Diploma Thesis

Design and Implementation of a Metadata Based Software Browsing Interface for the openSUSE Build Service

for reaching the degree of

Diplom Informatiker (FH)

Naturwissenschaftlich-Technische Akademie Prof. Dr. Grübler

Staatlich anerkannte Fachhochschule und Berufskollegs

University of Applied Sciences

Isny im Allgäu

written by David Mayr, thesis@davey.de

Matriculation Number: 8119

Prof. Dr. Dietrich Kuhn, kuhn@fh-isny.de

Realized at the SUSE Linux Products GmbH

Supervised by Klaas Freitag, freitag@suse.de

Abstract

One of the biggest advantages of open source is the sheer number of available software with fast update cycles. This advantage also comes with a cost: Software authors have to make sure that their software runs on the currently available systems. Users demand binary versions of the software because compiling takes a long time and is sometimes difficult to do.

The openSUSE Build Service was designed to provide software authors a means to easily build packages for multiple Linux distributions and to publish it to a broad user audience with very little effort.

As the number of software packages in the Build Service grows continuously, getting a quick overview of the most interesting software is nearly impossible. This makes the process of browsing through all packages exhausting and inefficient for users.

This thesis describes the design and implementation of an enhancement of the openSUSE Build Service: Metadata and statistics based browsing of the Build Service content. Users can browse the Build Service for the latest added and updated software, the most downloaded packages, the most active projects and the highest rated Build Service projects and packages.

Based on this information, top 10 lists are provided in two forms: first in the *webclient*, the primary Build Service user interface for human users and secondary in the Build Service *frontend API* for use in other applications.

The system implemented in the context of this thesis has advantages for three groups of users: The software authors and package maintainers can see how popular and utilized their software is, the project managers inside SUSE can use the information to determine how active the maintainers are and which packages are the most demanded ones and the end-user of the software can find and browse the content of the Build Service more convenient.

Annotations

This work was created at the **SUSE Linux Products GmbH**.
(Maxfeldstrasse 5, D-90409 Nürnberg)

Credits

I'd like to thank all the people who supported me while working on this diploma thesis and my parents, who made that all possible. Special thanks to my tutor Prof. Dr. Dietrich Kuhn of the nta and Klaas Freitag, my tutor at SUSE. Additional thanks to all my proofreaders.

Production Environment

This work was written with several computers all running the open source operating system Linux and the graphical user interface KDE. For typesetting this document I used TEX from Donald E. Knuth and LATEX from Leslie Lamport.

Statutory Declaration

Hereby I declare that the present Diploma Thesis was made by myself. Prohibited means were not used and only the aids specified in the Diploma Thesis were applied. All parts which are taken over word-to-word or analogous from literature and other publications are quoted and identified.

David Mayr — July 11, 2007

Contents

1 Introduction

Linux is the most propagated computer operating system these days. Huge companies and the municipalities of big cities decide to use Linux for their computers. Most of the software for Linux is free and *open source* (see glossary) like Linux itself, developed by thousands of individuals around the whole world. The necessity to collect this software and bundle it into *packages*, to be able to install it in an easy, convenient and uniform way, is the focus of Linux distributors like SUSE.

A big advantage of open source software is the huge number of available software with fast update cycles. This advantage also comes with a cost: Software authors have to make sure that their software runs on the currently available systems. Users demand binary versions of the software because compiling takes a long time and is sometimes difficult to do. But to keep all binaries up to date for the existing systems is very time consuming. Sometimes it is even impossible when the software author has no access to a particular architecture or distribution.

Project hosting services like SourceForge [Sou07] provide access to *compile farms* which consist of a number of hosts of different architectures that have different versions of distributions installed. But building packages with it is still a manual process. Other systems can create builds for only one distribution.

The openSUSE Build Service was designed to provide software authors a means to *easily* build packages *for multiple distributions* and have them automatically be rebuilt if the distribution changes.

Every software author can use the Build Service to build his software for multiple distributions and keep it automatically up to date. Users who like to experiment can create custom tailored package versions. People in search for a specific package can check the Build Service if it hosts a version that meets their needs.

The Build Service features a public programming interface through a REST-based web service (see section 3.2.4) using XML over HTTP. The interface is designed for flexibility and easy integration with other tools and frameworks to make Linux software authors, packagers and users creating and deploying packages efficiently and joyful.

