FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

A Tag Management System for Service- Oriented Environments

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Michael Leibrecht
Matrikelnummer 0025836

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Univ. Prof. Dr. Schahram Dustdar
Mitwirkung: Proj.-Ass. Dr. Daniel Schall

Wien, 29.10.2010

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Abstract

In the past few years social networks and social enhanced media applications have experienced an enormous growth in popularity and gain in users that are registered to the respective platform. People can easily participate in global collaborations focused on content creation and content exchange. In order to have the necessary means for structuring and classification of the data, users often fall back on a technique known as tagging. Tagging allows the user to describe any kind of resource with any keyword, to specify the resources content, its behavior and give a simplified description of its properties.

In comparison to social networks that are gaining more and more momentum, the evolution of Service Oriented Architectures precedes in a slower pace, mainly resulting from the complex requirements to establishing a general applicable work flow, and the economical obstacles that have to be considered when integrating third party services. Apart from these drawbacks there is room for improvement in other aspects as well, like the discovery and integration of published services.

Applying collaborative concepts of social networks to Service Oriented Architectures and web service centered environments might result in a benefit for the producer, as well as for the consumer of a resource, since they can actively exchange about a services behavior and its properties. Experimental approaches of recommending similar services, that exploit the contextual and semantic relation of tags, extend common techniques and add an supplementary level of abstraction to the discovery of services.

Kurzfassung

In den vergangenen Jahren haben soziale Netzwerke und Applikationen die soziale Aspekte integrieren, einen enormen Zuwachs an aktiven Benutzern, als auch einen Gewinn an Popularität erfahren. Benutzer können ohne großen Aufwand an globalen Kollaborationen, die sich mit der Erzeugung und dem Austausch von Inhalten beschäftigen, teilnehmen. Um ein geeignetes Mittel zur Strukturierung und Klassifikation der Daten zur Verfügung zu haben, greifen viele Benutzer auf eine Technik zurück, die als Tagging bezeichnet wird. Das Tagging erlaubt es dem Benutzer jede denkbare Ressource mit einem beliebigen Schlagwort zu versehen, um diese identifizierbar zu machen, respektive ihren Inhalt und ihre Eigenschaften zu beschreiben.

Vergleicht man die Progression sozialer Netwerke und Service Orientierter Architekturen zeigt sich deutlich, dass letztere in einem weitaus langsamerem Tempo voranschreitet, das vor allem von den komplexen Anforderungen her rührt, die sich ergeben, will man einen allgemein anwendbaren Arbeitsablauf etablieren. Auch die wirtschaftlichen Interessen die entstehen, wenn Services von Drittanbietern integriert werden sollen, sind Eigenschaften, die hohe Auswirkungen auf die Evolution von Service Orientierten Architekturen haben. Abgesehen von den genannten Herausforderungen sind Aspekte, welche die Entdeckung und Einbindung von freigegebenen Services betreffen, noch nicht vollständig untersucht.

Werden die kollaborativen Konzepte von sozialen Netzwerken auf Service Orientierte Architekturen und Umgebungen, deren zentrales Element Web Services darstellen, angewandt, könnte sich sowohl für den Erzeuger, als auch den Konsumenten einer Ressource, ein erhöhter Mehrwert ergeben, da sie in einen aktiven Austausch über das Verhalten und die Eigenschaften einer Ressource treten können. Experimentelle Ansätze, die kontextuelle und semantische Zusammenhänge der beschreibenden Tags ausnutzen, um Empfehlungen für ähnliche Services abzugeben, erweitern bestehende Techniken und fügen der Entdeckung von Services eine ergänzendes Level der Abstraktion hinzu.

Contents

List of Figures

CHAPTER 1

Introduction

Tim Berners Lee original idea of the World Wide Web was to create an open space of information exchange, where academic content was easily available to anyone. This networking space in its basic form offered the possibility to participate in knowledge creational processes, by contributing ones own content, and making it available by the same means, as content of other participants of the network has been consumed. see [45] for a summary of the initial intentions of the WWW.
Unfortunately the more the Web grew and became mature in its technological and functional facets, the more it became less applicable for unexperienced users. The fields of duty to participate in a collaborative process shifted from pure content creational tasks to technological requirements, that started to become a barrier for many enthusiasts.
Technologies like blogs, wikis, mailing lists, and recent technological evolutions, above all social networking platforms, resumed the initial idea of a space for people to exchange about their interests and needs, regardless of their geographic location or insight of the underlying technology.
With the available technologies the access to contribute has been eased significantly, but this ongoing evolution on the other hand creates certain drawbacks. From a maintenance point of view, technology has to serve several requirements to make a user experience acceptable. The barrier to interact has been shifted from the human participants to the underlying technology and applications, respectively the developers of such. It is their assignment to reduce the effort, to make full use of the features of an application, to transmit the users views and opinions.

1.1 Motivation

In service oriented environments one of the main challenges is to seamlessly integrate existing software components in ones own work flow. Unfortunately the desired scenario of a brisk exchange about the components at hand is seldom achieved. It is common practice that the usage of services is restricted to company network boundaries and that there is no cross organizational interaction regarding the exchange of services. Neglecting commercial and legal interests of an exterior party, that come with the integration of components that have not been developed and

maintained within a company, see [40], one of the remaining central issues of Service Oriented Architectures is the search and discovery mechanisms of available services.
In order to discover services and exposed functionality there exist a variety of methods that often do not lead to satisfying results and leave room for further improvement, see [27] and [56], respectively allow researchers to experiment with various techniques to ease the retrieval of services and components. Especially in regards to visually exploring the relation of a web service to another there has been little effort so far.
Another aspect that is a motivation for this research comes from the influence a community of users can exert on the structure of search results. Categorization of data sets is usually restricted to a small group of experts, that classify the data according to fixed principles. To reflect the social dynamics the exchange of a community on a certain topic has, traditional methods of categorizing data can not be applied. Instead we aim for restructuring search results to improve the view of the data in hand.

1.2 Problem Description

The development of social networks and various platforms, offering community based applications, has grown enormously over the recent years. Mobile devices and area-wide network support has lead to a high acceptance of self-organizing, always-on social networks where users can interact and exchange content. Access to and usability of these applications is at a level where everyone can participate.
In regards to web services, the recent technological and usability improvements have not been as significant as in social networks. Publishing and discovery has undergone minor changes since it initial implementation, hence is not adapting very well to current issues and requirements, that come with the integration of a service, e.g. the update of descriptions or the exploring of similar services. With the system we are implementing, we are trying to cross-grade the straight-forward and lightweight paradigm of a community based approach to the complex and rigid environment of web service publishing and discovery. The influence a community of users has on the system is reflected in the way the published web services are categorized by the means of tags. The tags that the users attach to the resources that represent web services, are implicitly categorizing the content. The algorithmic foundations to realizing these functionalities are based on information retrieval research and are adapted to obtain suitable results for this meta data oriented approach.

1.3 Approach

Our basic motivation to create a system, that allows users to manage and implicitly categorize web service descriptions, comes from the fact, that there is no system available, that equally tries to merge the functionalities of social networking platforms and web service registries and simplifies the application of a user community.
Although there are many applications and approaches, that offer the functionality to discover and explore the entries of service registries, e.g. like UDDI[1] and various P2P overlay networks see

[1]Universal Description Discovery and Integration

[27] for a description of this approach or [56] for an example. We see some major advantages in a more lightweight approach, that combines the concepts of social network collaboration and web service registries. Common registry technologies do not take into account the expertise of the costumers, that make use of a certain service. There is typically few to no interaction between the vendor of a service, that registers a resource, and the consumer that integrates the service into his workflows. Although this aspect is acceptable for most scenarios, often both parties would benefit from a means of communication, that allows them to exchange their experiences and expertise on a specific service. Common service registry technologies are often not designed to enhance or regularly update the meta data of a given resource, thus trying to update existing data of a service in some applications is a rather tedious and error prone task.
User generated data about a resource and additional descriptions of implementation details, often eases the decision for developers in what context a service can be applied to. In most applications that serve as registries for services this is not evident. The social interaction that our system is designed for, creates a network that generates implicit knowledge a community has agreed on regarding a specific resource.
From a technological point of view it is a drawback to only be able to publish and retrieve services, that are purely soap based. In regards to the properties a web 2.0 application has to fulfill, we aim for a lightweight approach instead, that does not distinguish between the communication style a service is implemented to use. Hence a vast variety of services can be integrated in the registration process of our framework.
Influenced on some previously released research, one of the main challenges is to implement an algorithm, that automatically extracts a hierarchical structure from the flat structure that our keyword based approach results in. The algorithmic abstraction of the tags that are added to a service creates an additional layer of information and structuring of the dataset. To demonstrate the usage of the informational improvement of that commonly known technique i.e. tagging, we create a framework that is based on integrating algorithmically improvements for the search and discovery of web services. The success of the implementation, respectively the acceptance of a community of users, is bound to several fundamental requirements.

- The application has to be offer an understandable and clear structured interface, to make full use of the frameworks functionality with a minimum of effort to become familiar with the web- based user interface.

- The search engine is a combination of information retrieval and recommendation algorithm, that is based on information that the users have specified by applying tags to the services. Thus search results are heavily depending on the participation of the users. The more the system is used, the more the search results become accurate and the semantic relation between two services becomes obvious.

- To gain a maximum of benefit from the research the components and algorithms implemented in the system, should be themselves published as services, to achieve interoperability and re usability in different projects and scenarios.

1.4 Thesis Contributions

Main focus of the thesis is how social networking principles can be implemented to cope with the requirements of a SOA based environment. The contributions can be broken down to the following elements.

- We present an architecture for an application that fulfills the given requirements of social networking principles,
- We outline the implementational decisions that have to be considered and
- Discuss the advantages and drawbacks of such an community based concept.
- From an implementational view there are various possibilities to achieve an appropriate result of a similar system. Therefore the implementation only reflects a minimal set of necessary functions.
- Instead of recreating very complex concepts of social networking platforms, we rather focus on experimenting to integrate algorithmic aspects, that create an additional benefit for searching and navigating in the dataset, thereby improving the user experience of the system. Again there is large potential in creating very sophisticated means of visualizing the underlying data structure, but it is not the scope of this thesis to research on visual concepts, but rather to give insight in underlying concepts of architecture and design decisions.

Summing up the contribution of this master thesis consists of a distributed framework to exchange meta data, about web services, based on the concepts derived from social networks. We integrate an algorithm that produces recommendations of similar services and conclude with an evaluation of performance of the co-operating software components. Apart from technological and architectural details, we also focus on social network properties and deal with the attributes that lead to a widely acceptable community solution.

CHAPTER 2

Related Work

Tagging or manual indexing, as it is sometimes referred to in [83] by Voss, started to become a common activity, as the popularity of social networking applications like delicious[1], CiteULike[2] and connotea[3], just to name a few representative platforms that serve as the scientific grounding for early research in the area of tagging, increased as of late 2006. Since then the activity to tag a certain resource, regardless of the type, has been integrated in various common work flows nowadays.
The popularity and the huge acceptance of the users of the activity of tagging, comes from its ease in use and its enormous benefits to the user experience, as stated by Begelman et al. in [6] that claim that *"tagging seems to be the natural way for people to classify objects, as well as an attractive way to discover new material."* Hoto et al. in [35] trace the immediate success of tagging and applications, that rely heavily on the activity of tagging, to *"the fact that no specific skills are needed for participating and that these tools yield immediate benefit for each individual user, without too much overhead."* Because of its simplicity tagging is gaining more and more importance in social collaboration, as it is a means of implicitly categorizing content and helps structuring huge data sets.
Although the contributions in research on the topic of tagging, that have a huge impact on the whole scientific field, have been performed several years ago now, we believe that there is still a lot of potential in regards of stretching the limits of that technology, especially in integrating these technologies in Service Oriented Architectures.

2.1 SOA and Web Services

An integral aspect of this thesis is, how a community of users can benefit from applying standard social network functionality to the storage and retrieval of web services, respectively integrate

[1]www.delicious.com
[2]www.citeulike.com
[3]www.connotea.org

this functionality in SOA environments, and how restructuring of search results of services can yield to an improved user experience.
As mobile devices have become cheaper and the availability of broadband network access dramatically has increased over the last years, using mobile applications has become an activity that is no longer restricted to a small group of technological enthusiasts. Instead it has become an activity that helps users conquer the challenges of modern life, as there are handy services available that touch every aspect of life. As the acceptance and usage of these applications has increased, we also sense the desire to let a community participate in ones activities and share ones interests. Therefore many recent applications that include aspects, where social activities are involved and information about a user is distributed over the internet, become more and more popular see [38], [70] and especially [37].
In the software engineering discipline there is a similar trend obvious. The responsibility to perform a computational task is no longer restricted to a single machine, but is divided into more manageable subtasks on multiple machines, for reasons of performance and reliability. Applications and work flows are no longer restricted to a single processing unit, but are spanned over various participants, be it machine or human, see [69]. As well as complex tasks are split into computational smaller problems, simple and easy to handle applications are merged into larger clusters of systems, to perform jobs that were not initially intended. Those so called mash up systems enjoy great popularity, as their areas of application are far more vast than a single component could ever be, see [64].
The possibility to add and exchange simple components in ones own application work flow, has lead to an increase in service availability and demand, thus claiming an extensive exchange about the properties of a service and its qualities.

Web Service Definition Papazouglu in [63] defines a web service as *"a self-describing self contained software module available via a network, such as the internet, which completes tasks, solves problems, or conducts transactions on behalf of a user or application. Web services constitute a distributed computer infrastructure made up of many different interacting application modules trying to communicate over private or public networks(including the internet and web) to virtually form a single logical system."*
And further Papazouglu defines the following properties that specify a web service: A web service ...

- ... *"consists of loosely coupled software modules."*
- ... *"can semantically encapsulate discrete functionality."*
- ... *"can be accessed programmatically."*
- ... *"can be dynamically found and included in applications."*
- ... *"is described in terms of a standard description language."*
- ... *"is distributed over the internet."*

We can further divide web services in two types of services. *"Simple Services"* perform a request-response interaction. These type of operations are referred to as atomic operations, as a request is either executed completely or not at all. *"Complex Services"* on the other hand involve interactions with various sub services to accomplish a goal that has a computational larger effort but also benefit.
Additionally we can divide web services further in regards to their interactional behavior. We divide between loosely or tightly coupled systems, where tightly coupled defines a system in which applications need to know how their partner application behave, i.e. know details of method signatures and quality attributes of a service. Loose coupling on the other hand is the architectural paradigm that is higher aspired, as it improves the reuse of components.

Service Oriented Architecture Definition SOA and web services are often treated as synonyms, but in fact SOA can be implemented without the use of web services, using different technologies e.g. like DCOM[4], CORBA[5] or REST[6]. Rather than depending on a specific implementation technology, SOA is a way of logically organizing software components, to provide access to services for end- users as well as other services, depending on the published functionality.
The discoverable interfaces of the services decrease the effort to access and integrate the components, allowing to move the interaction from a simple exchange of information to a more complex and sophisticated interaction paradigm. Besides the advantages that come with integration, the costs of the overall system are reduced, due to re usability of already developed components and services regardless with what technology they are implemented. Central to SOA is the concept of composing simple services to larger systems, that have comparably higher complexity than the integrated services on their own.
In standard literature like [63],[4], [85] three operations of systems implementing a SOA are defined.

- *"publish operation"* A service provider is publishing a service or component.
- *"find operation"* A service requester is querying service registries or comparable technologies to discover services that comply to his needs.
- *"bind operation"* A service requester is connecting to the discovered service and is integrating it by creating a connection to the service.

We see that web services and Service Oriented Architectures are closely related to each other. A SOA can be realized by the means of web services, but not every application of web services leads to a realization of a SOA. The publishing and the discovery of a service are integral parts of the architectural paradigm, simplifying the discovery of the required functionality and easing the retrieval of a service leads to an overall benefit for the participating stakeholders.

[4]Distributed Component Object Model
[5]Common Object Request Broker Architecture
[6]Representational State Transfer

Core Technologies

What is of special interest to us, is how SOA and the architectures that support web service usage can benefit from the research referenced in this chapter. Multiple disciplines have delivered the foundation for our approach, we integrate many various different results to orchestrate them to this framework. A very intuitive way of recommending a similar service comes from the research of Konduri et al. in [43], that try to identify web services and recommend similar services, based on a comparison of the services WSDL file. A similar approach can be found in the work of Dong et al. in [21] where multiple characteristics of a service, like web service operation signature, similarity of textual description of a service, etc., are used to identify a web service and its properties and derive closely related services from this information. Dong et al. also use an agglomerative clustering algorithm to create a structuring for the retrieved results. These approaches differ to our ideas, as the information that is gathered from the user community is totally neglected, when recommending a system. We on the other hand aim to integrate the aspects of evaluation of a taggable resource, that only an unbiased community can produce.
The framework that is central to this paper is used to demonstrate architectural properties that have to be considered, as well as to experiment with algorithmic improvements to support current technological trends, above all, trends that concern social collaborative networks. Equivalent and similar implementations are conceivable, using different approaches, from the ones we have chosen. In this chapter we will present some of the basic technologies, the system we implement is referring to.

WSDL

A key feature to realizing the concepts of SOA is the discovery of a suitable service. To describe what a web service does, where it is located and how it can be invoked the Web Service Description Language (WSDL) [13] is used. This platform independent XML based language is used for specifying the public interface a web service implements, and includes information like the supported message protocols, the data type of the used messages and the binding information of the service. Based on the specification of these properties, web services can be either manually integrated in a work flow by a service requester, or automatically found and invoked in other processes, due to its machine-processable structure. See Cauldwell et al. in [10].
The structure of a WSDL description can be split in two parts.

- **The interface description** is an abstract definition of interface of a service, that contains information on what types of messages are needed and what type of message format is necessary to establish communication between a client and a service.
- **The implementation description** on the other hand specifies the exact details where a system resides and what binding properties have to be fulfilled to enable interaction with the service.

The WSDL 2.0 specification, see [12] defines eight message exchange patterns of how communication with a service can be established. In comparison to this WSDL 1.1 only offers four

exchange patterns. That clearly shows the improvements in communicational techniques and the improvements of possibilities to exchange data.
From a historical perspective WSDL is an international W3C standard and has been updated as of 2007. The updated Version WSDL 2.0 is currently a recommendation, but is gaining more and more importance since it has some major improvements over the 1.1 version. Especially when it comes to RESTful services, respectively services that require different HTTP binding methods than GET and POST, version 2.0 has overcome the deficiencies of its predecessor and allows more flexible configurations in communication and binding. For more information about the details please refer to [13] for the initial standard or [12]for the current recommendation.

SOAP

The Simple Object Access Protocol (SOAP) is a communication protocol based on XML. It is used to enable communication over heterogeneous networks through its language and platform independence. As means for transportation over a distributed network like the internet HTTP, FTP, SMTP as well as JMS, is possible, although using HTTP minimizes the configuration effort, since interaction over firewalls can take place without any intervention. SOAP is used as a means of communication between a client and a service, that is to be invoked, as well as a client and an UDDI registry.
Every SOAP Message is constructed the following way. A **SOAP Envelope** containing one or more headers and a **SOAP Body** containing the actual content that is to be transferred. In a soap based-service environment, depending on the service' s WSDL definition, two types of message patterns are possible.

- **RPC style**
  referring to a remote procedure call, where the structure of the message body has to indicate the method name and contain a set of parameters, where as in

- **Document style**
  the content of the message does not have to conform to any regulations.

Further both styles can be sub divided by their encoding style. Either **literal encoding** that indicates that a message content conforms to a specific XML Schema or **SOAP encoded** symbolizing that the content uses a set of rules based on the XML Schema data types, but no specific Schema is applicable. Please refer to [84] or [63] for further information.

Service Registries

In their analysis about common service registries and their properties Treiber et al. in [22] identify three different types of registries.

Centralized Service Registries CSR are registries that have a single registry broker that processes incoming search and publish requests. Because of their single point of communication with the registry these systems are rather easy to maintain without investing much overhead on mirroring the available data, but lack of the capability to conquer increasing demands on the service. The scalability towards and increase of usage is not very performant, since the single point of access results in a delay in processing of requests.

Decentralized Service Registries DSR distribute the responsibility for maintaining service descriptions to the participants of a peer to peer network. Although scaling very well towards a large amount of users, because of the distributed structure of the system, availability of the actual data can not be guaranteed, as it is highly depending on the availability of a single network node. Overlaying a network on top of the actual communication layer, systems like CHORD, see [77], offer the means for very efficient search requests. On the other hand this system has to suffer from a large overhead to create the topological structure to perform efficient searches.

Hybrid Service Registries HSR try to combine the advantages of centralized and decentralized service registries. Various types of nodes make a reliable and performant management of services possible. Again the expenses for creating the network structure and the additional communication for coordination of the participating peers have to be considered.
Since the system we implement is representing a centralized service registry we concentrate on discussing two examples of this registry type.

UDDI

In order to publish and discover the potential of a service a vendor has to offer, a standardized mechanism is needed to store service relevant data. On the most basic level a service registry keeps track of what services an organization has published and what properties a given service has. Two types of registries are possible

- **document- based registry**
  Clients of the registry can publish information about the type of business and technical specifications of a service, by uploading XML based documents. Additionally clients have to provide descriptive information about each document they associate with the service in the form of meta data.
- **meta data- based registry**
  In this case service providers simply upload documents that contain service information. Any other documents are not stored in the registry, rather any meta data about a particular service is gathered from the existing service documents.

The Universal Description, Discovery and Integration (UDDI) is a possible realization of a document- based centralized registry. This platform independent service registry is based on XML and is used for describing and registering web services in terms of business categorization and technical information likewise.
Its core components are

- **White Pages** - that store information about contacts of a specific company.
- **Yellow Pages** - that contains a standardized categorization of the industrial affiliation of a company.

- **Green Pages** - that store information about technical aspects of services published by a certain company.

UDDI API

To query and access the very complex data UDDI offers, an API to handle the process of publishing and retrieving programmatically, is available. According to the UDDI specification, viewable under [82], the API supports 3 types of query patterns.

- **browse pattern**
  where the clients make use of queries that retrieve a large quantity of results. By browsing through the data rows, the user can search for a service that fulfills his needs.
- **drill down pattern**
  Once a key is available it is easy to query for detailed information and specifications of that service.
- **invocation pattern**
  Once invocation of a service fails, the first step is to query for the binding template of the corresponding key of the service, to verify that the information used to invoke the service is still up to date. Queries might fail for the reason of updates or server migration. To minimize the effort to invoke a service, the easiest way is to recheck for a given identity key and repeat the process of invocation.

ebXML

Besides UDDI another registry that is also a centralized service but metadata based registry, is electronic business using XML (ebXML).
In the technical architecture document in [57] the following goals of ebXML architecture are introduced. The ebXML architecture is aiming for ...

- *"... A standard mechanism for describing a Business Process and its associated information model."*
- *"... A mechanism for registering and storing Business Process and Information Meta Models so they can be shared and reused."*
- *"... A mechanism for registering the aforementioned information so that it may be discovered and retrieved."*
- *"... A standardized business Messaging Service framework that enables inter operable, secure and reliable exchange of Messages between Trading Partners."*

to name a few. Similar to UDDI ebXML offers means to interchange descriptions about business models and service descriptions but goes a step beyond to also provide a complete framework to enable secure and reliable communication for the exchange of services, especially in regards to integrating small and medium sized businesses into a global electronic market place.

REST

REpresentational State Transfer (REST) is an architectural style of how services and information systems can be accessed over distributed networks like the internet, see [24] for a reference to Roy Fielding' s dissertation, that is the basis for this technology. Central to the idea of REST is to address every accessible data via its own URI[7]. RESTful services are stateless, which means that the server does not store any information of previous requests from a specific client. Every data necessary to perform a given operation has to be sent on each request. The main benefit of the statelessness of the server is the capability to share the load on increasing requests to various servers.
The underlying transport protocol for interactions with REStful services is HTTP, which defines several methods that specify the type of request for a service. Possible methods, also referred to as verbs are

- **GET** - that requests data from the server.
- **POST** - that creates a new resource on the server.
- **PUT** - that signalizes an update of an existing resource.
- **DELETE** - that deletes a specific resource.

The data that is returned from a RESTful service is typically represented in JSON[8] or XML[9], although any other MIME[10]- Type is possible as a valid representation. The ease of use and the simplicity of its design principles make a RESTful architecture a more and more attractive choice for developers of how to design a web service.
In [23] Fielding et al. outline the advantages and present numerous applications of the REST architecture paradigm and conclude that *"The important point, however, is that REST does capture all of those aspects of a distributed hypermedia system that are considered central to the behavioral and performance requirements of the Web"*.

REST vs. SOAP- based Services

When comparing RESTful and SOAP - based services the main feature, they both have in common, is their platform and language independence, since both concepts are based on XML. Another benefit of these technologies is that they both rely on HTTP as their transport protocol, which allows communication to take place over firewalls throughout company boundaries.
SOAP The big advantage of SOAP based services is that communication is bound to a contract defined by a WSDL of a service. There are many initiatives, that try to even more improve the security and reliability of SOAP based approaches, see [55] as an example. Since the WSDL 2.0 recommendation, a better specification of the interface and the behavior of a service is possible, hence is also applicable for RESTful services, but this is a not very common practice, yet. The

[7]URI Uniform Resource Identifier
[8]Java Script Object Notation
[9]eXtensible Markup Language
[10]Multipurpose Internet Mail Extensions

definition of the SOAP Header makes it possible to define very complex scenarios, since multiple web services can be addressed in one SOAP Envelope, respectively a single message.
REST is a more flexible approach, that is not bound to a certain schema and messages do not contain any extra XML markup payload. The message content is solely focused on the necessary data for a request. The big advantage that makes this technology very attractive for developers, is the fact, that architecture is flexible enough to easily and quickly adapt to changes and services can be implemented without too much effort.
Their advantages are equally a possible disadvantage, although they have many commonalities, it seems that the more flexible approach implemented in the RESTful architecture, seems to gain more and more importance in recent service development.

2.2 Web 2.0 System Properties

Definition Web 2.0 is a term that denotes the technological evolution of the way users interact with distributed content. It describes the transition from a non- interactive distributed environment, primarily based on consumption of information, that is due to the creation of very few contributors, to a global- collaborative based approach to use the web, where members of various communities take the opportunity to participate in self organizing and content creational processes. As a matter of fact the term is not only restricted to technologies that are gathering information from active participation of users, but also from the users behavior in the web and the users habits.

Properties Tim O'Reilly defines several properties in [59] that make a successful web 2.0 application. One of the key features of such systems is to be in control of data sources that are *"hard-to-recreate"* and *"get richer the more people use them"*. The underlying dataset is what defines a system and its properties, and the need of participation also is a requirement for social networks. Participation and storage of the generated data directly results in non functional requirements like scalability and availability, that have to be taken care of.
Besides the data source specific properties there are architectural requirements that have to be considered, as well. *"Lightweight programming models"* as O'Reilly coins it and the *design for "hackability" and "remixability"* of applications are aspects that are crucial for the success of an application. Systems are supposed to implement not too complex architectural paradigms, to be able to still easily adapt to changes in the business model of a system.
Time to Market is increasing continuously and leads to an end of the common software release cycles. O'Reilly states that *"users must be treated as developers, in a reflection of open source development practices."* That means that software is released early without too much effort on testing and verification. Classical testing phases are replaced by beta releases of an application and are presented to a community without warranty of its liability. That reduces the costs for extensible testing cycles and also allows to directly integrate the feedback of the end user, to improve an application before its final release. Extensible usage of a community covers much more scenarios and unveils much more deficiencies than any testing suite ever could.
Applications that are produced under the aspects of web 2.0 have to cope with some requirements more extensible, than non distributed applications have to. Non- functional requirements like scalability and accessibility gain importance as the systems to be implemented depend more

and more on the participation of its users. In terms of larger processes it is an essential feature, that the architecture is designed for extensibility and reusability in other service, or mash- up systems, to be applied to new areas of applications. Hence software has to be available as a single computational unit respectively a service and *"not pre- packaged software"* that cannot be adopted to the infrastructural needs.
Another significant aspect is the fact that mobility of the users has increased and that this has to be reflected in the portability of a service to multiple devices. Applications are no longer restricted to a single point of access, but have to pay tribute to a society that aims for a perpetual stream of communication and interaction.

Folksonomy

When talking about social collaborative networks it is unavoidable to stumble upon the term folksonomy, that was originally coined by Thomas Vander Wal, who defines folksonomy as *"...the result of personal free tagging of information and objects (anything with a URL) for one's own retrieval. The tagging is done in a social environment (usually shared and open to others). Folksonomy is created from the act of tagging by the person consuming the information."* available on his personal blog at [80]. See Albrecht' s master thesis available at [3] for a comparison of folksonomies and ontologies and a further discussion of advantages and disadvantages of folksonomies. Ontologies and the what type of datastructures and areas of application this type of classification is working well, is handled in the work of Shirky available at [72], that stresses that categorization only can work, when everybody is participating in it and there is no limitation of how to describe a category. Shirky analyses tag usage and outlines that *"by forgoing formal classification, tags enable a huge amount of user-produced organizational value, at vanishingly small cost."*

Service Based Social Network

Another approach to discover services for ones one needs is followed by Treiber et al. in their work about social networks[81]. In their paper they present an approach to extend the definitions and properties of FOAF[11] [7] to make use of well known notification mechanisms, to integrate services in ones work flow. Existing infrastructure of web 2.0 research is a profound grounding for extending a technology for ones needs. The main difference to the current approach lies in the way how services are managed and recommended. In contrast to Treiber et al. that follows a de-centralized approach to distribute information about a service, we foster a centralized, registry based approach as we use an extensive similarity calculation algorithm to recommend similar services. We believe that the calculation cannot be distributed efficiently on several peers but has to remain on a central authority.

