TAMPERE UNIVERSITY OF TECHNOLOGY

Heikki Peltola

UTILIZING EXTERNAL SERVICES AND SHARING CONTENT IN A CONTENT MANAGEMENT SYSTEM

Master of Science Thesis

Examiners: Prof. Tommi Mikkonen,
M.Sc. Niko Mäkitalo
Examiners and topic
approved by Department
Council 08.02.2012

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
PELTOLA, HEIKKI : UTILIZING EXTERNAL SERVICES AND SHARING CONTENT IN A CONTENT MANAGEMENT SYSTEM
Master of Science Thesis, 61 pages
March 2012
Major: Software Engineering
Examiners: Prof. Tommi Mikkonen, M.Sc. Niko Mäkitalo
Keywords: content management, distributed system, external service, context

People are using more and more online services. Many of the services are used for storing content. Also the number of users' personal devices – mobile phones, laptops, cameras, and so forth – is increasing rapidly. Consequently, the amount of content stored in the services and devices people are using is constantly growing. Managing all of the content located in different places is therefore becoming an increasing problem. It is not easy to search for a certain piece of content, if one cannot remember in what service or device the content is stored at. Content can also easily become forgotten and lost in a rarely used device.

This thesis presents new and improved features to an existing content management system named VisualREST, which is designed for managing content from all of the user's devices. We are presenting new ways of importing content from user's devices, as well as importing content from external services, in order for the content management system to be as comprehensive as possible. Searching content is an important and often used feature of a content management system. Therefore, a new, user-friendlier interface for searching content is presented. Sharing content with other users should be quick and easy, and for that end a notion of context is presented. It is designed to help share content between a predefined group of people. All of these features have been implemented and tested in VisualREST.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
PELTOLA, HEIKKI: ULKOISTEN PALVELUIDEN HYÖDYNTÄMINEN JA SISÄLLÖN JAKAMINEN SISÄLLÖNHALLINTAJÄRJESTELMÄSSÄ
Diplomityö, 61 sivua
Maaliskuu 2012
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Tommi Mikkonen, DI. Niko Mäkitalo
Avainsanat: sisällönhallinta, hajautettu järjestelmä, ulkoinen palvelu, konteksti

Internet-palveluiden määrä kasvaa jatkuvasti tarjoten käyttäjille yhä enemmän valinnan varaa. Tärkeä ja suosittu palveluiden käyttötarkoitus on sisällön säilöminen. Myös henkilökohtaisten laitteiden – matkapuhelimien, kannettattavien tietokoneiden, kameroiden, ja niin edelleen – määrä on kasvanut nopeasti. Näin ollen myös palveluissa ja laitteissa olevan sisällön määrä jatkaa kasvuaan. Sisällön hallinta eri palveluiden ja laitteiden välillä on siksi kasvava ongelma. Halutun sisällön hakeminen ei ole helppoa, jos ei muista missä palvelussa tai laitteessa kyseistä sisältöä säilytetään. Sisältö voi myös helposti unohtua ja hävitä harvoin käytetyllä laitteella.

Tässä diplomityössä esitetään uusia ja paranneltuja ominaisuuksia olemassaolevaan sisällönhallintajärjestelmään nimeltään VisualREST, joka on suunniteltu hallitsemaan sisältöä kaikilta käyttäjien laitteilta. Työssä esitämme uusia tapoja ladata sisältöä toisista palveluista VisualRESTiin, jotta sisällönhallintajärjestelmässä olisi mahdollisimman kattavasti saatavilla kaikki käyttäjien tuottama ja omistama sisältö. Sisällön etsiminen on tärkeä ja usein käytetty sisällönhallintajärjestelmän ominaisuus, ja siksi esitämme uuden, käyttäjäystävällisemmän rajapinnan sisällön etsimiseen. Sisällön jakaminen muiden käyttäjien kanssa tulisi olla helppoa ja nopeaa, ja siksi esittelemme käsitteen context. Se on suunniteltu helpottamaan sisällön jakamista ennalta määrätyn joukon kesken tietyssä käyttökontekstissa. Kaikki edellä mainitut ominaisuudet ovat toteutettu ja testattu VisualRESTissä.

FOREWORD

I got onboard the VisualREST project in the year 2010, a year after it was started. Working for the project and studying at the same time has supported each other, in spite of occasional long working hours.

I would like to thank all of the people that have been part of the project from Aalto University, Nokia Research Center and Tampere University of Technology. Especially, I would like to thank Tommi Mikkonen for his encouraging and constructive guidance, and Niko Mäkitalo for recommending me to work in the project and his help and feedback for this thesis. I would also like to thank my parents Hannele and Harri for their support. Finally, I would like to thank my dear companion Marja for her love and encouragement.

Tampere, February 9, 2012

Heikki Peltola
heikki.peltola@tut.fi
+358 50 302 7856

TABLE OF CONTENTS

1. INTRODUCTION

The Internet is becoming more and more accessible to people all around the world. According to Internet World Statistics, in March 2011, 58.2% of population in Europe is using the Internet [1]. The amount of internet users has more than tripled since the year 2000. People connected to the Internet are using services that are offering content storing capabilities for pictures, documents and other types of content. People also have more and more personal devices in their use. There are desktop computers, laptops, cameras, mobile phones, smart phones, et cetera. The amount of content in those devices is constantly growing. New and better ways of managing the growing amount of content situated in all of the different services and devices are needed. All of the content must be made available from a single entry point.

In this thesis we present ways of extending an existing content management system. We present how content can be imported from users' devices and services that store users' content, into an existing content management system named VisualREST [2]. The search features of VisualREST are improved by offering a new content search interface. We also present a way of grouping and sharing content with other users. Our research problem in this thesis is *how to import all users' content to a content management system?* In order to import users' content as comprehensively as possible, users must be offered ways of importing content from different sources. Additionally, this thesis focuses on, *how to improve the search features of the existing content management system* and *how to facilitate sharing content with other people?*

The rest of this thesis is organized as follows. Chapter 2. gives background information about content management systems and the technologies used in this thesis. Chapter 3. describes content management system VisualREST and presents an improved query interface. Adding and importing content to VisualREST is presented in Chapter 4.. Chapter 5. describes how users can share their content using contexts. In Chapter 6., we evaluate the features presented in this paper. Chapter 7. finally concludes the paper by summarizing the main contributions to VisualREST.

2. BACKGROUND

This chapter gives background information on the key concepts and the most crucial technologies used in VisualREST. We start by describing what content management is and presenting basic principles content management systems should follow. We will also present technical background on the underlying technologies.

2.1 Content management system

The amount of content that people have in their numerous devices has increased rapidly in the recent years. Handling the growing amount of content would be an unbearable task without the help of a system designed to manage the content. Content management systems have been developed for answering this issue.

Content management system is a centralized service designed for helping users manage their content. Typically content management system is responsible for storing content, administrating access rights and offering a way for users to search, browse and access content. According to Boiko, the key properties of a content management system can be identified to be accessibility, reliability and security [3, p. 980-991]. Accessibility means that the users of the service can search, browse and access the content in the content management system. The service also needs to be reliable, so that it will stay online under heavy usage and possible hardware failures should not lead to the loss of any data. Security is a further important aspect, as users should have total control over who are allowed to access their content. Also the communication of sensitive data should be encrypted.

Different roles in a content management system are service provider, user, and content owner. Service provider is responsible for offering the service of managing content. The service offers an interface for users and it is responsible for ensuring that the key properties of a content management system are fulfilled. User is anyone using the service. Users must be offered proper interfaces for all of the required actions to the system. Content owner is also a user of the system but with certain privileges, content owners can add new content and define who are allowed to access their content.

2.1.1 Distributed system

Distributed system is a set of autonomous computers working interconnected to each other. Coulouris et al. define distributed system to be a system in which hardware or software components located on networked computers communicate and coordinate their actions only by passing messages [4, p. 2]. In a distributed system it is customary to hide unnecessary information from the users of the system. The services that are offered to users may need to utilize different computers, but to a user it might seem like only one computer is doing a given task.

The construction of a distributed system has many special challenges that must be addressed. Distributed systems need to consist of *independent computers*, and failure in one computer should be confined so that it will not affect any other computers. The system should be *scalable*, as it must be able to handle growing amounts of work by adding more computers to the system. The system needs to be *secure*, and communication needs to be encrypted accordingly. Another dimension of security is authentication and authorization, users need to be authenticated and authorized to resources with restricted access. The system must be *transparent*, so that application developers does not need to be concerned about the whole system, only the design of their particular applications. These are some of the important issues that must be taken into account when constructing and developing a distributed system. [4, p. 25]

Distributed content management is a way of managing content in more than one location. In a worldwide distributed content management system it is wise to keep in mind where the content should be stored at. If the content is only requested by users in Europe, it would most likely be best to store the content in European servers, because the data would not have to travel such a long way to the users. This improves response time from the servers and improves the quality of service.

2.1.2 Content, metadata and essence

As the name implies, content management systems are designed for managing content – photos, videos, textual and other kind of data presented and stored as files. According to [5] content consists of essence and metadata.

Essence is the raw programme material itself. Essence is the data that represents the photo, video, text, and other kind of data that is represented to users by applications. Metadata is the other part of the content: it is descriptive information about the essence. Metadata describes the essence of the content. Mauthe and Thomas propose that metadata can be classified into: *content-related metadata*; describing the actual content, *material-related metadata*; describing how the content can be used, and *location-related metadata*; describing location and how the content can be accessed [6, p. 4].

Metadata can be generated either using automated processes or by adding it manually. The automatically added metadata can be related to the time and place when the content was created. For example a photo can have metadata describing time the photo was taken or latitude and longitude of the physical location the photo was taken at. Automated metadata can also be created for example by processing the content with algorithms that are designed for recognizing people from photos [7]. When metadata cannot be added with automated processes, it must be created and added manually. Manually added metadata can be arbitrary metadata, it can be a comment or opinion about the content or it can be for example a tag describing the content.

A system that manages both essence and metadata of content is called a content management system. In a content management system it is important to have precise and comprehensive metadata about content. It is required for describing, searching and retrieving a content. [6, p. 5]

2.2 Technical background

This section describes the key technologies used in this thesis. We start with REST and Message passing architectural styles, followed by a general description of Ruby on Rails framework and OAuth authorization protocol.

2.2.1 REST

Representational state transfer (REST), as Fielding describes in his thesis [8], is an architectural style for building network-based software. Most often REST is utilized in web applications and implemented on HTTP protocol [9]. The key properties of REST are client-server interaction, statelessness, and uniform interface.

Client-server architectural style is used for communication between the client and the server. In client-server model the client initiates actions by making a request to the server and server responds to the requests. Usually, the server is a non-terminating process and often the server provides service to more than one client. Scalability of the service is achieved by running multiple server programs sharing their resources. Client programs do not share their resources with each other.

Statelessnes means that the server does not maintain the state of the client. The client needs to send all necessary information with every request to the server, including possible authentication parameters. Since the server does not store any client state information, subsequent requests sent by a client do not have to be processed by the same server program.

RESTful services are based on resources that are utilized through a uniform interface. Each resource is addressable by a unique URI. The resources are accessed and manipulated using these URIs. For example HTTP PUT request to a URI would create the resource, while HTTP GET request to the same URI would get a representation of the resource as a response.

If a service violates any of these constraints, it is not considered to be a RESTful service. According to Richardson and Ruby, many web services that are claiming to be RESTful are actually not [10, p. 17-18]. Often these services have elements of RESTful services, but in addition has features of RPC architectures [11]. These services have methods that do not follow the uniform interface constraint of REST.

2.2.2 Message passing

Message passing is communication where messages are sent from a sender to one or more recipients. Message passing can be implemented in different protocols, in this thesis we focus on *Extansible Messaging and Presence Protocol*, also known as *XMPP* [12]. XMPP is a near real-time messaging protocol, used in a number of services with millions of users. Facebook chat, Windows Live Messenger, and Skype are only a few examples of the services that utilize XMPP.

XMPP is based on delivering XML streams and stanzas. The streams are containers for exchanging the stanzas between any two entities over a network. Stream can be seen as a root XML element and the stanzas as first-level child elements. XMPP follows client-server model, in which a client connects and opens a stream to the server, which then opens a stream back to the client. After the initialization, both the client and the server are able to send stanzas over the stream. There are three types of stanzas: message, presence, and info/query. [13]

The architecture of XMPP is decentralized and similar to email; there is no central master server and anyone can run their own XMPP server. Connections in XMPP are encrypted with Transport Layer Security (TLS) [14]. XMPP is an open standard and there are lots of extensions available for it. One of the extensions is the Publish-Subscribe extension [15], which allows the creation of nodes. Users can register as listeners to these nodes and when information is published in those nodes, it is broadcasted to all the listeners.

2.2.3 Ruby on Rails

Ruby on Rails [16] is an open source web application framework for the Ruby programming language [17]. It was initially released in 2004 and reached version 3.0 in the year 2010. Ruby on Rails uses the Model-View-Controller architecture pattern (MVC), in which *model*, *view* and *controller* are isolated as their own components. *The model* manages data storing, maintenance and handling. *The view* renders the model into a user interface. *The controller* handles user input and instructs the model and view to perform actions based on the user input.

The design of Ruby on Rails is driven by two key concepts: DRY and convention over programming. DRY stands for *don't repeat yourself* – it aims at reducing repetition, every piece of knowledge should be expressed in just one place. This reduces the amount of code needed and makes it easier to implement changes into an existing program. The *convention over configuration* means that Ruby on Rails has reasonable default settings.

By following the conventions developers can write applications using less code than a typical Java web application uses in XML configuration. It is also made easy to override these conventions. [18, p. 2]

Ruby on Rails is typically deployed with a database server such as MySQL or PostgreSQL, and a web server such as Apache running the Phusion Passenger module. Nevertheless, Ruby on Rails does not tie the developer to certain modules. There are a wide range of modules that can be choosen depending on the developer's preferences.

There is a lot of literature about Ruby on Rails, for example [18], [19] and [20] to name a few. Ruby on Rails has a comprehensive API [21] available online and there are lots of message boards and blogs dedicated to Ruby on Rails. Ruby on Rails is well documented, and there is a wide range of people all around the world developing it.

2.2.4 OAuth

OAuth [22] is an open standard for authorization. With OAuth it is possible to share private resources stored on one service to another service without the need for giving username and password to a third party. OAuth allows users to hand out tokens. Token allows a service to access resources in another service with certain permissions. For example a token can authorize a service to read all of certain user's resources in another service, but not to modify any of the resources.

Figure 2.1 presents the basic idea of OAuth, without being an accurate representation of the protocol. The background story is that Alice wants to access her resources on Service B using Service A, without giving username/password to service A. The first step is for Alice to ask Service A to get an access token to service B. Alice is directed to Service B. Service B asks Alice to authenticate and allow Service A to access the resources on Service B. Alice authenticates to Service B and allows Service A to access the resources on Service B. Service B gives Service A an access token that can be used for accessing the resources on Service B. Service A confirms Alice that the access token has been received. From that point on Alice can use Service A to access her resources on Service B.

At the time when OAuth was designed many services used protocols that were very similar to OAuth, such as Flickr API [23], Amazon Web Services API [24], and Google AuthSub [25]. Each of these protocols provided a proprietary method for exchanging user credentials for an access token. OAuth was created by studying each of these protocols, extracting the best practices and commonality. OAuth was designed so that the existing protocols could easily switch over to OAuth as well as new implementations to start immediately using OAuth. After the OAuth 1.0 Protocol was published in 2007, many service providers have modified their authentication protocols to work according to the OAuth protocol.

