

Controller It has no particular task to do, in fact all the main activities are done by the framework, so that it inherits from the Resource controller and making life easier for the user developer.

```

class ArticlesController < ResourcesController
end

```

Views There are two mandatory partial views to provide with a custom layout, in fact the framework calls them on creating, editing and showing the resources:

`_form.html.erb`

```

<p>
  <b>Title:</b><br />
  <%= text_field(:resourceable, :title) %>
</p>
<p>
  <b>Content:</b><br />
  <%= text_area(:resourceable, :content) %>
</p>

```

`_show.html.erb`

```

<% article = resourceable %>
<p>
  <b>Title:</b>
  <span style="font-size: 25px;"><%= article.title %></span>
</p>
<p>
  <b>Content:</b>
  <%= article.content %>
</p>

```

The use of the term `resourceable` is mandatory.

Note: `.html.erb` is the new views extension adopted in Rails, however also the old `.rhtml` is compatible.

Migration Since all the generic fields such as author reference and creation date are provided by the Resource model of the framework, an article just has to be described with a couple of fields. The migration file then looks like:

```
class CreateArticles < ActiveRecord::Migration
  def self.up
    create_table :articles do |t|
      t.column :title, :string
      t.column :content, :text
    end
  end

  def self.down
    drop_table :articles
  end
end
```

Helper Finally the helper file:

```
module ArticlesHelper
end
```

To quickly create all sets of files, there are convenient generators that provides the basic creation of files, after which they just have to adapt as illustrated before.

```
$ ./script/generate model Article
$ ./script/generate controller Articles
```

## 6.5 How to customize the plugin views

The social\_platform plugin is implemented as a Model-View-Controller application, so that it has a separate **app** folder with all the models, views, controllers and helpers files. The plugin has extendible capabilities, the user developer for instance may want to change any views adapting it to the target project. In order to extend it, he just has to create the specific view file in the main **app** folder (outside of the plugin). For instance, in order to override a view file, he must create a new folder resources in app/views then create a file show.rhtml in order to have app/views/resources/show.rhtml.

He must then copy the content of the `show.rhtml` provided with the plugin and customize it. At this point the plugin automatically renders the customized view instead of the provided one.

It is strongly recommended to avoid the change of any code directly in the plugin, in order to keep the application consistent after new updates of the plugin.

## 6.6 Issues

### 6.6.1 Webnode not available

The prototype assumes that every server known as webnode is available. For sure this is not always possible and introduces the issue of solving references stored on offline webnodes. If a remote webnode is not online, the client webnode ignores it, so that the remote resources on the given webnode are no reachable.

Another problem exhibits when an available resource is owned by an author stored on an offline webnode. In this case the the resource is shown as normal, but the user details are shown as not available (n/a).

### 6.6.2 Execution expired

If the webserver is executing as a single threaded process, each request has to wait until the previous requests are performed. Due to this behavior, when webnode A performs a request of a resource on webnode B, if the author of the resource resides on the webnode A, a deadlock occurs. This problem can be solved running multi-threaded webserver.

## 7 Conclusions

The solution presented was developed as a bachelor project and lasted 15 weeks, from 12th March to 29th June 2007.

As we planned in our project proposal, we analyzed systematically the interactions within a set of successful web applications in order to extract common functionalities. After extracting these commonalities, we designed the basic framework structure and developed a starter kit for web 2.0 developers.

The **social\_platform** plugin for RubyOnRails focuses on social networking and resource management, providing fundamental functionalities present in such applications. The user web developers can customize and adapt the framework to their needs, generating new content models on top of it.

The interoperability analysis drove us to investigate in depth and to integrate RESTful controllers using the ActiveResource client module. This choice deviates from our initial plan to have a semantic web approach exploiting RDF. However, as we planned, we provided the functionalities of interoperability, able to exchange data structures and operate with local and remote objects.

Finally on top of the service we provide, as first web developers we give a proof of concept application, implementing a community web site ready to use. To prove the interoperability we launch the application in at least three different web servers, in a way that their data and functionalities can be seen as a distributed web application.

The project gave us a challenge to explore and learn the innovative technologies on the web application. We experimented successfully a way to convert an isolated to distributed web application, interoperating and interacting with disseminated data objects.