[11] friend-of-a-friend FOAF

2.3 Ranking and Clustering

Information Retrieval

As the number of documents and users of collaborative applications that use tagging increases, sophisticated means of information retrieval become more and more important, to keep the usability and performance of the system in use, at a reasonable level.
Hassan- Montero et al. in [31] separate between means of information filtering and information retrieval and define two characteristics of the latter.

- *"Push technologies Where technologies like RSS are being used to push the information to the user."*
- *"Pull technologies Where the user actively seeks for information by pulling the data by means of either querying or browsing."*

Bao et al. in [5] evaluate the use of an adapted PageRank called SocialPageRank, and identify the benefits of the use of social annotations to improve the search for documents.
Widely in use in many collaborative systems is an approach to *"enable visual browsing"* in a dataset, that is referred to as a *"tag cloud"*. Despite its simplicity and popularity among the tagging community, early research on that topic, as in [31] or [6], already disclose the limits of that technology and assert two major drawbacks. Additional compare the work of Sinclair et al. in [73] that investigate the areas of application of tag clouds.

- *"The method to select the tag set to display is based exclusively on the use of frequency, which inevitably entails that displayed tags have a high semantic density. In terms of discrimination value, the most frequently used words are the worst discriminators [68] ..., very few different topics, with all their related tags, tend to dominate the whole cloud."*

  Hence the more often a tag is used, the higher is its importance to the visualization and the less it becomes useful for differing between the tagged resources.

- *"Alphabetical arrangements of the displayed tags neither facilitate visual scanning nor enable infer semantic relation between tags."*

  The possibilities of tag clouds seem limited as additional information about the relation between the tags, forming the dataset, is lost. But *"although a folksonomy is commonly defined as a flat space of keywords without previously defined semantic relationship, different studies demonstrate that associative and hierarchical relationship of similarity between tags can be inferred from tag co-occurrence analysis"* see Hasan- Montero et al. in [31] and Salton et al. in [67] for a complete discussion on that topic.

Early research in information retrieval techniques showed very clearly the superiority of categorized, structuring of search results over a traditional list representation. Chen et al. in [11] state that categorizing data *"has the advantage of leveraging known and consistent category information to assist the user in quickly focusing in on task-relevant information."* thus avoiding time wasting engagement with humble structured data sets.

Clustering

Christiaens in [14] names two very opposing techniques of how to classify large sets of data.

- Using a **lightweight approach**: i.e. a technique like tagging.
- Using a **heavyweight approach**: i.e. techniques like taxonomies, facet classification and ontologies.

The great benefit of using heavy weight techniques to classify data sets, is the possibility to be able to fall back on structured query languages to retrieve results. The "rich meaning" of the data enables trained users to gain a maximum of semantic knowledge, especially from ontologies, allowing them to efficiently navigate through the data. The major drawbacks of this approach, although very powerful and superior in implicit meaning to the lightweight classified data, are the expenses in creation and maintenance that have to be considered as well as the steep learning curve to gain information from the data. Only very small groups of experts are involved in specifying the categories for a given technique and only few users can enjoy the benefits from the rich dataset. Thus heavyweight classification is not applicable to a rapid changing dataset that is produced for a large group of users. On the other hand one has to admit that lightweight classification of data tends to deliver rather shallow results, as users tend to use categories that are too unspecific for a larger community.
From a semantic view there is plenty of room for improvement in displaying the relationship between the elements in a data set of a collaborative tagging application. One approach to improve the yield of the information of the available data is to create a different data structure out of it. Steinbach et al. in [75] state, that *"clustering has been proposed for use in browsing a collection of documents"* and further more *"document clustering has also been used to automatically generate hierarchical clusters of documents."* Additionally Steinbach et al. in the same document in [75] have shown that *"clustering techniques can improve the user experience of tagging systems."* Compare the work of Tan et al. in [79] for a discussion of different clustering techniques.
More recently the approach, of extracting additional structural information from a flat set of tags, has been adopted from several researchers using various techniques.
In the work of Heymann et al. in [33] the similarity of tags is used to create a taxonomy-like structure from an unstructured data set. Ramage et al. in [65] in cooperation with Heymann and Garcia-Molina state that *"tagging data improves the performance of automatic clustering algorithms when compared to clustering on page text alone"*
Shepisten et al. in [71] have chosen a slightly different approach of adding an additional layer of information to the given data set structure. In their work they extend a recommendation algorithm that also makes use of the similarity of tags, to a more personal recommendation based on a user profile. Tags are therefore combined to larger clusters, that represent certain topics within the network. Hierarchical tag clustering also is applied to the research of Gemmel et al. in [28] that show the benefits of this technique while navigating through large sets of data from folksonomies.
In [6], Begelman, Keller and Smadja determine the relation among tags, using a very simple technique of co-occurrence of two tags describing a certain resource, and create clusters to give a meaningful structure to the unsorted data. Jäschke et al. in [39] propose a tag recommending system, and compare the effectiveness of two different strategies. In their work they stress the advantages of graph based approaches.

CHAPTER 3

Tagging Framework Design Principles

In order to implement an application that meets the requirements of a system we aim for, we have to direct the discussion towards the properties and characteristics that shape a network and define its final behavior. In this section we want to outline these aspects, that have been defined in essential contributions on the topic of tagging and social networking applications. We conclude this chapter by introducing the mathematical grounding of the similarity calculation, and an example of how a tree representation can be used to re- structure the data used to tag several web services.

3.1 Characteristics

Besides the requirements of web 2.0 systems that have been defined in [59], Marlow et al. as well as Szomszor et al. have identified some characteristics in [49], and [78] that are specific to social collaborative networks. These properties reflect fundamental design issues that have to be considered, when implementing such a system.

"Tagging Rights" A very important question while designing a collaborative social network system, is who is allowed to tag what resource. i.e. How open is the data set towards changes. Marlow et al. spotted two opposing attitudes towards this aspect.

- ***"self tagging"*** Systems that restrict the users possibilities to add tags to resources, that they have not created themselves, can be categorized as self tagging systems.(e.g like Technorati[1])
- ***"free-for-all"*** While self tagging restricts the possibilities of the users to tag content, the free-for-all approach offers a broader access to editing various resources. In systems like Flickr[2] for example every user is allowed to tag any resource, which according to Marlow' s

[1] http://technorati.com/
[2]http://www.flickr.com/

et al. research, *"broadens tags that emerge, both in magnitude of the group of tags assigned to a resource, and in the nature of the tags assigned."* Depending on the amount of different types of user, the tags that are used can have a huge variety.

"Tagging Support" Depending on the area of application a system has, we can identify three types of very different mechanisms, that have a major influence on the outcome of the data pool.

- ***"blind tagging"*** A user can randomly choose any character combination he wants to add to a certain resource and is not offered any information about the tags other users applied to a resource. On the one hand that broadens the usage and the diversity of various tags, but on the other hand also leads to the known problems of folksonomies, like synonyms and homonyms.

- ***"viewable tagging"*** A user can see what tags are applied to a resource by other users. The systems influence is reasonable.

- ***"suggestive tagging"*** The system actively suggests tags that can be applied to a certain resource, based on various methods, like *"machine-suggested synonyms"*, etc.

"Aggregation" Is a parameter that has statistically influence on the outcome of term frequency analysis and therefore a main influence of similarity calculations of resources.

- ***"bag model"*** The term bag model coins an approach where a system permits the usage of duplicate entries of tags to a certain resource.

- ***"set model"*** Where as if a the set model is applied to a system it is not possible to add a tag more than once.

"Types of Objects" Another aspect that characterizes a collaborative network, is the type of resource that is to be tagged by the community. Although not restricted to any type at all, common types include web sites[3], images[4], videos[5], academic articles[6] or web services how it is applied in practice in our system.

"Source of Material" Determining the source where the system draws the resources from, is another integral part of a social network. Several systems, like the system we are implementing, allow users to upload the resource themselves, while other systems prescribe the resources that are available. To leave the choice of what resources can be uploaded in a system to the user, results in a very flexible approach with the drawback that control over the actual data is lost. The validity and consistency of uploaded data has to be taken care of by the community itself.

[3]Delicious - http://www.delicious.com
[4]Flickr - http://www.flickr.com
[5]YouTube - http://www.youtube.com
[6]CiteUlike - http://www.citeulike.org

"Resource Connectivity" Depending on the type, resources themselves may be connected among each other. The connectivity between resources is divided in three categories: linked, group or none. While linked resources connect to a single resource, e.g. via hyper links, group connectivity spans a larger amount of resources that are related in some way. What has to be considered is the fact that resource connectivity directly influences the complexity of a system. Tagging support is also depending on this property as suggestions and viewable tags rely on the connections that are spanned among the resources.

"Social Connectivity" Similar to resource connectivity there is a social connectivity that equally influences a folksonomy. Again the categories: linked, group or none are applicable. Linked relatedness is a typical functionality in many social networks that is often referred to as Buddy list. Group connectivity is a property that allows user to group their known social contacts to groups they can create, define themselves that helps them to restructure larger amounts of contacts, respectively creating sub- folksonomies of a larger structure.

3.2 Usage of Tags

Golder and Huberman have defined several scenarios in [29], of how tags are used. See a reference from Szomszor et al. of some of the characteristics in [78].

- ***"Identify the Resource"***. The most common usage of tags is as identifiers that describe the resource itself, *"including common nouns of many levels of specificity, as well as many proper nouns, in the case of content discussing people or organizations."*

- ***"Identify the Resource' s type"***. Not only are tags used to identify the content of a resource, but also the type of a resource(e.g. article, blog, news feed, etc.).

- ***"Identify the Owner of a Resource"***. *"Some bookmarks are tagged according to who owns or created the bookmarked content."*

- ***"Identify Refining Categories"***. Some tags are used to *"refine or qualify existing categories"*.

- ***"Identify Qualities or Characteristics"***. Adjectives like interesting, useful, etc. are used to reflect the importance or quality of a resource.

- ***"Identify Relation"***. To specify a self reference to a resource, tags are often used to denote a certain relation like, myphotos, mypaper, etc.

- ***"Organizational Requirements"***. A final yet very important aspect, is to use tags to group resources according to certain commonality. Tags like toread, jobsearch, etc. are used to organize relations among the gathered resources.

3.3 Similarity and Tree Structure Algorithm

The algorithm to determine the similarity between a query and a resource that is identified by the tags that are attached to it, is based on a variety of papers and research topics. The original idea to harvest implicit knowledge from unstructured data has evolved over time.
Starting as an approach to improve the precision in information retrieval tasks, the idea of an algorithm to re- structure the user generated meta- data has not changed significantly. In the year 2000, Chen et al. in [11] already noticed the great benefit of clustering for large datasets and state that *"organizing search results allow users to focus on items in categories of interest rather than having to browse through all the results sequentially"*. Steinbach et al. in [75] discuss different clustering techniques and outline the differences between K- means clustering that is a partitional clustering technique and agglomerative hierarchical clustering that is used in the current approach. Although K- means has a better runtime and produces clusters of equal quality as stated in [75], it assumes that the number of clusters in specified in advance, thus restricting the structure that might arise without such a limitation. Rather recent research includes the work of Brooks et al. in [8] that have evaluated the benefit of similarity of tags and hierarchical clustering techniques introduced in [19] by Cutting et al. to promote clustering as a superior technique for browsing large datasets over classical non- hierarchical information retrieval methods. Similar to this Begelmann et al. show in [6] *"clustering can improve the tagging experience and the use of the tagspace in general."*
There are several techniques to determine a semantic relatedness between two tags. In [9], Cattuto et al. apply several measures of relatedness including co-occurrence, and distributional measures that are based on cosine similarity calculation. Their results yield that *"globally and meaningful tag relations can be harvested from an aggregated and uncontrolled folksonomy vocabulary"*. Different similarity measurements are evaluated in [48] by Markines et al., Heymann et al. in [33] also outline cosine similarity as a very effective way of determining the similarity of tags and discuss the effect on data resulting from large social networks in [34]. Shepisten et al. in [71] extend their clustering efforts and introduce a recommendation strategy where they include the similarity they can infer from personal information of the user profiles of a system. Skopik et al. in a paper about trust of services in [74] were able to adopt a clustering based recommendation approach to the topic of security and trust of web services. Sujit Pal in [62] and [61] delivers the implementation grounding for the similarity calculation and hierarchical clustering, and Garcia in [26] outlines the mathematical foundation.
In the current implementation we re- focus on the social aspects that arise when dealing with a community of users, and their impact on the data we can gather from the interaction of users when dealing with web services and their retrieval.

Mathematical Definition

When dealing with users that apply tags to resources we have to deal with a data structure that is commonly known as a folksonomy. Hotho et al. in [35] defines a folksonomy formally as:

a four tuple F:=(U,T,R,Y,) where

U,T, and R are finite sets, whose elements are called users, tags and resources, resp.

Y is a ternary relation between them called tag assignments, i.e. $Y \subseteq U \times T \times R$

From a mathematical perspective this structure is referred to as a *"triadic context"* cf. [46] and [48] as well as *"tripartite hypergraph"* see [53] and [30]. In an earlier discussion of the underlying data concerning the visualization of the data structure, we could see that, because of its complexity, it is a non trivial issue to visualize and handle such data. Mika et al. in [53] shows further, how to avoid the drawbacks of this data structure, by splitting the tripartite representation it into three bipartite representations of the problem. *"Tripartite graphs and hyperedges are rather cumbersome to understand and work with. However, we can reduce such a hypergraph into three bipartite graphs(also called two-mode graphs) with regular edges".*

Vector Space Model

When calculating the similarity of two resources, respectively the similarity of a tag to a certain resource, we refer to the cosine similarity as the measurement of choice. It is a very commonly adopted approach and there exist open source libraries that ease the use of this computational model. To be able to calculate similarities between documents, the persisted resources from the systems have to be transformed into a vector space model representation. Salton et al. in [68] supply the results of research, that serve as a grounding for this type of representation. Resources respectively the documents that represent the resources from the systems are modeled as a vector over a set of describing tags.
Setting up the resources as vectors and creating the necessary data structure to calculate the similarity is a very time consuming task and is therefore performed as an off line step that is repeated continuously in the Similarity Service. The runtime of the algorithm to calculate the similarity is tightly coupled to the amount of resources and the amount of tags applied to a single resource, hence calculation does not scale very well with a rising number of resources. To overcome this drawback we decided to perform every subsequent operation on a snapshot of the data retrieved from the database. Users that submit a search request to the BackEnd component, have to consider that the search results might change slightly from one request to the other, as an update of the data set might have occurred.

Term Frequency- Inverse Document Frequency

G.Salton et al. in [66] and in [68] describe a technique that is used to determine the weight of each tag that in a certain document, also compare [71] where Shepisten et al. follow a similar approach. Term frequency is a measurement that is used to identify how often a tag has been used to describe a certain resource. We write :

$$\text{tf(t,r)} = |\{a = \langle u, r, t \rangle \in A : u \in U\}|$$

where t denotes a tag, r denotes a resource, and A the set of annotations of the user u.

Inverse document frequency is introduced to keep the influence of a certain term at a reasonable level. The user might expect a tag that is used more often to describe a resource to be more expressive, than a term that is used less often. In fact this circumstance results in an effect that has the opposite influence on the data. A term that is used significantly more often tends to dominate the similarity structure and is therefore less expressive. A solution to this is the introduction of the inverse document frequency, which qualifies the impact of a single term on the document.

$$tf * idf(t,r) = tf(t,r) * log(N/n_t)$$

where N denotes the total number of documents, divided by the number of resources a certain tag was applied to.

Cosine Similarity

While the first steps in calculating the similarity structure to perform search requests on, is done in an off line step. The following calculation can be performed in real time. We can calculate the similarity between a resource and a query as follows.

$$\text{cos(q,r)} = \frac{\sum_{t\in T} tf(t,q)*tf(t,r)}{\sqrt{\sum_{t\in T} tf(t,q)^2} * \sqrt{\sum_{t\in T} tf(t,r)^2}}$$

Tree Structuring

As a final step in each query we create a hierarchical structure, by applying an agglomerative clustering algorithm on the result of the calculation. This step is performed to create a structure that maps the semantically relatedness of the items among each other and the search query. Agglomerative clustering is a bottom up technique where every element, to be clustered, is represented in a cluster of its own. In every step clusters are compared against each other and merged, depending on a linkage criteria, into a bigger cluster containing more elements. The algorithm is finalized when all elements are merged into a single cluster. The tree structure that is produced is also known as a *"dendogram"*. Hastie et al. in [32] give a very detailed insight in different clustering techniques, and outline their algorithmic properties. To determine the similarity between to elements we again refer to the cosine similarity. In [60] Sujit Pal delivers the grounding for the calculations and the implementation. The algorithms are based on the research of Konchady et al. that can be found in [42].

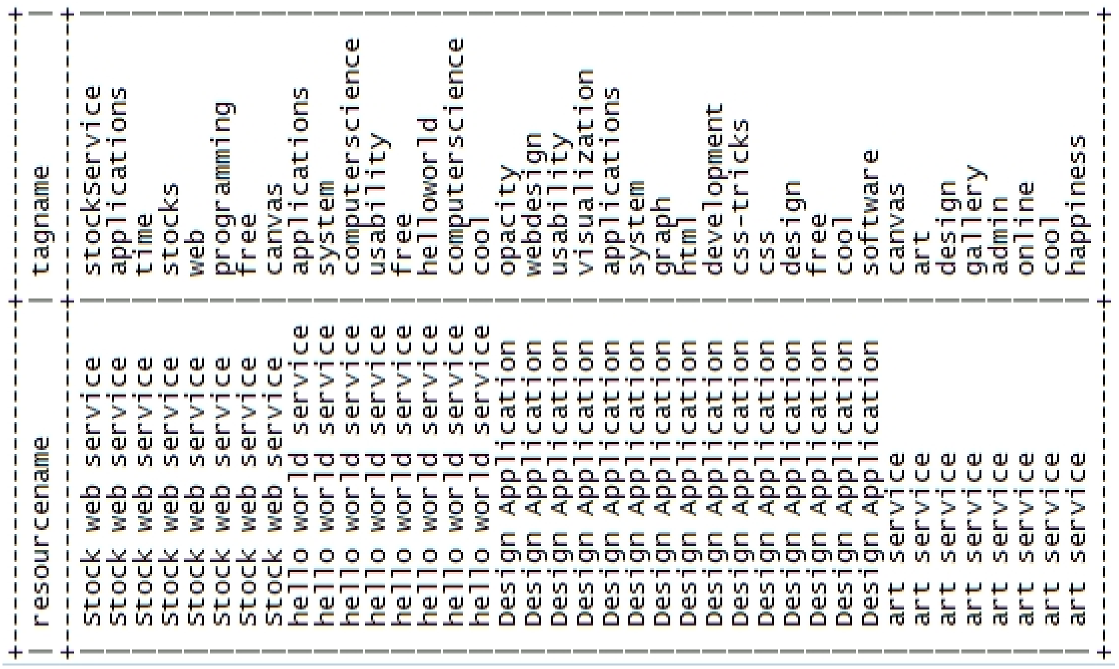
Example

In this section we will give a small example to give a better understanding on how the process to recommend services, based on a given query string, works. The example is kept small on purpose to be able to show structures and effects on the data structure more easily. We show the prerequisites of the system, perform a typical query, and try to explain the results of the visualization.

Setting

In Figure 3.1 we can see the database structure of the system, after a handful of users have attached tags to some resources. On the left hand side of Figure 3.1 you can see a number of total four resources, i.e. "stock web service","hello world service", "Design Application" and

"art service". On the right hand side we can see the tags that have been attached to the given resources. In total there are thirty- nine tags attached to the four resources, where some of the tags are attached to more than one resource.
In the current example the resource "Design Application" is tagged nearly as often as "stock web service" and "hello world service" together. "Art service" happens to have the same amount of tags attached as "stock web service" and "hello world service". The tags "free", "cool", "applications", "usability", and "system" are the most common.



```
+---------------------+-----------------+
| resourcename        | tagname         |
+---------------------+-----------------+
| stock web service   | stockservice    |
| stock web service   | applications    |
| stock web service   | time            |
| stock web service   | stocks          |
| stock web service   | web             |
| stock web service   | programming     |
| stock web service   | free            |
| stock web service   | canvas          |
| hello world service | applications    |
| hello world service | system          |
| hello world service | computerscience |
| hello world service | usability       |
| hello world service | free            |
| hello world service | helloworld      |
| hello world service | computerscience |
| hello world service | cool            |
| Design Application  | opacity         |
| Design Application  | webdesign       |
| Design Application  | usability       |
| Design Application  | visualization   |
| Design Application  | applications    |
| Design Application  | system          |
| Design Application  | graph           |
| Design Application  | html            |
| Design Application  | development     |
| Design Application  | css-tricks      |
| Design Application  | css             |
| Design Application  | design          |
| Design Application  | free            |
| Design Application  | cool            |
| Design Application  | software        |
| art service         | canvas          |
| art service         | art             |
| art service         | design          |
| art service         | gallery         |
| art service         | admin           |
| art service         | online          |
| art service         | cool            |
| art service         | happiness       |
+---------------------+-----------------+
```

Figure 3.1: Database View

Listing 3.1 shows the response of the system to an incoming similarity query, if there are suitable results for the given querystring. The XML structure that is presented here, serves as the input for the Tree Visualization applet.

Results

In Figure 3.2 and in Figure 3.3 we can see a comparison of the structures that result from the queries for the tags "system", "cool", and "free". It shows that (i) a resource that is comparable tagged more often, like the resource "Design Application" is likely to produce a branch on its own, from the root node.
(ii) Tags that are used comparable more often, like the tags in our example, i.e. "free", "system", "cool", "usability" on more than one resource, have an significantly higher relevance in the results, than tags that are used to describe only one resource, like the tag "gallery". In other words the more often a tag is used to describe different resources, the more importance it gains in the resulting structure. A tag that is attached to a resource more than once on the other hand,