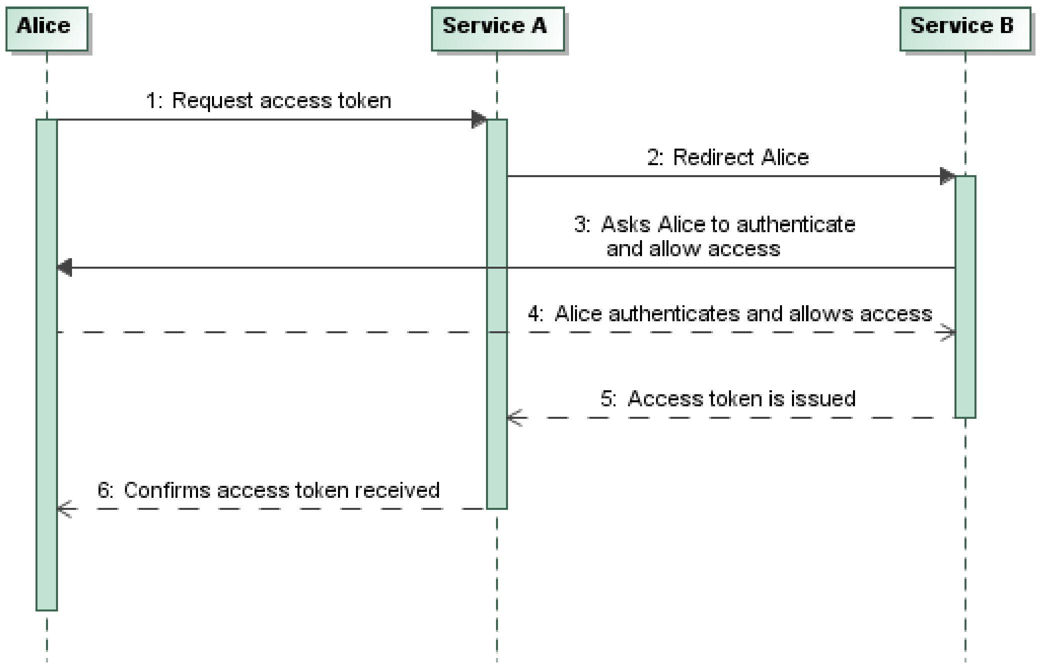

Alice
Service A
Service B
1: Request access token
2: Redirect Alice
3: Asks Alice to authenticate and allow access
4: Alice authenticates and allows access
5: Access token is issued
6: Confirms access token received

Figure 2.1: OAuth access token message flow.

3. CONTENT MANAGEMENT SYSTEM – VISUALREST

This chapter describes a content management system called VisualREST, more closely described in [2, 26]. From the VisualREST system, we give an overview, describe what kinds of containers there are, and how they work. Then we discuss how content can be searched. We also describe how XMPP is used for sending notifications to clients and how XMPP nodes can be used.

All of the features described in the subsequent chapters have been implemented in the VisualREST content management system. The features have been tested and used by a small group of researchers and developers.

3.1 Overview

VisualREST is a distributed content management system, implemented using Ruby on Rails. The idea of the system is to keep track on content from all of the users' devices. Figure 3.1 provides an overview of the system, showing three users, Alice, Bob, and Charlie. Each user has different kinds of devices, all of them connected to the VisualREST system. Each device uploads the metadata of their content to the VisualREST system. The actual essence of the content is kept and stored on the devices. User's device is represented in VisualREST as a container.

VisualREST can be used with web browsers or client programs designed to utilize the system. For web browsers, VisualREST offers a web interface. A client program can be designed for importing content to the system, accessing content, or both. Communication between the server and the client is based on HTTP protocol. However, client programs can also use an additional XMPP interface. The additional interface is used for requests from server to the clients and for notifying clients about content they are interested in.

The resource hierarchy of VisualREST is presented in Figure 3.2. The hierarchy is as

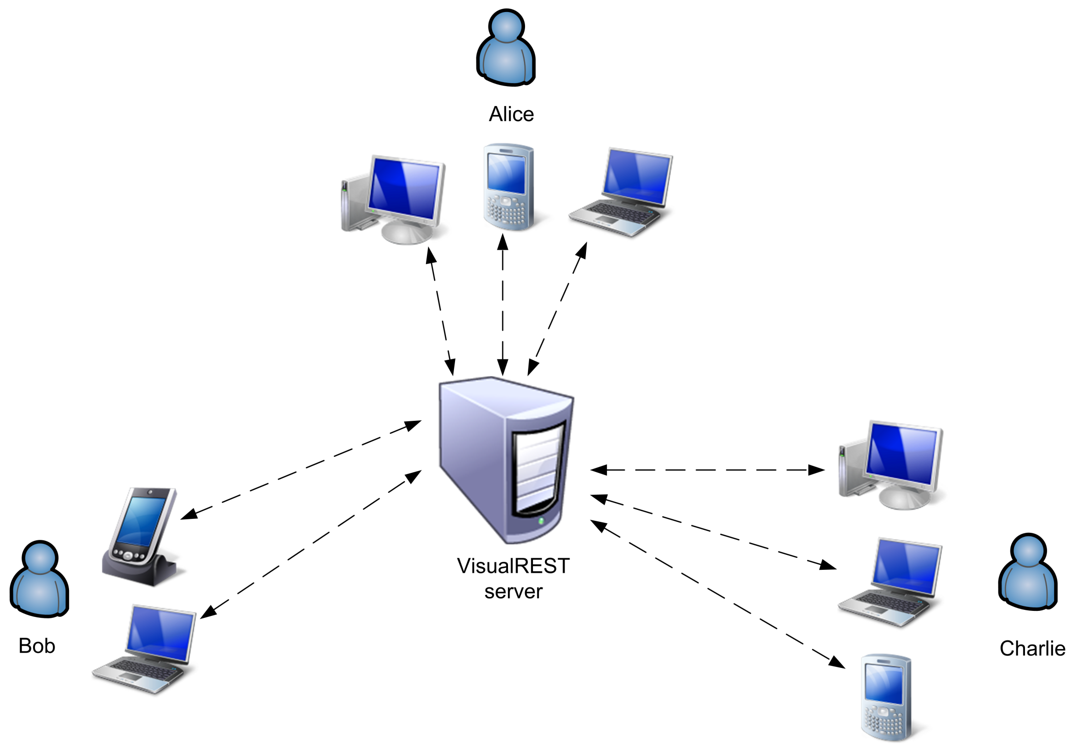


Figure 3.1: VisualREST overview.

follows: The VisualREST system has users. Users have containers that hold the content. Each content has an essence. The essence is described by metadata.

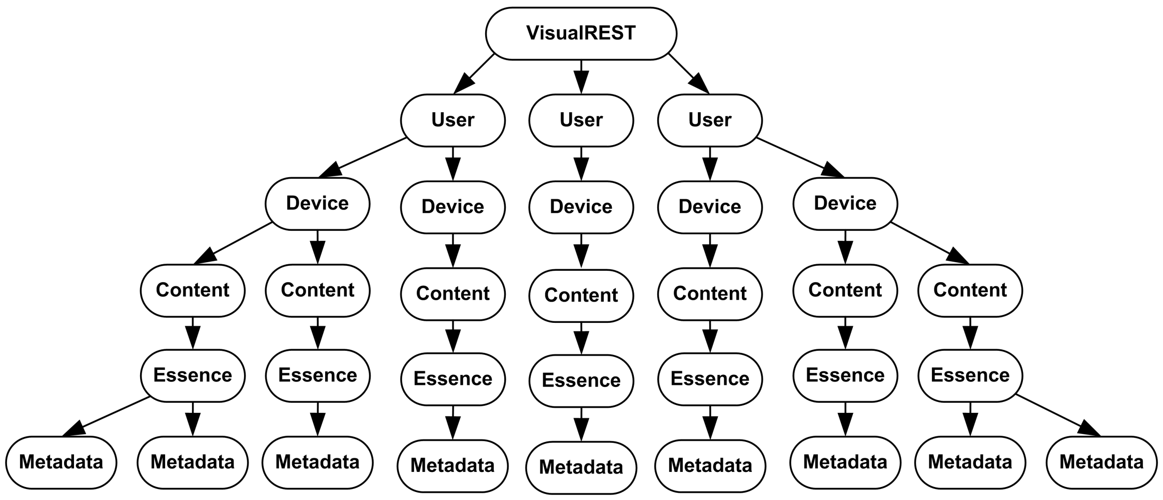


Figure 3.2: Resource hierarchy in VisualREST.

The main components of the VisualREST database model are presented in the class diagram given in Figure 3.3 using the Unified Modelling Language [27]. The parts of the database that are irrelevant within the context of this thesis are not shown, because they would only complicate the more important parts and would not bring any extra value.

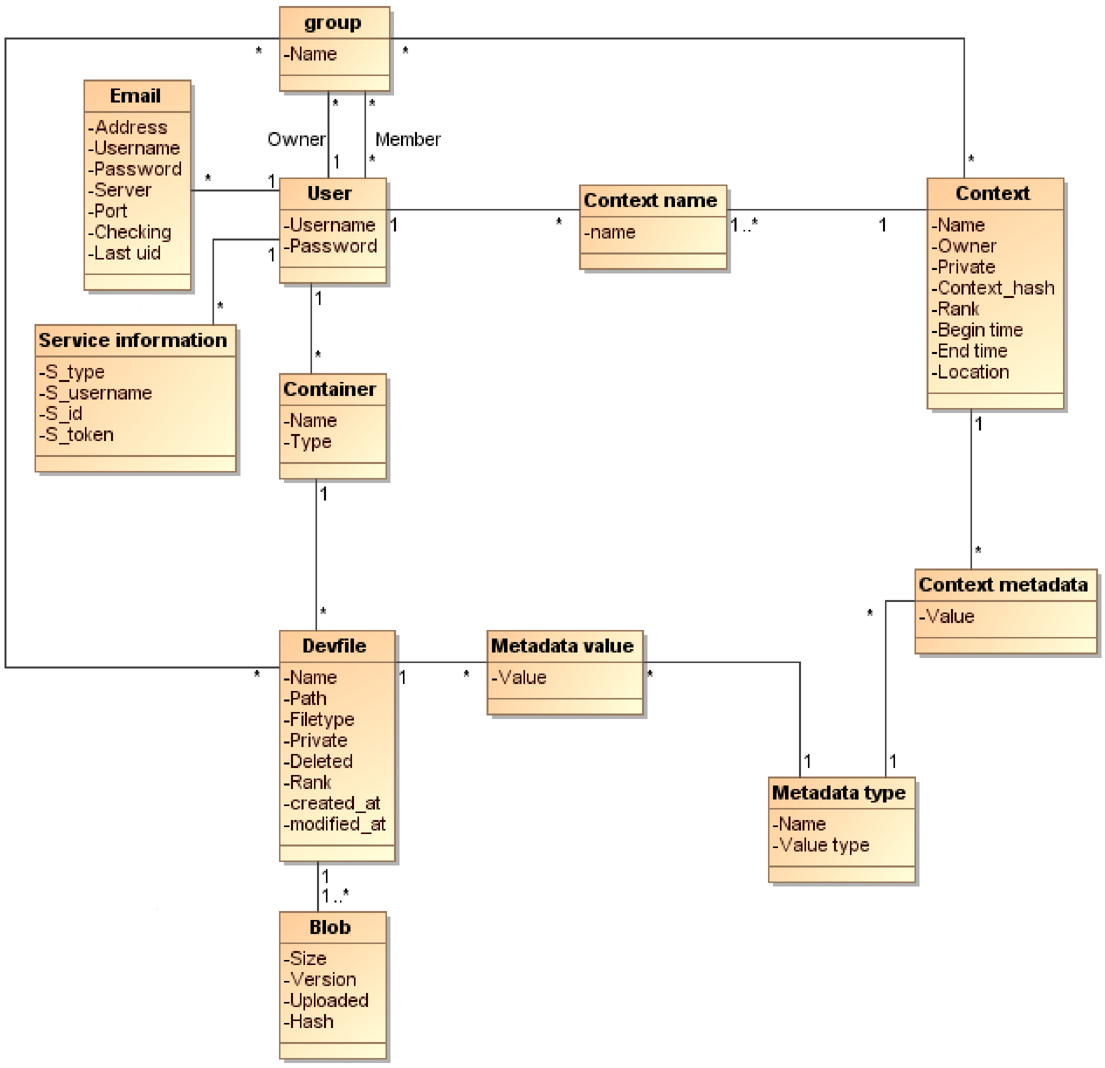


Figure 3.3: Simplified database model of VisualREST.

A more thorough description of the database can be found at [2]. The following will summarize the classes in the database and describe their function.

User: The most central part of the database. All the other classes are related and dependant in the User class. This class has attributes *Username* and *Password*.

Email: Users can add their email information to the system. The information can have only email *Address* or all authenticating parameters needed for accessing an email account, such as *Username*, *Password*, *Server* address and *Port* number. The *Checking* parameter is a boolean, defining whether to actively scan the mail box and *Last_uid* the ID of last fetched email.

Service information: When importing content from other systems to VisualREST, this class can be used for storing the authentication parameters for OAuth type of authentication. *S_type* tells the type of service, such as Flickr [28] or Facebook [29]. *S_username* is the username in that service. *S_id* and *S_token* are authentication parameters stored in VisualREST.

Container: *Name* of the container and *Type* of the container, such as "x86_64-linux", "Nokia-N900" or "virtual_container".

Devfile: Initial metadata that every content has — *Name*, *Path* on the container and *Filetype*. *Private* defines if the file is available for everybody or only a certain limited group of users. Content cannot be deleted, but it can be marked as *Deleted*. Every Content has a *Rank* value, based on the contents popularity. Content also has timestamps for *Created_at* and *Modified_at*, which is automatically updated when content is modified.

Blob: Blob is a representation of a version of the content. Blob has version related metadata: *size* and *version* number. *Uploaded* tells, whether the essence is already uploaded on the server. *Hash* is a SHA1-hash [30] of the content's essence.

Metadata type: Types of metadata added by users to the system. *Name* and *Value type* are needed.

Metadata: User added arbitrary metadata for content. Metadata is always of some metadata type and is related to a content.

Context: Context is a tool for grouping and sharing content. Context has *Name*, *Owner*, *Private* value, individual *Context_hash* and *Rank* value. Optional values are *Begin_time*, *End_time* and *location*.

Context name: Every member of a context has a possibility to give a personal *name* for the context.

Context metadata: Context can have metadata defining the content that are part of this context.

Group: Groups are for handling access rights. Every group has an *Owner* and a *Name*. Group can be related to Devfile or Context. Users in a group are allowed to access Devfiles and Contexts the group is related to.

3.2 Authentication

Authenticating to the VisualREST is done with two methods. In the web user interface, the authentication is based on sessions. Ruby on Rails offers a way to store session

IDs in user's web browser for uniquely identifying a client browser. In the web user interface user gives his/her username and password. The server verifies the authentication parameters and stores logged in user information. The information is linked with the web browser's session ID. The authentication parameters must be sent only once at the beginning of each session and the authentication will be valid for the duration of the whole session.

For the RESTful interface, VisualREST offers a different way to authenticate. Client programs using the REST interface must provide authentication parameters with every HTTP request. Table 3.1 presents the required authentication parameters. The *auth_timestamp* parameter implies how long the *auth_hash* is valid and it will also ensure that every request to VisualREST will get a different *auth_hash* parameter. This prevents a malicious user from capturing authentication parameters and using them for fabricating requests to the server.

Table 3.1: Authentication parameters.

parameter	**description**
auth_username	Username
auth_timestamp	Unix time. Seconds since January 1st, 1970
auth_hash	SHA1-hash calculated from: *auth_timestamp* + user's *password* + *path* part of the URI
i_am_client	Value *"true"*, defines the request to be from a client program

3.3 Container

VisualREST's hierarchy, as presented in Figure 3.2, shows how containers are connected in the system. A container is owned by a user and a container has content. The purpose of a container is to group and make it easier to handle content. One content can belong to one container.