experiences a loss of impact on the structure, because of the inverse document frequency, that has been mentioned in this section.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<node id="C00+C01+C06+C07+C02+C03+C04+C05">
  <node id="C00">
    <tag tagid="59">css</tag>
    <tag tagid="35">css-tricks</tag>
    <tag tagid="60">design</tag>
    <tag tagid="25">development</tag>
    <tag tagid="14">graph</tag>
    <tag tagid="23">html</tag>
    <tag tagid="28">opacity</tag>
    <tag tagid="83">software</tag>
    <tag tagid="82">visualization</tag>
    <tag tagid="34">webdesign</tag>
  </node>
  <node id="C01+C06+C07+C02+C03+C04+C05">
    <node id="C02">
      <tag tagid="77">free</tag>
    </node>
    <node id="C03">
      <tag tagid="81">cool</tag>
    </node>
    <node id="C04">
      <tag tagid="10">system</tag>
    </node>
    <node id="C05">
      <tag tagid="67">usability</tag>
    </node>
    <node id="C01+C06+C07">
      <node id="C01">
        <tag tagid="7">applications</tag>
      </node>
      <node id="C06">
        <tag tagid="18">canvas</tag>
        <tag tagid="37">programming</tag>
        <tag tagid="16">stocks</tag>
        <tag tagid="2">stockservice</tag>
        <tag tagid="12">time</tag>
        <tag tagid="33">web</tag>
      </node>
      <node id="C07">
        <tag tagid="43">computerscience</tag>
        <tag tagid="91">helloworld</tag>
      </node>
    </node>
  </node>
</node>
```

Listing 3.1: Example XML Response

(iii) A search for tags that have a similar importance, based on distribution and occurrence in the dataset, results in similar graph structures, c.f. the structure between the sub branch that contains the tags "system", "cool", "usability" and "free" and the nodes that contain the tags "applications", and "helloworld" and "computerscience" in Figure 3.2 and a similar structure in Figure 3.3(a) that differs in a sub branch structure, resulting from the attachment of the tag "cool" to the resource "Design Application" and finally in a similar structure in Figure 3.3(b) that slightly differs in a

lower branch structure, resulting from the attachment of the tag "free" to the resource "stock web service".
(iv) A resource that is tagged with totally different tags, than the tag that is searched for, i.e. has low to no similarity at all, is neglected in a result tree, i.e. in our example a relation to the resource "art service". Only in Figure 3.3(a) a structure that is related to this resource appears, since the queried tag is directly attached to the resource. In the other structures the resource "art service" is only reachable over the relation, that is produced based on the concurrent application of the tag "canvas" to the resources "stock web service" and "art service".

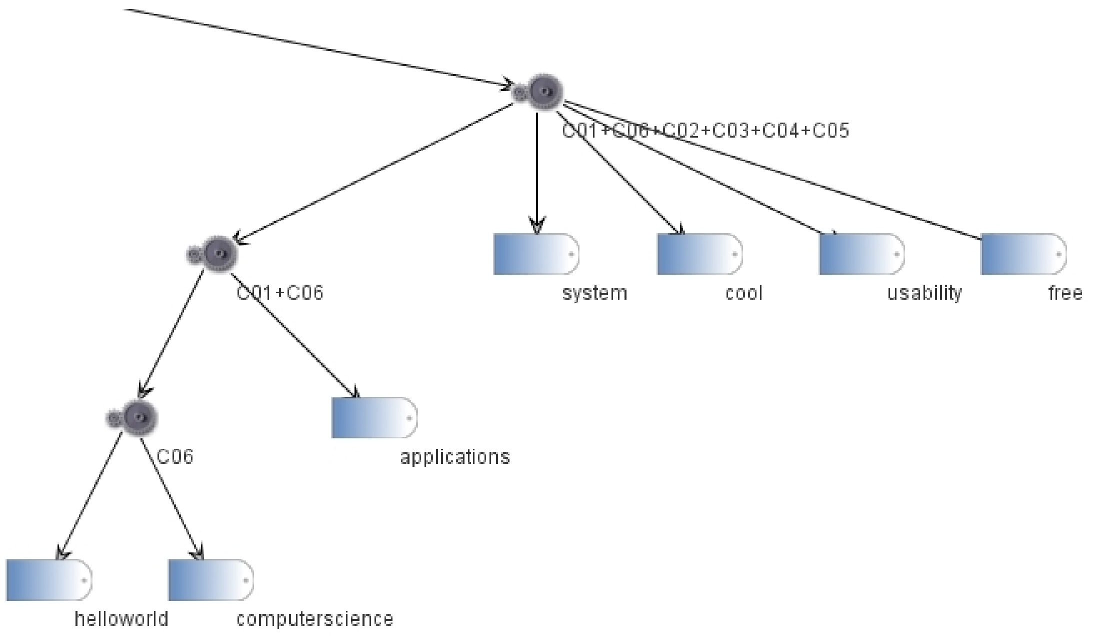
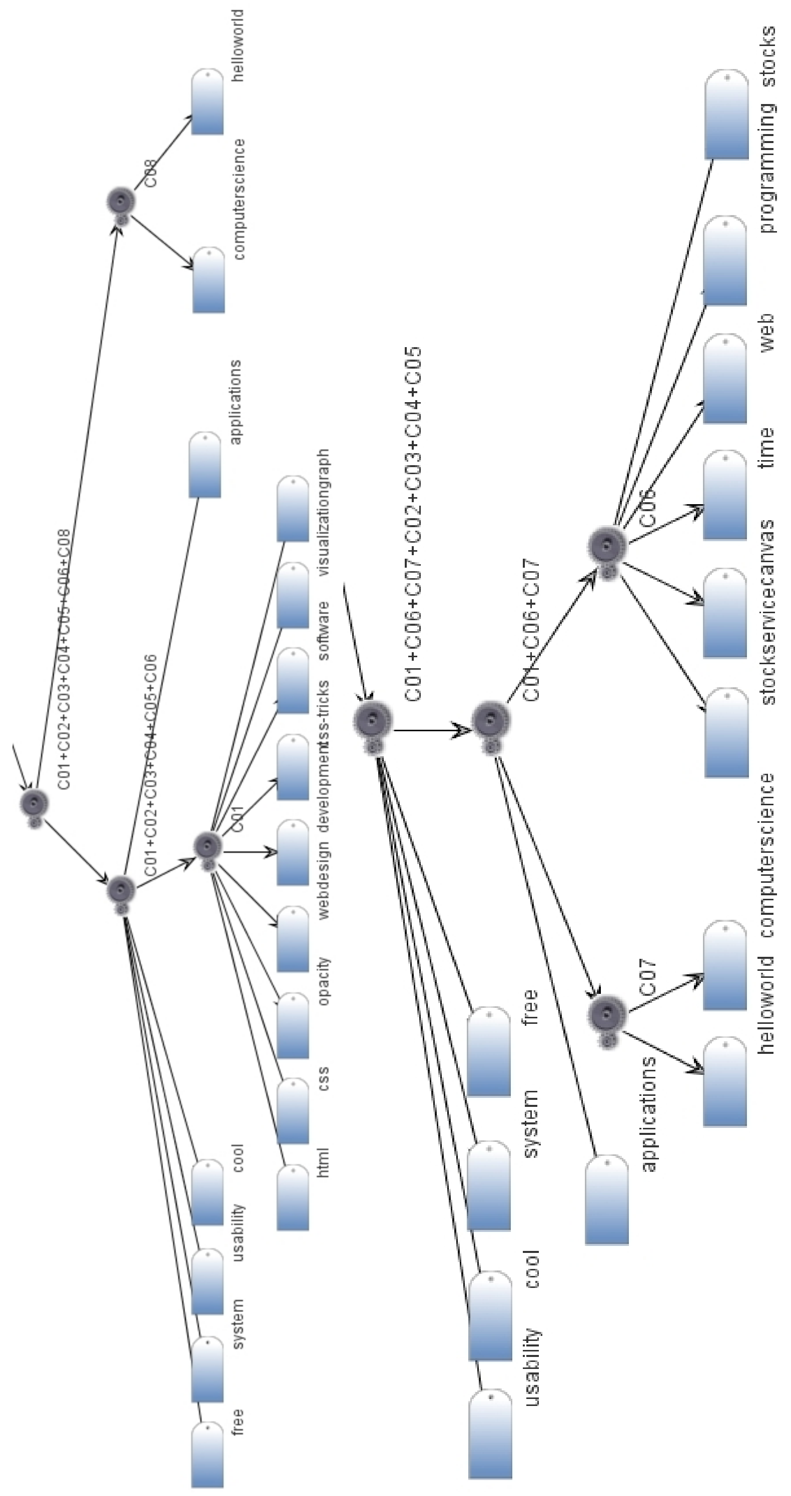


Figure 3.2: Search Results for tag "system"

(v) Tags that are more similar than other tags are closer located to the search query tag in the tree structure, i.e. sibling and/or parent child relation ship, but even relations that are not as equally strong are represented in the visualized graph.

Example Conclusion

Although the number of entries is kept very small, still we can experience, how the system is working on real data. We can clearly see the effects of the distribution of tags, and the co occurrence with other resources in the system, on the structural presentation of the similarity data. The data that is produced from search queries, results in the visualization of non trivial structures, due to the complexity of the relations among the items. With the interactive user interface, the user is given a suitable tool to discover and explore a vast amount of data.



(a) Search Results for tag "cool" (b) Search Results for tag "free"

Figure 3.3: Comparison of the Results

CHAPTER 4

Conceptual Approach and Use Case Description

In the previous chapters we could see what characteristics define the behavior of social networking applications. We identified the general conditions these systems are operating under and pointed out several scenarios and properties of the usage of tags. Further more we introduced the idea of a similarity algorithm, to improve common discovery strategies for resources, that are tagged in a collaborative network.
In this section we focus on the concepts we want to apply to our implementation, and how the real world challenges can be mapped to a suitable system architecture. We define the layers and components of the framework, and create various use case descriptions and sequence views, to demonstrate the features of the system and its properties. Additionally we introduce a complex use case scenario that will serve as a grounding for evaluating the applicability of the framework, in order to determine the systems boundaries and limitations, such and implementation has to cope with.

4.1 Concepts

In Figure 4.1 we identified the core elements of the system. Main interest is how the social interaction of a user community, that is exchanging data and information about web services that they have created themselves and intended to publish, can be optimized. The framework is the container that provides the means to store and retrieve the information about the services the users are uploading. Via the user interface, only accessible to users that are registered to the system, the various features of the system can be engaged.
Another main feature the system is providing, is the possibility to tag the content the users have uploaded, in order to identify the resources more easily for later retrieval. From a technological perspective the framework is responsible for persisting the associations between the created tags and the uploaded resources. Users apply the tags to describe the properties and the intended

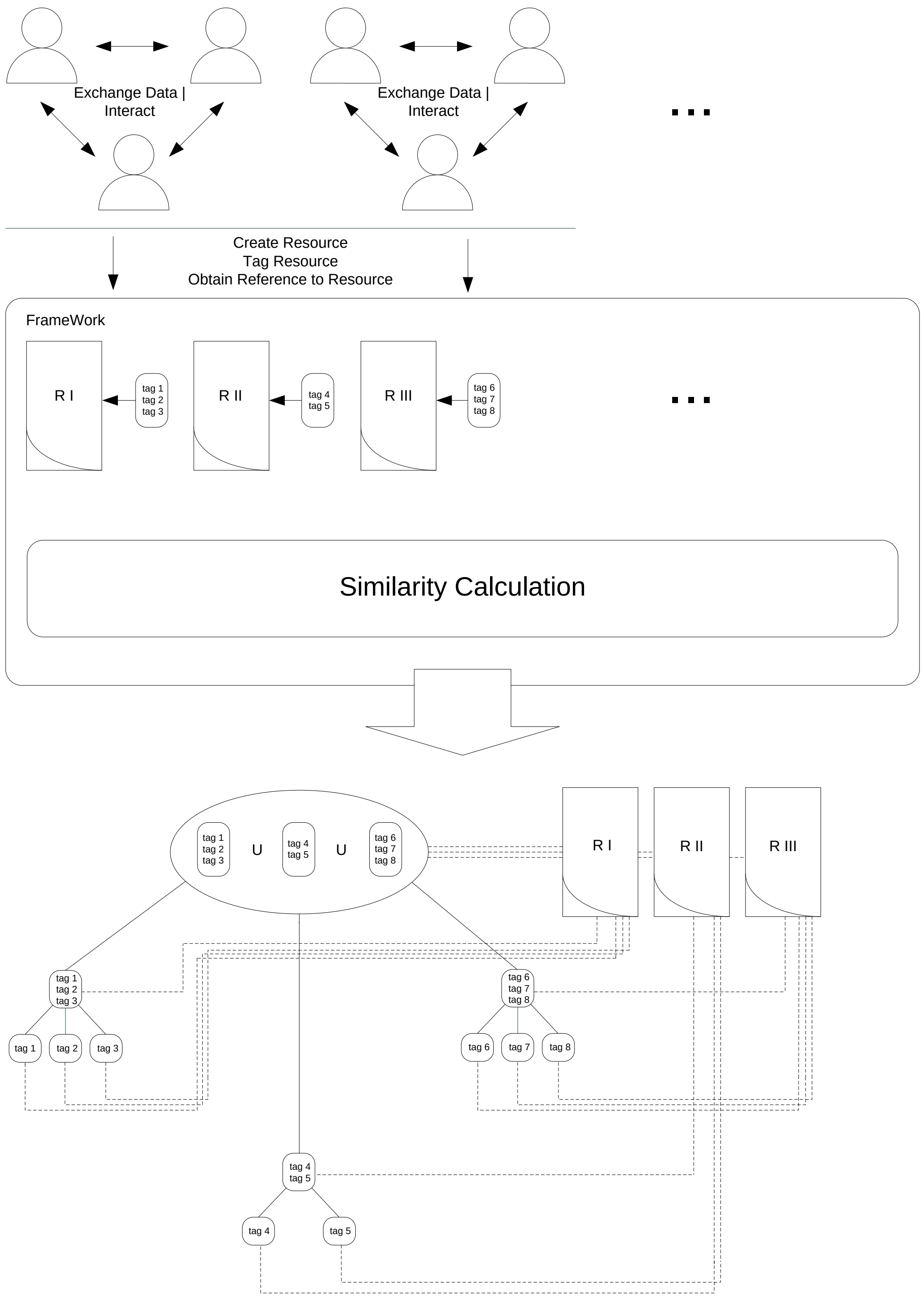


Figure 4.1: Identification of Elements of the Problem Domain

purpose of a service. This behavior results in a dense linkage between the users, the resources they tag and the applied tags respectively.
On the bottom of Figure 4.1 we can see the outcome of the similarity calculation in a schematic visualization, that is the result of the querying mechanism of the framework. Whenever a user is performing the search for a specific tag, the algorithm determines the semantic relationship between the queried keyword and the tags the community has applied to the resources, and creates a hierarchical structure from this information. On top of this data structure is a unification of the set of tags that have a semantic relevance to the queried keyword. Associated to each tag there is a list of resources that the tag is applied to. Moving down the branches of the tree structure the unification of the tags breaks up into single tag representations, that still contain a reference to the tagged resource.
Main purpose of this type of visualization is to provide a flexible yet information rich structuring for the generated data. Compared to common visualization mechanisms, e.g. like tag clouds, the data structure we present is very rich in information about the semantic relation between the tags, respectively the context of a resource.
With the provided mechanism the user can easily identify a suitable resource that is fulfilling his requirements and can obtain a reference to a service in a WSDL or an equivalent XML representation.
It is an essential feature of many successful social networking applications to navigate through a list of related users, as already stated in the description of characteristics. This buddy list functionality helps to foster the interaction between users and encourages them to exchange on similar interests. We equally integrate this feature in our network, as it helps to explore what resources a user has favored and what tags he has applied to. This concept allows users to browse through the existing set of users and resources and helps a user to explore resources that might be of equal importance to him.

4.2 System Architecture

Based on the concepts we have motivated in this chapter we want to introduce the architecture for the system to implement. The framework consists of three main layers that are connected and exchange data over various means of communication. In order to demonstrate the many facets of web services and the distributed environments they operate in, we try to integrate a variety of technologies to show the cooperative character of the architectural paradigm. Despite the technological diversity of the implemented components, we want to outline the benefits of such an approach. Each of these subsystems has limited functions and a rather simple architecture, only when the tasks they handle are combined, the benefit of the system increases to fulfill the requirements of a social networking community.
One of the main ideas of the web service technology is to separate the responsibility for a computational problem into smaller sub problems and distribute it to different peers. The approach to divide and conquer, that has been common in information science from early on, has been adopted in our framework as well, cf. [20] and [44] for a discussion of this principle.
Instead of handling the user requests in one single massive application, we can exploit the advantages of multiple, computational less complex implementations and make use of their

features, i.e. load sharing, reusability, portability, etc. to create a lightweight framework that can easily adapt to changes in the users behavior and in the applied business models.

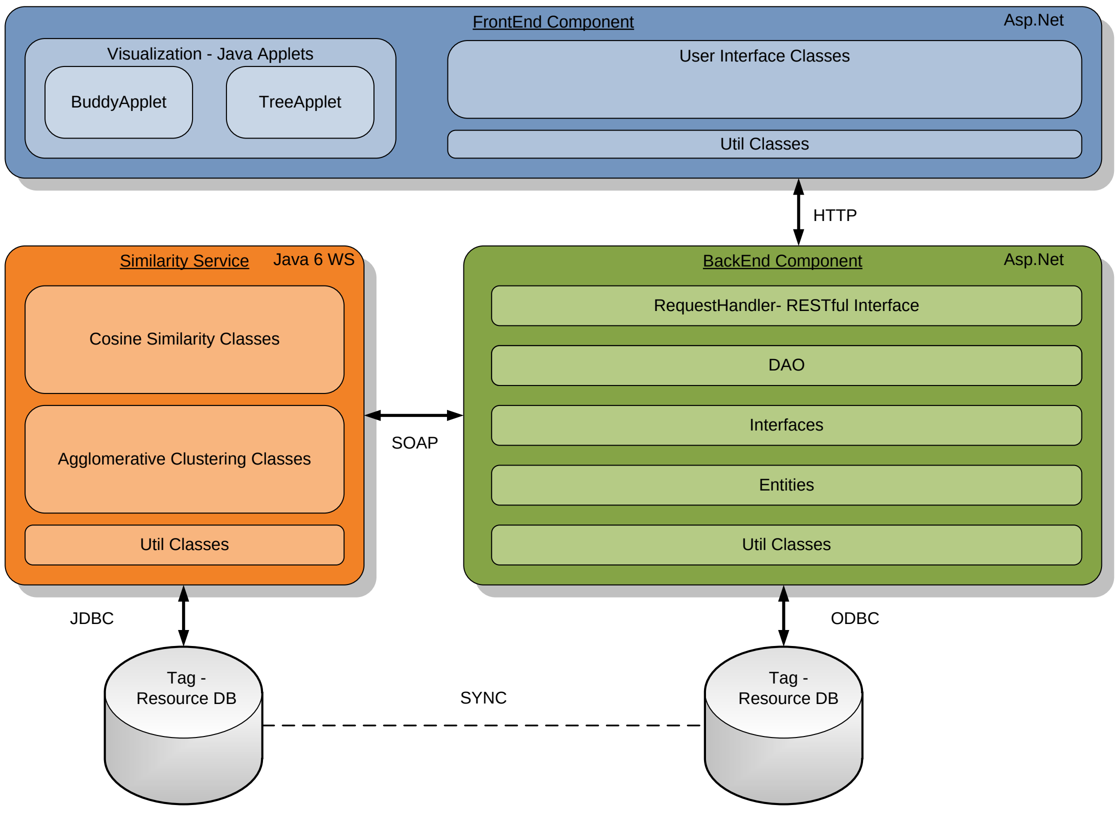


Figure 4.2: Conceptual Architecture of Service Tagging Framework

In Figure 4.2 we can see the conceptual architecture of the framework including the type of technology that is in use for each of the components and how these layers communicate. The FrontEnd is primarily the component the user is interacting with. Implemented in Asp.Net it is providing means to visualize and manipulate the data basis. Over HTTP the FrontEnd is communicating with the BackEnd, that is the core component, dealing with persistence and forwarding of requests. The Similarity Service that is implemented in Java 6 and publishing its capabilities as a web service is the component, that is responsible for upholding the necessary infrastructure to perform a similarity calculation on an incoming search request. Over the means of SOAP the request and the reply is transfered from the BackEnd to the Similarity Service and back. To reduce the amount of communication between these two components and minimize the network load, the Similarity Service is pulling the information that is necessary in advance from a mirror of the original database where the structural information is stored. Eventually the restructured data is transfered to the FrontEnd where the TreeApplet is used to visualize the dependency among the elements.

4.3 Use Case Description

The description of the use cases of the system to be designed can be split into three parts. The first part covers the description of basic use cases that are common in most systems, that allow similar functionality. These use cases are necessary for a fluid work flow and are crucial for the basic social network functionality. The area of application of these use cases range from simple login and logout functionality to more complex tasks as creation of a new user account, creation of a resource and such. The second part deals with more complex functionalities that are specific to the structure of social networks or folksonomies. Mainly navigation through the data structures that evolve through search mechanisms, is handled in this subsection. The third part covers use cases, that have to be performed to keep the data basis consistent and up to date. Since similarity computation cannot be performed on a dataset that is computed in real time, but has to be performed on a snapshot of the data basis, the system automatically has to perform several maintenance tasks, that are explained in this subsection.
The numbering of the use cases is completely independent of the document structure and solely serves the purpose of identifying a specific use case and its description.

Actors

Throughout the system we have identified four actors that interact with the framework, this part will briefly describe their role in the system and capabilities.

User- unregistered A User is a human actor that is interacting with the framework. As a precondition to carry out any of the use cases, a user has to have access to an internet connection and a Java script capable browser. Unregistered users cannot log on to the system, which is the pre-condition for any other use case, and can therefore perform solely a single use case; 2.1 Create a new user account. As a postcondition of performing use case 2.1 the status of the actor is changed from an unregistered user to a registered user.

User- registered Registered users already are in possession of an user account to the framework, and can log on to the system and perform use case that are available to this type of actor. The execution of use case 2.3 yields to a deletion of the profile data of a user, and changes the status of a user back to unregistered. The registration process has to be repeated in order to regain a status where interact with the system is possible.

System- passive The "System passive" actor represents the system, as an environment that is accepting requests and commands from the user via the user interface. It is solely responding to requests initiated by the user and is not commencing any work flows itself.

System- active The "System active" actor describes a part of the framework, that is actively participating in the work flow. Since the system is designed to actively pull information, about the tags that have been applied to resources, from the database, we introduce this part of the framework as a actor by itself. Controlled by a timing thread this "System active" constantly is busy, keeping the similarity structure up to date and consistent with the requests.

Basic Use Cases

This subsection covers the description of typical use cases that are common in many social networks. Ranging from basic CRUD[1] functionality to search mechanisms. A key feature of social networks is to maintain list of known users, for simplified access to more personal information about a user.

1. Standard Use Cases

1.1	Login
Goal:	Validates the user credentials, and logs in a user into the system.
Summary:	The user enters his credentials, the system validates the data, and logs in the user.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to be registered(see use case 2.1).
Description:	
1.	Via the user interface(UI) the user navigates to the login screen.
2.	The system waits for user credentials.
3.	The user enters his credentials and submits the data.
4.	The system validates the data.
5.	The system grants access and displays the welcome screen.
Alternatives:	
4.a	The entered user credentials are invalid.
5.a	The system does not display the welcome screen and asks to re-check for a valid username password combination.
PostCondition:	The user is logged in and can make use of the system.

1.2	LogOut
Goal:	Logout the User.
Summary:	At any time the user can use the logout button to leave the system. Any previously initiated action, is cancelled.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see use case 1.1).

[1]CRUD is an acronym that abbreviates the basic storage functions (C)reate, (R)ead, (U)pdate and (D)elete performed on a given set of data.

Description:	
1. 2. 3.	Via the user interface(UI) the user selects the logout button. The system logs out the user. The system presents the login screen.
Alternatives:	
	none.
PostCondition:	The user is logged out and cannot use any part of the system.

2. User- specific Use Cases

User specific use cases refer to functions of the system that involve users and their profile details. Additional this section covers the use cases that deal with buddy list functionality , i.e. user to user interaction. In Figure 4.3 you can see how two users interact to outline their social relation. This not only has a huge social impact, but also improves the usage of the system, as both users have direct access to personal information only visible to their buddies.

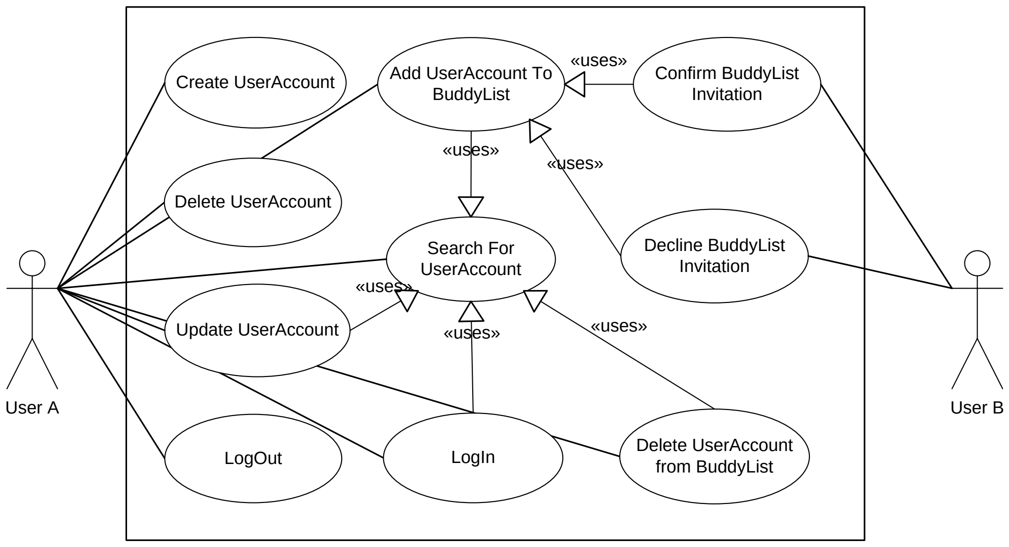


Figure 4.3: User Specific Use Cases Overview

2.1	Create UserAccount
Goal:	A new User Account identified through an unique username is created.
Summary:	The unregistered user decides to join the community of active users and creates an account. He chooses an unique username, enters personal information like lastname, firstname emailaddress and password and submits the information to the system. Additionally a user can upload an image to represent his account, although this is a n optional task, since every account is already equipped with an image. Once the uniqueness of the chosen username is verified by the system, a new user account is created and the user can start participating in the community.

Dependency:	
Actor(s):	User unregistered(primary), System passive(secondary)
PreCondition:	The username that is chosen is unique. The system has to take care that the chosen username is not used twice.
Description:	
1.	A not registered user signals to create a new user account.
2.	The not registered user enters the necessary personal data in the web form, including firstname, lastname, email address, and desired username as well as password.
3.	The system checks for availability of the chosen username.
4.	The username is registered for the user.
5.	An email is sent to email address of the newly registered user, confirming the creation of an user account.
Alternatives:	
2.a	The not registered user omits one or more of the necessary fields for registration.
3.a	Step 2 of the main sequence is performed until all necessary data is gathered.
3.b	The username is not available, as it is already in use by another user.
4.b	The system suggests to choose a different username.
5.b	Step 2 of the main sequence is performed until all necessary data is gathered.
2.c	The user decides to upload a new image file.
3.c	The system verifies the file extension.(Possible extensions are .jpeg, .gif, .png)
3.c.I	The system cannot verify the file extension.
4.c.I	Step 2.c of the main sequence can be performed until the data can be verified.
4.c	The system stores the binary data of the image.
PostCondition:	A new user account is created and the user can log on to the system and start actively participating in community activities.

2.2	Update UserAccount
Goal:	The data of an existing user account is modified.
Summary:	An existing user is logging in to the system, is choosing to modify his account details, and submits the changed data to the system.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1).
Description:	
1.	Via the user interface(UI) the user navigates to the section, where he can modify his personal information.
2.	The system displays the information that is currently stored about the user, including the an uploaded image.

3.	The user enters the new data he desires to change. (Possible data includes:firstname, lastname, email address, or selects a new image to upload)
4.	The user submits the changed data to the system.
5.	The system stores the updated data.
3.a	The user uploads an image file that has a file extension that cannot be verified by the system.
4.a	The system informs the user that the file extension is incorrect.
5.a	The data currently stored about a user is not changed.
PostCondition:	The data of an existing user is changed and stored in the database.

2.3	Delete UserAccount
Goal:	The data of an existing user account is deleted.
Summary:	An existing user is logging in to the system, and is choosing to delete his personal information. After executing this step the user is no longer registered and its status changes to user- unregistered. The user then can no longer log in and use the functionality of the system.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see use case 1.1)
Description:	
1.	Via the user interface(UI) the user navigates to the section, where he can modify his personal information.
2.	The system displays the information that is currently stored about the user.
3.	The user selects the interface mechanism to delete his user account.
4.	The system displays a warning and asks for confirmation.
5.	The user confirms to delete his user account.
6.	The system deletes the user account data.
7.	The system automatically logs out the user. (see use case 1.2)
Alternatives:	
5.a	The user declines to delete his user account.
6.a	The system returns to the welcome page, no data is deleted.
PostCondition:	The user account of the user is deleted, he therefore can no longer log in (see use case 1.1) to the system. The status changes from user- registered to user- unregistered.

2.4	Search for UserAccount
Goal:	The system presents a list of user accounts meeting previously specified criteria.
Summary:	The user chooses to search for other user accounts to add them to his list of known users(i.e. BuddyList). The user enters search criteria like username. The system presents a list of users that meet the given criteria.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1).
Description:	
1.	Via the user interface(UI) the user navigates to the section, where he can search for other users.
2.	The user enters data that is used to specify the search results (e.g. username).
3.	The system returns a list of user accounts that meet the specified search criteria.
Alternatives:	
3.a	No entries meet the given search criteria. The system informs the user that no results could be found.
4.a	Step 2 of the main sequence is performed until the user aborts this use case.
PostCondition:	The user is presented a list of user accounts he can add to his buddy list.

2.5	Add UserAccount to BuddyList
Goal:	Initializes the adding of a user account to a list of known users(i.e. BuddyList) for faster access to their profile and contributions.
Summary:	The user initiates contact with a user account returned as a search result from the previous use case Search for UserAccount (see use case 2.4). After adding the user to his list of known users, the system generates a invitational message and awaits the decision of the invited user to either accept or decline the invitation.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see use case 1.1). The user has to perform use case Search for UserAccount(see 2.4).
Description:	
1.	The user select an entry from the search results performed in the precondition.
2.	The system displays a detailed account information of that user account.
3.	The user selects to add the user account to the list of known users(i.e. BuddyList)
4.	The system generates a message inviting the selected user to join the users list of known users(i.e. BuddyList)

5.	The system informs the user that until the user has not confirmed his invitation(see use case 2.7) the user account is not added to the list of known user accounts(i.e. BuddyList)
6.	The system returns to the list of known user account(i.e. BuddyList)
Alternatives:	
3.a	The user does not select to add the user account to the list of known users(i.e. BuddyList)
4.a	The user returns to the list of search results.
PostCondition:	An Invitation to join the list of known users is issued to a specific user. The invited user has the possibility to either accept the invitation(see use case 2.7) or decline the invitation(see use case 2.8).

2.6	Delete UserAccount from BuddyList
Goal:	Delete a previously added user account from a users list of known users(i.e. BuddyList).
Summary:	Delete a previously added user account from a users list of known users(i.e. BuddyList).
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see use case 1.1). The user account has been added to the list of known users using use case Add User To BuddyList(see use case 2.5) and has accepted the invitation performing use case confirm buddy list invitation (see use case 2.7).
Description:	
1.	Via the user interface(UI) the user navigates to the section, where he can see his list of known users(i.e. BuddyList).
2.	The user selects a specific user.
3.	The system displays a detailed view on the data of the user account.
4.	The user selects the interface mechanism to delete the user account from his list of known users(i.e. BuddyList).
5.	The system displays a warning and asks from confirmation.
6.	The user confirms to delete the user account from his list of known users(i.e. BuddyList).
7.	The system deletes the user account from the users list of known users.
8.	The system returns to a view of the list of known users. The data has been deleted.

Alternatives:	
6.a	The user declines to delete the user account from his list of known users(i.e. BuddyList).
7.a	The system returns to a view of the list of known users(i.e. BuddyList). No data has changed.
PostCondition:	The specified user account entry has been deleted from the users list of known users.

2.7	Confirm BuddyList Invitation
Goal:	Accepts an invitation to join the list of known users of a the user that started the invitation process(see Add User To BuddyList use case 2.5)
Summary:	After having received an invitation from a user to join its list of known users(i.e. BuddyList), the invited user decides to accept the invitation. The system generates a message and from that moment on the user that asked for confirmation is able to view detailed information about the user that joined its list.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1). A user has performed use case Add User To BuddyList(see use case 2.5),and is awaiting confirmation.
Description:	
1.	The system indicates that a user has requested to add this user to his list of known users.
2.	The user views the invitational message.
3.	The user accepts the invitation.
4.	The system sends a message to the user that initiated the contact that the user has accepted his invitation.
5.	The system suggests to add the user that initiated the contact to the list of known users.
6.	The user accepts to add the user that initiated the contact.
7.	The system sends an invitational message to the user that initiated the contact.
Alternatives:	
3.a	The user declines the invitation. See use case Decline BuddyList invitation (use case 2.8).
6.b	The user declines to add the user that initiated the contact, to his list of known users.
PostCondition:	From the moment the user accepts the invitation to a list of known users, he grants access to a detailed view on his personal profile.

2.8	Decline BuddyList Invitation
Goal:	Declines an invitation to join the list of known users(i.e. BuddyList) of a user that started the invitation process(see Add User To BuddyList use case 2.5).
Summary:	After having received an invitation from a user to join its list of known users (i.e. BuddyList), the invited user decides to decline the invitation. The system generates a message, and the user that asked for confirmation is not able to view detailed information about the invited user.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1). The user has created the specific resource(see use case Create Resource 4.1).
Description:	
1.	The system indicates that a user has requested to add this user to his list of known users.
2.	The user views the invitational message.
3.	The user declines the invitation.
4.	The system sends a message to the user that initiated the contact that the user has declined his invitation.
Alternatives:	
3.a	The user accepts the invitation. See use case Confirm BuddyList Invitation (see use case 2.7).
PostCondition:	The user did not join the list of known users from the user that initiated the contact. The user did not grant access to a detailed view on his personal profile.

3. Tag- specific Use Cases

Tag specific Use cases refer to functions of the system that involve tags and the resources they are applied to. These functions are tightly coupled to the manipulation of resource data since a tag cannot exist without being attached to a resource.

3.1	Create Tag
Goal:	A new Tag is created and is associated to a resource.
Summary:	The user searches for a resource(see use Case Search for Resource 4.4) chooses to modify a specific resource and adds a tag to the resource.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)

PreCondition:	The user has to log in using his username and his password (see usecase 1.1). the user has to perform a search for a resource(see use case Search for Resource 4.4). Tags can only be created, associated to a certain resource.
Description:	
1.	The user performs use case Search for resource 4.4.
2.	The user selects a resource from the list of search results.
3.	The user selects the user interface mechanism to add a tag to the specific resource.
4.	The user enters the name of the tag and submits the data.
5.	The system stores the created data.
6.	The system displays the newly created tag.
Alternatives:	
	none.
PostCondition:	A Tag associated to a resource is created.

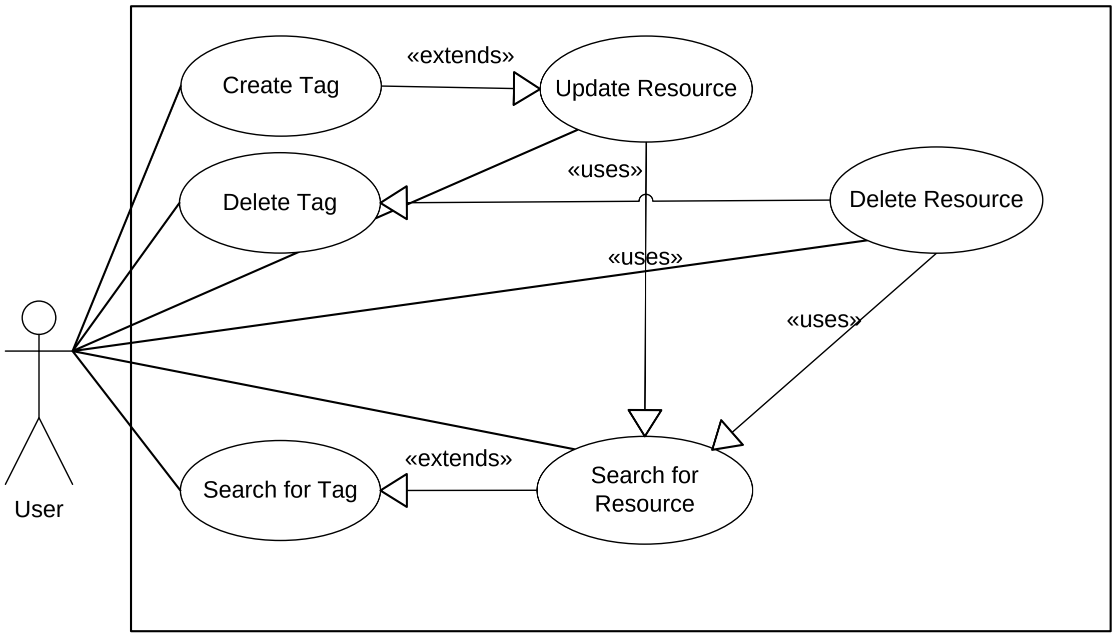


Figure 4.4: Tag Use Cases Overview

3.2	Delete Tag
Goal:	An existing tag is deleted.
Summary:	The creator of a resource is logged in and chooses to delete a tag. After confirming the deletion, the data of the tag is deleted.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)

PreCondition:	The user has to log in using his username and his password (see usecase 1.1). The user has created the specific tag(see use case Create Resource 4.1), or the user that created the resource the tag is associated to, deletes the corresponding resource (see use Case Delete Resource 4.3).
Description:	
1.	Via the user interface(UI) the user navigates to the section where he can modify the details of a resource.
2.	The system displays the information that is currently stored about the resource including all associated tags.
3.	The user selects the interface mechanism to delete a tag he created.
4.	The system displays a warning and asks for confirmation.
5.	The user confirms to delete the tag.
6.	The system deletes the tag.
Alternatives:	
3.a	The user tries to delete the data of a tag that was created by another user.
4.a	The system displays the information that is currently stored about the tag, but the user is not able to delete the available data.
5.b	The user declines to delete the tag.
6.b	The system return to the start page, no data is deleted.
PostCondition:	The specific tag is deleted.

3.3	Search for Tag
Goal:	The system presents a list of tags meeting previously specified criteria.
Summary:	The user enters various keywords to identify a tag. The system displays a list of tags that meet the given criteria.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1).
Description:	
1.	Via the user interface(UI) the user navigates to the section, where he can search for tags.
2.	The user enters data that is used to specify the search results (e.g. tag name, or resource name)
3.	The system returns a list of resources that meet the specified search criteria.
Alternatives:	
3.a	No entries meet the given criteria. The system informs the user that no results could be found.
4.a	Step 2 of the main sequence is performed until the user aborts this use case.
PostCondition:	The user is presented a list of tags he can view the details of or view the associated resource.

4. Resource- specific Use Cases

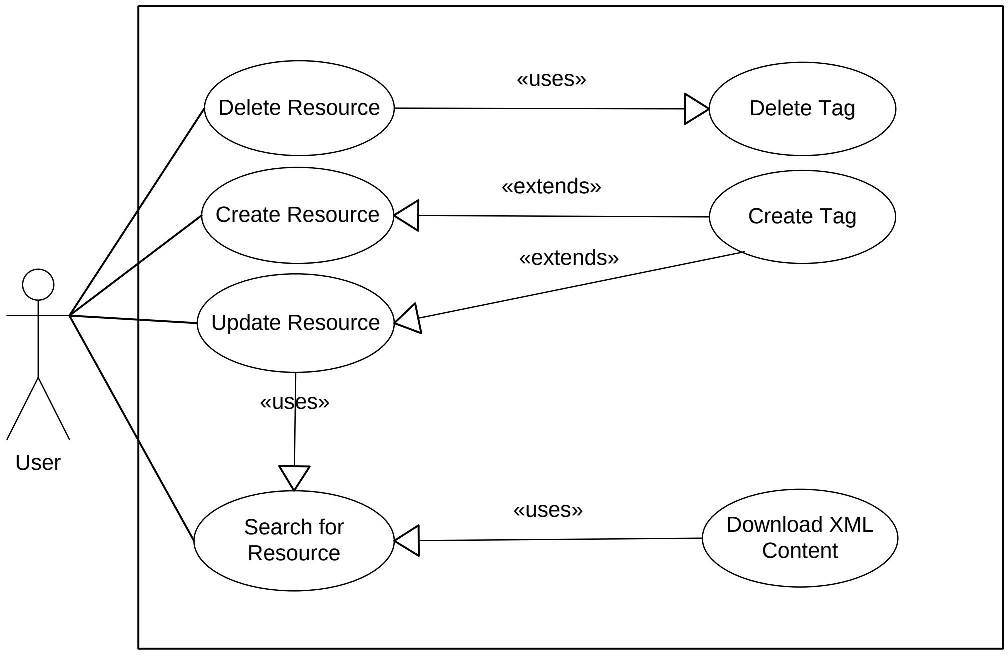


Figure 4.5: Resource Use Cases Overview

4.1	Create Resource
Goal:	A new resource is created.
Summary:	A logged in user specifies the data of a new resource, and the system creates a new resource entry.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1).
Description:	
1.	Via the user interface(UI) the user navigates to the section, where he can create a new resource.
2.	The user enters the necessary data to create a new resource, like name and a short description, as well as the corresponding xml description file. and submits the data.
3.	The system stores the created data.
4.	The system displays the newly created resource.
Alternatives:	
	none.
PostCondition:	A new resource is created. From this moment users can add tags to the resource.

4.2	Update Resource
Goal:	The data of an existing resource is modified.
Summary:	A user is logged in to the system, is choosing to modify the details of a resource he created, and submits the changed data to the system.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1).The user has created the specific resource(see use case Create Resource 4.1) he wants to modify.
Description:	
1.	Via the user interface(UI) the user navigates to the section where he can modify the details of a resource he created.
2.	The system displays the information that is currently stored about the resource.
3.	The user enters the new data he desires to change. (Possible data includes: name, description)
4.	The user submits the changed data to the system.
5.	The system stores the updated data.
Alternatives:	
1.a	The user tries to modify the data of a resource that was created by another user.
2.a	The system displays the information that is currently stored about the resource, but the user is not able to alter the available data, because he did not create the specific resource.
PostCondition:	A new resource is created. From this moment users can add tags to the resource.

4.3	Delete Resource
Goal:	An existing resource is deleted.
Summary:	The creator of a resource is logged in and chooses to delete a specific resource. After confirming the deletion, the data of the resource is deleted.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1). The user has created the specific resource(see use case Create Resource 4.1).
Description:	
1.	Via the user interface(UI) the user navigates to the section where he can modify the details of a resource he created.
2.	The system displays the information that is currently stored about the resource.

3.	The user selects the interface mechanism to delete the resource.
4.	The system displays a warning and asks for confirmation.
5.	The user confirms to delete the resource.
6.	The system deletes the resource data and all related tags.
Alternatives:	
1.a	The user tries to delete the data of a resource that was created by another user.
2.a	The system displays the information that is currently stored about the resource, but the user is not able to delete the available data.
5.b	The user declines to delete the resource.
6.b	The system returns to the start page, no data is deleted.
PostCondition:	The specific resource and all tags that are related to that resource are deleted.

4.4	Search for Resource
Goal:	The system presents a tree structure of resources meeting previously specified criteria, respectively have a similarity relation to the search criteria.
Summary:	The user chooses to search for resources to view their details. The user enters search criteria. Based on a similarity of the keywords that the user entered and a relation of tags to the resource, the system presents a list of resources that meet the given criteria.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1).
Description:	
1.	Via the user interface(UI) the user navigates to the section, where he can search for resources.
2.	The user enters data that is used to specify the search results (e.g. tag name), or other keywords that might describe the resource.
3.	The system returns a tree structure of resources that meet the specified search criteria.
Alternatives:	
3.a	No entries meet the given criteria. The system informs the user that no results could be found.
4.a	Step 2 of the main sequence is performed until the user aborts this use case.
PostCondition:	The user is presented a list of resources he can view the details of.

4.5	Download XML Content
Goal:	The XML content of a resource, including any associated tags is downloaded from the system.
Summary:	A logged in user views the details of a resource. Over a designated interface mechanism the user is able to download the XML content of the resource and save it on his hard disk.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1).
Description:	
1.	Via the user interface(UI) the user navigates to the section, where he can view his resources.
2.	The user views the details of a specific resource.
3.	The user signalizes the intention to download the content of the specific resource.
4.	The system ask for a location to store the content.
5.	The user specifies the location to store the content and the system starts to transmit the data.
Alternatives:	
6.a	The user does not specify a location to download the content of the resource.
7.a	The download of the content is aborted.
PostCondition:	The XML content that is associated to a resource is downloaded and stored on the hard disk or a similar storage location.

Advanced Use Cases

This part covers the description of advanced functionalities more specific to the navigation through social networks. The structure that evolves from the participating entities, users, resources and applied tags, spans a graph that can be traversed to discover relationships of the elements.

5.1	Create Message
Goal:	The user writes a personal message to another user in order to foster a relationship.
Summary:	Either for communication about a certain topic or to invite a user to ones BuddyList, the user can send a message to another user.
Dependency:	
Actor(s):	User registered(primary), User registered(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1).
Description:	

1.	Via the user interface(UI) the user navigates to the section, where he can compose a new message.
2.	The user can select the recipient from a list of users.
3.	The user enters a subject and the text he wants to send.
4.	The user submits the message to the selected user.
Alternatives:	
3.a	The message is a reply to a previously received message, and the subject is already inserted automatically.
PostCondition:	A message is sent to the selected user. When the recipient logs in into his account, the system automatically displays the message.

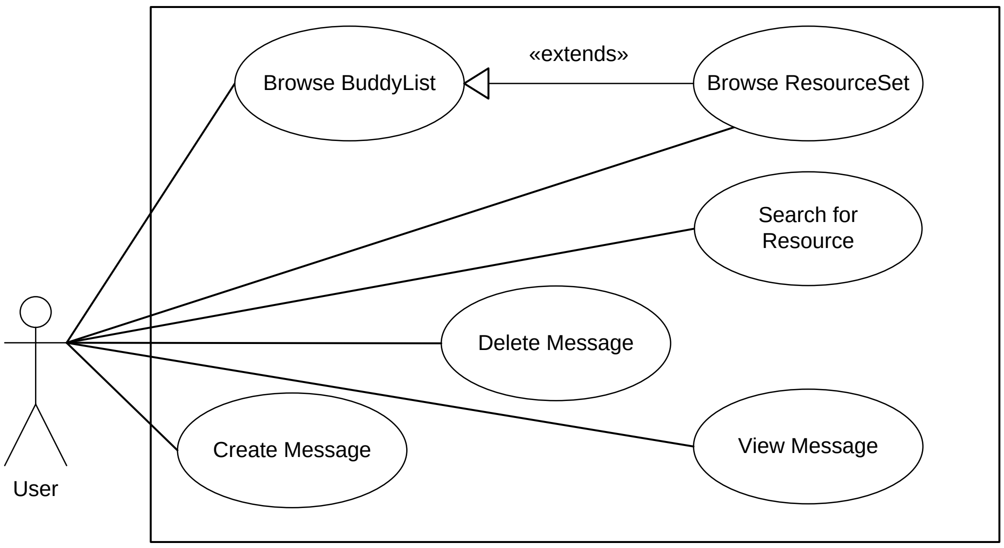


Figure 4.6: Advanced Use Cases Overview

5.2	Delete Message
Goal:	The user deletes a message from the list of incoming messages.
Summary:	Once a read message is no longer of use for a user. He is able to delete the message by selecting it from the list of incoming messages.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1). The user has to have received a message from another user.
Description:	
1.	Via the user interface(UI) the user navigates to the section, where he can view incoming messages.

2.	The user selects a message he wants to delete.
3.	The system asks for confirmation.
4.	The user confirms his decision to delete the selected message.
5.	The system deletes the selected message.
Alternatives:	
4.a	The user declines to delete the message.
5.a	The message is not deleted.
PostCondition:	A previously received message is deleted.

5.3	View Message
Goal:	The user views the content of a message.
Summary:	Once a user is logged in he is presented a list of incoming messages automatically. Via the list representation he is able to read the content and who sent the message.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1). The user has to have received a message from another user.
Description:	
1.	Via the user interface(UI) the user navigates to the section, where he can view incoming messages.
Alternatives:	
	none.
PostCondition:	The user can reply to the message via the context menu.

6.1	Browse Resource- Set
Goal:	The user browses through a set of resources, to view their details and to mark them as favoured.
Summary:	Having found a resource by either browsing through ones BuddyList or browsing through a similarity structure, the user can view the details of a resource.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1).
Description:	
1.	The user selects a resource entry from the graph of resources
2.	The system displays a list of tags associated with the selected resource.
Alternatives:	
	none.
PostCondition:	The user is navigating through a set of resource.

7.1	BrowseBuddy List
Goal:	The user navigates in his list of known users(i.e. BuddyList) to discover what user marked which resource with what tag.
Summary:	Having added some users to the list of known users(i.e. BuddyList) the user is able to have a detailed view on what resources are tagged by a specific user. Having displayed a user and the details about the tags and resources the user modified.
Dependency:	
Actor(s):	User registered(primary), System passive(secondary)
PreCondition:	The user has to log in using his username and his password (see usecase 1.1).
Description:	
1.	Via the user interface(UI) the user navigates to the section, where he can have the list of known users(i.e. BuddyList) displayed.
2.	The user can select users and view their details.
3.	The system displays the details about tags that the selected user used and resources that the user tagged.
Alternatives:	
4.a	The user selects a resource from the list of resources that the user tagged.
PostCondition:	The user has viewed details about how a known user has tagged a resource what tags he has used to describe a certain resource.

PreComputational Use Cases

This part covers tasks that are performed on a regular basis, but with no interaction of a user, neither registered nor unregistered, to keep the data basis up to date and consistent with the modifications that the users have performed on the stored data. Because of the very time consuming algorithm, to create a similarity tree structure out of the tags and resources, it is not possible to include changes made by the users in real time. Instead the requests submitted by a user are executed on a snapshot of the data basis and the following use cases are performed regularly to maintain a consistent data basis. The following use cases are substance to the actor System- active.

8.1	Update Similarity Structure
Goal:	Timing thread to ensure the similarity structure is recalculated on a regular basis.
Summary:	Since it is a very time consuming process to generate a similarity tree and it is not possible to add single elements in an already created tree structure. This timed thread guarantees that the data is updated regularly.
Dependency:	
Actor(s):	System active(primary)

PreCondition:	Users have created one or more resources(see use case Create Resource 4.1), and have applied several tags(at least one) to each of the created resources to describe a resource.
Description:	
1.	The system connects to the shared database, where the data about created resources and applied tags is stored.
2.	The system performs use case calculate similarity structure(see use case 9.1)
3.	The system updates the internal data structure, and keeps the data in memory.
Alternatives:	
	none.
PostCondition:	A similarity structure is created and kept in memory for further calculation.

9.1	Calculate Similarity Structure
Goal:	The system calculates a new similarity structure, based on the data of the shared database.
Summary:	The system queries for the data that is entered by the users. It calculates the similarity among the tag and resource elements and constructs a data structure. The newly calculated structure is stored in memory, to provide a data structure for the search, respectively the recommendation mechanisms.
Dependency:	
Actor(s):	System active(primary)
PreCondition:	The system has to perform use case 8.1 Update Similarity Structure.
Description:	
1.	The data from the database is parsed to a structure used for further processing.
2.	The structure is used to create a vector representation of the data.
3.	The vector representation is transformed to a matrix that holds the similarity measurements between on document and another.
4.	The matrix is transformed according to the indexing method.
5.	A Search object is created.
6.	Properties like the identifiers of the documents and the applied tags are added to the search object.
7.	The similarity measurement is chosen and applied to the structure.
8.	The structure is set up and incoming requests can be queried against this structure.
Alternatives:	
	none.
PostCondition:	A structure to calculate the similarity is created from scratch and held in memory.

4.4 Scenario of Tagging Framework

To create a more realistic usage scenario for the implemented system, we describe a use case that is compound from some previous described use cases, to demonstrate the usage and performance of the framework. This composite use case is designed to address the typical functions of the system. Besides its representational character this use case also serves as a grounding to evaluate the runtime of the similarity algorithm, as well as the overall performance of the system.
Based on the data we can gather from this more complex task, we aim to identify the parameters of the system that can be used to improve the scalability and the overall performance of the framework, in a possible extension of the system.

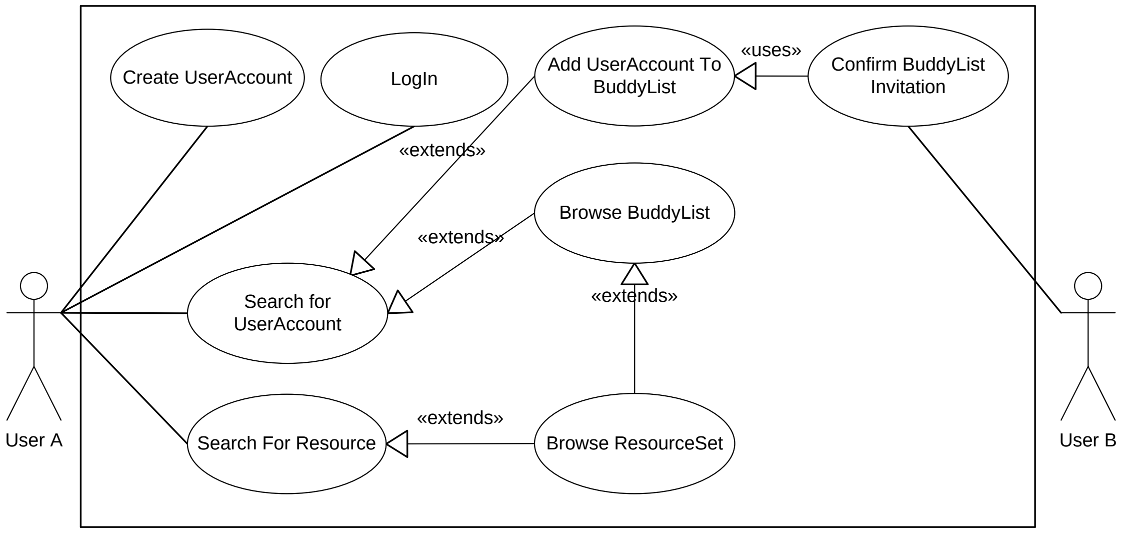


Figure 4.7: Composite Use Case Overview

Composite Use Case

Goal:	The goal of the composite use case is to demonstrate a typical work flow when interacting with the system. This scenario is set up to identify the responsibilities and possibilities of the system.
Summary:	A user that is not yet registered to the framework, decides to join the community that is publishing and tagging their web services. Tagging a web service helps other users to identify the properties and the area of application of a service, yet increases the chances for a service to be discovered. Via the login form a non registered user can reach the registration page, that allows a user to create a new user account. After deciding for a userName, the user enters some personal information like lastname, firstname, and email address, chooses a password and optionally a userimage to upload and submits the data to the system. Once the uniqueness of the username is verified by the system, a new user account is created and the user can join the community.

	The user, from now on referred to as User A, has decided to search for services that contain the functionality to make reservations for movies, since User A wants to integrate such a service in his work flow. The first step to achieve this goal is to perform a search on available resources, that are tagged with the keywords movies, cinema, etc. The system computes the similarity between the keyword and the data structure that is kept in memory and is updated regularly. Unfortunately User A cannot find a service that fulfills his needs. User A changes his approach and starts a search for users that used the keywords, movies, cinema, etc. to tag their services. The search for users returns the profile of a community member that sounds promising, later referred to as User B. User A initiates a BuddyList relationship by inviting user B to join user A's list. After user B logs in the system, he can read the message that has been sent to him, asking him to join user A's BuddyList. User B accepts the invitation, and the system adds user B to user A's BuddyList and vice versa. Both users can browse through each others creates resources and view the details. On his next login user A is able to find a resource which has only been recently uploaded and therefore has only very few tags attached yet, but is much more suitable for his needs. User A can obtain a reference to the service and can successfully integrate it in his work flow.