Users have content in their devices that needs to be imported to the VisualREST system. Users' devices have a client program for that purpose. Container is representing content in user's device that the user has imported into the VisualREST. Client programs can be implemented for almost any kind of device with an internet connection.

VisualREST's design philosophy is that content or any version of the content is always preserved. According to this there is no possibility to delete content. However, a user might wish to forget about some content, therefore it is possible for the user to mark content as deleted. When content is marked as deleted, it is not shown in search results, unless explicitly specified.

As an addition, we also made it possible to remove containers from the system. This feature is needed since devices with container programs might get lost, broken or otherwise forgotten. When a container is removed from the system, also all of the containers content is removed. Including all metadata and possibly uploaded essence.

Users' devices are not the only places that users nowadays have their content. People are using lots of services that have content uploaded. This content can usually be exported from one system and imported into another. To make it possible for users to import content to VisualREST without the need for an actual device with a running client program, we added a new kind of container called a **virtual container**. A virtual container is designed to look like any other type of container to users, and the only main difference is that the essence of the content is stored on the server instead of the user's device. Because the essence is stored on the VisualREST server, it is always available to the users.

3.4 Metadata

All content added to VisualREST must have certain initial metadata. The required initial metadatas are: name, filedate, size, filetype, and path. In addition to initial metadata, users can also add arbitrary metadata to the content. In order for user to add arbitrary metadata, the metadata type must be added to VisualREST. All of the metadata types that users have already created can be listed with HTTP GET request to the URL:

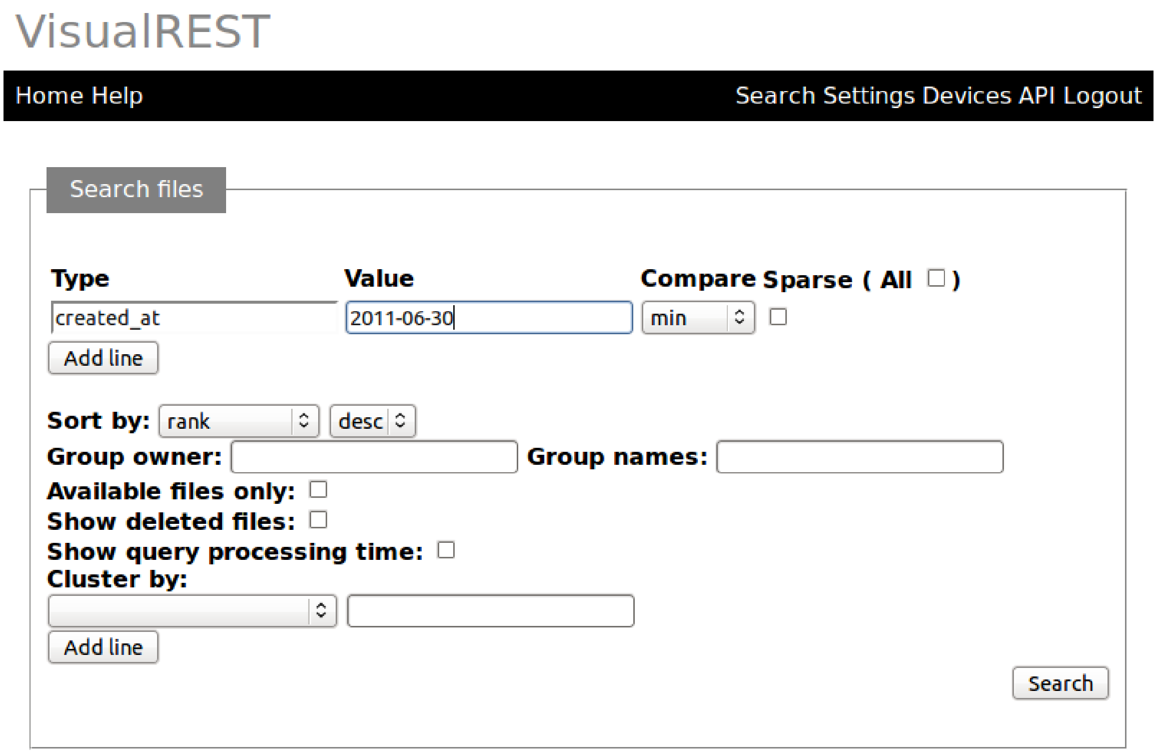
```
http://visualrest.cs.tut.fi/metadatatypes
```

New metadata type can be created using HTTP PUT request. Parameter `value_type` is needed to state the value type, possible value types are: `string`, `number`, `date` and `datetime`. The URL for creating new metadata type is:

```
http://visualrest.cs.tut.fi/metadatatype/{new_metadata_type}
```

3.5 Searching content

Content can be browsed and searched from the VisualREST system. Searching is done with queries to the system. The queries can be done by client programs or by using the web interface. Figure 3.4 provides a screen shot of the search query form, which is available in the web interface. In this example query, we are searching for content that is created_at the date 2011-06-30 or later. The query form can be used for experimenting how the query parameters are formed.



Type	Value	Compare	Sparse (All ☐)
created_at	2011-06-30	min	☐

Sort by: rank desc
Group owner:
Group names:
Available files only: ☐
Show deleted files: ☐
Show query processing time: ☐
Cluster by:

Figure 3.4: File search query form.

The search is commonly done for all content in the system. However, if needed the search can be narrowed down to include only certain groups, users, or containers. Search results have only content that the user is authorized to see. Therefore, search results may vary for different users.

When a user has found an interesting content, the essence of the content can be requested from the server. Figure 3.5 illustrates the messages sent to and from the server when essence is requested.

1. Alice makes a search query for content on the server.

2. Server returns search results to Alice.

3. Alice finds an interesting content and requests the essence from the server.

4. The essence of the requested content is located at Bob's device. The server requests the essence from Bob's device with XMPP message. XMPP message is used to get fast reaction from Bob, with HTTP this would be inefficient.

5. Bob's device pushes the requested essence to the server.

6. The server returns the essence to Alice, who requested it in the beginning.

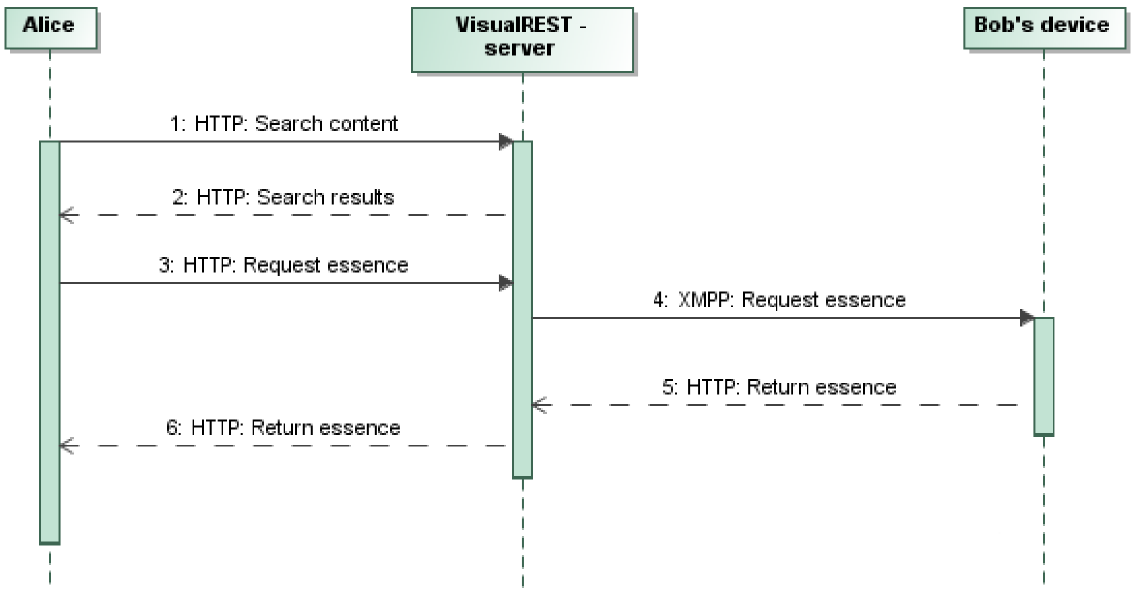


Figure 3.5: Requesting content from VisualREST.

As we can see in the above example, Alice does not directly send any messages to Bob's device. All of the interaction is done through the VisualREST server. This means that Alice does not need to know where the actual essence of the content is really located at in order to request it.

3.5.1 Search parameters

Search parameters are the basic building blocks of a search query. Search parameter consists of a key-value pair. Key tells what type of metadata we are searching for in a content, and value describes the value of the metadata in a content. Search keys available in the systems are listed in Table 3.2. Additionally, all user added metadata types can be used as search parameters.

Table 3.2: Possible search keys.

GET /files
created_at (Optional) Time when content was created.
modified_at (Optional) Time when content was last modified.
user (Optional) Username, multiple users separated by '+'.
device (Optional) Container or device name, multiple values separated by '+'.
filename (Optional) Filename or part of it.
type (Optional) Filetype, for example 'video' or 'image'.
size (Optional) Size of content's essence in bytes.
path (Optional) Path of content or part of it.
rank (Optional) Rank value of content.
tag (Optional) User added tag.
context_hash (Optional) Context is added to search query with context_hash value.

In addition to metadata types, also values have types. Available value types in search parameters are listed in Table 3.3. When searching for date type, value can be given as `year`, `year-month` or `year-month-day`. With datetime value can also be of form `year-month-day hour:minute:second`. If only year is given, search is for values within that year. If year and month given, search is for values within that year and month.

3.5.2 Query interface

Content is searched with queries and a query is sent to the server using HTTP GET method. Queries can be made as an authenticated user or as a non-authenticated user. If the query is made as an authenticated user, the user must either log in with the web

Table 3.3: Available value types.

Type	description	example
string	letters or numbers	tampere
number	for file size integer, for user added metadatas float	1532 / 24.42
date	yyyy-mm-dd / yyyy-mm / yyyy	2011-12-24 / 2011-12 / 2011
datetime	yyyy-mm-dd nn:nn:nn	2011-12-24 18:00:00

user interface, so that the browser keeps the session details, or with client programs by giving authentication parameters with every HTTP request. If user is querying as a non-authenticated user, search results will only present public content that is available to everyone.

Queries can be made with client programs or with the earlier presented query form or even by writing the query URL to a web browser's navigation bar by hand. The query interface was designed to be as readable as possible for humans, and also keeping it easy for programs to parse. In Ruby on Rails parameters with the same format can be easily parsed, by default they can be accessed as a list.

VisualREST query interface has eight different parameter formats, presented in Table 3.4. Next we will go through all of the different formats and represent example queries for content.

Table 3.4: Search parameter format.

Parameter	description
q[key]=value	Common usage. Used when searching for matching value.
qmin[key]=value	Comparison operator, value bigger than or equals.
qmax[key]=value	Comparison operator, value smaller than or equals.
qsmaller[key]=value	Comparison operator, value smaller than.
qbigger[key]=value	Comparison operator, value bigger than.
qsparse[key]=value	Setting sparse true for a parameter.
qgroup[username]=groupname	Searching content from users within a group.
qcluster[key]=value	Clustering search results.
qoption[key]=value	Giving addiotional conditions to the query.

q[key]=value

The format is used when searching for matching search parameter value. When searching for matches with string type, a partial match is enough. In other words, search value `tampe` would match with metadata `tampere`. All metadata value types listed in Table 3.3 are allowed with this format. An example URL for searching content is presented in Figure 3.6. We can divide the URL into smaller pieces and go through it part by part. First part is the request method: `http`. The second part is the server address `visualrest.cs.tut.fi`. After that is the resource we are requesting: `files`. Finally, after the `?` is the most interesting part, search parameters separated from each other by `&`. The example query is for searching content with metadata `tag=cloud` and also `tag=system`. All content that has both of those metadatas would be returned as search results.

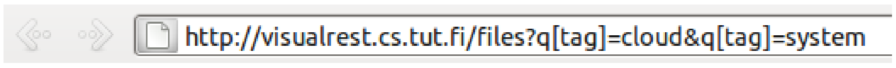

http://visualrest.cs.tut.fi/files?q[tag]=cloud&q[tag]=system

Figure 3.6: Common example query.

qmin[key]=value

Qmin is a comparison operator for searching values that are at the minimum the value given in the parameter. In other words, metadata values that are the same or bigger than the parameter value. Allowed value types are number, date, and datetime. Strings cannot be searched with this, since there is no other intuitive and practical way for comparing strings than partial or exact matches.

In Figure 3.7 the query is for searching content that have metadata rank with value 12 or bigger. All content that have rank value lower than 12 is not returned as a search result.

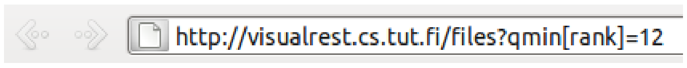

http://visualrest.cs.tut.fi/files?qmin[rank]=12

Figure 3.7: Example query – qmin.

qmax[key]=value

Qmax can be used for number, date and datetime types. It is used to search for content with metadata that is at most the value given or smaller. In Figure 3.8 the example query is for searching content that has essence size at most 50 000 bytes.

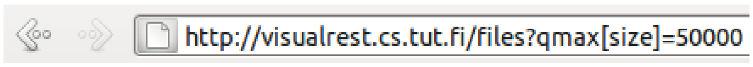

http://visualrest.cs.tut.fi/files?qmax[size]=50000

Figure 3.8: Example query – qmax.

qsmaller[key]=value

Qsmaller can be use used for number, date and datetime types. Content that has metadata with the same key and smaller value would be accepted search results for this parameter. The example query in Figure 3.9 is for searching content that has been modified before february 5th, 2011.

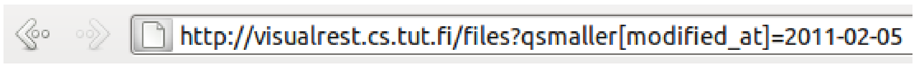

http://visualrest.cs.tut.fi/files?qsmaller[modified_at]=2011-02-05

Figure 3.9: Example query – qsmaller.

qbigger[key]=value

Qbigger can be used for number, date and datetime types. This parameter is for searching content with metadata values bigger than the search parameter value. In Figure 3.10 the query is for searching content that has been created after the year 2010. Files created in the year 2010 are not accepted.

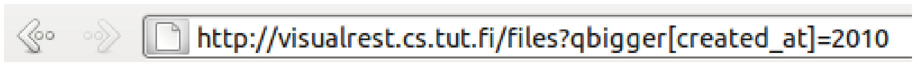

http://visualrest.cs.tut.fi/files?qbigger[created_at]=2010

Figure 3.10: Example query – qbigger.

Range with multiple parameters

Searching for metadata value ranges can be done by giving the beginning and ending values in separate parameters. The query in Figure 3.11 would be a search for content that are created at the minimum in the year 2010 and at the maximum in the year 2010. Therefore this query would be a search for content created in the year 2010.

In Figure 3.12 the example query is for content that has rank value bigger than 10 and smaller than 100. In other words, rank values within the range of 11-99.

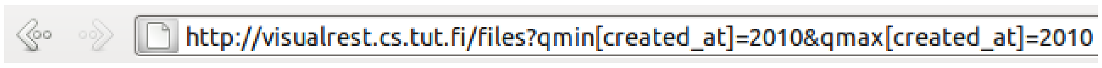

http://visualrest.cs.tut.fi/files?qmin[created_at]=2010&qmax[created_at]=2010

Figure 3.11: Example query – date value range

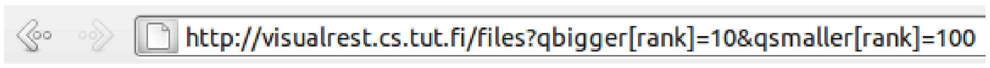

http://visualrest.cs.tut.fi/files?qbigger[rank]=10&qsmaller[rank]=100

Figure 3.12: Example query – number value range.

qsparse[key]=value

Sparse is a parameter option intended for metadata types that are added by users. Normally if content does not have the metadata type of a search parameter, the content is not shown in search results. When sparse is set to true, we will see in search results all content with matching metadata and also content that does not have that specific metadata type at all. Therefore, the only content not accepted in the search results is content that have the queried metadata type, but the value does not match.

Making a query with sparse option selected, might be time consuming in some cases. This is due to additional processing that needs to be done on the server side. However, this does not present an alarming concern since sparse option has not been very widely used among the users of VisualREST. When user is searching for content with certain metadata, usually there is no need to select the sparse option for the search parameter.

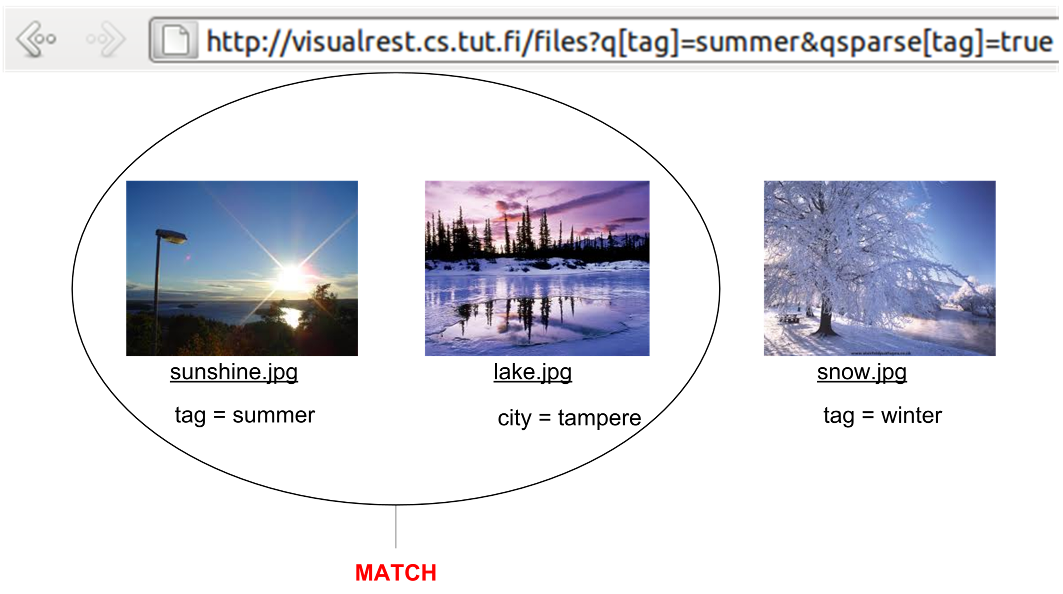

http://visualrest.cs.tut.fi/files?q[tag]=summer&qsparse[tag]=true

Figure 3.13: Sparse example content.

Figure 3.13 has an example query for content that has metadata `tag=summer`. The query also defines sparse true for metadata type `tag`. To demonstrate how the sparse option affects the search results, the figure also has three content with metadata. The first content `sunshine.jpg` is in the search results, because it has matching metadata `tag=summer`. Also the second content `lake.jpg` would be in the search results, because it does not have any metadata with the metadata type `tag`. However, the third content `snow.jpg` would not be part of the search results, since it has metadata type `tag`, but the value does not match.

qgroup[username]=groupname

With qgroup option users can narrow down the search results to be only content owned by users in the specific groups. The parameter consists of username, the owner of the group, and group name. Multiple group names can be given separated by +.

This option is made available for users to make it easier to search for content added to the system by their friends or a desired group of people. In Figure 3.14 we give a search query for all available content that users in Alice's groups called friends and family have.

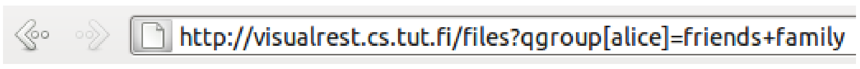

http://visualrest.cs.tut.fi/files?qgroup[alice]=friends+family

Figure 3.14: Example query – qgroup.

qcluster[key]=value

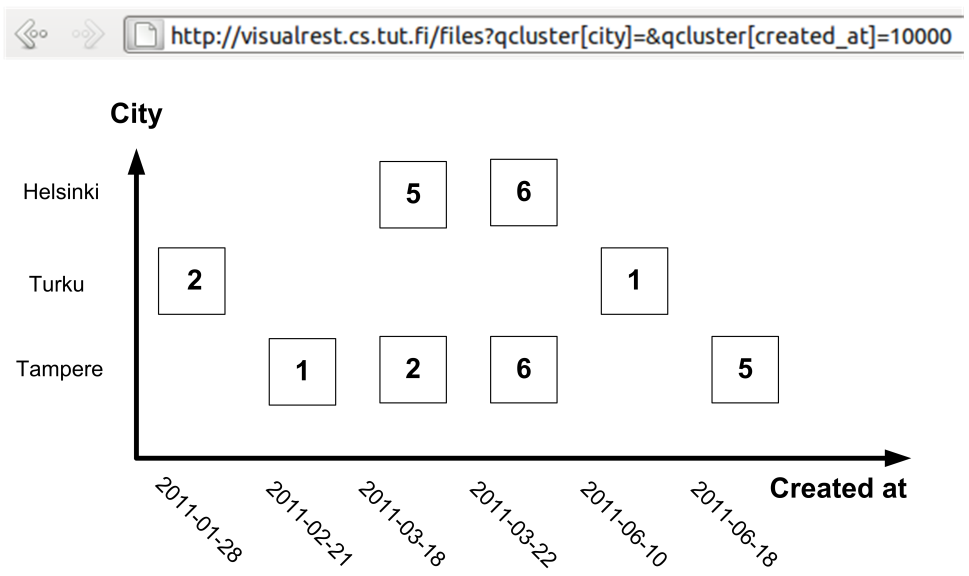
Clustering is a way to represent search results to users. The idea of clustering is to group search results according to some similarities in the content. This helps navigate the results and hopefully makes it easier to find the content that the user is looking for.

Clustering is done to the search results according to clustering parameters given in the query. Clustering can be done in one or multiple dimensions. Each cluster has similar content, according to the clustering parameters. For every cluster, a surrogate is chosen to represent the content in that cluster. Based on the surrogates, the user can easily see in what cluster seems to be the content that the user is interested in. All content in one cluster can then be queried in its entirety.

The clustering parameter consists of key and value. Key is for the metadata type that clustering will be based on. If value type is *string*, the query parameter does not need a value. Afterall, for strings there is no easy way to specify ranges, therefore only values that are exact matches belong in the same cluster.

If value type is a number, the value parameter is a float number. With date and datetime type, the value is the number of seconds. Metadata of the content are compared with each other. The distance of metadata from each other is compared. If the metadatas of contents are closer to each other than the parameter value, the contents belong to the same cluster.

In Figure 3.15 is a clustering example according to creation date and metadata type `city`. For created_at we give value 10000, meaning 10 000 seconds. Creation dates that are less than 10 000 seconds (about 3 hours) apart from each other fall in the same cluster. Since metadata type `city` has value type of string, only exact matches belong to the same cluster and parameter value is not needed.



http://visualrest.cs.tut.fi/files?qcluster[city]=&qcluster[created_at]=10000

Figure 3.15: Clustering example with two dimensions.

In the example we get 8 clusters represented by boxes. The amount of content in each cluster is shown inside the box. Helsinki and Turku have two clusters and Tampere has four clusters. Each of these clusters represents content created in the same city less than three hours apart from each other. Helsinki has a cluster with five content created on the same day, quite near each other. This is already a quite good indication that the content probably has something in common. When we have found a promising cluster, we can request a full list of content that belongs into that cluster and examine the content in more detail.

qoption[key]=value

The following options are available for the query. Possible keys and values are listed in Table 3.5.

Table 3.5: Possible query option (qoption) keys and values.

key	value
sort_by	created_at / modified_at / size / rank / user / device / path / filename
order	asc / desc
available_files_only	true
show_deleted_files	true
query_processing_time	true
sparse	true

Sort_by and **order** are closely related to each other. *Sort_by* parameter defines according to what metadata the search results are sorted. *Order* tells wheter to put the search results in descending or ascending order. By default the order is set to descending.

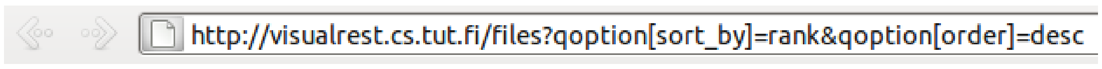
Available_files_only parameter makes it possible to not include content that is not available at the moment. This means content that is not accessible at the moment because the device holding the essence of the content is not online and the essence has not already been uploaded to the VisualREST server.

Show_deleted_files, as the name suggests, accepts in search results also content that has been marked as deleted. By default deleted content is hidden from the query search results. With this parameter only the owner of the content can see the content that is marked deleted.

Query_processing_time can be returned with the query results. This option is not necessarily designed for end users, as much as for developers who are making clients for the system. A client program might be suffering from a slow connection or some other efficiency problem, this option might help determine what is causing the problem.

Sparse true sets the sparse value true for all search parameters. This option is not recommended since with complex queries it might slow down the query.

The example query using qoption, presented in Figure 3.16, is a query that has options for sorting and ordering. The query results are returned sorted by rank value in a descending order.



http://visualrest.cs.tut.fi/files?qoption[sort_by]=rank&qoption[order]=desc

Figure 3.16: Example query – qoption

3.6 Notifications

VisualREST uses XMPP for delivering messages initiated by the server. The previously presented request for uploading content is a direct message from the server to a certain user's device. In addition to this one-to-one type of notification, VisualREST has nodes for delivering notifications from one-to-many. Figure 3.17 presents how XMPP messages are used in node communication. VisualREST server or any of the users can push messages into a node. All listeners of the node are notified about the new message in near real-time. VisualREST has three types of nodes: **/files**, **/contexts**, and **/context/<contextname>**. The different types of nodes are described in the following.

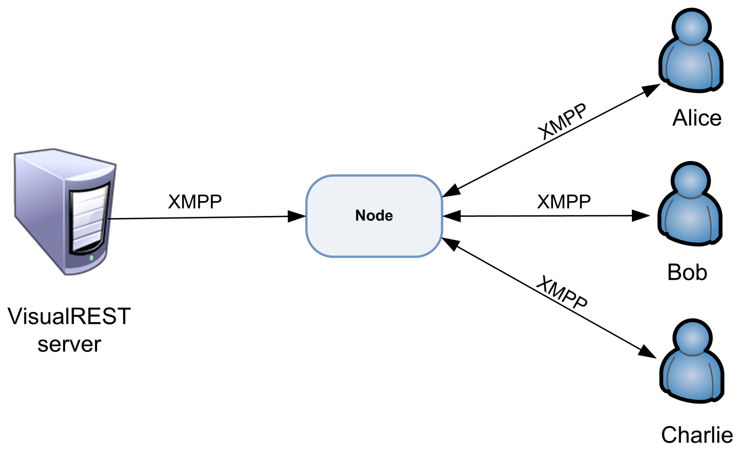


Figure 3.17: Node communication.

/files node is dedicated to notifying users about new publicly available content added to VisualREST. Every user is allowed to register as a listener to this node. An entry is created to the node every time a new public content that is created to the system. Node listeners can implement their own filters for finding content that most likely is interesting to the user. Filters can be based on allowing only certain entries or by filtering out entries that the user is not interested in. For example, the filtering can be used to show content to the user that is: added by friends, marked to be from a certain city or content that has been tagged with certain values.

/contexts node is also public, every user can read all of the entries added to it. This node is for notifying users about new contexts. When new context is created and shared with other users, an entry will be made to this node. An entry has information about the new context and members of the context. This means that also users that are not allowed for a context, will get the notifications. Therefore filtering unwanted entries is encouraged. Privacy must be taken into consideration when sharing information about contexts that are not public. A piece of future work will be to limit the notifying about contexts only to members of the context. This is not possible with only one node and multiple nodes would mean more complexity. Also sending notifications directly to users has certain difficulties, because we would not know what device the notification should be sent to or should it be sent to all of the user's devices.

/context/<contextname> is a node that every context in VisualREST has. The node is used for notifying about changes in the context. Members of the context can listen to the node and get notifications when new content is added to the context. Also if context parameters are modified, all members using the node will be notified.

4. GETTING CONTENT TO THE CONTENT MANAGEMENT SYSTEM

People have an increasing amount of devices and they are using more and more services where content can be stored and shared. In order for content management system to be as comprehensive as possible, content needs to be imported from all of the places users are keeping their content. This means importing content from different types of users' devices and services, such as Flickr or Facebook.

In this chapter we will discuss how content can be imported directly from users' devices. Followed by how content can be harvested and imported from other services into one content management system.

4.1 Importing content from devices

Users' devices are usually the places that content is created and stored at. To make the content easily accessible and shared regardless of place and time, a good solution is to import the content into a content management system. VisualREST offers many ways of importing content into the system. We present three possible ways of importing content from user's devices:

- Client program running on user's device.
- Sending content as mail attachment to the content management system.
- Uploading content on web interface.

All of these methods are supported in VisualREST. Different ways of importing content are needed, because there is no single way of importing content that would be the best in every possible use case.

4.1.1 Container program

For end users the easiest way of importing content to a content management system is to have a container program running on the user's device. This way the importing process can be automated and the user does not need to do anything in order for the content to get imported. Container program updates metadata of new content to VisualREST and also monitors and updates changes in content that is already added to VisualREST.

Negative sides in this approach are that the container program must be installed on the user's device and some settings must be set in order for the container program to know what content the user wants to import into the content management system. Another downside might be that if the user is not carefull with container program settings, some content might get added to the content management system by accident.

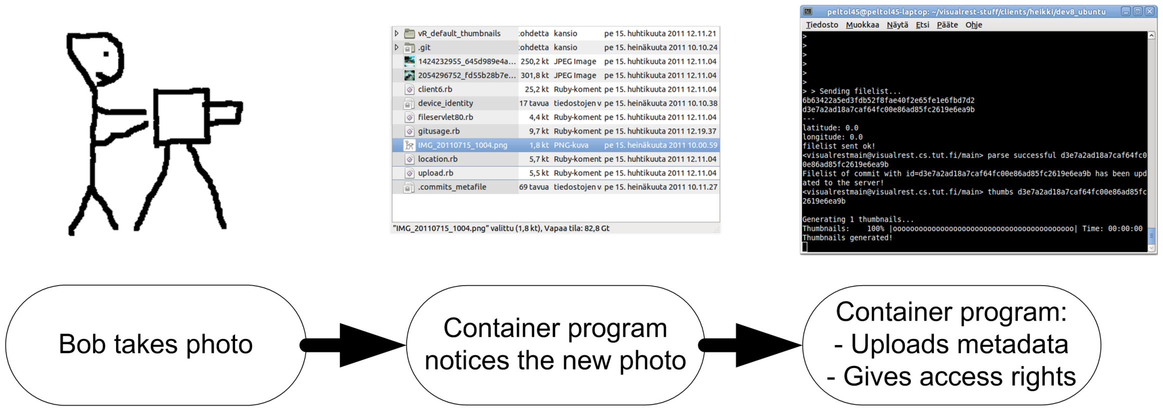
In VisualREST it is up to the client program whether to upload also the essence of the content, or only the metadata. If the essence is not uploaded, the container program needs to implement XMPP interface and upload the essence of the content when the server asks it to do so.