Dependency:	
Actor(s):	User unregistered(primary) (i.e. user A), User registered(primary)(i.e. user A), User B registered(secondary), System passive(secondary)
Precondition:	The username that is chosen is unique. The system has to guarantee that the chosen username is not used twice. The user needs to provide a valid email address.
Description:	
1.	A not registered user signals to create a new user account.
2.	The not registered user enters the necessary personal data in the web form, including firstname, lastname, email address, and desired username as well as password.
3.	The system checks for availability of the chosen username.
4.	The chosen username is available and is registered for the user.
5.	An email is sent to the email address of the registered user, confirming the creation of an user account.
6.	The user logs in the system.
7.	The user performs a search for a resource specifying some keywords to identify a resource.

8.	The system displays a tree structure of similar results.
9.	The user can view the details of the resources.
10.	The user performs a search for users specifying some keywords.
11.	The system display a list of users that used the specified keywords to tag a resource.
12.	The user chooses to add user B to his BuddyList.
13.	User B accepts the invitation. From now on user A can view all resources this user has created and vice versa.
14.	User A browses user B's list of resources.
15.	User A finds a resource that fulfills his requirements of a service.
16.	User A gets a reference to the resource and uses it for one of his work flows.
Alternatives:	
4.a	The chosen username is not available. Return to step 2.
8.a	The system cannot display a list of results because the specified keywords do not match any resource. Return to step 7.
11.a	The system cannot display a list of results because the specified keywords do not match any users. Return to step 10.
13.a	User B declines the invitation. User A cannot view the List of created resources from user B.
PostCondition:	
	User A has successfully registered to the system. User A has established a BuddyList relationship with user B. User A has retrieved a reference to a service user B has uploaded and is able to integrate this service into his own work flow.

CHAPTER 5

Implementation Description of Tagging Framework

5.1 Component View of Tagging Framework

This section gives an overview of the components, that the framework consists of. We describe the structure of components, their interfaces and the dependencies among each other, to fulfill the desired functionality. We also try to put some emphasis on the design patterns that we have adhered to, while designing the elements of the system. In all our architectural efforts we aim for *"loose coupling"* and *"high cohesion"* of the components. The interested reader please refer to [25] and [76] for an elaborate listing of design patterns.
Figure 5.1 shows the four main components of the framework that can be distributed on various machines depending on the deployment strategy. As a minimal requirement the BackEnd component and the FrontEnd component, both have to be hosted in a ASP.NET capable web application server, e.g. IIS. The **FrontEnd** component communicates via an RESTful interface with the **BackEnd** component. The BackEnd has direct access to the database over the data access classes and is forwarding incoming search request concerning resources, to the **Similarity Service** that is exposing a Java 6 web service interface and is accepting SOAP calls. The **Similarity Service** component is responsible for calculating the structure the similarity algorithm is operating on.

FrontEnd Component

The FrontEnd is primarily the component the user is interacting with. We designed this component to enrich the user experience, and implemented it in ASP.NET as this is a state of the art technology for web applications that handle dynamic content. Despite its lack of visualizing complex data structures, this technology seemed appropriate for our needs as it is constantly being adapted to respond to technological evolutions. Not only did we include Ajax based extensions to improve the interactivity of the pages, but we also included technologies like Java applets to enhance

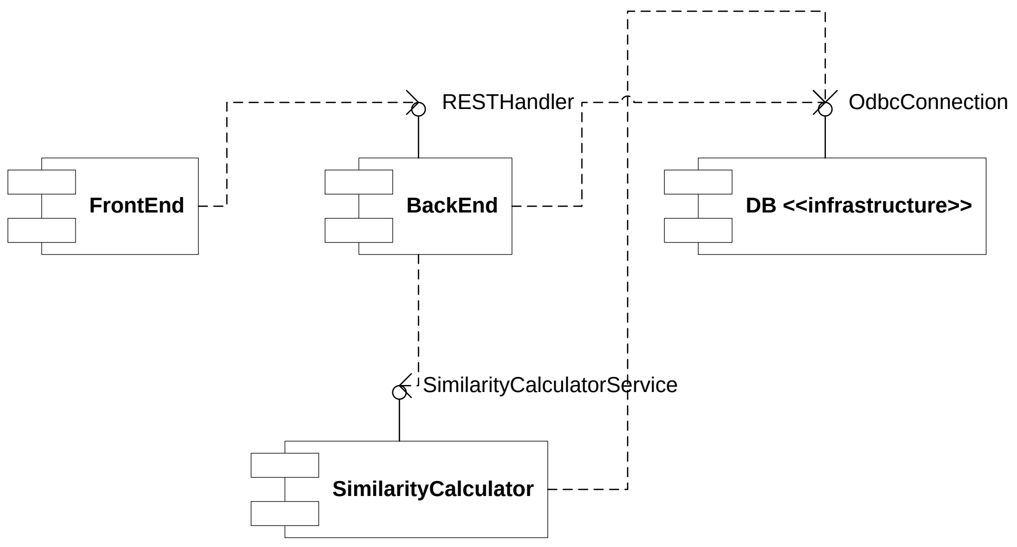


Figure 5.1: Component Overview

key features of the service tagging framework, especially in regards to visualizing the similarity structures where there is limited support in ASP.NET.
To keep this component exchange- and upgradeable with a minimum of implementation effort, we centrally manage interaction between the User Interface classes and the BackEnd component over a dedicated communication package, contained within the Util classes, see Figure 5.2. The Java technology subcomponents contain their own engine to exchange data with the BackEnd, since it is a non trivial task to manage communication between Java applets contained within an ASP.NET page. Every communication with the BackEnd component is established over the TCP using HTTP to guarantee reliable exchange, as lost and corrupted packages of data are retransmitted by the transport protocol.

FrontEnd Package View

Package UI

The **UI** package serves as a container for the user interface and code behind files of the main entities of the system: **User**, **Resource**, **Tag** and **Message**. Within the **UI** package there are sub packages for each of the main entities that contain files that are specific to the classes, to achieve readability and aim for loose coupling of the components. The **UI** package includes the pages responsible for login and logout process.

Package Visualization

The **Visualization** package contains the **Buddy** Java applet as well as the **Tree Visualization applet** that are used in the FrontEnd to improve the visual experience for the user. In order to

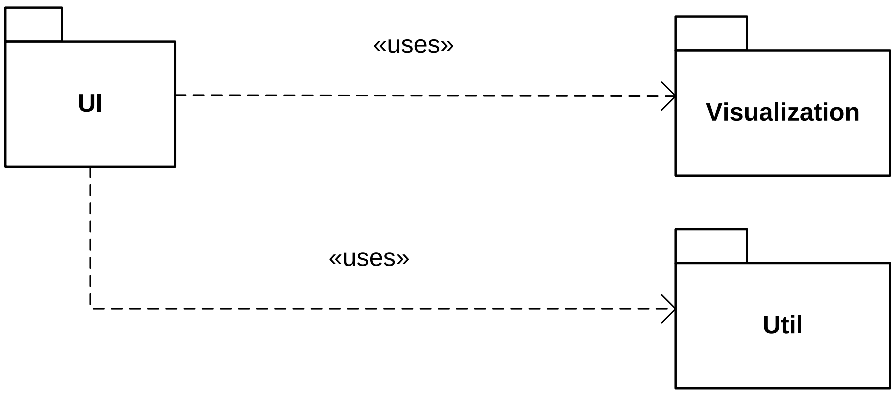


Figure 5.2: FrontEnd Package View

make full use of the graph visualization, we integrated code of a framework called JUNG[1] see [58] for more information on this framework. JUNG allows us to create undirected graphs for our **Buddy applet** and tree structures for the dependencies of the **Tree Visualization applet**.
For parsing through the data structures received from the BackEnd component, we make use the JDOM package that can be found under [36]. The package has a well structured API that allows us to easily traverse the XML document, to receive the items we need for interaction in the FrontEnd.

Package Util

The class **Constants** serves as a container for implementation specific configurations. It contains information like the RESTful-Interface endpoint address and necessary details to construct the logical path to the requested resources.
HTTPConnection is a class that makes communication with the BackEnd possible. It implements the singleton design pattern, that is used to generate a unique instance of a class, to make the infrastructure for transmitting HTTP requests, available.

FrontEnd Applets Package View

Additional to the packages from the ASP.NET implementing classes, we integrated Java applets to handle the visualization of the graph structures, that result from the interaction with other users, but also from the search for similar services within the framework.

Tree Visualization Applet

In Figure 5.3 we present an overview of the Tree Visualization applet, which is a sub component of the FrontEnd implementation. Its main purpose is to create an interactive representation of the search queries a user can perform in the system.

[1]Java Universal Network/Graph Framework

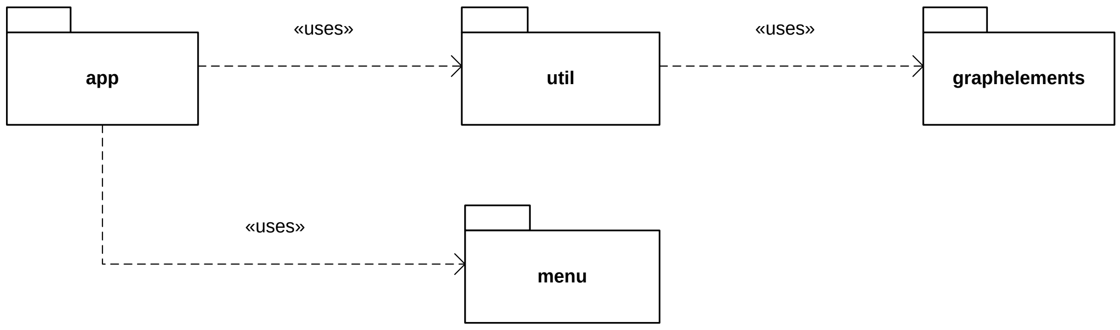


Figure 5.3: Tree Visualization and Buddy Applet Package View

Similar to the Buddy applet, the FrontEnd implementation of the framework hosts another Java applet, that is used to visualize the relational dependency of resources and tags that is gathered from the Similarity Service component of the framework. From an architectural view, both applets have the same architecture and segmentation of packages. Although Figure 5.5 and Figure 5.4 contain identical packages structure, there are some differences within the packages that are outlined in this section.

Package App

This package contains the main class of the applet, that has to be supplied with several parameters from the ASP.NET page, the applet is hosted in. Equally to the Buddy applet this applet has to be provided with the user id of the user that is performing a request. But unlike the Buddy applet, where some of the computational effort can be performed directly, the calculation of the similarity of services is too complex and resource intense to be performed on the client side. It is therefore, that the result of a search request has to be supplied, additionally to the user id, to start the visualization process. The computational logic that is left to perform the visualization of the data is contained within the main class, **TreeVisualization**, and the XMLParser class within the util package.

Package Graphelements

Similar to the Buddy applet this package contains the elements that are used to visualize the graph. The **BaseVertex** class is not extended by classes that are specific to the domain model, i.e. Resource and User, since in this graph we only operate on data of resources. Instead we divide the nodes in **MultiNode** and **SinlgeNode** classes, that are representations of nodes that contain more than one child and single nodes that represents the leaves of the tree structure. That allows us to efficiently create the graph structure.

Package Menu

Similar to the Buddy applet this package contains classes that allow the user to interact with the data and perform several functions on the nodes. **ShowCorrespondingResourceMenuItem**,

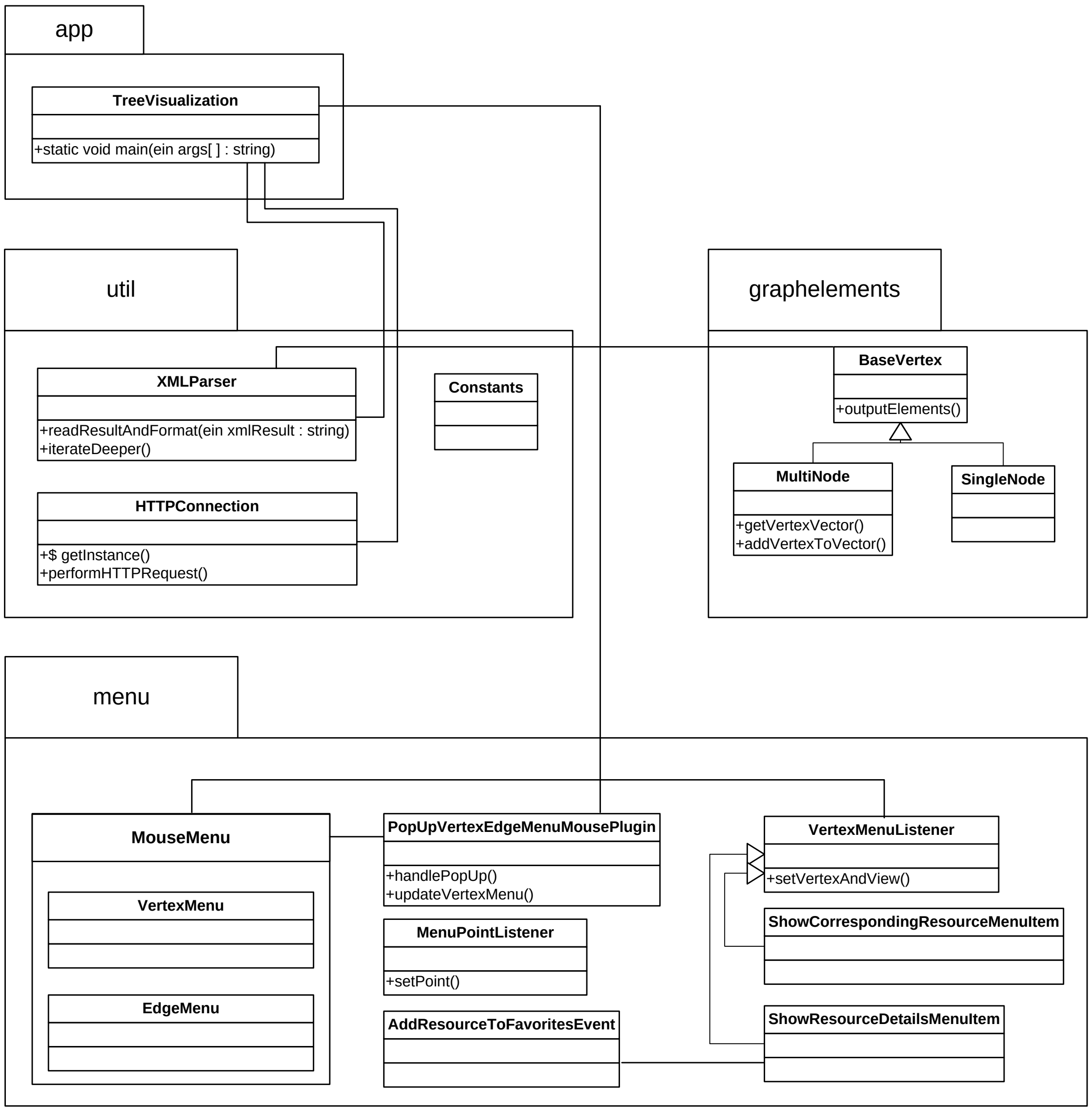


Figure 5.4: Tree Visualization Package and Class View

ShowResourceDetailsMenuItem and **AddResourceToFavourites** are classes that extend the **VertexMenuListener** class to perform functions that are specific to the framework.

Package Util

Util is a package that contains helper classes that enable certain functions. **HTTPConnection** again is a class that is forwarding requests to the BackEnd component to retrieve detailed data about resources. **Constants** is a class that provides certain data and addresses that are unlikely to change even in later releases and are therefore gathered in a central instance of this class.

As already mentioned **XMLParser** is a class that contains computational logic to perform the parsing of the search data to generate the tree graph.

Buddy Applet

Buddy applet is part of the FrontEnd implementation and serves as a container for visualizing the relations to another user and the resources they tagged, respectively created.
Similar to the Tree Visualization applet, this applet is designed to support the visual features of the system. As the package structure is the same as in the Tree Visualization applet, the representation of packages in Figure 5.3 is also valid for the Buddy applet. The applet is based on the concept of a representation of users that are connected to the current user over a so called Buddy relationship, that symbolizes the intention to share personal data with a specific person. Once this relationship is established between two users both can view the resources they are dealing with and how these have been tagged.

Package App

This package contains the main class of the applet that has to be supplied with a necessary parameter from the ASP.NET hosting page. The id of the user that is currently operating on the system is of great importance, as this value is used to specify the data of some of the features of the applet.

Package Graphelements

Contains several classes that represent the elements that are used within the graph drawing of the applet. Additional to the **BaseVertex** class this package contains classes that extend the BaseVertex class. **ResourceVertex** that represents an instance of a resource and **UserVertex** that represents an User of the system in the graph are available. At the moment there is no need for representing instances of edges, since this functionality is not implemented, but could be integrated in a later release of the prototype of the framework.

Package Menu

Within this package we can find several classes that are specific to the JUNG framework [58] to enhance the visualizations with interactive features to enable manipulation of the data structures. Extending the **VertexMenuListener** class we added several MenuItem classes, **ShowConnected-TagMenuItem**, **ShowResourceDetailsMenuItem**, and **DeleteBuddyRelationMenuItem**, that offer functionality tailored to the interaction with the framework. At the moment these classes are sufficient to fulfill the requirements of the system. In order to achieve an interface that is even more interactive and equipped with more complex tasks in a potential later release of the framework, this is the package to include the necessary classes.

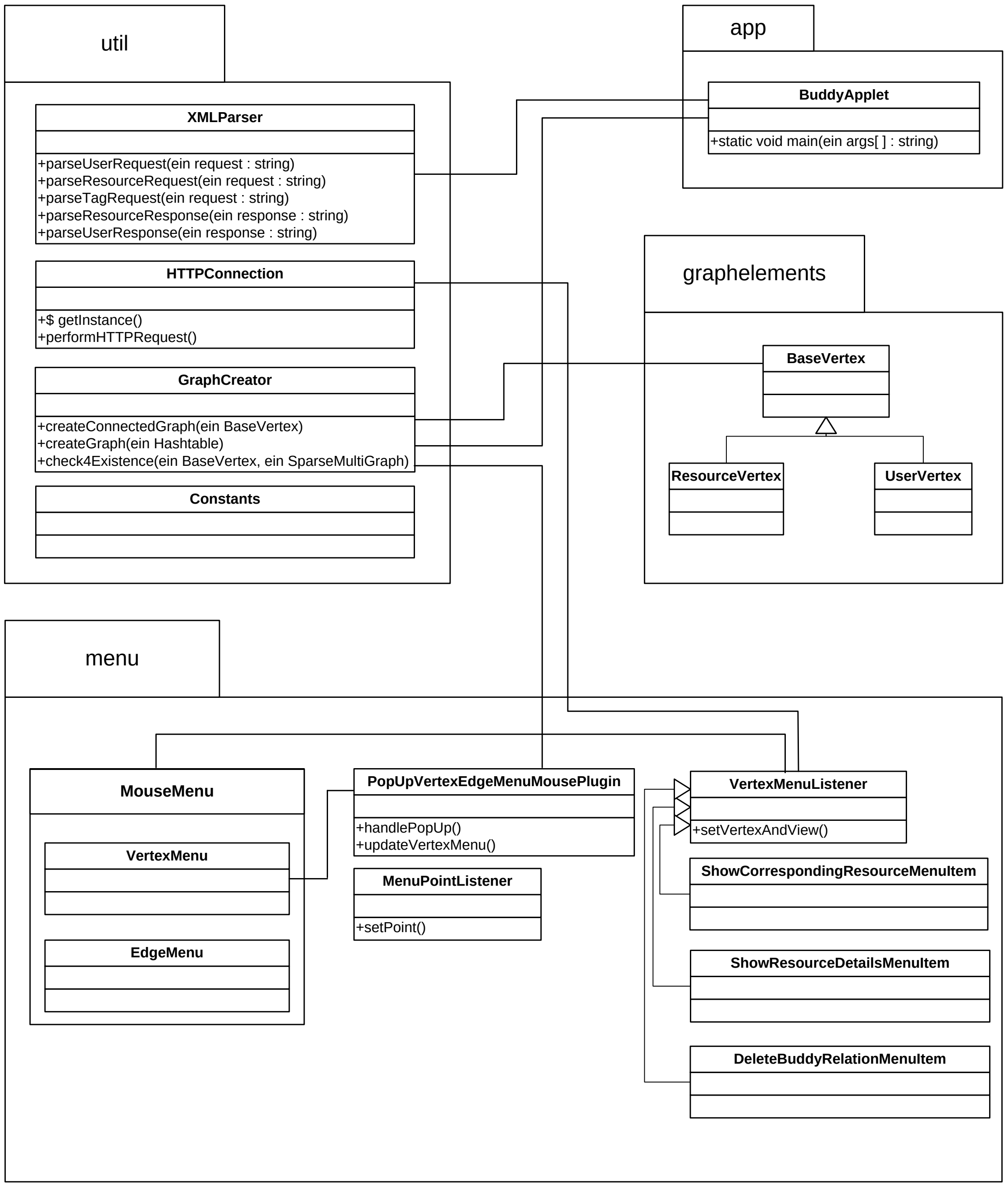

util
XMLParser
+parseUserRequest(ein request : string)
+parseResourceRequest(ein request : string)
+parseTagRequest(ein request : string)
+parseResourceResponse(ein response : string)
+parseUserResponse(ein response : string)
HTTPConnection
+$ getInstance()
+performHTTPRequest()
GraphCreator
+createConnectedGraph(ein BaseVertex)
+createGraph(ein Hashtable)
+check4Existence(ein BaseVertex, ein SparseMultiGraph)
Constants
app
BuddyApplet
+static void main(ein args[ ] : string)
graphelements
BaseVertex
ResourceVertex
UserVertex
menu
MouseMenu
VertexMenu
EdgeMenu
PopUpVertexEdgeMenuMousePlugin
+handlePopUp()
+updateVertexMenu()
MenuPointListener
+setPoint()
VertexMenuListener
+setVertexAndView()
ShowCorrespondingResourceMenuItem
ShowResourceDetailsMenuItem
DeleteBuddyRelationMenuItem

Figure 5.5: Buddy Applet Package and Class View

Package Util

The util package offers several classes that are necessary to guarantee communication with other components. **HTTPConnection** is a class that is the equivalent of a class in the ASP.NET framework implementation, and serves the same service, namely establishing a HTTP connection with the BackEnd component. To make the system more manageable many of the properties necessary to connect to the BackEnd endpoint are contained within the **Constants** class. **GraphCreator** is the class that is responsible for creating the links that represent the relations among the graph elements. There is no explicit data structure that contains the data to visualize the buddies of a user and their tagged resources. Instead, all of the relational data has to be calculated within this class. To achieve efficient parsing of the queried XML data from the BackEnd component, **XMLParser** class can be used. It constructs a JDOM representation of the data that is returned from the BackEnd and extracts the desired data.

BackEnd Component

The BackEnd component is the central interface for communication between the FrontEnd component, that is accepting user commands, and the Similarity Service component that enhances the search and recommendation features of the framework. The BackEnd component is responsible for many of the basic persistence functions of the system, i.e. maintenance of useraccounts, management of services and tags, and dispatching of search requests. Many features of the business logic are realized within this component.

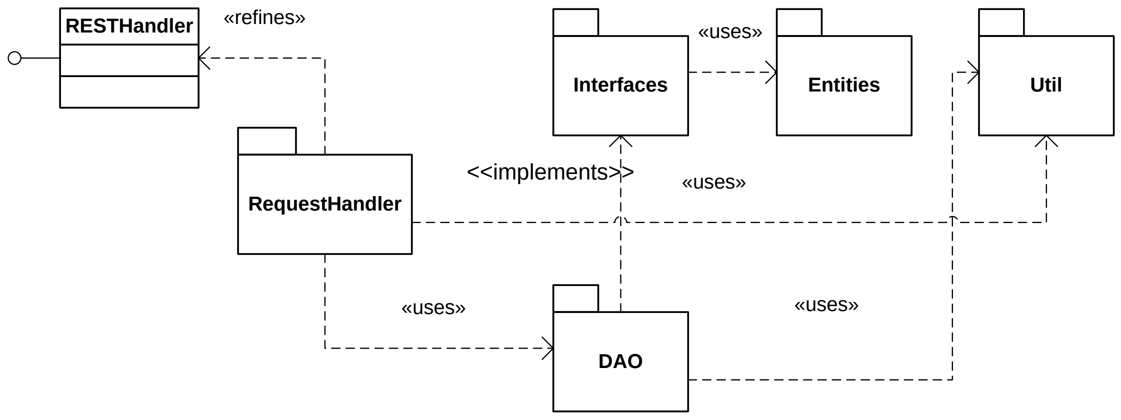


Figure 5.6: BackEnd Package View

Although most of the functionality of the system is managed over this component one of our key contributions, the feature of recommendation of related resources, is computed in the Similarity Service component, that exposes its capabilities over a web service interface, where communication is established over SOAP messages.
Although our prototype implementation physically shares the same database, from a conceptual perspective the framework is designed to operate on two independent databases that mirror the same data. The main reason for this concept is the following. Sending the data to calculate the structure with every incoming request is not feasible as the amount of tags and the described resources grows exponentially with increasing users, resulting in an response time for search requests that is not justifiable. The initial steps for the similarity calculations are performed independent of any incoming requests to keep the response time, for further calculations, at

a reasonable level. To avoid too complex coupling between the BackEnd component and the Similarity Service component, through implementing sophisticated notification mechanisms, we rely on synchronous message exchange. Communication between the two components is therefore reduced to the transfer of the request data, i.e. the incoming search keyword, and the response, i.e. the structured similarity data in XML representation.

BackEnd Package View

The design of this subsystem is influenced by the Core J2EE design pattern of Data Access Objects, see [51], that abstracts the access to the persistence layer of a system, to allow for exchangeability of the storage mechanisms and separate concerns of each of the participating classes.

Package Interfaces

This package contains classes that contain the interface definitions of the DAO classes. All exposed methods can be viewed from within these classes, that serve as a contract between the implementing classes and the programmer.

Package Entities

Within the entities package we can find the classes that represent the entities in use in this system. The containing classes contain their basic properties and methods to access this data programmatically. Available entities within this package are the main entities of the system: User, Tag, Resource and Message.

RESTHandler Component - RequestHandler Package

The **RESTHandler** is one of the key elements of the framework, as the whole communication is transacted over this component. Every request that is formatted as a HTTP request is parsed and validated for further processing. Depending on the request type of the incoming HTTP request, the data is forwarded to one of the refining RequestHandler classes contained within the **RequestHandler** package. One RequestHandler for the corresponding request type is processing the input for the corresponding type of resource, either Tag, Resource, User, Message or Search. We divide between four request types and their corresponding classes, **PUTRequestHandler** for update requests, **POSTRequestHandler** for create requests,
DELETERequestHandler to delete data on the server, and **GETRequestHandler** for reading data of a certain instance.
To maintain access to the entities of the system, the active RequestHandler uses the classes provided in the **DAO** package. The RESThandler interface in combination with the RequestHandler package implements a structural design pattern known as facade, that exposes access to subcomponents of a system via a single interface and delegates client requests to appropriate subcomponents, which in turn promotes loose coupling, cf. interface- segregation. See [25] and [50] for more information on design patterns.

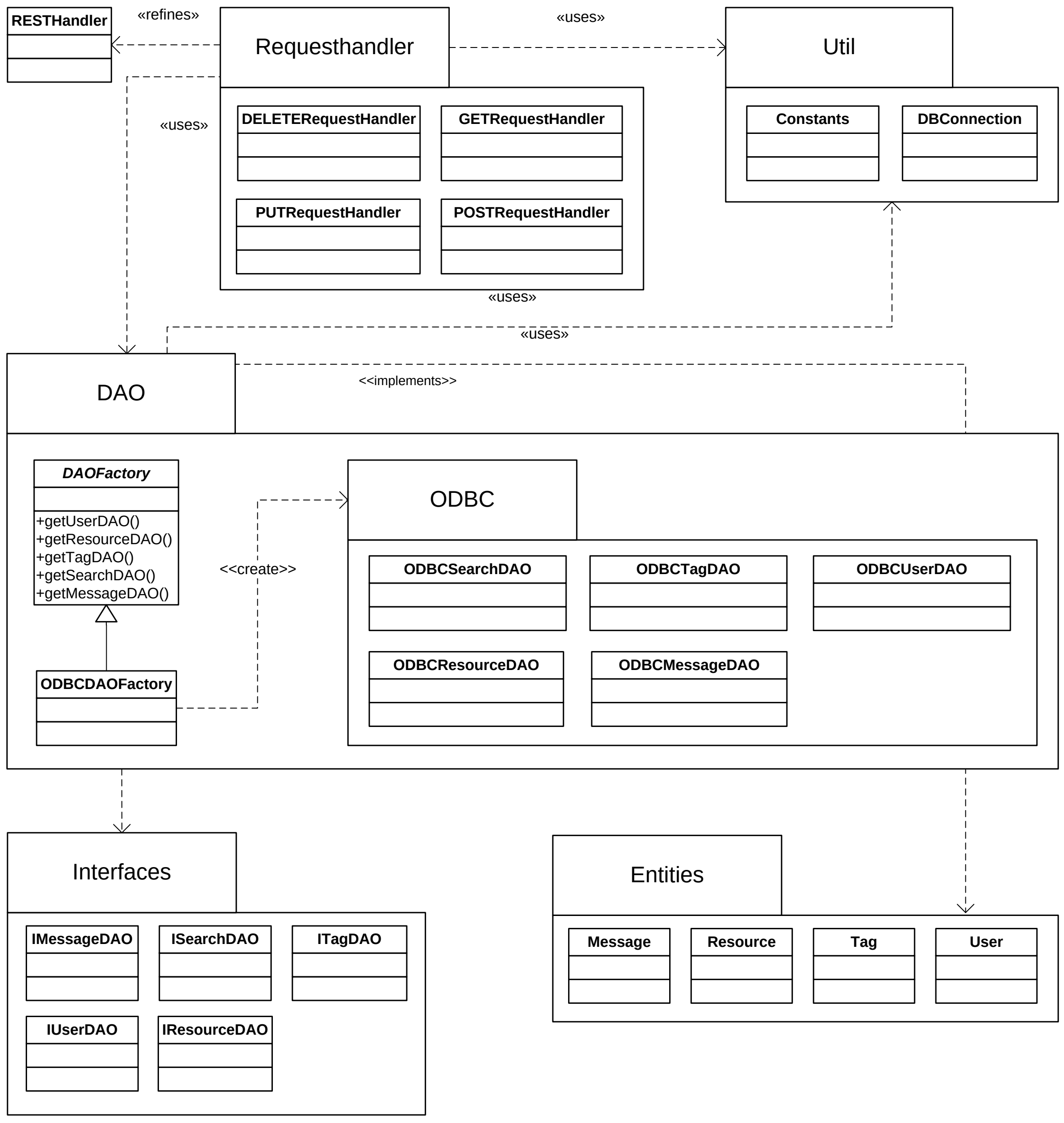


Figure 5.7: BackEnd Package and Class View

Package DAO

The DAO package is constructed referring to the J2EE access pattern of Data Access Objects [2], that offers access to components via a structured interface see [51]. The **abstract DAOFactory** class defines methods for retrieving a DAO class for a specific instance of entities that implement the corresponding interface. In the framework currently only ODBCDAO classes are implemented,

[2]Data Access Object - DAO

which are located in the **ODBC** sub package, but using this approach it is easy to extend the system to use various means for persistence, e.g. XML storage mechanisms etc. Inside the ODBC sub package we can see a data access object correlating to each one of the entities: **OSDBCUserDAO**, **ODBCResourceDAO**, **ODBCTagDAO** and **ODBCMessageDAO**. Additionally this package includes a DAO class that is responsible for providing the application logic for the search functionality, contained within the **ODBCSearchDAO** class. The methods that have to be implemented in every DAO no matter what storage technique is used, are defined in the **Interfaces** package.

Package Util

The **Constants** class serves the purpose to gather component wide constants and configurations to be used by the classes of the component. The **DBConnection** class is implemented as a singleton, to shield the database from arbitrary access. The current implementation is designed to work with a database as its main means for persistence.

Addressable Resources and Parameters

In this section we will describe what kind of classes from an OOP[3] -view can be addressed, and what parameters have to be specified in the request, to perform the desired operations. The properties in each paragraph define the behavior of an object of a specific class and are required unless indicated otherwise. Values that are omitted in the following list, but are included in the database description indicate values that are added by the system itself.

Users

Is the class responsible for representing user entities of the system. The following properties define an object of this class.

- userName
- lastName (optional)
- firstName (optional)
- password
- emailAddress
- user Image (optional)

Tags

A tag is a single keyword of arbitrary length up to 255 characters, that is used to describe the function of a service and/or its properties. Users add tags to resources. Every resource can be tagged multiple times from different users with the same tag.

[3]Object-Oriented Programming

Resources

A resource is a representation of a service that is created by the users of the system. User can attach as many different tags as they want to describe a resource. Adding tags to a resource increases the benefit of the whole system, especially the search for resource functionality, as the similarity algorithm is depending on the attached tags to determine a resource.

- name
- description (optional)
- xmlcontent

Search

Via the search resource the system is queried for search requests. Depending on the structure of the request, the system can identify what resource is involved in the search request. Additionally the user can specify up to three keywords to limit the results of a search request. If no keyword is included in a search request the system assumes it is a search request for all items of a specific resource type.

Addressable Operations and RESTful Verbs

Although the HTTP specification, available under [2], defines more methods for communication, we only make use of the following four request types, as all the requirements of the system can be mapped with these basic methods. The remaining methods of the HTTP 1.1 specification like CONNECT, TRACE, HEAD, and OPTIONS shall be mentioned just for the sake of completeness, but do not affect the systems data that is persisted, nor the systems responses to requests from clients.
Every request in the system is constructed in the same manner. We basically address every request to a service endpoint where the BackEnd component is located. All incoming request are in a first step distinguished by their request type, respectively the operation to be performed, and in a second step by the addressed resource.
The service endpoint address is the part of an URI that is exactly the same in every request, no matter what operation on what resource is performed, and is constructed as follows:

```
http://hostAddress:listeningPort/RESThandler.ashx/
```

The allowed operations have to be specified in the HTTP request directly as the request type, whereas the available resources have to be addressed over the additional part of the URI of the request that is appended to the above definition of the endpoint address. In the following listing we specify what address triggers what function in the BackEnd component. For better readability we abbreviate the service endpoint address, that is the same in every request, with [sea][4] - for service endpoint address.

[4] [sea] is the abbreviation for service endpoint address

POST

A POST request creates a new instance of the addressed class at the BackEnd component that is persisted in the database. In the previous section we could see what classes can be addressed. In order to transmit a valid request, the properties that define a requested object on the BackEnd component have to be specified in the HTTP request. Besides the creation of an instance this request type is also needed for transmitting search requests.

user

[sea]/user	creates a new user.
[sea]/user/login	performs a login.
[sea]/user/userid/user2addId	adds a user to the buddylist.
[sea]/user/userid/user2addId/true	confirms invitation.
[sea]/user/userid/binarycontent	uploads binary content(image file) to the user account.
[sea]/user/userid/favoured/resourceid	creates a relation between a user and a resource that symbolizes that the user is interested and likely to tag that resource, but a tagging action as not been performed yet.

resource + tag

[sea]/resource	creates a new resource.
[sea]/resource/resourceid/tag	creates a new tag to a specific resource.
[sea]/resource/resourceid/tagid	associates an existing tag to specific resource.
[sea]/resource/resourceid/binarycontent	uploads binary content (xml description file) to the resource

search

[sea]/search/user	performs a user search.
[sea]/search/resource	performs a resource search.

message

[sea]/message/userIdFrom/userIdTo	Creates a message from a user to a certain user, to allow users to foster their social relationship with other users and/or to initiate contact with a unknown user.

PUT

A PUT request is very similar to the POST requests of the system, except that an entity of a specific class is updated and already has to exist. Therefore the main difference is in the URI part of the request, since the object that is to be updated has to be identified in the address via its id.

user

[sea]/user/userid	updates the user account identified by userid.
[sea]/user/userid/binarycontent	updates the binary content (image file) of an useraccount.

resource

[sea]/resource/resourceid	updates the resource identified by resourceid.

GET

A GET request delivers all the data records of a requested class, i.e. either user, resource, tags or messages. When specifying the id of an object, in the URI, this method solely returns the specified item, and its details. Further we can retrieve the binary content of user accounts and resources.

tags

[sea]/tag	returns all available tags in the system.
[sea]/tag/tagid	returns a specific tag.
[sea]/tag/tagid/resource	returns all resources that are tagged with a specific tag.
[sea]/tag/tagid/user	returns all users that used a specific tag.

resources

[sea]/resource	returns all resources.
[sea]/resource/resourceid	returns a specific resource.
[sea]/resource/resourceid/tag	returns all tags associated to a specific resource.
[sea]/resource/resourceid/tag/tagid	returns a specific tag associated to a specific resource.
[sea]/resource/resourceid/tag/tagid/resource	returns all other resources that are associated to a specific tag associated to a specific resource.
[sea]/resource/resourceid/user	returns the users that are tagging a specific resource.
[sea]/resource/resourceid/user/userid	returns a specific user record of users that are tagging a specific resource.
[sea]/resource/resourceid/user/userid/resource	returns all other resources of a user that tagging a specific resource.
[sea]/resource/resourceid/owner	returns the owner, the user who created the resource.
[sea]/resource/resourceid/binarycontent	returns the binary content(xml description file) of a specific resource.

user

[sea]/user	returns all user records.
[sea]/user/userid	returns a specific user record.
[sea]/user/userid/binarycontent	returns the binary content of specific user record.
[sea]/user/userid/message	returns all messages that a certain user has received.
[sea]/user/userid/tag	returns all tags of a specific user record.
[sea]/user/userid/resource	returns all resources of a specific user record, no matter if the user is the creator or not.
[sea]/user/userid/resource/created	returns all resources of a specific user that this user has created.
[sea]/user/userid/resource/favoured	returns all resources of a specific user that this user has added to his favourites.
[sea]/user/userid/resource/tagged	returns all resources of a specific user that this user has tagged but not created.
[sea]/user/userid/buddies	returns all buddies of a specific user record.
[sea]/user/userid/buddies/userid	returns a single buddy of a specific user record.
[sea]/user/userid/buddies/userid/tag	returns all tags of a single buddy of a specific user record.
[sea]/user/userid/buddies/userid/tag/tagid	returns a specific tag of a single buddy of a specific user record.
[sea]/user/userid/buddies/userid/tag/tagid/resource	returns the resource associated with a specific tag of a single buddy of a specific user record.
[sea]/user/userid/buddies/userid/resource	returns all resources of a single buddy of a specific user record.
[sea]/user/userid/buddies/userid/resource/rid	returns a single resource of a single buddy of a specific user record.
[sea]/user/userid/buddies/userid/resource/rid/tag	returns all tags of a single resource of a single buddy of a specific user record.
[sea]/user/userid/buddies/userid/buddies	returns all buddies of a single buddy of a specific user record.

DELETE

A DELETE request removes a data record from the database specified in the URI, permanently.

user

[sea]/user/userid	deletes a user account and its associated tags and resources.
[sea]/user/userid/tagid	deletes a specific tag from a specific user account.
[sea]/user/userid/resourceid	deletes a specific resource from a specific user account.
[sea]/user/userid/favoured/resourceid	deletes a specific resource the user has added to his favourites.
[sea]/user/userid/message/messagid	deletes a certain message a user has received from another user.
[sea]/user/userid/buddies/userid	deletes the BuddyList relation between two users.

resource

[sea]/resource/resourceid	deletes a specific resource and all associated tags.
[sea]/resource/resourceid/tag/tagid	deletes a specific tag from a specific resource.

ReturnValues

As the communication with the BackEnd component is handled over HTTP, every request submitted to this component has a certain return value. In the following we present the responses to certain types of request. The status codes are equivalent to the HTTP status codes.
200 - OK Is the status code that is sent if a request like **GET** or a **PUT** succeeded.

201 - CREATED In case of a successful creation of an entity the BackEnd component returns this status code.

400 - BAD REQUEST If a request is malformed or necessary parameters are missing the system responds with this status code.

404 - NOT FOUND If a resource is requested that is not available the BackEnd component responds with 404.

500 - INTERNAL ERROR In case of an internal error the system responds with a 500 status code.

Similarity Service Component

The Similarity Service component is, from a communicational perspective, cut off from direct interaction with the user, since it cannot be directly contacted from the FrontEnd component.

Every communication, in this case search requests and similarity calculations of resources, has to be transmitted over the BackEnd component. The BackEnd component therefore has comparatively more responsibility for maintaining the communication, i.e. the exchange of search and similarity data, with this component, but can decrease its claims for computational resources, as the Similarity Service takes over the computational intense task to keep the similarity information up to date. See Figure 5.8 for a view on the included classes.

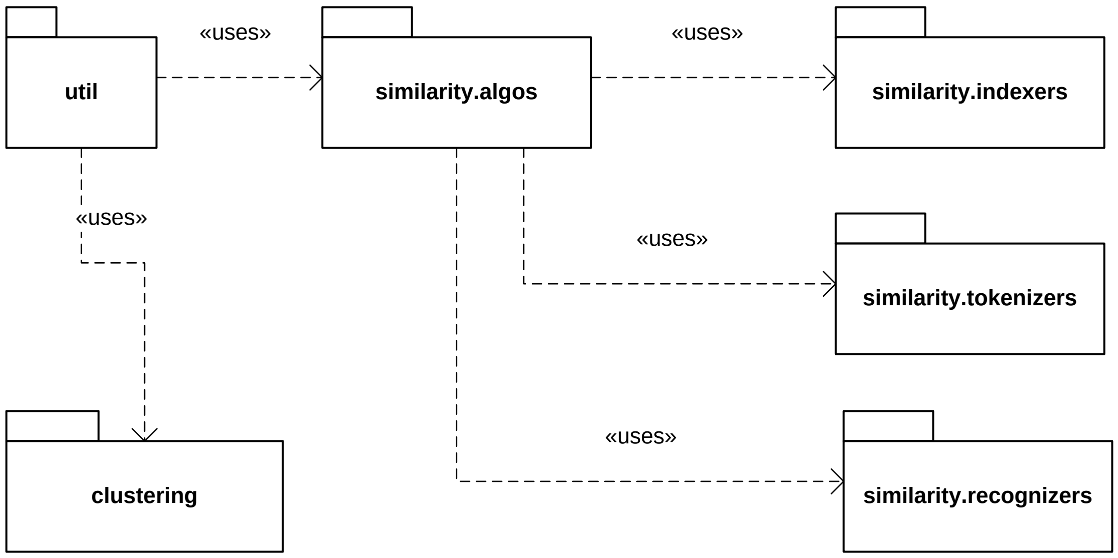


Figure 5.8: Similarity Service Package View

On a regular basis the data about resources and tags is pulled from the database. This information is retrieved throughout a JDBC Connection since it is more effective, to query the data directly from the database, than to transfer the data XML encoded, over the RESTful interface of the BackEnd component and keeps the component independent from any triggering mechanisms. In contrast to the BackEnd component and big parts of the FrontEnd component, that are developed using the ASP.NET framework, this service is implemented using Java 6 technology, to publish a web service interface. The similarity of resources is calculated by falling back on information retrieval research and refers to available java archives. Details of the recommendation and the similarity calculation features of the framework are handled in a later section of this document.

Similarity Service Package View

The Similarity Service is the component that is responsible for the calculation of the similarity among resources that are persisted in the framework. When a user is querying for a service, in a first step this components identifies the tags that are most similar to the query string. In a second step, based on the similarity among the describing tags, a tree structure is created, that represents the relation ship of the tags and the associated resources. This component heavily is depending on code derived from [61] and [60] that is as an implementation realization of the research on similarity and clustering of documents found in [42].

Package Util

In the util package we find two classes responsible for the access to the shared database. As already mentioned is far more effective to query for the data directly than to transfer it via the RESTful interface of the BackEnd component. **DBConnection** is the class that manages the connection, whereas **DBResultParser** is used to create an internal representation of the results of the data base queries.

Package Similarity.Indexers

Once the text, respectively the tags that describe a resource, is reduced to the essential elements, omitting stop words, punctuation, and other special characters that distort the calculation, the classes of the indexers package is used to transform the remaining text elements to a vector representation using the **VectorGenerator** class. **IdfIndexer** class is used to reduce the influence of re occurring tags on the overall similarity structure as this increases the precision of the results.

Package Similarity.Tokenizers

Tokenizers is a helper package that includes the classes **WordTokenizer** a class that performs the step of reducing the text to words that can be used in the calculation step as stop words and special characters influence on the calculation and are eliminated. **Token** is the representation of a chunk of the read input of a resource and **TokenType** is the differentiation of the types of a token, including abbreviations, stop words and regular words, just to name a few.

Package Similarity

The package that contains the classes that implement the web service interface and the main method of the service is the similarity package. **SimilarityService** is the class that runs the main method, in which a **TimingThread**, that is responsible for a perpetual update of the similarity matrix computation, and a **SimilarityCalculatorService**, that creates a web service interface to the **SimilarityCalculator**, is instantiated. SimilarityCalculator has two methods that are essential **calculateQuerySimilarityTFIDF()** that is exposed over the web service interface and as the name implies, calculates the similarity of a query to the previously calculated resource matrix, and **updateMatrix** that performs an update of the underlying calculation matrix. As a subsequent step an instance of **TagClusterCalculator** in the package clustering is called for the computation of the tree structure.

Package Similarity.Algos

Within this package we have the classes that contain the similarity algorithms. The original implementation of the included similarity framework, allows for different similarity algorithms. The base class that further algorithm classes have to be derived from is **AbstractSimilarity**. In the current version we concentrate on the usage of **CosineSimilarity** as the means for similarity measurement. In future releases there might potentially be different algorithms in use. **Searcher** is the class where all necessary parts for a calculation are gathered. Once the similarity of the

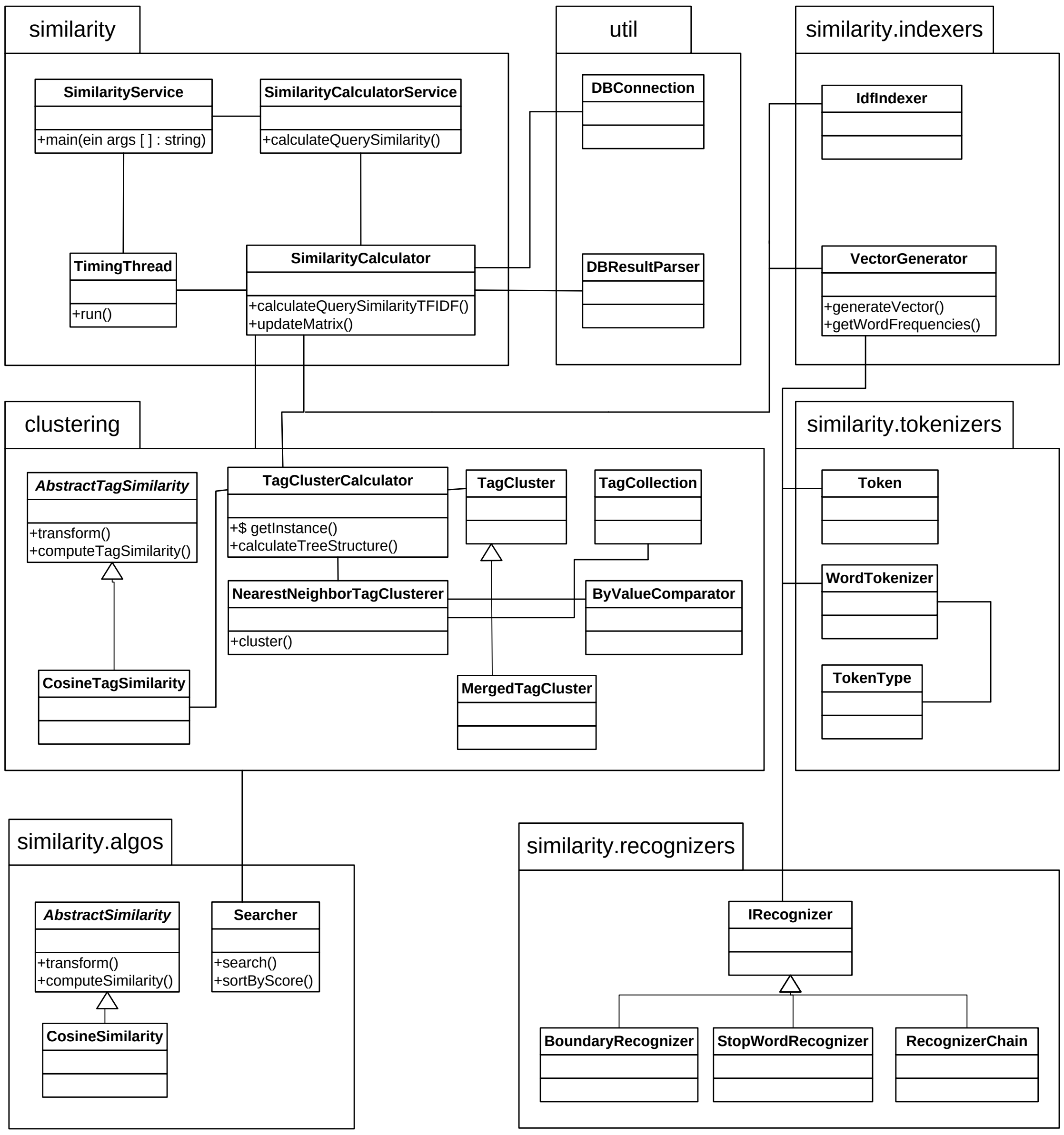


Figure 5.9: SimilarityService Package and Class View

query string and the existing tags is determined. The most similar tags are used as input to the secondary step of tree structuring the data.

Package Similarity.Recognizers

Recognizers is a package that includes the classes to perform the actual parsing for stop words and punctuation of the documents that define a resource. In addition with the **RecognizerChain** class that implements the **IRecognizer** interface, these classes perform the actual identification of text elements to tokens and the respective token types.

Package Clustering

The clustering package contains classes to perform the step of clustering the data from the initial similarity calculation. As this also includes calculating the similarity among the elements in further steps, these package also includes classes for calculating similarity. Namely **AbstractTagSimilarity** and **CosineTagSimilarity** which contain slightly different implementations of the cosine similarity algorithm that can be found in the similarity.algos package. **TagCluster**, **MergedTagCluster** and **TagCollection** are structural classes, that help in the process of calculating the clusters of tags. The actual code to perform the clustering step is contained in the **NearestNeighborTagClusterer** class that performs a nearest neighbor analysis of the elements of a cluster to determine which cluster is to combine next. Although there are potentially algorithms with lower computational costs, this is a quite popular algorithm that is in use in many areas of application, and suites our needs quite well.

5.2 Interactions of the Components

As already stated in the schematic system architecture in section 4.2, the system consists of three main components, that interact with each other to fulfill the previously mentioned functions. The FrontEnd is basically used for accepting input from the user, formatting the data to HTTP requests to trigger processes in the BackEnd implementation, and displaying the returned results in various formats. The FrontEnd component thus is acting as a gateway to the RESTful BackEnd component, and can be exchanged with any other possible implementation, as long as the endpoint address of the BackEnd component is known.
Since basically every interaction of a user with the system, will trigger a very similar sequence of participating objects in FrontEnd as well as BackEnd component, we have decided to split the complete sequence view of a request into two representational parts. Sequence diagram in Figure 5.10 gives an overview of how the FrontEnd component accepts interface commands, and creates a request to communicate with the RESTful BackEnd component via HTTP. The Sequence diagram in Figure 5.11 shows how the BackEnd component will accept incoming requests, and how they are handled to perform the required computational logic.
The Sequence diagram in Figure 5.12 shows how the Similarity Service component is automatically updating the data structure, to be able to calculate the similarity of the created resources stored in the database. This process helps to speed up any incoming search requests as it is time consuming to create the underlying dataset for this task.

FrontEnd Sequence View

The FrontEnd is a component that is using the ASP.NET technology[18], and AJAX extensions [16] to create a highly interactive user interface for the user. ASP.NET and its code behind structure allows for very efficient and well factorized code, since the controller classes are separated from the viewing classes.

Visualization

In order to visualize the complex data structures, primarily the similarity data structure as well as the buddy relation of a user, we decided to integrate a JAVA based software package called JUNG[5], that has especially been developed to visualize graph data of any kind, be it directed or undirected. See [58] for a reference to the original implementation, available under the BSD[6] license. The JUNG archives we integrated allow us to visualize the hierarchical data the Similarity Service calculates from search requests for resources, in a tree representation, and the undirected connected graphs for the buddy list functionality.
Direct function calls of Java code from ASP.NET pages is hardly realizable without accepting architectural compromises. The solution to realize the desired functionality with a minimum of drawbacks on either architecture and technological possibilities, was to integrate the code in Java Applets and integrate those in the ASP.NET pages. Necessary parameters are passed to the Applets via encoded parameters. Unfortunately the engine to communicate available in the ASP.NET implementation, has to be duplicated in the Java applets since this code cannot be used directly. To guarantee security in the system, the integrated Applets have to be digitally signed to verify the authenticity of the vendor. For the sake of simplicity this is not performed by a Certification Authority (CA), as it would be necessary to guarantee the authenticity of the executed applets, is realized by signing the Applets by our selves.

Communication with the BackEnd Component

As already mentioned, the FrontEnd contains almost no computational logic. The only changes that are performed to the dataset, is parsing the retrieved data to be displayed in ASP.NET components. The FrontEnd is only responsible for accepting input commands, and displaying either the relations of the entities in the database or the results of search request. In order to perform changes on the data set or to interact with the system, the FrontEnd has to parse the user input to a suitable format and establish communication with the BackEnd implementation.
The BackEnd is exposing a RESTful interface paradigm and is based on HTTP as its underlying communication protocol. Therefore every request from the FrontEnd to the BackEnd has to be parsed to an HTTP Request. Most communication is based on transferring html encoded data to and retrieving XML formatted data back from the BackEnd. The only exception to this communication mechanism appears when binary data is included. As you can see in the description of the data base structure in section 5.3, we store the data of the user image that is associated to an user account and the data of the describing XML file of a resource in binary format. Within the class that is responsible for communication **HTTPConnection**, we created two different methods that handle the requests depending on the transferred data. Especially when requesting binary image data we can benefit from this technique, as we can output the binary data on a ASP.NET page and can directly include this page as the source for an image tag in the html code. Transformation of the file type is implicitly handled through the means of the framework, and must not be dealt with manually. This way multiple file types like .jpg, .gif, or

[5]Java Universal Network/Graph Framework
[6]Berkeley Software Distribution

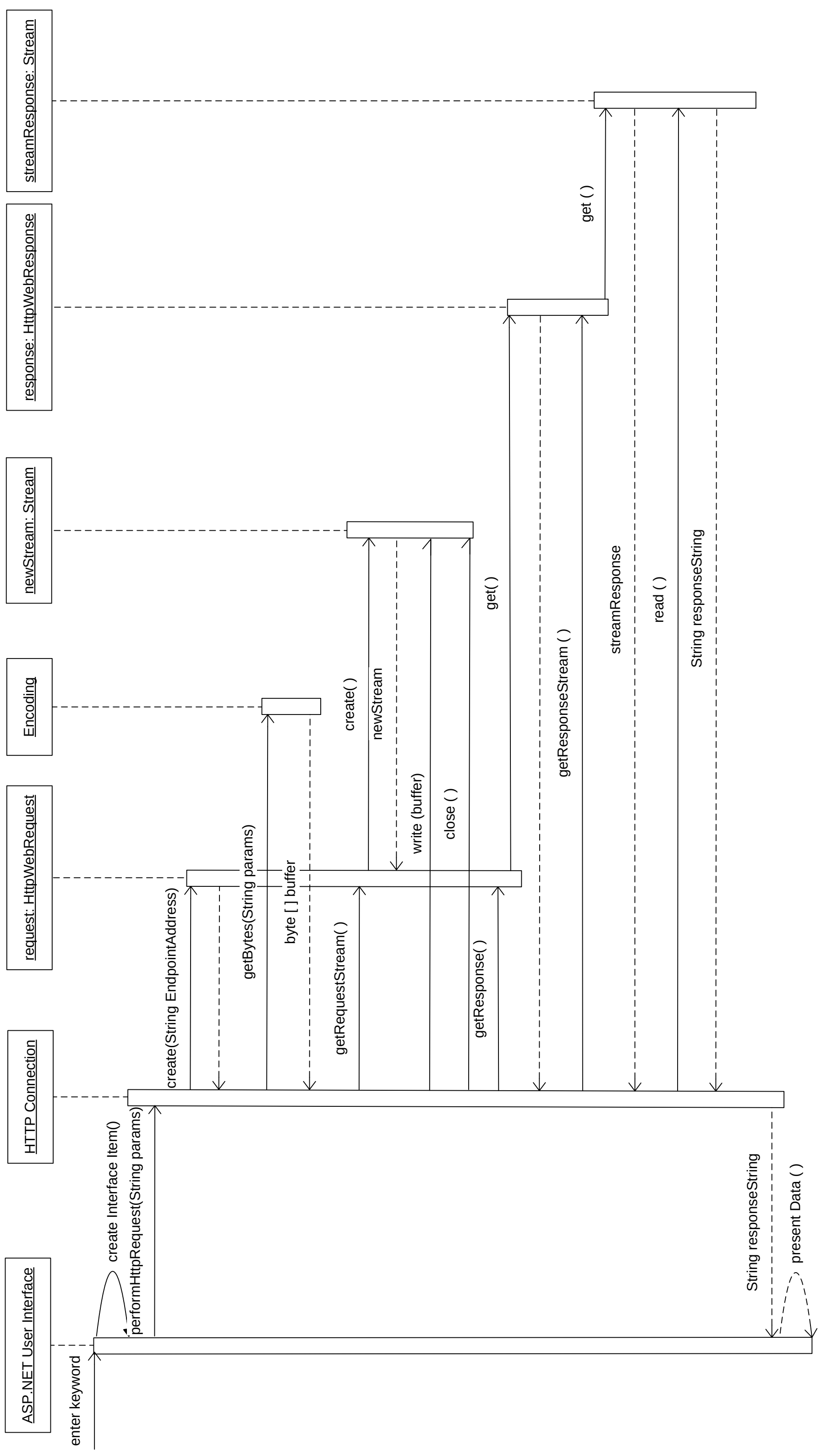


Figure 5.10: Representational FrontEnd Sequence View

.png for images, and .wsdl or .xml for resource description can be handled by the system, without the risk of determining an incorrect file type.
In figure 5.10 we can see what entities are involved in creating a search request for a resource. The FrontEnd component creates a **HTTPConnection** object that is responsible for handling the rest of the communication. A **HttpWebRequest** object is created to address the endpoint address of the BackEnd component. Depending on the type of the request we have to set the corresponding type in the HTTP request to yield to the desired functionality, possible parameters are GET, POST, DELETE and PUT. The parameters for the request, in this case keywords for the search for resources, are transformed from their string representation into a **byte array**, that serves as an input for the **RequestStream**. From the **HttpWebRequest** object one can determine a **HttpWebResponse** object, that is used for receiving the **ResponseStream** from the service endpoint. Once this stream has been read, we can deliver the *responseString* as an input for the ASP.NET FrontEnd, where the string is parsed into a **DataSet** and serves as the input to various data components.
In our configuration the BackEnd component is deployed on an IIS[7] version 7.5 on windows 7. In order to make a RESTful interface component available on such a system, we had to take into account several configurations, especially concerning the request type of the HTTP request. The allowed request types are restricted to GET and POST by default. The usage of any other request type results in a exception of the IIS, because of a security violation. When deploying a component that implements a RESTful interface these restriction have to be considered. Recently as of mid 2009 there is some effort to include RESTful interfaces into ASP.NET especially with the help of MVC package of ASP.NET. See [17] for a definition and [41] for an implementation, another open source alternative can be found at [47].

BackEnd Sequence View

Similar to the FrontEnd component, every request that is accepted, does not differ significantly from any other in its structure. The main difference of the requests is the request type or verb and the resource that is addressed. Depending on what action is performed on an object, we separate the request in different classes that are responsible for the request type. The basic RESTHandler class forwards the request to the following classes based on the request type. For every request type there is a corresponding RequestHandler, that is **POSTRequestHandler** for the creation of resources and search requests, **PUTRequestHandler** responsible for updates of resources, **DELETERequestHandler** for deletes of resources, and **GETRequestHandler** for retrieving data of resources.
In each of the RequestHandler, we further divide the request by what type of resource is addressed, i.e. user, tag, resource or search objects. For Every object there is a corresponding DAO class that is managing communication with the database. This architectural pattern is based on a J2EE paradigm to decrease the implementation effort to perform separated request to various databases. Although we only address a single database in our implementation we still aim for extensibility of the framework, and try to include as many features as possible for further improvement. In

[7]Internet Information service

the figure 5.11 we see a typical request and the entities that are involved in the work flow. The example demonstrates how a resource object is created.

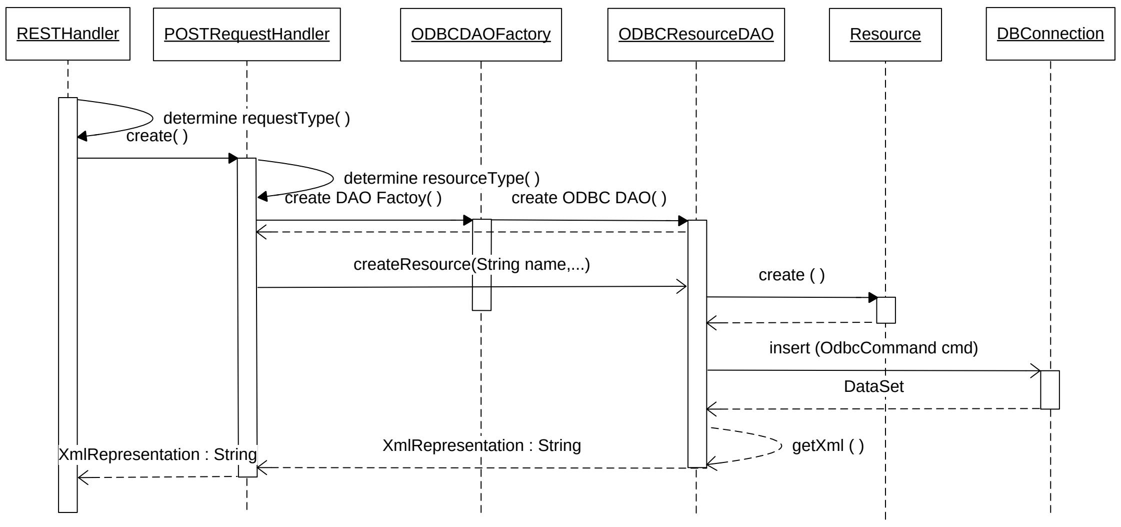


Figure 5.11: Representational BackEnd Sequence View

The BackEnd component is designed as a RESTful service, that awaits a HTTP request. Once the **RESTHandler** object has determined what HTTP method should be performed[8] it forwards it to the appropriate **RequestHandler** object, in this case **POSTRequestHandler** as we create a new resource. Within the **POSTRequestHandler** object a DAO Factory is created that is responsible for creating DAO classes for the requested type of storage mechanism. In this case we are relying on an ODBC connection as out favorite means of persistence and therefore create a **ODBCDAOFactory**, but different types of DAO Factories can be included in this scenario to address multiple persistence layers at once. Resources are stored in the ODBC database in this scenario, but other mechanisms, like XML storage, might be included to stress the peculiarities of the XML content, that is associated with each resource. The ODBCDAOFactory is derived from the abstract class **DAOFactory** that contains the definitions methods for accessing the specific DAO objects. In the ODBCDAOFactoy an ODBCDAO for the corresponding resource is created. Each DAO contains the necessary method to perform persistence and data manipulation on the corresponding datasets. The **ODBCResourceDAO** is responsible for instantiating a **Resource** object and storing the data in the database. Finally a **DataSet** in XML representation can be returned to the requesting user. The architecture of the storage mechanism is designed according to the J2EE DAO pattern that can be found at [51].

Binary Data

As already mentioned in the FrontEnd Sequence View subsection, there are two different types of requests. Requests that return, XML formatted data and request that return binary data. Binary

[8]supported methods are PUT, GET, DELETE, POST

data request have to be handled in a different manner, in order to preserve specialized characters. Requests that store binary data are used to store images for user accounts, and XML description files when creating a resource object. The **GETRequestHandler** and the **POSTRequestHandler** and **PUTRequestHandler** classes are involved in binary data transfers. When requesting the data for a certain user account without including the FrontEnd component, e.g. by using a command line tool like curl[9] to perform HTTP request, the binary data is not automatically included. For retrieving the binary data of a user account image or a XML description file, one has to explicitly query for this data.

SimilarityService Sequence View