Two different container programs has already been implemented for VisualREST. The first one is done with Ruby, and the other one is done with Python [31] and Qt [32]. The Ruby container program is made simple and efficient, it is text-based, and it does not have graphical user interface. With the text-based interface, it is easy to debug and there are less parts that can break down. It has proven to be very reliable and fast. The program listens to changes in certain folder on user's device and updates them to VisualREST.

The other container program is called MIST [33]. MIST is developed for Maemo platform and it runs on Nokia N900. It is designed to work with the native camera application, observing for new photos taken. As soon as a photo is taken, metadata and the actual essence of the photo is added to the VisualREST.

Figure 4.1 presents a use case from container program's point of view. Bob is taking a picture with his device. The picture is saved on Bob's device which has a container program running. The container program notices the newly created photo and starts processing it. Metadata is extracted from the photo and uploaded into VisualREST. Also possible predefined user access rights can be given to Bob's friends. Essence of the picture is kept on Bob's device. Now Bob's friends can access metadata of the picture on

VisualREST server. If they also wish to access the essence of the picture, it is requested from the server and the server forwards the request to Bob's device.


Bob takes photo
Container program notices the new photo
Container program:
- Uploads metadata
- Gives access rights
peltol45@peltol45-laptop: ~/visualrest-stuff/clients/heikki/dev8_ubuntu
Tiedosto Muokkaa Näytä Etsi Pääte Ohje

vR_default_thumbnails	:ohdetta	kansio	pe 15. huhtikuuta 2011 12.11.21
.git	:ohdetta	kansio	pe 15. heinäkuuta 2011 10.10.24
1424232955_645d989e4a...	250,2 kt	JPEG Image	pe 15. huhtikuuta 2011 12.11.04
2054296752_fd55b28b7e...	301,8 kt	JPEG Image	pe 15. huhtikuuta 2011 12.11.04
client6.rb	25,2 kt	Ruby-koment	pe 15. huhtikuuta 2011 12.11.04
device_identity	17 tavua	tiedostojen v	pe 15. heinäkuuta 2011 10.10.38
fileservlet80.rb	4,4 kt	Ruby-koment	pe 15. huhtikuuta 2011 12.11.04
gitusage.rb	9,7 kt	Ruby-koment	pe 15. huhtikuuta 2011 12.19.37
IMG_20110715_1004.png	1,8 kt	PNG-kuva	pe 15. heinäkuuta 2011 10.00.59
location.rb	5,7 kt	Ruby-koment	pe 15. huhtikuuta 2011 12.11.04
upload.rb	5,5 kt	Ruby-koment	pe 15. huhtikuuta 2011 12.11.04
.commits_metafile	69 tavua	tiedostojen v	pe 15. heinäkuuta 2011 10.11.27

"IMG_20110715_1004.png" valittu (1,8 kt), Vapaa tila: 82,8 Gt

```
>
>
>
>
>
>
> > Sending filelist...
6b63422a5ed3fdb52f8fae40f2e65fe1e6fbd7d2
d3e7a2ad18a7caf64fc00e86ad85fc2619e6ea9b
---
latitude: 0.0
longitude: 0.0
filelist sent ok!
<visualrestmain@visualrest.cs.tut.fi/main> parse successful d3e7a2ad18a7caf64fc0
0e86ad85fc2619e6ea9b
Filelist of commit with id=d3e7a2ad18a7caf64fc00e86ad85fc2619e6ea9b has been upd
ated to the server!
<visualrestmain@visualrest.cs.tut.fi/main> thumbs d3e7a2ad18a7caf64fc00e86ad85fc
2619e6ea9b

Generating 1 thumbnails...
Thumbnails:    100% |ooooooooooooooooooooooooooooooooooooooooo| Time: 00:00:00
Thumbnails generated!
```

Figure 4.1: Use case – New picture in container program.

4.1.2 Sending content as email attachment

If user does not have a container program already installed and wants to quickly add some content to VisualREST, sending the content as an email attachment is one possibility. Sending content as email attachment is an easy and controllable way of adding content to the content management system.

VisualREST server has a dedicated email account for receiving content as mail attachments. The mail address of the server is `attachments@visualrest.cs.tut.fi`. In order for user to be able to send content as email attachment to the server, the user needs to register his/her email address on the server. The only required information is user's email address, additional information can be used for accessing content in the user's mail account, presented in the following section. The server keeps lists of users' email addresses, this is an early development version for security with emails. Mails sent to the server from unknown addresses are ignored and not processed.

RFC 5322 [34] describes the Internet Message Format, a syntax used in emails. The syntax allows the use of `tag` inside the email address. User sending attachments to VisualREST must use this `tag` and provide his/her username with it. When Bob is sending attachments from his mail account to VisualREST, Bob would send the email to `attachments+bob@visualrest.cs.tut.fi`

When the server has received the email, first the provided `tag` with the username is checked. After the user has been found, the sender mail address is compared with the mail addresses added by the user. If the mail address is found, the attachments of the email are fetched and saved to the user's virtual container dedicated to all mail attachments of the user. All user's content added from mail attachments are found in a virtual container dedicated for that purpose.

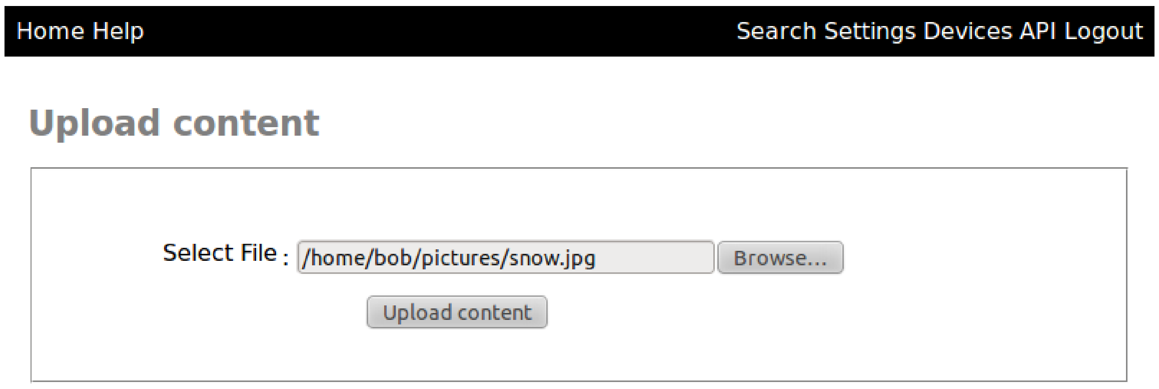
A major downside in sending content as email attachment compared with the container program is the amount of manual work that needs to be done every time content is imported into VisualREST. Nevertheless, if a user is only once importing content from a certain device, using this functionality might be worth the trouble.

Security is an issue with emails. Future work is needed for making sure that the user is really the one that the email is from. Mail headers can be easily manipulated and sender information fabricated. Some kind of authentication parameter is needed while still preserving the easy usage, but for now it is left for future work in VisualREST.

4.1.3 Uploading content with web browser

Content can also be uploaded to the server by using a web browser. The web interface offers an easy way for selecting what content to upload from user's device, and to what virtual container the content will be stored at on the server.

In order for user to be able to upload content with the web interface, the user needs to be signed in. If the user does not yet have a virtual container, it needs to be created. On the web interface user can see all his/her own containers, including all virtual containers. The user can go to the settings of a virtual container the content will be added to. The top of the settings view is presented in Figure 4.2.



Select File : /home/bob/pictures/snow.jpg

Figure 4.2: Content upload view.

From the `browse` button a window opens in which the user can select the file that will be uploaded to VisualREST. When the file is selected, the uploading begins after pushing the "Upload content" button. Virtual containers store the essence of their content on the VisualREST server, and therefore the metadata and essence of the uploaded content are both stored in VisualREST.

As with the previous functionality, sending content as email attachment, also uploading content using the web interface has its constraints. Content needs to be uploaded one file at a time. With large amount of files this would not be a very user friendly solution.

4.2 Importing content from other services

If content is already on some other service or content management system, it can be imported into a virtual container on VisualREST. Nevertheless, the basics of importing from external services might be similar, there is always some work that needs to be done and some parameters adjusted.

In this section, we discuss how content can be imported from different services into VisualREST. We focus on user's point of view, as well as VisualREST's point of view. The example services we import content from are:

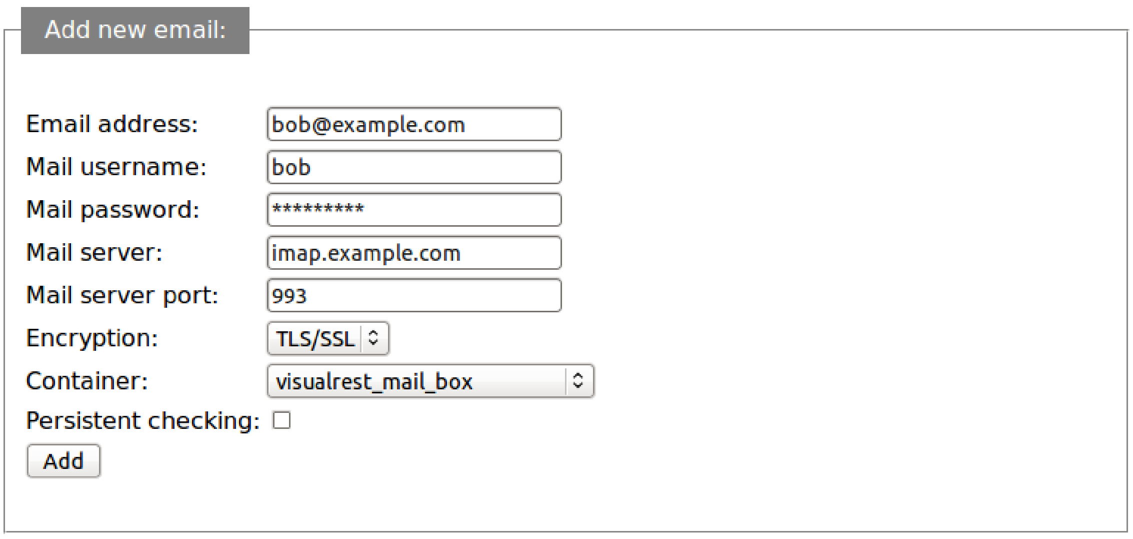
- Mail account,
- Flickr,
- Facebook and
- Dropbox [35].

Each of these services are designed for different kind of uses. They all still have something in common: Each of them has a lot of content. Searching through all of them separately to find the content one is looking for would be time consuming and difficult. By importing all of the content into one content management system, we are offering one interface from where users can access all of their content.

4.2.1 Importing from mail account

Nowadays almost everybody is reachable by email. People are used to sending pictures and other files via email to their friends and family. It is very common to receive the first photos of your sister's new born baby or photos taken at your best friends wedding as an email attachment. Usually, people have many mail accounts with different service providers. New mail accounts are created and old ones are forgotten. Not only the amount of mail accounts, but the amount of emails in those accounts can be very large. Those emails can have a lot of content that could easily get lost. One way of making sure that content received by email is not lost, is to import the content into a content management system.

We have implemented in VisualREST a method that goes through all emails in user's mail account and saves all attachment files into VisualREST. It is possible to let a worker actively check for new emails or do a one-time sweep of user's mail account. Figure 4.3 presents the form in the web interface that is used for adding new email address information to VisualREST. When importing mail directly from user's mail account, all of the fields in Figure 4.3 are mandatory. Table 4.1 describes all of the fields and describes the purpose of each of these parameters.


Add new email:

Email address:	bob@example.com
Mail username:	bob
Mail password:	*********
Mail server:	imap.example.com
Mail server port:	993
Encryption:	TLS/SSL
Container:	visualrest_mail_box
Persistent checking:	☐

Figure 4.3: Adding email information.

When the user email account details are added to VisualREST, the email account is scanned and attachments are fetched. The background processes are not shown to the

Table 4.1: Email parameters.

Parameter	**Description**
Email address	User's email address.
Mail username	Username to identify the user's mail account. Used for authenticating to the mail server.
Mail password	Password to user's mail account. Used for authenticating to the mail server.
Mail server	Address of the mail server.
Mail port	Mail server port number.
Encryption	The method of encryption used for traffic between VisualREST and the mail server. Possible values are TLS/SSL and None, if no encryption is used.
Container	Name of the virtual container the content will be saved to. Container can be choosen from a drop-down list.
Persistent checking	States whether to have persistent checking or not. If not selected the mail account will be checked only once. If selected the mail account will also be checked for new emails at a regular interval.

user. The process from VisualREST's point-of-view proceeds as follows:

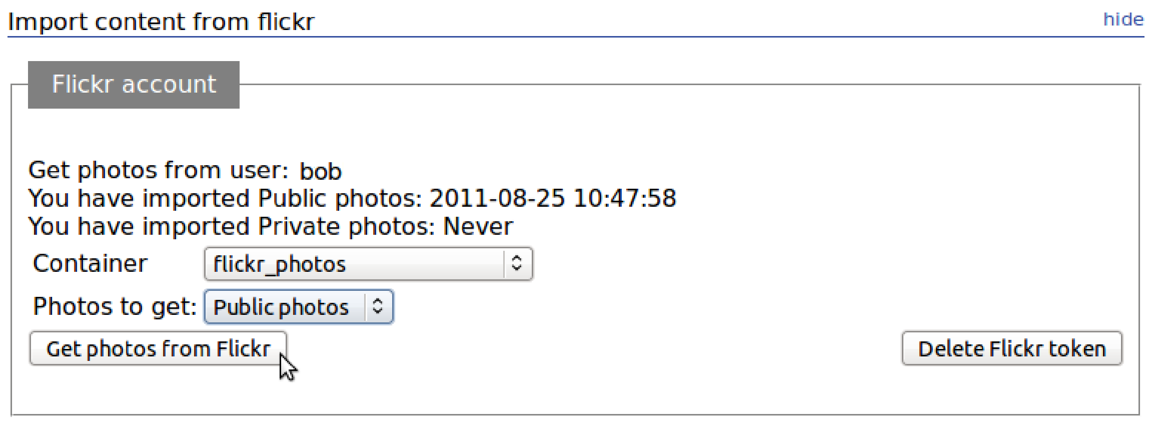
1. Connect to the server with the information provided by the user (server address, port, encryption, username and password).
2. When connected, select INBOX and get a list of all emails.
3. Get each email and check each of them for attachments.
4. If there are attachments, fetch them and save to the virtual container user has selected.
5. Add metadata to the content, such as mail topic, mail sender and mail date.
6. When all of the emails has been processed, save `uid` of the last email. It defines the last mail that has been processed. This will be used when the mail account is checked for new emails.

User's username and password to the mail account are stored in VisualREST database. If a user is not comfortable with having own email account information stored, the user can at any time remove the email information from the VisualREST system.

Spam messages are a well-known problem for everyone these days. Some kind of filtering of spam emails is needed to avoid unwanted pictures ending up in the system. This functionality is left for future work.

4.2.2 Importing from Flickr

Flickr is an image hosting service that is widely used for storing and sharing personal photos. In August 2011, Flickr announced that they were hosting six billion images on Flickr, and the number of uploads has grown 20 percent per year for the last five years [36]. Flickr has a comprehensive and user-friendly API. The API can be used for importing user's photos from Flickr to VisualREST. Figure 4.4 illustrates the interface in VisualREST for importing content from Flickr.


Import content from flickr

Flickr account

Get photos from user: bob
You have imported Public photos: 2011-08-25 10:47:58
You have imported Private photos: Never
Container: flickr_photos
Photos to get: Public photos

Figure 4.4: Get photos from Flickr.

The implementation of importing user's photos from Flickr has been designed to be as straightforward and user-friendly as possible. The following four steps are required from the user:

1. **OAuth authorization for user's photos in Flickr**: User is directed to Flickr login page. After the user has signed in, he/she is asked to grant VisualREST access to his/her photos. When the access has been granted, the user is directed back to VisualREST.

2. **Select what content to import into VisualREST**: At the moment it is possible to select all user's private or public photos from Flickr. By default the imported photos are only accessible to the user.

3. **Select where to import the content**: Content can be imported to any one of the user's virtual containers.

4. **Ask VisualREST to get the selected content**: After the previous steps has been taken user needs to press the "Get photos from Flickr" button. VisualREST will start the importing process and notify the user the amount of photos that will be imported.

VisualREST database has a class for storing information about external services and their authentication parameters. Table 4.2 describes the available attributes. The attributes are for storing all of the information needed for making authenticated requests with certain permissions to the external services. The *extra_1* and *extra_2* fields can be used in different services for different purposes. These fields are needed since the authorization processes and information that needs to be stored on VisualREST server may vary between different systems.

Table 4.2: Flickr information in VisualREST database.

Parameter	**Description**
service_type	Type of service, such as Flickr or Facebook.
service_username	Username to Flickr.
service_id	Distinctive id representing the user.
service_token	Needed for making authenticated API calls. It ties VisualREST to the Flickr user account with reading permission.
extra_1	Last time public photos were imported.
extra_2	Last time private photos were imported.

From the server's viewpoint, importing content consists of two individual processes. The two processes must be done in the corresponding order:

1. **Getting authorization to user's Flickr content**: The authorization process begins with redirecting user to Flickr website. The user authenticates to Flickr and is asked if he/she grants permissions for VisualREST. In this case, we are only requesting read permission. After that the user is forwarded back to VisualREST with a `frob` parameter. The server needs to convert the `frob` parameter to an access token with an HTTP request to Flickr. Flickr returns the access token, which will be saved to VisualREST. Access token is needed for making authenticated and authorized requests to Flickr.

2. **Importing the content from Flickr to VisualREST**: When we have the Flickr access token, we can begin importing content. It is possible to import user's own public or private photos into VisualREST. The user selects what content to import and where to save it. Next VisualREST gets a list of content metadata from Flickr. The list items are processed one-by-one, the metadata of the content is saved on the VisualREST server and the essence on the virtual container. We get metadata such as description, time when picture was taken, time when picture was uploaded to Flickr, tags and URL to the photo on Flickr.

The authenticated requests to Flickr are build according to the Flickr API. The structure is similar in every request but the parameter part varies. The following is an example request to Flickr:

```
http://api.flickr.com/services/rest/?api_sig=<api_signature>
&api_key=<flickr_api_key>&auth_token=<service_token>
&method=<method>&privacy_filter=<privacy_setting>
&user_id=<service_id>
```

Api_signature is a hash value calculated from all the other parameters. It is used for security purposes, so that the request cannot be altered. *Api_key* is an application key, used by Flickr to see which application is making the requests. *Auth_token* is an access token that was earlier requested from Flickr and accepted by the user. *Method* is the action we are executing. The full list of all available methods are described in Flickr API. *Privacy_filter* is related to the method, in this request we are asking for user's private or public photos depending on the value of the privacy filter. *User_id* is an identification value for the user's account.

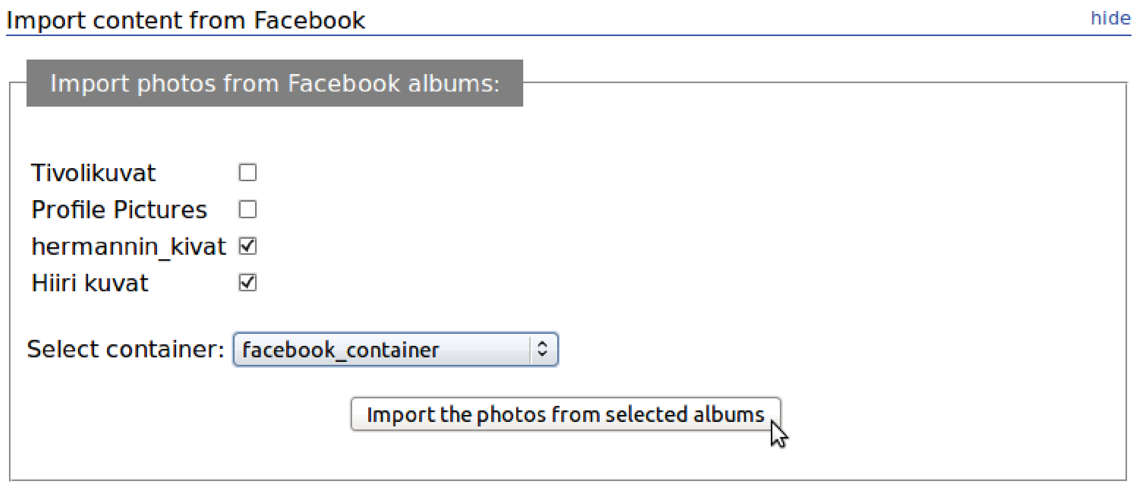
Security issues need to be taken seriously, especially when we are dealing with connecting services so tightly. Authorization parameters are stored on VisualREST database. It is made possible for users to delete all information related to authorization, such as *service_token* and *service_id*.

4.2.3 Importing from Facebook

Facebook is a social networking service with over 500 million users [37]. It allows users to create user profiles, connect with friends, exchange messages and share content, such as photos. In Facebook user's photos are grouped into albums. The photos have optional metadata such as caption, comments from other users, and list of people tagged in the photo. Facebook has a comprehensive API [38] for developing applications using the Facebook interface.

Importing content from Facebook is very similar to importing content from Flickr. The first step is to redirect user to Facebook login page and ask if the user wishes to grant access to his/her content in Facebook. When the authentication is completed, the user is redirected back to VisualREST. After that VisualREST can request the access token from Facebook and store it in the database.

Figure 4.5 presents a screen shot of the importing interface in VisualREST. In this Figure, the user has already authorized VisualREST to get photos from the Facebook account and VisualREST has requested and received the access token from Facebook. We can see a list of albums the user has in Facebook. User checks the content that will be imported to VisualREST. Also the virtual container where the content will be stored at must be selected. After the user presses the "import" button, VisualREST starts importing the selected albums.


Import content from Facebook
hide
Import photos from Facebook albums:
Import the photos from selected albums

Tivolikuvat ☐
Profile Pictures ☐
hermannin_kivat ☑
Hiiri kuvat ☑

Select container: facebook_container

Figure 4.5: Get photos Facebook.

VisualREST requests album details separately for each of the selected albums. The album information includes captions and URLs referring to the photos of the albums. The essence of each photo is requested from Facebook and saved with its metadata to the selected virtual container.

The Facebook access token can be deleted from VisualREST, as well as all other user authentication parameters related to Facebook. This can be done with a single press of a button. If the access token has been deleted, a new one must be requested in order to allow VisualREST to have any access to the user's content in Facebook.

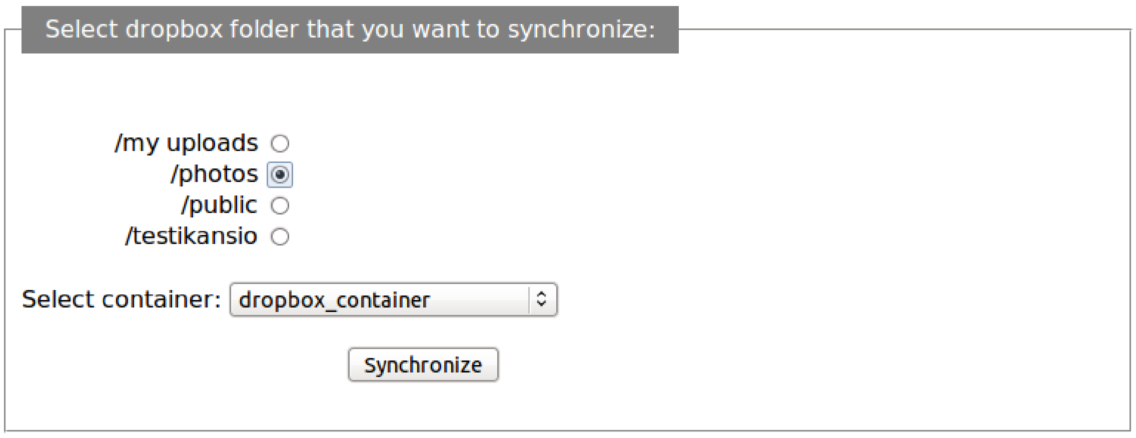
4.2.4 Importing from Dropbox

Dropbox is a file hosting service that utilizes cloud storage. It enables users to store and share files and folders with other users across the internet using file synchronization. Dropbox synchronizes complete folders between the service provider servers and users' computers. Files in Dropbox can also be accessed through a REST interface [39].

The importing process from Dropbox is somewhat different from the previously presented ones. The username and password must be given to VisualREST and with those VisualREST will request the access token to Dropbox. VisualREST does not save the username or password related to Dropbox. Instead only the access token that is requested from Dropbox is stored.

Figure 4.6 presents an example view of the Dropbox importing interface. After the access token has been received, all user's first level Dropbox folders are shown. One of the folders can be selected and a virtual container the content in that folder will be synchronized to. When the "Synchronize" button is pressed, VisualREST will start a poller worker that will retrieve all content in the selected folder into the chosen virtual container. The poller worker will also keep polling at regular intervals for changes in Dropbox. When new content is created in Dropbox, it is also imported to VisualREST. When the essence of a content is modified on Dropbox, it is also updated to VisualREST.

If content is removed from Dropbox, it is marked as deleted in VisualREST. By starting multiple poller workers, we can keep multiple folders from Dropbox synchronized with the selected virtual containers. It is possible to stop and delete these poller workers that are synchronizing content from Dropbox to VisualREST. Also Dropbox access token can be deleted from VisualREST, as well as all user's authentication parameters related to


Select dropbox folder that you want to synchronize:

/my uploads ○
[✓ /photos] ●
/public ○
/testikansio ○

Select container: dropbox_container

Figure 4.6: Get photos from Dropbox.

Dropbox. Even if the poller workers are deleted or the authentication parameters are deleted from VisualREST, all of the synchronized content will still be preserved on the virtual container.

5. SHARING CONTENT USING CONTEXTS

Now that we have presented how content can be easily imported to a content management system, it would be useful if the content could also be easily shared with other people. Access rights to content can be set by hand one file at a time, but this is not very user friendly or efficient way.

We present a notion of context for sharing content in a more automated way among a predefined set of people. This chapter defines what we mean by context in the Visual-REST system. We also describe how content relates to context and how the provided interface can be used.

5.1 What is context

Context is a tool used for grouping content that are somehow related with one another. Contexts can be used for grouping content and also for sharing the grouped content between a defined set of users. Users can jointly produce the content and bring all of the outputs together and share them.

People have a natural tendency of grouping things related to one another [40]. Context makes use of this tendency by making it as natural as possible to use contexts. Context can be related to a certain place, time, people, happening or anything else. Also a combination of the previous is possible. As a matter of fact, there are no restrictions to what kind of parameters or combinations a context is formed of. New types of parameters can be created to meet different kind of needs.

Context can be created for example for a meeting. All of the people attending the meeting would be granted access to that context. All users would be able to add content related to the meeting into the created context. Before the meeting the context could have information such as agenda of the meeting, time schedule and instructions on how to get to the meeting location. During the meeting users could add their notes about the meeting or photos of the whiteboard. After the meeting everyone involved would be able to access all of the collectively created content by the people in the meeting.

Context is defined by its metadata parameters. Context parameters define what content belongs into a context. Every context has a unique *context_hash*. It is used for identifying a context as well as for referencing a certain context. Another important property in a context is *query_uri*. It is used for requesting all of the content that belong into a context. *Query_uri* consists of *context_hash* and all other context parameters. *Query_uri* for context is a normal query for files, with the speciality of *context_hash* as one of the search parameters.

Figure 5.1 presents an example context. The context is owned by Alice and she has named the context `london`. Below the icon are the parameters that every context has, such as *query_uri*, *context_hash* and *rank*. The next parameters are describing the context, but are not mandatory. The parameters describing this particular example context are geographical metadata. Followed by a list of users that are authorized to access this context, also called members of a context. This context is shared with Bob and Charlie. Finally we have metadata describing the content that belongs in this context, in this example we have only one piece of metadata `tag=vacation`.

5.2 Content in Context

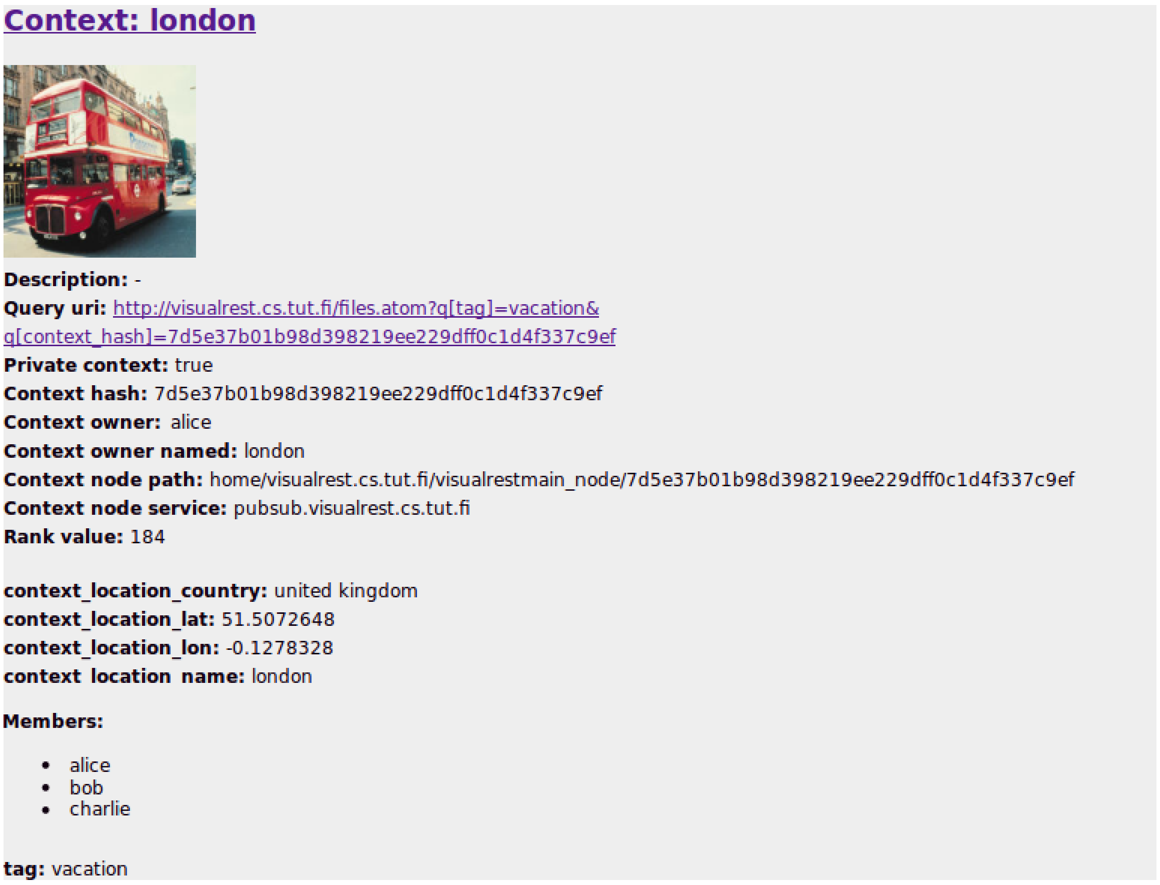
Content is the reason we have contexts. Therefore it is important to fully understand how content gets to be part of a context. In fact, there is no reason why content could not be part of multiple contexts at the same time.