The construction of the algorithmic structure, necessary to perform the calculation of the similarity of a resource element to a search query is a resource and time consuming task. In order to still make it possible to retrieve search results for resources in a reasonable time, we designed the service to take a snapshot of the dataset as a foundation for the calculations that are less time consuming and can be performed in real time. On a regular basis we update this structure to keep information consistent with the database. Figure 5.12 shows how this is achieved and what entities are involved in this necessary task.
The Server implementation of this service starts within the **Similarity Service** class that is containing the main class of the Java application. This class instantiates a **SimilarityCalculatorService** object and publishes the services methods as web service operations. The **SimilarityCalculatorService** instantiates a class called TimingThread that extends a TimerTask. As the name suggests this object is responsible for timing the method call for keeping the dataset up to date. Once an instance of the singleton pattern implementing class **SimilarityCalculator** is obtained the method **updateMatrix** triggers an update of the similarity data set.
The **SimilarityCalculator** object is connecting to the database via a **DBConnection** object. As a result for the query for resources and attached tags in the database, a resultSet is obtains, that is directly forwarded to a **DBResultParser** object. This class is responsible for structuring the data from the database, to be usable in the forthcoming operations. The returned documents object representing a Map object, is forwarded to a *VectorGenerator* object that is converting each resource of the database into a vector representation for further calculation.
The method call getMatrix returns a first matrix structure that is used to hold the similarity data between the resources. The **Indexer** object which type is the **IdfIndexer** class, is used to reduce the weight of the tags that are attached to the resources according to their inverse document frequency. These operations are performed on the initial matrix, of resources. Additionally the algorithm explicitly retrieves the documentNames, an identifier of the resources, and the attached tags from the matrix.
A final step is to create a **Searcher** object that is used in later steps to perform the actual similarity calculation between a request and the matrix. The **Searcher** object is equipped with the documentNames, the tags and the termDocumentMatrix and a similarity object. Any incoming request for calculation is now performed on the recently calculated similarityMatrix, stored in the **Searcher** object.

[9]http://curl.haxx.se/

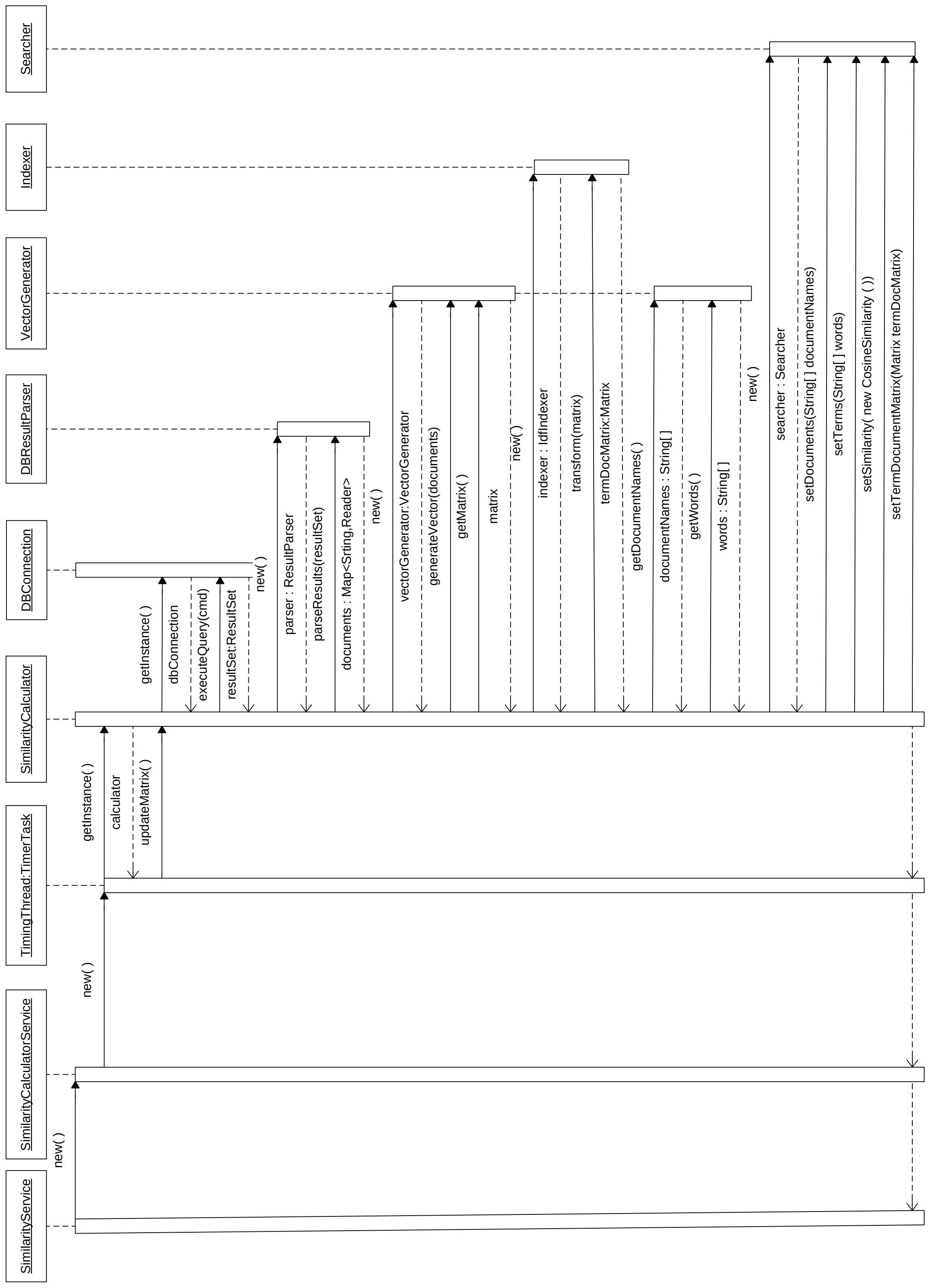


Figure 5.12: Update of Similarity Service DataSet Sequence View

5.3 Database Description

Throughout this section of the paper we will explain the representation of entities in the database. We will discuss the properties and attributes of each table as well as the relations among the connected tables. In Figure 5.13 we can see an UML class diagram of the underlying database scheme of the system.
The storage facility in use in the current implementation of the framework, makes use of a MYSQL database version 5.1. Because of the formatting of the data that is transferred other databases or technologies that have more native XML support are thinkable. Because of the architecture of the system, the database can be easily exchanged with a minimum of effort.

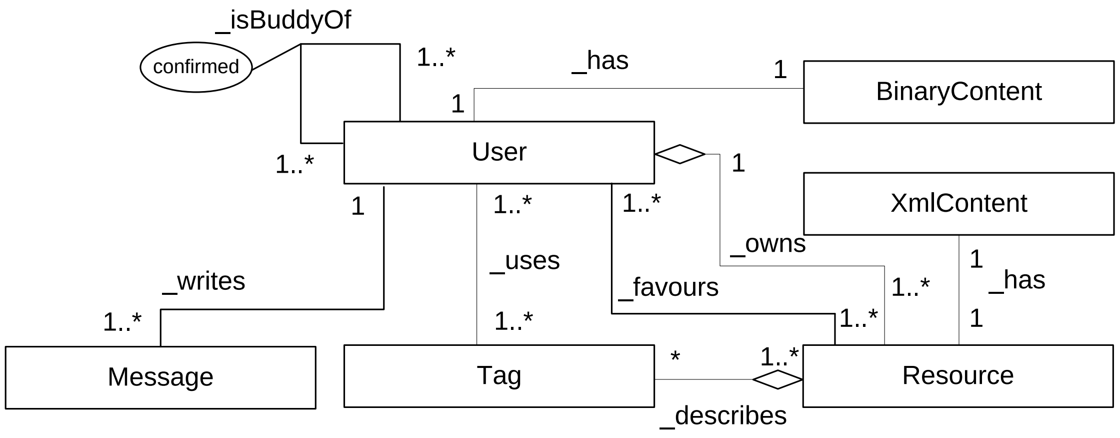


Figure 5.13: Database Description

User - Table Structure

id	INT AUTO INCREMENT PRIMARY KEY
userName	VARCHAR(255) UNIQUE
firstName	VARCHAR(255)
lastName	VARCHAR(255)
email	VARCHAR(255)
password	VARCHAR(255)

Relation _isBuddyOf : User - User

The m:n relation _isBuddyOf describes how users are connected to each other, over their BuddyList. This relation requires for the creation of a new Table in the database: user_user. The primary key is composited from the id's of the user tables. Additionally we introduce an attribute "confirmed" that represents the status of an BuddyList invitation. If the invitation to the buddy list of one user has been accepted by the invited user this field is set to true.

Tags - Table Structure

id	INT AUTO INCREMENT PRIMARY KEY
name	VARCHAR(255)

Resource - Table Structure

id	INT AUTO INCREMENT PRIMARY KEY
name	VARCHAR(255)
description	VARCHAR(255)
uploadDate	DATETIME
ownerId	INT

Ternary Relation _ User-Tag-Resource

A Tag can only exist if it is connected to a resource. We see a m:n relation between the tags table, that is aggregated to the resource table. Additionally every entry of a tag to a resource, has to be identified by a user id, to guarantee identifiability of ones tags. The primary key therefore is compounded of the primary keys of the involved tables: user, tags and resource. A cascade delete is triggered when a user is removing his account.

UserContent - Table Structure

id	INT AUTO INCREMENT PRIMARY KEY
description	VARCHAR(255)
userId	INT
contentLength	INT
binContent	MEDIUMBLOB

The UserContent table is responsible for storing images that the users can upload, to improve the personal experience of the system. Especially in the search functionality for users this a useful feature, as it helps to identify a user via his uploaded image. We decided to store this image in a binary representation as this guarantees a maximum of portability to different platforms. The contentlength attribute has to be specified in order to correctly parse the data when read from the database. The size of a MediumBlob is calculated as follows

$$MediumBlob = L + 3Bytes, whereL \leq 2^{24}$$

which results in approximately 16 MB of storage for each file. We restrict this amount over the FrontEnd implementation.

XmlContent - Table Structure

id	INT AUTO INCREMENT PRIMARY KEY
description	VARCHAR(255)
resourceId	INT
contentLength	INT
binContent	MEDIUMBLOB

This table is used to persist a binary representation of the XML description file that is associated to a resource. While in use, the system is constantly adding the tags that are applied to a resource directly into the structure of the XML file. Additionally a copy of each tag is stored in the database. The information that is gathered in the system, can therefore be moved to different systems with a single mechanism of querying for the binaryContent of a resource.

Message - Table Structure

id	INT AUTO INCREMENT PRIMARY KEY
from_user	INT
to_user	INT
subject	VARCHAR(255)
text	VARCHAR(255)
type	VARCHAR(255)

The Message table is used to persist messages from one user to another, as the means of communication among the participating users. The early intention to use emails to establish communication was dismissed as this approach is totally independent of any third party applications. Id represents the internal id of a message. From_user identifies the user a message is sent from. To_user identifies the user a message is delivered to, whereas subject and text are self explaining. The type of a message defines if it is an invitational message, i.e. the message type is set to "invitational" or just a standard message.

5.4 Mapping of Use Cases to Architectural Elements

To get a better understanding of how the required functions are realized in actual classes, we set up the following table, that combines the use cases with their corresponding architectural elements, respectively their source code files. For a detailed description of the components and architectural elements please refer to section 5.1. For a detailed view on the use cases please refer to section 4.3.

Communication Classes

In order to keep the following list readable we outline the classes that are responsible for enabling communication between the components. Since in every use case a communication between the FrontEnd Component and the BackEnd Component is established, we do not explicitly add these entries to every use case mapping. Instead the reader may be advised, that in fact every use case includes the following classes, to maintain communication.

Component	Package	Implementing Class
FrontEnd	TaggingFrontEnd.Util	HTTPConnection.cs
FrontEnd	TaggingFrontEnd.Util	Constants.cs
BackEnd	TaggingBackEnd	RestHandler.ashx
BackEnd	TaggingBackEnd.Util	Constants.cs

User Dependent Use Case Mapping

The following use case mappings deal with the manipulation of user data. For the sake of simplicity and readability, we outline the classes that are affected in every use case concerning the user entity. These classes are included in every of the following use case mappings. User dependent use cases can be identified by the UcId starting with 1.x and 2.x.

Component	Package	Implementing Class
BackEnd	TaggingBackEnd.DAO	UserDAO.cs
BackEnd	TaggingBackEnd.Interfaces	IUserDAO.cs
BackEnd	TaggingBackEnd.Entities	User.cs

Tag Dependent Use Case Mapping

Similar to the user entity use case mappings, there are classes in the BackEnd component that are solely responsible for the management concerning tag related functionalities. Tag dependent use cases are identified by the UcId starting with 3.x.

Component	Package	Implementing Class
BackEnd	TaggingBackEnd.DAO	TagDAO.cs
BackEnd	TaggingBackEnd.Interfaces	ITagDAO.cs
BackEnd	TaggingBackEnd.Entities	Tag.cs

Resource Dependent Use Case Mapping

Similar to user and tag use case mappings there are classes in the BackEnd component that are solely responsible for the handling of resource concerning functionalities. Resource dependent use cases are identified by the UcId starting with 4.x.

Component	Package	Implementing Class
BackEnd	TaggingBackEnd.DAO	ResourceDAO.cs
BackEnd	TaggingBackEnd.Interfaces	IResourceDAO.cs
BackEnd	TaggingBackEnd.Entities	Resource.cs

HTTP Dependent Use Case Mapping

Depending on what HTTP method the user wants to perform, every request that is accepted by the RESThandler interface is forwarded to a specific requesthandler class.

Component	Package	Implementing Class
BackEnd	RequestHandler	POSTRequestHandler.cs
BackEnd	RequestHandler	PUTRequestHandler.cs
BackEnd	RequestHandler	DELETERequestHandler.cs
BackEnd	RequestHandler	GETRequestHandler.cs

UCId	UCName	Component	Package	Implementing Class
1.1	LogIn			
		FrontEnd	TaggingFrontEnd.UI	Login.aspx; Login.aspx.cs
		BackEnd	TaggingBackEnd.RequestHandler	POSTRequestHandler.cs
1.2	LogOut			
		FrontEnd	TaggingFrontEnd.UI	Lougout.aspx; Logout.aspx.cs
2.1	Create UserAccount			
		FrontEnd	TaggingFrontEnd.UI.user	UserCreate.aspx; UserCreate.aspx.cs
		BackEnd	TaggingBackEnd.RequestHandler	POSTRequestHandler.cs
		BackEnd	TaggingBackEnd.Util	DBConnection.cs
2.2	Update UserAccount			
		FrontEnd	TaggingFrontEnd.UI.user	UserUpdate.aspx; UserUpdate.aspx.cs
		BackEnd	TaggingBackEnd.RequestHandler	PUTRequestHandler.cs
		BackEnd	TaggingBackEnd.Util	DBConnection.cs
2.3	Delete UserAccount			
		FrontEnd	TaggingFrontEnd.UI.user	UserOverview.aspx; UserOverview.aspx.cs
		BackEnd	TaggingBackEnd.RequestHandler	DELETERequestHandler.cs
		BackEnd	TaggingBackEnd.Util	DBConnection.cs
2.4	Search for UserAccount			
		FrontEnd	TaggingFrontEnd.UI.user	UserSearch.aspx ; UserSearch.aspx.cs
		BackEnd	TaggingBackEnd.DAO	SearchDAO.cs
		BackEnd	TaggingBackEnd.Interfaces	ISearchDAO.cs
		BackEnd	TaggingBackEnd.RequestHandler	POSTRequestHandler.cs
2.5	Add UserAccount to BuddyList			
		FrontEnd	TaggingFrontEnd.UI.user	UserSearch.aspx
		BackEnd	TaggingBackEnd.RequestHandler	POSTRequestHandler.cs
		BackEnd	TaggingBackEnd.Interfaces	IMessageDAO.cs
		BackEnd	TaggingBackEnd.DAO	MessageDAO.cs

2.6	Delete User from BuddyList			
		FrontEnd	TaggingFrontEnd.BuddyApplet.app	BuddyApplet.java
		FrontEnd	TaggingFrontEnd.BuddyApplet.util	HTTPConnection.java
		FrontEnd	TaggingFrontEnd.BuddyApplet.menu	MouseMenu.java
		FrontEnd	TaggingFrontEnd.BuddyApplet.menu	DeleteBuddyRelationMenuItem.java
		BackEnd	TaggingBackEnd.Entities	User.cs
		BackEnd	TaggingBackEnd.Interfaces	IUserDAO.cs
		BackEnd	TaggingBackEnd.DAO	UserDAO.cs
		BackEnd	TaggingBackEnd.RequestHandler	DELETERequestHandler.cs
		BackEnd	TaggingBackEnd.Util	DBConnection.cs
2.7	Confirm BuddyList Invitation			
		FrontEnd	TaggingFrontEnd.UI.message	MessageOverView.aspx; MessageOverView.cs
		BackEnd	TaggingBackEnd.RequestHandler	POSTRequestHandler.cs
		BackEnd	TaggingBackEnd.Util	DBConnection.cs
		BackEnd	TaggingBackEnd.Interfaces	IMessageDAO.cs
		BackEnd	TaggingBackEnd.DAO	MessageDAO.cs
2.8	Decline BuddyList Invitation			
		FrontEnd	TaggingFrontEnd.UI.message	MessageOverview.aspx; MessageOverview.cs
		BackEnd	TaggingBackEnd.RequestHandler	POSTRequestHandler.cs
		BackEnd	TaggingBackEnd.Util	DBConnection.cs
3.1	Create Tag			
		FrontEnd	TaggingFrontEnd.UI.tag	TagCreate.aspx; TagCreate.aspx.cs
		BackEnd	TaggingBackEnd.RequestHandler	POSTRequestHandler.cs
		BackEnd	TaggingBackEnd.DAO	ResourceDAO.cs
		BackEnd	TaggingBackEnd.Interfaces	IResourceDAO.cs
		BackEnd	TaggingBackEnd.Util	DBConnection.cs

3.2	Delete Tag			
		FrontEnd	TaggingFrontEnd.UI.tag	TagOverview.aspx; TagOverview.aspx.cs
		BackEnd	TaggingBackEnd.RequestHandler	DELETERequestHandler.cs
		BackEnd	TaggingBackEnd.DAO	ResourceDAO.cs
		BackEnd	TaggingBackEnd.Interfaces	IResourceDAO.cs
		BackEnd	TaggingBackEnd.Util	DBConnection.cs
3.3	Search For Tag			
		FrontEnd	TaggingFrontEnd.UI.tag	TagOverview.aspx ; TagOverview.aspx.cs
		BackEnd	TaggingBackEnd.RequestHandler	POSTRequestHandler.cs
		BackEnd	TaggingBackEnd.DAO	SearchDAO.cs
		BackEnd	TaggingBackEnd.Interfaces	ISearchDAO.cs
4.1	Create Resource			
		FrontEnd	TaggingFrontEnd.UI.resource	ResourceCreate.aspx; ResourceCreate.aspx.cs
		BackEnd	TaggingBackEnd.RequestHandler	POSTRequestHandler.cs
		BackEnd	TaggingBackEnd.Util	DBConnection.cs
4.2	Update Resource			
		FrontEnd	TaggingFrontEnd.UI.resource	ResourceUpdate.aspx; ResourceUpdate.aspx.cs
		BackEnd	TaggingBackEnd.RequestHandler	PUTRequestHandler.cs
		BackEnd	TaggingBackEnd.Util	DBConnection.cs
4.3	Delete Resource			
		FrontEnd	TaggingFrontEnd.UI.resource	ResourceOverview.aspx; ResourceOverview.aspx.cs
		BackEnd	TaggingBackEnd.RequestHandler	DELETERequestHandler.cs
		BackEnd	TaggingBackEnd.Util	DBConnection.cs

4.4.	Search for Resource			
		FrontEnd	TFE[a].UI.resource	ResourceSearch.aspx;
		FrontEnd	TFE.UI.resource	ResourceSearchResult.aspx;
		FrontEnd	TFE.TreeVisualization.app	TreeVisualization.java
		FrontEnd	TFE.TreeVisualization.util	XMLParser.java
		FrontEnd	TFE.TreeVisualization.util	HTTPConnection.java
		FrontEnd	TFE.TreeVisualization.util	Constants.java
		FrontEnd	TFE.TreeVisualization.menu	MouseMenu.java
		FrontEnd	TFE.TreeVisualization.menu	PopUpVertexEdgeMenuMousePlugin.java
		FrontEnd	TFE.TreeVisualization.menu	ShowResourceDetailsMenuItem.java
		FrontEnd	TFE.TreeVisualization.graphelements	MultiNode.java
		FrontEnd	TFE.TreeVisualization.graphelements	SingleNode.java
		BackEnd	TBE[b].RequestHandler	POSTRequestHandler.cs
		BackEnd	TBE.DAO	SearchDAO.cs
		BackEnd	TBE.Interfaces	ISearchDAO.cs
		SimilarityService	SimilarityService.similarity	SimilarityCalculator.java
		SimilarityService	SimilarityService.clustering	TagClusterCalculator.java
5.1	Create Message			
		FrontEnd	TFE.UI.message	MessageCreate.aspx; MessageCreate.cs
		BackEnd	TBE.RequestHandler	POSTRequestHandler.cs
		BackEnd	TBE.Util	DBConnection.cs
		BackEnd	TBE.Interfaces	IMessageDAO.cs
		BackEnd	TBE.DAO	MessageDAO.cs
5.2	Delete Message			
		FrontEnd	TFE.UI.message	MessageOverview.aspx;
		FrontEnd	TFE.UI.message	MessageOverview.cs;
		BackEnd	TBE.RequestHandler	DELETERequestHandler.cs
		BackEnd	TBE.Util	DBConnection.cs
		BackEnd	TBE.Interfaces	IMessageDAO.cs
		BackEnd	TBE.DAO	MessageDAO.cs

[a]TaggingFrontEnd
[b]TaggingBackEnd

5.3	View Message			
		FrontEnd	TFE.UI.message	MessageOverview.cs;MessageOverview.apsx
		BackEnd	TBE.RequestHandler	GETRequestHandler.cs
		BackEnd	TBE.Util	DBConnection.cs
		BackEnd	TBE.Interfaces	IMessageDAO.cs
		BackEnd	TBE.DAO	MessageDAO.cs
6.1	Browse ResourceSet			
		FrontEnd	TFE[a].UI.resource	ResourceSearch.aspx;
		FrontEnd	TFE.UI.resource	ResourceSearchResult.aspx;
		FrontEnd	TFE	TreeVisualizationApplet.jar
		BackEnd	TBE[b].RequestHandler	POSTRequestHandler.cs
		BackEnd	TBE.DAO	SearchDAO.cs
		BackEnd	TBE.Interfaces	ISearchDAO.cs
		SimilarityService	SimilarityService.similarity	SimilarityCalculator.java
		SimilarityService	SimilarityService.clustering	TagClusterCalculator.java
7.1	Browse BuddyList			
		FrontEnd	TFE.BuddyApplet.app	BuddyApplet.java
		FrontEnd	TFE.BuddyApplet.util	XMLParser.java
		FrontEnd	TFE.BuddyApplet.util	HTTPConnection.java
		FrontEnd	TFE.BuddyApplet.util	GraphCreator.java
		FrontEnd	TFE.BuddyApplet.graphelements	ResourceVertex.java
		FrontEnd	TFE.BuddyApplet.graphelements	UserVertex.java
		FrontEnd	TFE.BuddyApplet.menu	MouseMenu.java

[a]TaggingFrontEnd
[b]TaggingBackEnd

		FrontEnd	TFE.BuddyApplet.menu	PopUpVertexEdgeMenuMousePlugin.java
		FrontEnd	TFE.BuddyApplet.menu	ShowCorrespondingResourceMenuItem.java
		FrontEnd	TFE.BuddyApplet.menu	ShowResourceDetailsMenuItem.java
		BackEnd	TBE.Entities	User.cs
		BackEnd	TBE.Interfaces	IUserDAO.cs
		BackEnd	TBE.DAO	UserDAO.cs
		BackEnd	TBE.Util	DBConnection.cs
8.1	Update Similarity Structure			
		SimilarityService	SimilarityService.util	TimingThread.java
9.1	Calculate Similarity Structure			
		SimilarityService	SimilarityService.util	DBConnection.java
		SimilarityService	SimilarityService.util	DBResultParser.java
		SimilarityService	SimilarityService.util	TimingThread.java
		SimilarityService	SimilarityService.util	SimilarityCalculator.java
		SimilarityService	SimilarityService.util	SimilarityCalculator.java
		SimilarityService	SimilarityService.indexers	VectorGenerator.java [a]
		SimilarityService	SimilarityService.indexers	TfIndexer.java
		SimilarityService	SimilarityService.recognizers	BoundaryRecognizer.java
		SimilarityService	SimilarityService.recognizers	StopWordRecognizer.java
		SimilarityService	SimilarityService.recognizers	RecognizerChain.java
		SimilarityService	SimilarityService.recognizers	IRecognize.java
		SimilarityService	SimilarityService.similarity	AbstractSimilarity.java
		SimilarityService	SimilarityService.similarity	CosineSimilarity.java
		SimilarityService	SimilarityService.similarity	Searcher.java
		SimilarityService	SimilarityService.tokenizer	Token.java
		SimilarityService	SimilarityService.tokenizer	TokenType.java
		SimilarityService	SimilarityService.tokenizer	WordTokenizer.java

[a] Additional source files from GNU Library or Lesser General Public License (LGPL) available code, see [61] and [62]

CHAPTER 6

Evaluation and Discussion

In the following section we introduce a series of screen shots of the actual implementation, that are supposed to give an insight how the framework is going to perform and how certain features are realized. We refer to the complex use case scenario in section 4.4 and outline the features that seem to be the most important. Further we motivate a discussion on certain design issues and what alternatives exist to the current solution.

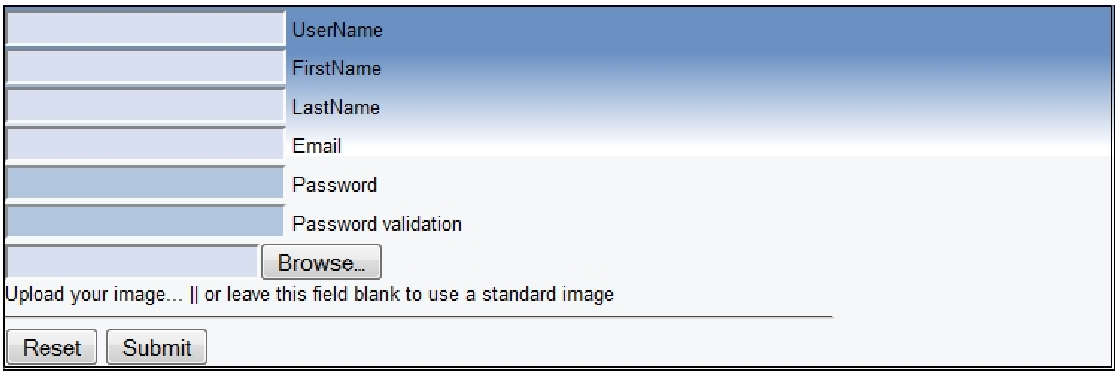
6.1 Realization Aspects

Registration Process

As a precondition to every activity in the community, every user has to provide a set of certain properties. In Figure 6.1 you can see the data that can be gathered, to make the experience for the user more personal. The Username is one property that is required and is validated for uniqueness against existing user accounts, before a new user account is created. The second property that can not be omitted is the email address, as it is in use if the user requests a new password. The email address is solely used for the purpose of reseting ones password, since communication within the framework is managed over an internal messaging system. Another optional feature that is introduced to make the user experience more personal is the functionality to upload personalized user images, to identify ones user account. If this field is left free within the registration process, a standard image is applied to the user account.

Creation of a Resource

Once the user has successfully created a new user account and logged in into the system, a user can publish services, that can be integrated in one others work flow, by creating a new resource. In Figure 6.2 you can see the realization of this use case. The user specifies an name to identify a resource, can optionally add a description to give provide a better understanding of the features of the resource, and has to specify an description of the service in xml format. In most cases that

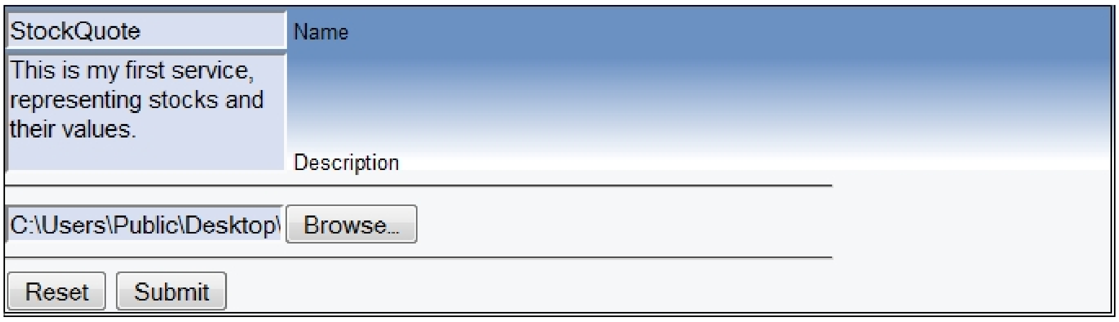


	UserName
	FirstName
	LastName
	Email
	Password
	Password validation

Browse...
Upload your image... || or leave this field blank to use a standard image
Reset Submit

Figure 6.1: Registration Process

will be a WSDL file, but other description files with different structure can be included in the system. The only prerequisite is a valid and well formed xml description file. Once the resource data is submitted to the system, the user can start tagging the just created resource. Without applying any tags to the resource, the service is not going to be found in any search requests. Additionally the resource is more likely to be found the more varying tags are applied to it.



StockQuote	Name
This is my first service, representing stocks and their values.	Description

C:\Users\Public\Desktop\ Browse...
Reset Submit

Figure 6.2: Create a Resource

Search Results in Tree Visualization

The search for a resource, i.e. a service that has been published by another user, results in the screen in Figure 6.3. In this scenario the user was searching for resources that are related to the tag "cool", cf. the example of the similarity calculation in section 2. We can see a tree representation of the search results, with the leaves being representations of the tags, combining to larger clusters, up to the root node. Each node offers a context menu that can be opened with a right click on the specific item. At each node we can see a list of tags that are contained within this node and the associated resources to each tag. On a click on one of the entries the user can add a resource to his list of favored items. Within this list he can apply tags to the resources. To give the user the chance to re arrange the graph for better view ability, the user can pick nodes from the graph and move them to another location within the frame.

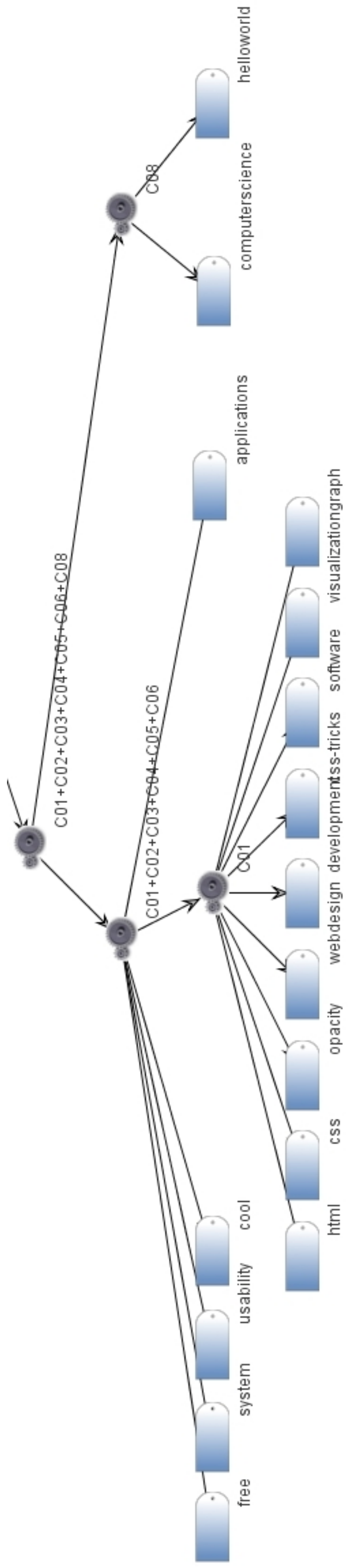
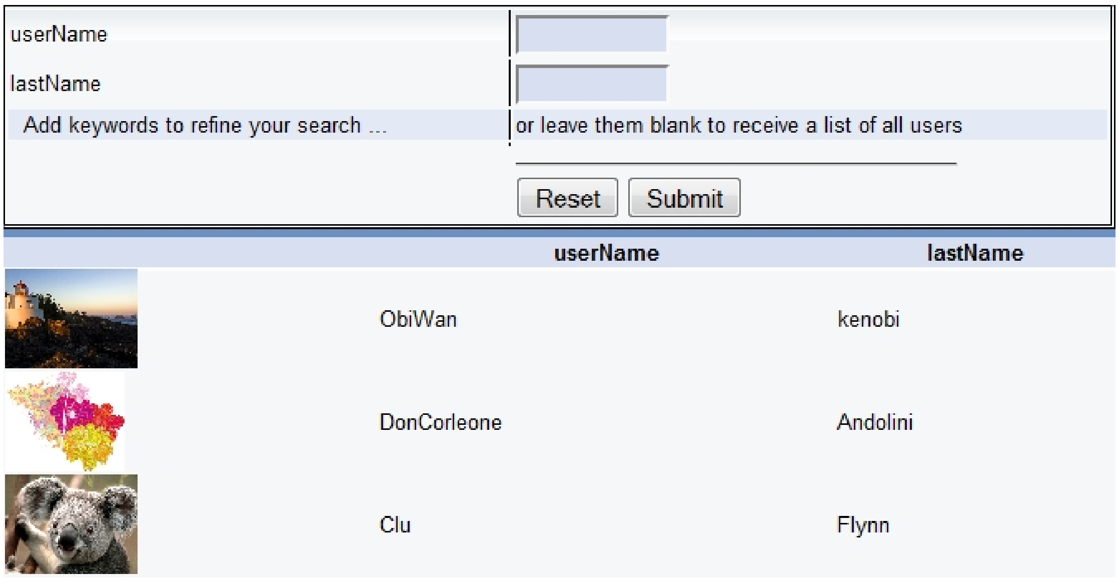


Figure 6.3: Example Search Result

Search Results Searching for Users

The search for a user is represented by the screen shot in Figure 6.4. Searching for users is the entry point to adding users to ones BuddyList. Without specifying any search criteria the system will return a list of all available users. To limit the results to a smaller amount the user can specify certain properties, that narrow the retrieved entries. Besides the username and the lastname, the user can additionally add keywords to the search. Keywords represent the tags a user has used to identify a resources in the system. Up to three keywords that are logically combined, can be attached to each search query. Once the search results are returned from the system, the user can view a list of the users that meet the given criteria. When hovering with the mouse over the entries a context menu pops up, displaying the details of the users. On a click on the "Send Buddy Request" Button the user initiates a BuddyList request process. The newly requested BuddyList relationship is only visible if the invited user accepts the invitation to join the users list.

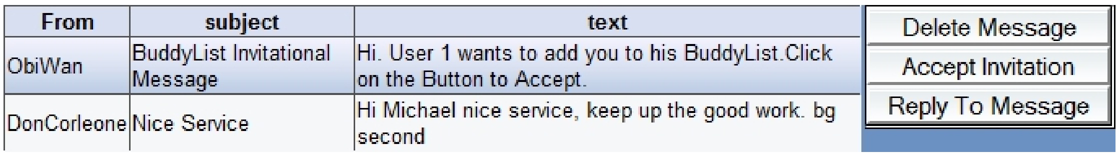

userName
lastName
Add keywords to refine your search ...
or leave them blank to receive a list of all users
Reset
Submit

	userName	lastName
	ObiWan	kenobi
	DonCorleone	Andolini
	Clu	Flynn

Figure 6.4: Search Result Searching for all Users

Messaging Functionality

To complete the BuddyList invitational process an invitational message is sent to the invited user. Directly after login into the system the user is forwarded to the list of incoming messages. When hovering over a message entry, a context menu pops up, offering the following choices. (i)The message can be deleted, if it is not of use any longer. (ii) The user can post a reply to the message, or depending if the message was an invitational message can accept the invitation and finish the BuddyList invitational process. On accepting the BuddyList invitation both users can see the tags a user has tagged. Besides invitational messages the users can use the messaging subsystem, to communicate within the system, to foster good relation ship.



From	subject	text
ObiWan	BuddyList Invitational Message	Hi. User 1 wants to add you to his BuddyList.Click on the Button to Accept.
DonCorleone	Nice Service	Hi Michael nice service, keep up the good work. bg second

Figure 6.5: Messaging Overview

BuddyList Visualization

Once a BuddyList relationship has been established between two users, both can see the tags they have been attaching to resources as is visible in Figure 6.6. On a right click a context menu pops up and offers the user to delete the BuddyList relation, or if it is a resource the user has clicked on, offers the possibility to view details and add it to ones own list of favored resources.

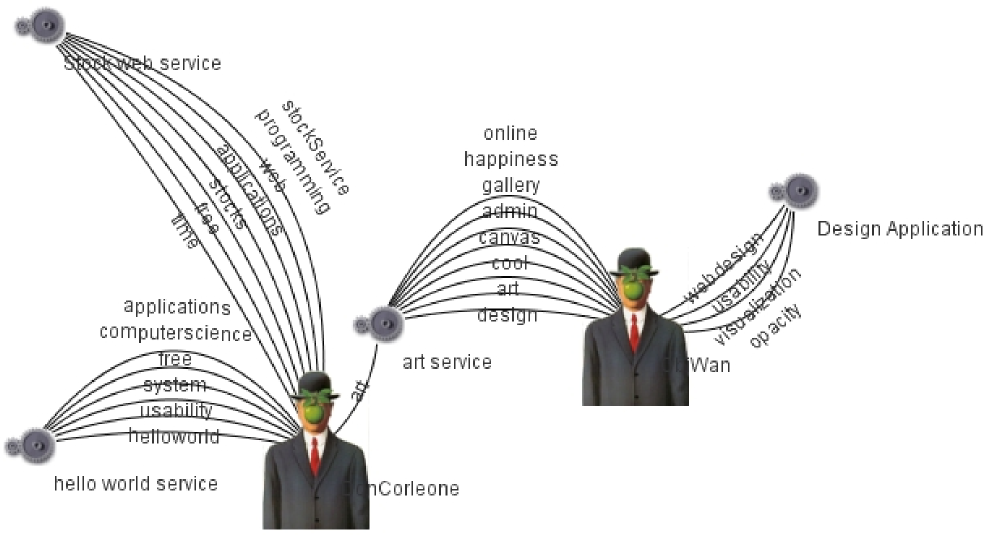


Figure 6.6: Browse Through BuddyList

6.2 Mapping of Properties to Implemented Features

Throughout the document we presented properties that define a successful web 2.0 application and introduced a set of design principles that have to be considered while creating a social networking application. In this section we want to map conceptual properties to implementation facets and architectural aspects of the system.

Web 2.0 Principles

In regard to the web 2.0 principles defined as of O' Reilly are realized in the framework as follows. The architecture of the framework aims to realize the principle for a *lightweight programming model* as (i) the data that is to be stored can be of *arbitrary XML structure*, similar to the approach of ebXML, which makes the framework *open* for a variety of data, as long as the description file is based on a valid XML structure. (ii) The architecture is designed to be extensible towards future extensions of the framework and because of this flexible approach, allows the main components to be deployed on various machines. (iii) The interface exposes a RESTful communication paradigm that is designed for simplicity in use and interoperability with other services and components.
The data that is gathered in common social networks is of great benefit for many parties, either for commercial but also for non-profit interests. But in contrast to our framework the data is usually not further processed to add an *additional layer of meta information* the user can benefit of. With the similarity structure we calculate, as response to the search requests, the system creates a *source of data* that is of more value than common data about the structure of applied tags, as it yields even more implicit information of how the resources are related to each other.