VisualREST does not keep lists that keep track of all content that belongs to a certain context. Everytime context's content is requested using the *query_uri*, VisualREST finds all content in that context and returns a list of the content. Content that belongs to a context, is found with two methods:

1. Matching metadata.

2. Content explicitly marked to be part of a context.

When all content belonging to a context is requested, VisualREST will first find all



Context: london

Description: -
Query uri: http://visualrest.cs.tut.fi/files.atom?q[tag]=vacation&q[context_hash]=7d5e37b01b98d398219ee229dff0c1d4f337c9ef
Private context: true
Context hash: 7d5e37b01b98d398219ee229dff0c1d4f337c9ef
Context owner: alice
Context owner named: london
Context node path: home/visualrest.cs.tut.fi/visualrestmain_node/7d5e37b01b98d398219ee229dff0c1d4f337c9ef
Context node service: pubsub.visualrest.cs.tut.fi
Rank value: 184

context_location_country: united kingdom
context_location_lat: 51.5072648
context_location_lon: -0.1278328
context_location_name: london

Members:

- alice
- bob
- charlie

tag: vacation

Figure 5.1: Example context.

content that have matching metadata with the context. All content that the user is authorized to access will be shown to the user. Content whose metadata matches with the context are considered being part of the context.

Another way that content can belong to a context is if a content has been explicitly marked to be part of a context. In this case metadata of the content and context are not compared. All of the content found with these two methods are part of a context and returned to a user that made the request with the *query_uri* of the context.

The reason why content can belong to a context with both of these ways is to make the process of connecting content to a context more automated. If content fulfills certain descriptions of a context, it will not need to be explicitly marked to be part of a context because the metadata can be used for that connection. As long as content has enough metadata, this process can be automated. Content that the automated process does not work for, can still be explicitly marked to be part of a context.

Figure 5.2 presents an example of how content can belong to a context. In the figure there are two pieces of content that are part of the context. The context has metadata `tag=work` and `city=tampere`. The above content has the same metadata and therefore belongs to the context. The below content does not have the same matching metadata, but instead it has been explicitly marked to be part of the context. The content has metadata *context_hash* with the same value as the context.

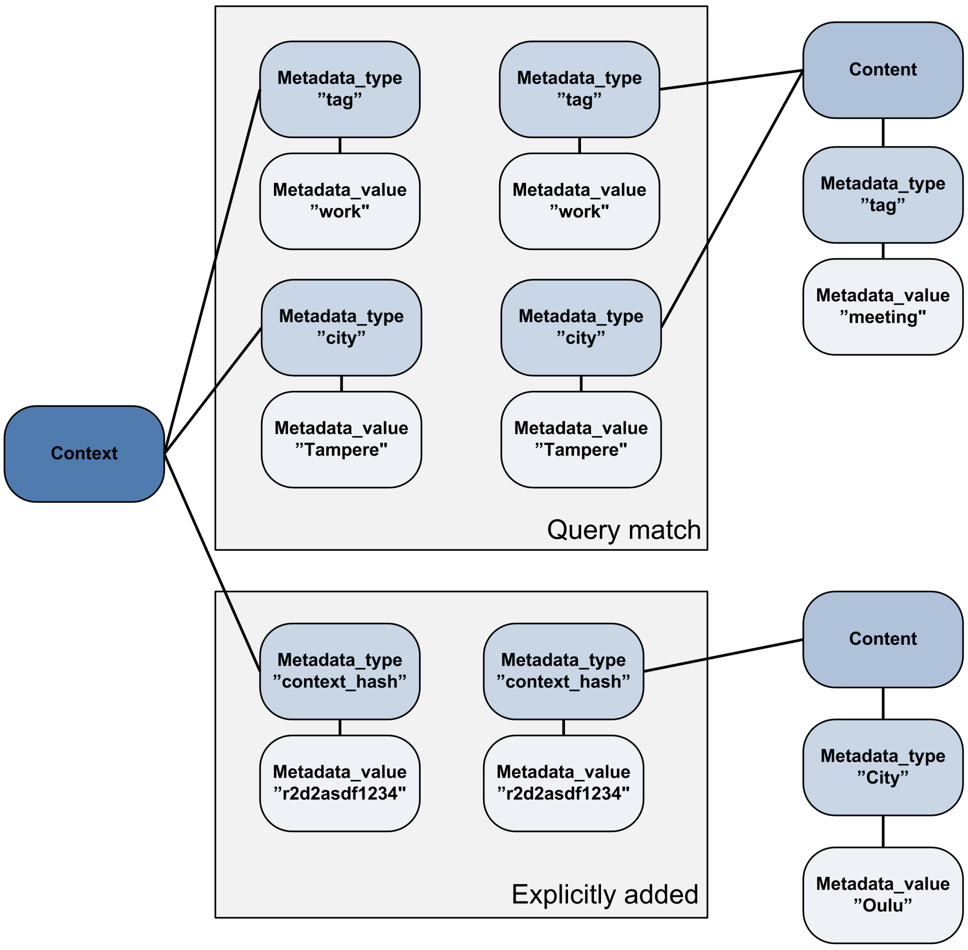


Figure 5.2: VisualREST – content in context.

5.3 Context access rights

The owner of the context can decide whether the context is public or private. Public context is available for every user in VisualREST. Private context is available to the owner of the context and users or groups the owner has decided to share the context with.

When content is explicitly added to be part of a context, the access rights of the content are automatically changed to be the same as the context has. This is why all explicitly added content is available to all users who are authorized to access the context that the content belongs to. On contrary, access rights to content that belongs to a context based on matching metadata will not be automatically changed.

Content that belongs to a context may vary for different users. It is possible that Alice has content and she is the only user that has access to it. If the content has the same metadata as a context, Bob would not see the content as part of the context but Alice would.

5.4 Context interface

This section describes all available HTTP-requests concerning contexts. The context interface is designed to be used by container programs, client programs and web browsers. We present how to create, modify and delete contexts. We also describe how context naming behaves and how contexts can be searched and viewed.

All of the following requests require authentication parameters, except for the search and get functions. The search and get functions can also be used without the authentication parameters, but it will significantly reduce the amount of results since then only public contexts will be available.

5.4.1 Create context

Contexts are created by the users of VisualREST. The creator of a context automatically becomes the owner of the context. The owner of a context cannot be changed. Context can be shared by naming all the members of a context individually or by adding groups of users to become members.

Table 5.1 presents the parameters that are required and available for creating new context. The first line defines the HTTP request method and address for this action. On the following lines are all available parameters and descriptions of the parameters.

The metadata parameter includes arbitrary metadata describing the content in this context. The same metadata types that are used for describing content, are here used for describing content that belongs in this context. The location parameter is used for describing geographical constraints to content in a certain context. Use of the location parameter is not restricted to certain rules. The location can be for example a continent, a country, a city or a building. The location parameter allows a loose way of defining locations.

VisualREST uses OpenStreetMap [41] for connecting locations to cities, countries, and coordinates. At the moment the location parameter is not used for determining whether a content belongs to a context, but it is for users to see what kind of content the context is designed to have.

Table 5.1: Interface for creating context.

POST /contexts
contextname (Required) Name of the context. User that creates the context cannot have contexts with the same name.
metadata (Optional) Metadata defining the context. Context uses the same metadata types as content.
begin_time (Optional) Describes the context to have a beginning time. For example in a context created for a vacation, this could be a date when the vacation begins.
end_time (Optional) Describes the context to have an ending time.
description (Optional) Description of the context. This is used for describing what kind of content the context is supposed to have.
icon_data (Optional) An icon for the context, given in binary data. Icons are used to make contexts more identifiable for users.
location (Optional) Location that the content in this context are related to.
private (Optional) True if it is a public context and false if a private context. Private context is default.
user (Optional) List of users that are granted access to this context and will become members of the context.
group (Optional) List of groups that are granted access to this context and will become members of the context.

5.4.2 Modify context

With the context modifying interface it is possible to change or add context parameters. When context metadata is changed, the *query_uri* is modified accordingly. This means that content belonging to a context might change after the context is modified.

Context parameters can only be modified by the owner of the context. However with this same function it is also possible to explicitly add content to a context. Adding content to a context is only made possible to user with access right to the context and ownership of the content, because it will change access rights to the content and it is not desirable to give users the power of defining the visibility of other user's content.

Table 5.2 presents the interface for modifying context. Context can be referenced in two ways: with the *context_hash* or *username-contextname* pair. Content is added to a context with the *file_uri* parameter. The content that will be added to the context is identified by giving the content's URL in the *file_uri* parameter. Content can be added to a context one at a time, multiple content needs to be added with multiple requests. Parameter referring Bob's photo `cloud.jpg` in Bob's device laptop would be:

```
file_uri=http://visualrest.cs.tut.fi/user/bob/
device/laptop/files/cloud.jpg
```

Table 5.2: Interface for modifying context.

POST /contexts/<context_hash> **POST** /user/<username>/context/<contextname>
file_uri (Optional) Adds a content to the context.
Other parameters are the same as the optional parameters in table 5.1.

5.4.3 Delete context

The number of contexts can increase significantly when users start testing the use of contexts. To get rid of unwanted and redundant contexts, Table 5.3 presents the interface for deleting contexts.

Table 5.3: Interface for deleting context.

DELETE /user/<username>/context/<contextname>
No parameters.

The only user allowed to remove a context is the owner of the context. When context is deleted all context information and references to the context will also be removed.

Nonetheless, content added to the context will not be removed. Content will still remain in their own containers.

5.4.4 Name context

Contexts can be named individually for every user that has been authorized for that context. This means that two users can have a different name for the same shared context. The reason why it is important to allow individual naming of a context is that a context does not necessarily mean the same thing for different people. Alice could name a context `my-wedding` when Bob would name the same context `alices-wedding`. Another problem without individual naming would appear if Bob and Charlie gave the same name for their own contexts and shared them with Alice. Without individual naming Alice could get confused with two different contexts having the same name.

Figure 5.3 presents a high-level illustration of context naming. The Figure has a context with metadata `country=finland` and `tag=holiday`. Context's *context_hash* value is `6e57f2ed24d57a713`. Alice is the creator of the context and she has decided to share the context with a group that consists of Bob and Charlie. Alice has named the context *holiday*, Bob has named it *holiday-in-finland* and Charlie has named it *finland*.

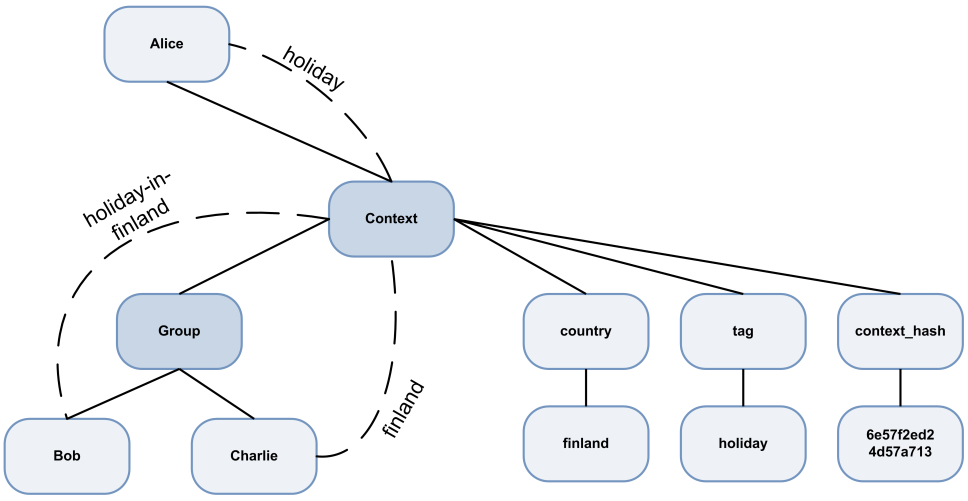


Figure 5.3: High-level example of context naming.

A context can have multiple names given by different people. Therefore, referencing a context by name can also be done with multiple names. In addition to referencing context by name, context can also be referenced using the unique *context_hash*. When using user given context names, the format must be "username.contextname". Different possible ways of referencing the example context presented above are:

- Alice's given name: `alice.holiday`
- Bob's given name: `bob.holiday-in-finland`
- Charlie's given name: `charlie.finland`
- Unique context_hash: `6e57f2ed24d57a713`

Table 5.4 presents the interface for changing contextname to a user. The request uses HTTP PUT method. The address consists of user's <username> and the new name given to the context in <contextname>. In order to identify the context that will be named, parameter *context_hash* or *old_name* is required.

Table 5.4: Change contextname for a user.

PUT /user/<username>/context/<contextname>
context_hash (Required context_hash or old_name) Unique context_hash.
old_name (Required context_hash or old_name) Old name of the context in the form "<username>.<contextname>", for example "alice.holiday".

5.4.5 Search contexts

Contexts can be searched from VisualREST with HTTP GET method. Table 5.5 presents the interface for searching contexts. Contexts can be searched from the whole system or only contexts that a certain user has created or named. All parameters are optional. Without search parameters all available contexts are returned. Search results can be narrowed down by *context_name* or *username*. Results can be sorted in ascending or descending order.

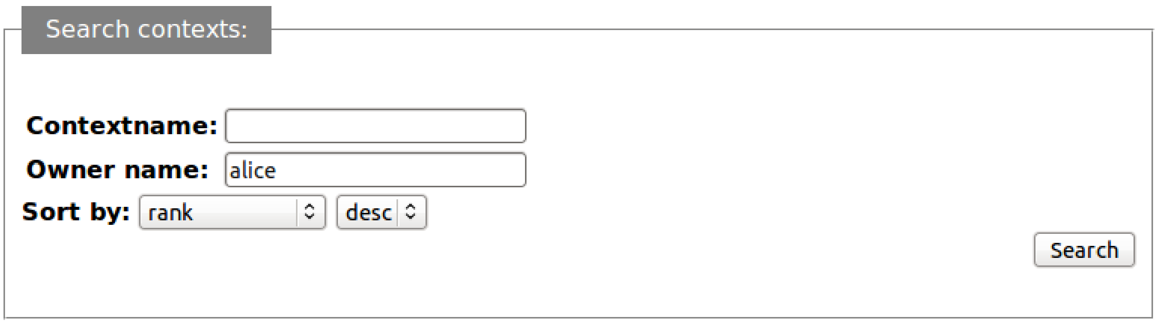
Contexts can be searched as a signed in user or as a guest. A guest user does not provide any authentication parameters. Guest user will only find contexts that are public and accessible by anyone. As a signed in user public and private contexts the user is a member of are searched according to the search parameters. Figure 5.4 presents the web

Table 5.5: Search for contexts.

GET /contexts **GET** /user/<username>/contexts
context_name (Optional) Name of the context or part of the name.
username (Optional) Returns only contexts that this user is a member of.
sort_by (Optional) Parameter the results will be sorted by. Possible values are: *date_added*, *date_updated*, *name* or *rank*.
order (Optional) Value can be *DESC* or *ASC*, meaning descending or ascending order. Default is descending.
format (Optional) The format in which the information is returned. Possible values are *html* and *atom*. Html is default.

interface VisualREST offers to users. The Figure has an example for searching all Alice's contexts sorted by rank value in descending order. The presented example context search query would be to URL:

```
http://visualrest.cs.tut.fi/contexts?username=alice
&sort_by=rank&order=desc
```


Search contexts:

Contextname:

Owner name: alice

Sort by: rank desc

Figure 5.4: Context search web interface.

Search results are returned in HTML or Atom-feed. The search results have a list of found contexts and provide information and links to the contexts. Context search can be used for finding content in an interesting context or for finding a context that the user can add his/her content to.