From an information retrieval perspective we follow a *pull technology* based approach as the users of the framework have to *actively seek for information* they are interested in. In contrast to the work of Treiber et al., in [81] where feeds and notifications are used as a distributional mechanism, we rather rely on a static search engine located on a specific machine. The algorithm to create the meta data about the services and their semantic relation is not suited to be distributed over several peers, thus has to be *maintained on a central authority*.

Tagging Framework Principles

Tagging adds another facet to the framework, as it realizes the principles of a *lightweight approach to classify large sets of data*, introduced in the clustering section of this document and in [14] by Christiaens et al. Implicit categorization and data set structuring is achieved by applying the similarity calculations to the given data.
The system we implemented realizes the ideas of a *free for all tagging framework* with *viewable tagging principles*. Users can tag any resource with arbitrary descriptions even if they are not the creator of a specific resource. To extend the tagging mechanism users can also choose from a set of previously used tags to identify a certain resource. The complexity of the tag recommendation to tag a resource, is comparably low to other systems, as we do not implement any strategy to recommend a tag for a resource. In a future work this could improve the handling of the framework even more.
To be able to perform the computation to determine the similarity of one resource to another, we have to rely on the *bag model*, we introduced in an earlier section of the document. The data every single user adds to the resource set is essential for the calculation of the similarity, hence *duplicate entries of tags to a resources is permitted*, but a single user cannot use a keyword to tag the same resource more than once.
We follow a user centric approach when it comes to the *sources of material* of the system, as the users have total control over the uploaded data. In addition to the lightweight programming model, we mentioned earlier, this results in a very flexible system, that allows the users to gain

access to a wide range of content.
In the design process of the framework we did not intend any *resource connectivity* as defined in the design principle section. Although resources might contain cross references to other resources that are available in the system, we did not actvily seek to integrate these aspects in the framework. Instead it was our intention to keep the resource connectivity as low as possible as this artificially increases the complexity of the relation of the resources, and has no effect on the calculation of the similarity. If the types of objects that are handled within a social networks are not exclusively service descriptions, but more dynamic content, a system might benefit from a high degree of resource connectivity, but in this implementation we do not think this is of relevance.
On the other hand *social connectivity* plays an essential role in the framework, as it is the main feature of collaborative networks. To realize this feature we applied the concept of buddy lists to our framework, that helps improve the usage of the system, as it supports the discovery of services.

6.3 Similarity Calculation

The algorithm to calculate the similarity structure is a non trivial problem, and due to the nature of the data structure that is necessary for the calculation, the amount of users participating in the tagging process and the amount of tags and resources, has a certain amount of influence on the runtime that can not be neglected. To minimize the impact of the complex data structure on the scalability we aspired a distributed architecture, to exploit the advantages of multiple machines and their resources. The off line step to calculate the similarity structure the search requests are operating on, results in an improved user experience. The response time of the actual request is kept at a reasonable level, since important structures are already available.
The similarity calculation creates a data structure that is different to common representations for search results. Although the retrieved data requires a certain amount of initial training, the output appears promising enough to improve the traditional ways of discovery and retrieval of references to services.
The resulting structure is purely based on the activities of the using community and implicitly reflect the habits and the preferences of the users of the system.

Runtime

In order to evaluate the runtime of the similarity calculation and the tree structure construction, we set up a scenario that is used to predict the behavior once the usage of the systems tends to rise. In order to derive a potential progress we performed three iterations. As a constant we created ten resources that are associated with a varying number of tags. In the first pass we associated 10 tags to the existing 10 resource in a uniform way, resulting in 100 associations. The tags are are the numbers from 1 to 10, every resource is tagged the equal amount with the same tags thus the expected result for a search query is as likely as any other. The average runtime for a search and the tree structure creation is: 1205,4 ms. In the second iteration we associated 100 tags, numbered from 1 to 100, to the 10 resources we stored in the data base, resulting in 1000 associations. Again the tags are equally distributed over the resources. The average runtime for a search and the subsequent tree creation is: 2979,6 ms. In the third and final pass, we created 1000 tags and associated them in a uniform way to the existing 10 resources, resulting in an average runtime for search and tree creation of: 8462,1 ms.

6.4 Architectural Alternatives

Communication Pattern

From a communication pattern view the system is currently implementing a synchronous messaging pattern where each response is immediately followed by a response. The BackEnd component is requesting results from the similarity service which in returns the similarity structure for the search request. From the runtime evaluation we can conclude that this communicational pattern will reach its boundaries as soon as there is a certain amount of usage of the system. As an communicational alternative, to improve the scalability of the system a possible solution would be to switch the subsystem to search for resources to a asynchronous messaging pattern. Incoming search request would be queued in the similarity service and when the computation has finished, a notification could be sent to the request or including the results for the query. We argument that including a notification strategy would increase the overhead for the search request and is only an improvement for the system if the number of association reaches a certain amount. Until then performance and usability of the system is probably better when using a synchronous messaging pattern.

Security and Authorization Mechanisms

In the current state of the implementation, which is only at a prototype level, we did not include any security or authorization mechanism, since these are features that only have significance for a deployed system and do not affect the behavior of the interface but increase the overall overhead and decrease the performance of the features of the system. We primarily focused on a basic architecture for such a system and the adaption of information retrieval algorithms to add new facets to social networking applications. If a system like this is reaching a development level where it is going to be deployed outside a secure environment, certain features have to be considered.

When SOAP communication is being used, a set of specifications known as WS- Security can be found at [54] that enable a secure channel. Since our architecture follows a different paradigm this is not applicable in full. When implementing a RESTful architecture it is common practice to incorporate a SSL/TLS[1], which is hybrid cryptographic protocol to secure communication between two peers, to avoid certain security risks, e.g. like interception of data or data injection. These security layer can be established between the FrontEnd and the BackEnd component of the system, but also when the BackEnd is communicating with the similarity service.

[1]Secure Socket Layer/Transport Layer Security

To avoid unauthorized access to the system we implemented a very basic authentication mechanism that identifies a user by his username and his password, similar to the basic authentication of the HTTP. To avoid human readable data is transfered over non secure channels this type of basic authentication uses Base64 to encrypt the data. Basic authentication is only applicable if it is used in combination with SSL/TLS since it is still possible to intercept the data and decrypt it because of the weak encryption algorithm in use. Another alternative is to use digest access authentication that has a superior security concept included already, cf.[1] for further information on that topic.

The RESTful architectural paradigm proves to be very efficient for our needs. As already stated it is a lightweight communication pattern that reduces the communication overhead and focuses on rather simple mechanism. Additional features, that possibly reside on separate peers, can be integrated in a straight forward way, by simply adapting the process of parsing the incoming HTTP request and forwarding the necessary information. That makes the BackEnd component a broker that is dispatching certain events to various components, be it on the local machine or remote.

Database Issues

The conceptual architecture, we introduced in an earlier section, suggests that the data, to set up the necessary structure to query the similarity for a certain keyword, is pulled directly from the database without having to interact with an additional component. Since this communication is read- only there is no need for a synchronization mechanism. The big advantage of this approach is that querying the data from the database directly is much more efficient than having to interfere with a component that has to preprocess the data.

For each resource that is created we store the XML data in the database. Instead of using a database for that purpose there was the opportunity to store the data directly on the file system. From a computational perspective it does not make a difference whether the XML data is read from the file system or from a database since the XML structure has to be mapped to a memory representation either way, when dealing with the content. The advantage of using only a relational database management system as a means of storage is that the responsibility for the data is in a single centrally managed unit. In an architecture that is operating in an service oriented environment, it might be desirable to distribute the responsibility for computational complex tasks to different peers, but the gain in reliability is comparably higher than the loss of independence of the components, when centrally managing the data basis.

CHAPTER 7

Conclusion and Future Work

We have analyzed the properties that have to be considered to achieve the requirements of a successful social networking application, and have adopted these insights to demonstrate what an architecture has to look like to realize essential features of social networks, with the use of state of the art technologies. Not only has the architecture been designed to cover the necessary features of modern web 2.0 applications, but also did we strive for a maximum of re- usability and maintainability, especially in regards to potential future extensions of the system.

Since the system at hand is only a prototype implementation there is still room for improvements. Especially the algorithm to create a similarity structure is limited in its capability to serve the demands for a massive user community, as the analysis of the runtime has shown. We started to discus how to overcome the deficiencies of the similarity algorithm, and pointed out architectural alternatives for certain elements of the framework. In our implementation we adapted various programming models and communication patterns that are essential to service oriented environments, and created a representational framework, that reflects the ongoing tendencies in the evolution of web applications.

As a preliminary step to the future work our first task has to be to gather feedback from a community of users, that is making actual use of the framework and the features that are currently implemented. The input we receive from this initial interaction is the basis for identifying the requirements of a future revision of the framework in regards to performance of the overall system and functions that potentially might improve the usability and ease the interaction with the system. Furthermore the impact of the integration of the similarity algorithm on search requests is not yet obvious in full, but seems promising enough to keep further investigating about the influence these aspects have on the habits a community of users has, using social networking applications.

Bibliography

[1] The Internet Society (1999). HTTP Authentication: Basic and Digest Access Authentication, 1999. http://tools.ietf.org/html/rfc2617.

[2] The Internet Society (1999). Hypertext Transfer Protocol – HTTP/1.1, 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[3] Christine Albrecht. Folksonomy. Master's thesis, TU Wien, 2006.

[4] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Gariapathy, and K. Holley. SOMA: A Method for Developing Service-Oriented Solutions. *IBM Syst. J.*, 47:377–396, 2008.

[5] Shenghua Bao, Guirong Xue, Xiaoyuan Wu, Yong Yu, Ben Fei, and Zhong Su. Optimizing Web Search Using Social Annotations. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 501–510, 2007.

[6] G. Begelman, P. Keller, and F. Smadja. Automated Tag Clustering: Improving Search and Exploration in the Tag Space. In *Collaborative Web Tagging Workshop at WWW2006, Edinburgh, Scotland*, 2006.

[7] Dan Brickley and Libby Miller. The Friend of a Friend (FOAF) project, 2010. http://www.foaf-project.org/.

[8] Christopher H. Brooks and Nancy Montanez. Improved Annotation of the Blogosphere via Autotagging and Hierarchical Clustering. In *Proceedings of the 15th International Conference on World Wide Web (WWW2006*, pages 625–632, 2006.

[9] Ciro Cattuto, Dominik Benz, Andreas Hotho, and Gerd Stumme. Semantic Grounding of Tag Relatedness in Social Bookmarking Systems. In *International Semantic Web Conference*, pages 615–631, 2008.

[10] Patrick Cauldwell. *Professional XML Web Services*. Wrox Press, Birmingham, England, 2001.

[11] H. Chen and S.T. Dumais. Bringing order to the web: Automatically categorizing search results. In *Proceedings of CHI-00, ACM International Conference on Human Factors in Computing Systems, Den Haag*, pages 145–152, 2000.

[12] Roberto Chinnici, Jeans-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, 2010. http://www.w3.org/TR/wsdl20/.

[13] Erik Christensen, Francisco Cubera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL), 2010. http://www.w3.org/TR/wsdl.

[14] Stijn Christiaens. Metadata Mechanisms: From Ontology to Folksonomy ... and back. *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, pages 199–207, 2006.

[15] Christina Yip Chung, Raymond Lieu, Jinhui Liu, Alpha Luk, Jianchang Mao, and Prabhakar Raghavan. Thematic Mapping - from unstructured documents to taxonomies. In *CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management*, pages 608–610, New York, NY, USA, 2002.

[16] Microsoft Corporation. AJAX Control Toolkit, 2010. http://ajaxcontroltoolkit.codeplex.com/.

[17] Microsoft Corporation. ASP.NET MVC The official Microsoft ASP.NET Site, 2010. http://www.asp.net/mvc.

[18] Microsoft Corporation. The official Microsoft ASP.NET Site, 2010. http://www.asp.net/.

[19] Douglass R. Cutting, David R. Karger, Jan O. Pedersen, and John W. Tukey. Scatter/gather A Cluster-Based Approach to Browsing Large Document Collections. In *Proceedings of the Fifteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Interface Design and Display, pages 318–329, 1992.

[20] E. W. Dijkstra. Ewd 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[21] Xin Dong, Alon Halevy, Jayant Madhavan, Ema Nemes, and Jun Zhang. Similarity search for web services. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 372–383, 2004.

[22] Schahram Dustdar and Martin Treiber. A View Based Analysis on Web Service Registries. *Distributed and Parallel Databases*, 18(2):147–171, 2005.

[23] Roy T. Fielding, Day Software, and Richard N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology*, 2:115–150, 2002.

[24] Roy Thomas Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, 2000.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Reading, MA, USA, 1995.

[26] E. Garcia. An Information Retrieval Tutorial on Cosine Similarity Measures, Dot Products and Term Weight Calculations, 2010. http://www.miislita.com/information-retrieval-tutorial/cosine-similarity-tutorial.html.

[27] John Garofalakis, Yannis Panagis, Evangelos Sakkopoulos, and Athanasios Tsakalidis. Web Service Discovery Mechanisms: Looking for a Needle in a Haystack? In *In: International Workshop on Web Engineering*, 2004.

[28] Jonathan Gemmell, Andriy Shepitsen, Bamshad Mobasher, and Robin Burke. Personalizing Navigation in Folksonomies Using Hierarchical Tag Clustering. In *DaWaK '08: Proceedings of the 10th international conference on Data Warehousing and Knowledge Discovery*, pages 196–205, 2008.

[29] Scott A. Golder and Bernardo A. Huberman. Usage Patterns of Collaborative Tagging Systems. *Journal of Information Science*, 32:198–208, 2006.

[30] Harry Halpin, Valentin Robu, and Hana Shepherd. The complex dynamics of collaborative tagging. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 211–220, New York, NY, USA, 2007. ACM.

[31] Y. Hassan-Montero and V. Herrero-Solana. Improving Tag-Clouds as Visual Information Retrieval Interfaces. In *InScit2006: International Conference on Multidisciplinary Information Sciences and Technologies*, 2006.

[32] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction.* Springer, 2 edition, 2009.

[33] Paul Heymann and Hector Garcia-Molina. Collaborative Creation of Communal Hierarchical Taxonomies in Social Tagging Systems. Technical Report 2006-10, Stanford University, 2006.

[34] Paul Heymann, Georgia Koutrika, and Hector Garcia-Molina. Can Social Bookmarking Improve Web Search? In *WSDM '08: Proceedings of the International Conference on Web search and Web data mining*, pages 195–206, 2008.

[35] Andreas Hotho, Robert Jäschke, Christoph Schmitz, and Gerd Stumme. Information Retrieval in Folksonomies: Search and Ranking. In *Proceedings of the 3rd European Semantic Web Conference*, pages 411–426, 2006.

[36] Jason Hunter. JDOM, 2010. http://www.jdom.org/.

[37] Google Inc. Google Trends on E-Commerce and Social Networking, 2010. http://www.google.com/trends?q=e-commerce,+social+networking.

[38] Sarah Radwanick Comscore Inc. The 2009 U. S. Digital Year in Review, 2010. http://www.comscore.com/Press_Events/Presentations_Whitepapers/2010/The_2009_U.S._Digital_Year_in_Review.

[39] Robert Jäschke, Leandro Balby Marinho, Andreas Hotho, Lars Schmidt-Thieme, and Gerd Stumme. Tag Recommendations in Folksonomies. In *Knowledge Discovery in Databases: PKDD 2007, 11th European Conference on Principles and Practice of Knowledge Discovery in Databases, Warsaw, Poland, September 17-21, 2007, Proceedings*, volume 4702 of *Lecture Notes in Computer Science*, pages 506–514. Springer, 2007.

[40] Claus Pahl Juan Pablo Garcia-Gonzalez, Veronica Gacitua-Decar. Service Registry: A Key Piece for Enhancing Reuse in SOA, 2009. http://msdn.microsoft.com/en-us/architecture/aa699419.aspx.

[41] Nate Kohari. Painless REST via ASP.NET MVC, 2010. http://github.com/enkari/siesta.

[42] Manu Konchady. *Text Mining Application Programming*. Thompson, Toronto, Ontario, Canada, 2006.

[43] Aparna Kondury and Chan Chien-Chung. Clustering of Web services based on Wordnet Semantic Similarity, 2010. http://dspace.lib.fcu.edu.tw/bitstream/2377/10830/1/CE07NCS002007000126.pdf.

[44] Phillip A Laplante. *Real-time systems design and analysis: an engineer's handbook; 2nd ed.* IEEE Computer Society Press, Piscataway, NJ, 1997.

[45] Tim Berners Lee. WorldWideWeb - Summary, 2010. http://www.w3.org/History/19921103-hypertext/hypertext/WWW/Summary.html.

[46] Fritz Lehmann and Rudolf Wille. A Triadic Approach to Formal Concept Analysis. In *ICCS*, volume 954 of *Lecture Notes in Computer Science*, pages 32–43, 1995.

[47] Open Source MIT License. OpenRasta Resource- Oriented Framework for .NET, 2010. http://trac.caffeine-it.com/openrasta.

[48] Benjamin Markines, Ciro Cattuto, Filippo Menczer, Dominik Benz, Andreas Hotho, and Gerd Stumme. Evaluating Similarity Measures for Emergent Semantics of Social Tagging. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 641–650, 2009.

[49] Cameron Marlow, Mor Naaman, Danah Boyd, and Marc Davis. HT06, Tagging Paper, Taxonomy, Flickr, Academic Article, To Read. In *HYPERTEXT '06: Proceedings of the Seventeenth Conference on Hypertext and Hypermedia*, pages 31–40, New York, NY, USA, 2006.

[50] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[51] Sun MicroSystems. Core J2EE Patterns- Data Access Object, 2010. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html.

[52] Sun Microsystems. Java SE Downloads, 2010. http://java.sun.com/javase/downloads/index.jsp.

[53] Peter Mika. Ontologies are us: A Unified Model of Social Networks and Semantics. *Web Semantics*, 5:5–15, 2007.

[54] OASIS. Oasis: WSS- SOAP Message Security. http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf.

[55] OASIS. OASIS: Advancing Open Standards for the Global Information Society, 2010. http://www.oasis-open.org/.

[56] OASIS. Oasis: WS- Discovery Specification, 2010. http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.pdf.

[57] UN/CEFACT OASIS. ebXML Technical Architecture Specification v1.04, 2010. http://www.ebxml.org/specs/ebTA.pdf.

[58] Joshua O'Madadhain. JUNG Java Universal Network/Graph Framework, 2010. JUNG http://jung.sourceforge.net/.

[59] Tim O'Reilly. What is Web 2.0, 2005. http://oreilly.com/lpt/a/6228.

[60] Sujit Pal. Salmon Run: IR Math with Java: Experiments in Clustering, 2010. http://sujitpal.blogspot.com/2008/10/ir-math-in-java-experiments-in.html.

[61] Sujit Pal. Salmon Run: IR Math with Java: Similarity Measures, 2010. http://sujitpal.blogspot.com/2008/09/ir-math-with-java-similarity-measures.html.

[62] Sujit Pal. Salmon Run: IR Math with Java: TF, IDF and LSI, 2010. http://sujitpal.blogspot.com/2008/09/ir-math-with-java-tf-idf-and-lsi.html.

[63] Michael P. Papazoglou. *Web Services: Principles and Technology*. Pearson, Prentice Hall, Upper Saddle River, NJ, USA, 2008.

[64] Sunilkumar Peenikal. Mashups and the Enterpise, 2009. http://www.mphasis.com/pdfs/Mashups_and_the_Enterprise.pdf.

[65] Daniel Ramage, Paul Heymann, Christopher D. Manning, and Hector Garcia-Molina. Clustering The Tagged Web. In *WSDM '09: Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pages 54–63, New York, NY, USA, 2009.

[66] G. Salton and C. Buckley. Term Weighting Approaches in Automatic Text Retrieval. Technical report, Cornell University, Ithaca, NY, USA, 1987.

[67] G. Salton and M. J. McGill. *Modern Information Retrieval*. McGraw-Hill, New York, NY, USA, 1983.

[68] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18:613–620, 1975.

[69] Daniel Schall. Human-Provided Services in Mixed Systems, 2010. http://www.infosys.tuwien.ac.at/prototyp/HPS/HPS_index.html.

[70] Stan Schroeder. The Web in Numbers: The Rise of Social Media, 2009. http://mashable.com/2009/04/17/web-in-numbers-social-media/.

[71] Andriy Shepitsen, Jonathan Gemmell, Bamshad Mobasher, and Robin Burke. Personalized Recommendation in Social Tagging Systems using Hierarchical Clustering. In *RecSys '08: Proceedings of the 2008 ACM Conference on Recommender systems*, pages 259–266, 2008.

[72] Clay Shirky. Ontology is Overrated: Categories, Links, and Tags, 2005. http://www.shirky.com/writings/ontology_overrated.html.

[73] James Sinclair and Michael Cardew-Hall. The Folksonomy Tag Cloud: When is it Useful? *J. Inf. Sci.*, 34(1):15–29, 2008.

[74] Florian Skopik, Daniel Schall, and Schahram Dustdar. Start Trusting Strangers? Bootstrapping and Prediction of Trust. In *WISE*, volume 5802 of *Lecture Notes in Computer Science*, pages 275–289, 2009.

[75] Michael Steinbach, George Karypis, and Vipin Kumar. A Comparison of Document Clustering Techniques. In *KDD-2000 Workshop on Text Mining, August 20*, pages 109–111, Boston, MA, USA, 2000.

[76] W. Stevens, G. Myers, and L. Constantine. Structured Design. *Classics in Software Engineering*, pages 205–232, 1979.

[77] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[78] Martin Szomszor, Ciro Cattuto, Harith Alani, Kieron O'Hara, Andrea Baldassarri, Vittorio Loreto, and Vito D.P. Servedio. Folksonomies, the Semantic Web, and Movie Recommendation. In *4th European Semantic Web Conference, Bridging the Gap between Semantic Web and Web 2.0*, pages 71–84, 2007.

[79] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[80] VanderWal Thomas. Folksonomy Coinage and Definition, 2004. http://vanderwal.net/folksonomy.html.

[81] Martin Treiber, Hong Linh Truong, and Schahram Dustdar. SOAF - Design and Implementation of a Service-Enriched Social Network. In *ICWE*, volume 5648 of *Lecture Notes in Computer Science*, pages 379–393. Springer, 2009.

[82] UDDI. UDDI Version 2.04 API Specification, 2010. http://www.uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm.

[83] Jakob Voss. Tagging, Folksonomy & Co - Renaissance of Manual Indexing? *Open Innovation Neue Perspektiven im Kontext von Information und Wissen Beiträge des 10. Internationalen Symposiums für Informationswissenschaft und der 13. Jahrestagung der IuK-Initiative Wissens*, pages 243–254, 2007.

[84] W3C. Simple Object Access Protocol (SOAP) 1.1 Specification, 2010. http://www.w3.org/TR/soap/.

[85] Haibin Zhu. Challenges to Reusable Services. In *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*, pages 243–244, 2005.