5.4.6 Get context

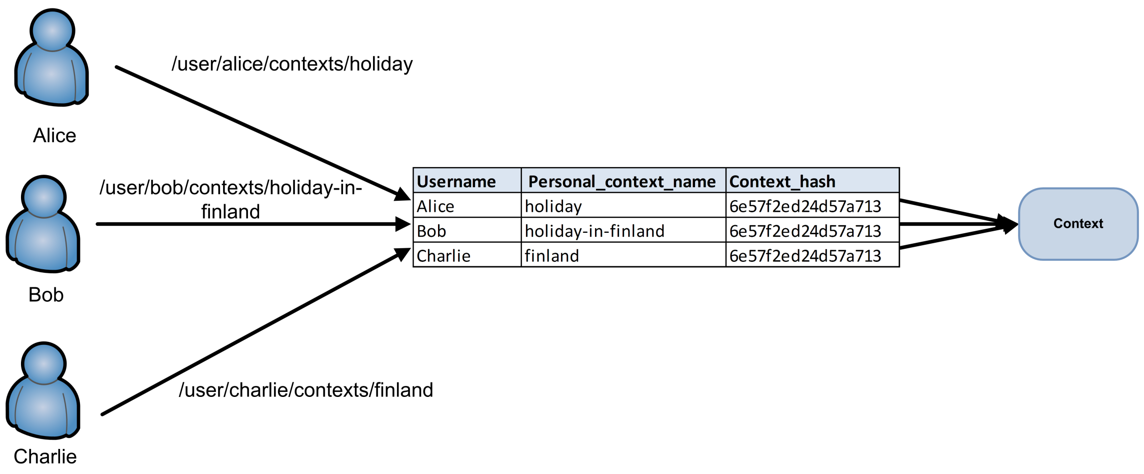
Getting context is not the same thing as getting content that belongs to a context. When requesting a context, the response has all available information about the context, such as *name*, *query_uri*, *context_hash*, *owner*, list of users allowed to access the context, *rank* value and metadata of the context. With the *query_uri* all content that belongs to that context can be requested.

Table 5.6 presents the interface for requesting information about a certain context. Context can be requested either with <*username*>-<*contextname*> pair or by using the *context_hash*.

Table 5.6: Get context information.

GET /user/<username>/contexts/<contextname> **GET** /contexts/<context_hash>
format (Optional) The format in which the information is returned. Possible values are *html* and *atom*. Html is default.

A context that has been named by several different users, can be referenced by several different URLs. Figure 5.5 presents an example of how Alice, Bob and Charlie each have their own URL which points out to the same context. The URL consists of a username and a personal context name. VisualREST keeps track of context names and connects them using the unique *context_hash* values.


Alice
/user/alice/contexts/holiday
Bob
/user/bob/contexts/holiday-in-finland
Charlie
/user/charlie/contexts/finland
Context

Username	Personal_context_name	Context_hash
Alice	holiday	6e57f2ed24d57a713
Bob	holiday-in-finland	6e57f2ed24d57a713
Charlie	finland	6e57f2ed24d57a713

Figure 5.5: Contextname and URL affiliation.

6. EVALUATION

This chapter focuses on evaluating the presented and implemented features of VisualREST. The three main features of the VisualREST content management system this thesis has presented are *content query interface*, *importing content*, and *contexts*. Next we will discuss these features in detail and evaluate how they meet their purpose.

6.1 Content query interface

VisualREST already had a content query interface, but it was considered too complicated and deficient. Query parameters were not as descriptive and unambiguous as they should have been. Also new features needed to be added and it was not possible with the old query interface.

The biggest issues with the old query interface were with comparison operators and date values. The comparison operators were given to number parameters by appending the parameter name with `min` or `max`, and for date types with `before` or `after`. The parsing of the parameters was not always uniform and the parameters got confusing with the added appendixes. Another issue that was noticed was with date values. It was possible to give a date in multiple formats. This was designed to be user friendly so that the value can be given in the format the user wishes to. But it turned out to be more of a burden than a benefit, both users and developers found this to add unnecessary complexity and not to bring any added value to the system. Therefore, we allowed the date value to be given in a unified format that was the most intuitive.

The main idea of the enhanced query interface was to add prefixes to all query parameters. The prefixes are another dimension to the regular `type-value` pair. The prefixes allow us to use the combination `prefix-type-value` for specifying each query parameter and their purpose in more detail. Furthermore, the prefixes allow us to add descriptive parameters to the query, such as the previously presented `qoption` and `qcluster` parameters. Additionally, the prefixes allow us to describe more precisely comparison among numerical and time related parameters.

The new query interface has proven to be more readable and easy to use than the previous one. The query URI that is used for making queries is longer than it used to be for the same query. Nevertheless, the readability of the queries has increased significantly. Queries are more readable to users and more easily parsed and processed by client programs and the server. Not only the usability of the interface has improved, but also we were able to add more features to the interface that would not have been easily implemented into the old interface.

6.2 Importing content

Users have large amounts of content in different devices and services. In order to make all of the content in different places as valuable as possible to the user, all of the content should be accessible through one access point. For this end, we have implemented several ways of importing content to the VisualREST content management system.

Users create a lot of content on their devices, therefore it is natural to implement ways of importing content directly from their devices that are connected to the internet. Content can be imported from users' devices using container programs, sending the content as email attachment or by uploading the content on the web interface. All of these methods have their advantages and disadvantages. The container programs are good for automatic importing, when new created content gets automatically added to the content management system. The other two methods are good for more infrequent importing when content is manually added to the content management system.

Importing users' existing content from other systems to VisualREST has been implemented for mail accounts, Flickr, Facebook and Dropbox. Importing content from a mail account differs from the others, since users do not have so much control over what content are sent to their mail account. Of course users can delete unwanted emails and attachments. Users can even send attachments to their own mail accounts. The content in mail accounts is static, as new content can be sent but existing content cannot be modified. Our implementation of importing email attachments offers the possibility of actively checking for new emails. With this function it is possible to make sure all new email attachments get added to the content management system.

For accessing user's mail account, the user must provide his/her username and password to VisualREST. Safety aware users might not be comfortable giving their usernames and passwords to a third party. This issue should be taken into consideration in future work, but for the time being there is no alternative way of getting access to users' mail accounts without the users giving their usernames and passwords to their mail accounts. Also importing content from Dropbox requires the user to give his/her username and password to VisualREST. The Dropbox API was launched in July 2010 [42] and is still under development. The possibility of OAuth type of authentication is expected to become available in the near future.

Content that is imported from external systems to VisualREST might get modified. If we want to keep the content in VisualREST synchronized with the modified content in its original place, we must have a way of knowing about the changes in the content. With Dropbox it is possible to add listeners that automatically update all changes in the content to VisualREST. Neither Flickr nor Facebook provides a good way of getting notifications about changes in the content. The only way is to manually poll each content for changes. This is obviously not a very good solution, since when the amount of content increases significantly, the amount of polling would take too much resources from the server. For this reason VisualREST does not offer automatic synchronization of content in Flickr or Facebook, instead content can be manually updated by importing the same content again to VisualREST.

Previously, it was only possible to have containers that hold the essence on users devices. In order to import content to VisualREST from other services, there needed to be a way of storing content only on the server. For this purpose we created virtual containers. A virtual container is designed to work like any other container, except that the essence and the metadata both are stored on the server. Virtual container is a natural addition to the existing containers. Virtual containers require more storing capacity from the server, but on the other hand requesting essence from a virtual container is much faster than from a container on user's device.

6.3 Using contexts

Content management systems are designed for managing content. It is important to store the users' content as well as to offer appropriate ways of accessing and sharing the content. In VisualREST contexts were developed for making it easier to share content with other users. Contexts are used for grouping similar content together. Content belongs to a context if the owner of the content explicitly defines the content to be in a context or if the content has matching metadata with the context. Contexts are very useful for meetings, parties and other events as they can be used for distributing content related to the event and for sharing content that has been produced for the event or at the event.

In our tests, contexts have proven to be a valuable tool for sharing content easily with a predefined group of people. The content in a context can be collectively created and shared with the people allowed for the context. Even though contexts have proven to be very useful, some concerns have emerged related to context naming and content belonging to a context. As described in the previous chapter, contexts can be individually named for different users. This feature was implemented to give users more power over how they want to name contexts they are using. This can be confusing if two users have different names for the same context and are talking to each other about the context. Users need to be aware of this kind of functionality. When requesting context information from the server, the response includes a name the user has given to the context and the name the owner of the context has given to the context. Context can also always be referenced with a unique *context_hash*, but that is not very user friendly. Using the *context_hash* is more suitable for being used by application programs.

Test users have also had some issues related to content belonging to a context. Without knowing how contexts are designed to work, it might not be clear how content is determined to belong to a context. If content is explicitly marked by the content owner to be part of a context, the content access rights are modified so that all users allowed to the context are also allowed to access the content. On the other hand if a private content belongs to a context because the content has matching metadata, the content belongs to the context only from the content owner's point of view. Since the content owner is the only user allowed to access the content, it will not be shown to other users as part of the context.

7. CONCLUSION

This thesis has presented development made for an existing content management system named VisualREST. The presented developments are improved query interface for content, new ways of importing content to the content management system and contexts for making it easier to share content with other users.

Content is searched from the VisualREST with search queries. The search queries are formed by adding search parameters next to each other. The new and improved content query interface has made the queries more transparent and expressive. It can be more easily parsed with application programs and it is also more readable to users. The introduction of prefixes in search parameters has made it possible to target parameters more precisely. For example a search parameter can be targeted to describe the whole query or only one search parameter.

Users have content stored in many of their devices and they are using different services that are storing their content. In order for users to be able to easily access all of their content from the different locations, VisualREST is offering ways to import all of their content into one content management system. The thesis presents several ways of importing content into VisualREST from users' devices and services that are storing users' content. Users are provided different ways of importing content from their devices for different kinds of needs. Content can be imported one at a time or by having a container program running on user's device automatically importing all new content. Functionality for importing content from external services is implemented for mail accounts, Flickr, Facebook and Dropbox. Importing content from external systems follows the same principles, importing from new services can be implemented relatively easily.

Sharing content with other users is made easier with the introduction of contexts. Contexts are designed for grouping and managing access rights to content. Contexts are a good way of sharing content between a group of people in such situations as meetings or parties. The creator of a context defines users that are members of the context. All members of the context are allowed to add content to the context and access content added by the other members.

REFERENCES

[1] Internet usage statistics. http://www.internetworldstats.com/stats.htm. [Referenced 17.10.2011].

[2] Niko Mäkitalo. *RESTful Content Management System for Distributed Environment*. Master of Science thesis, Tampere University of Technology, 2011.

[3] Bob Boiko. *Content Management Bible*. Wiley Publishing, Inc, Indianapolis, USA, 2005.

[4] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Designs*. Pearson Education Ltd, Essex, England, 2001.

[5] Philip Laven and Mike Meyer. EBU / SMPTE Task Force for Harmonized Standards for the Exchange of Programme Material as Bitstreams, August 1998.

[6] Andreas Mauthe and Peter Thomas. *Professional Content Management Systems: Handling Digital Media Assets*. John Wiley & Sons, Ltd, West Sussex, England, 2004.

[7] W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld. Face recognition: A literature survey. *ACM Computing Surveys*, 35:399–458, December 2003.

[8] Roy Thomas Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

[9] Network Working Group. *RFC 2616 – Hypertext Transfer Protocol – HTTP/1.1*, June 1999.

[10] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, Inc., Sebastopol, CA, May 2007.

[11] Network Working Group. *RFC 5531 – RPC: Remote Procedure Call Protocol Specification Version 2*, May 2009.

[12] Jabber Software Foundation. *RFC 3920 – Extensible Messaging and Presence Protocol (XMPP)*, October 2004.

[13] Peter Saint-Andre. Streaming xml with jabber/xmpp. *Internet Computing, IEEE*, 9(5):82 – 89, september 2005.

[14] Network Working Group. *RFC 5246 – The Transport Layer Security (TLS) Protocol*, August 2008.

[15] Peter Millard, Peter Saint-Andre, and Ralph Meijer. *XMPP Standards Foundation. XEP-0060: Publish-Subscribe*. XMPP Standards Foundation.

[16] Ruby on Rails web application framework homepage. `http://rubyonrails.org/`. [Referenced 08.09.2011].

[17] Ruby programming language homepage. `http://www.ruby-lang.org/`. [Referenced 11.09.2011].

[18] Sam Ruby, Dave Thomas, and David Hansson. *Agile Web Development with Rails*. Pragmatic Bookshelf, Raleigh, North Carolina, 2009.

[19] Michael Hartl. *Ruby on Rails 3 Tutorial: Learn Rails by Example*. Addison-Wesley Publishing Co., Boston, Massachusetts, 2010.

[20] Obie Fernandez. *Rails 3 Way*. Addison-Wesley Publishing Co., Boston, Massachusetts, 2010.

[21] Ruby on Rails API documentation. `http://api.rubyonrails.org/`. [Referenced 08.09.2011].

[22] Internet Engineering Task Force. *RFC 5849 – The OAuth 1.0 Protocol*, April 2010.

[23] Flickr API documentation. `http://www.flickr.com/services/api/`. [Referenced 20.09.2011].

[24] Amazon Web Services API documentation. `http://aws.amazon.com/`. [Referenced 20.09.2011].

[25] Google AuthSub API documentation. `http://code.google.com/apis/accounts/AuthForWebApps.html`. [Referenced 20.09.2011].

[26] Niko Mäkitalo, Heikki Peltola, Joonas Salo, and Tuomas Turto. VisualREST: A Content Management System for Cloud Computing Environment. In *Proceedings of the SEAA 2011 – 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 183–187, 2011.

[27] Object Management Group. *Unified Modeling Language: Infrastructure 2.0*, March 2006. `http://www.omg.org/spec/UML/2.0/`.

[28] Flickr homepage. `http://www.flickr.com/`. [Referenced 14.09.2011].

[29] Facebook homepage. `http://www.facebook.com/`. [Referenced 14.09.2011].

[30] Network Working Group. *RFC 3174 – US Secure Hash Algorithm 1 (SHA1)*, September 2001.

[31] Python programming language homepage. `http://www.python.org/`. [Referenced 23.09.2011].

[32] Qt framework homepage. `http://qt.nokia.com/`. [Referenced 23.09.2011].

[33] Subhamoy Ghosh, Juha Savolainen, Mikko Raatikainen, and Tomi Männistö. Cloudifying user-created content for existing applications in mobile devices. In *Proceedings of the COMPSAC 2011 – 35th Annual IEEE Computer Software And Applications Conference*, pages 4–11, 2011.

[34] Network Working Group. *RFC 5322 – Internet Message Format*, October 2008.

[35] Dropbox homepage. `http://www.dropbox.com/`. [Referenced 14.09.2011].

[36] Flickr Boasts 6 Billion Photo Uploads. `http://news.softpedia.com/news/Flickr-Boasts-6-Billion-Photo-Uploads-215380.shtml`. [Referenced 19.10.2011].

[37] Facebook Statistics, Stats and Facts for 2011. `http://www.digitalbuzzblog.com/facebook-statistics-stats-facts-2011/`. [Referenced 08.10.2011].

[38] Facebook API documentation. `https://developers.facebook.com/docs/reference/api/`. [Referenced 16.09.2011].

[39] Dropbox API documentation. `https://www.dropbox.com/developers/docs`. [Referenced 16.09.2011].

[40] Matti Rintala and Jyke Jokinen. *Olioiden Ohjelmointi C++:lla*. Talentum Media Oy, Helsinki, Finland, 2005.

[41] OpenStreetMap homepage. `http://www.openstreetmap.org/`. [Referenced 19.09.2011].

[42] Dropbox API Updates. `https://www.dropbox.com/developers/announcements/6`. [Referenced 27.01.2012].