Problem Description and Purpose

While the Build Service grows and software authors from all over the world add their software to the openSUSE Build Service, the number of software packages grows continuously. For getting an overview of the most interesting software in the Build Service, going through all the software packages, reading and comparing their description and purpose is nearly impossible. This makes the process of browsing through all packages hard and inefficient for end-users that are looking for new software.

Thousands of people already use the Build Service and its products. Most of them are very experienced: They know what they are looking for. The approach of this thesis to face the fact that the Build Service needs an easier way to find the most exciting and useful software in the masses of packages.

This is done by generating statistics of the *usage* and *popularity* of software packages and visualize them, so that Build Service users easily know which packages other users consider useful and interesting. It does not guarantee that everyone will consider it interesting. But this gives the users an idea what is worth to take a closer look at including a much higher chance that it turns out to be better than just trying out anything randomly.

The enhancements that are necessary to help users browsing the Build Service more efficient for interesting software, will be specified, designed and implemented in this thesis.

Structure of this Document

This thesis is structured in six main chapters.

After the introduction in chapter one, I will explain the most important theoretical basics needed in order to understand this thesis in chapter two.

In the third chapter I introduce the company, the team and project where this thesis was written. I also present the tools and main technologies used to accomplish this work and explain the architecture of the openSUSE Build Service.

The fourth chapter is the main part of this thesis. It includes the specifications of the work with use case analyses, the design of the planned features the implementation and integration into the Build Service.

Chapter five points out some aspects of the runtime experience after the implementation went to the production systems. Stated is also, what was done to face performance issues, as well as what was done to ensure that regular tests of the code is established. Furthermore the deployment of how the code is handed over to the production servers is explained.

The last chapter covers the conclusion of this work. It also gives an outlook of what could be improved in addition, but was out of the temporal scope of this thesis.

2 Theoretical Basics

In the following chapter the underlying theoretical basics for metadata based software browsing are described. The main abstract concepts of this thesis – metadata, statistics and browsing – will be brought out.

2.1 Metadata

The National Information Standards Organization defines metadata in their booklet 'Understanding metadata' (see [Nat04]) as follows:

"Metadata is structured information that describes, explains, locates, or otherwise makes it easier to retrieve, use, or manage an information resource. Metadata is often called data about data or information about information.
The term metadata is used differently in different communities. Some use it to refer to machine understandable information, while others use it only for records that describe electronic resources. In the library environment, metadata is commonly used for any formal scheme of resource description, applying to any type of object, digital or non-digital."

The three main types of metadata are:

- *Structural metadata* indicates how compound objects are put together, for example, how pages are ordered to form chapters.
- *Descriptive metadata* describes a resource for purposes such as discovery and identification. It can include elements such as title, abstract, author and keywords.
- *Administrative metadata* provides information to help manage a resource, such as when and how it was created, file type and other technical information and who can access it.

"Metadata can describe resources at any level of aggregation. It can describe a collection, a single resource, or a component part of a larger resource. Metadata can also be used for description at any level of the information model laid out in the IFLA (International Federation of Library Associations and Institutions) Functional Requirements for Bibliographic Records: work, expression, manifestation, or item. For example, a metadata record could describe a report, a particular edition of the report, or a specific copy of that edition of the report." [Nat04]

Metadata can be embedded in a digital object or it can be stored separately.

Storing metadata with the object it describes ensures that the metadata will not get lost, obviates problems of linking between data and metadata and helps to ensure that the metadata and object will be updated together. The openSUSE Build Service stores most of its metadata like titles, descriptions and timestamps embedded in the objects they belong to.

However, it is impossible to embed metadata in some types of objects. Additionally storing metadata separately can simplify the management of the metadata itself and facilitate search and retrieval. Especially when the amount of metadata belonging to an object is rather large as for example rating scores of thousands of users for a software package, it is better to store the metadata separately from the actual objects.

The openSUSE Build Service stores different kinds of metadata for its software projects. Most of these metadata is descriptive metadata like names, titles and descriptions. There is also structural metadata stored with each package and project, because software packages belong to projects and projects can belong to other projects. Furthermore the Build Service uses administrative metadata to store the creation and update timestamps for packages and projects.

Another kind of metadata that was not mentioned yet, is generated and collected statistical metadata for the projects the Build Service hosts. The data collected are ratings and download counters for software. Activity values of software packages inside the Build Service could be described as *calculated metadata*.

2.2 Statistics

A good explanation of how statistics are defined can be found at [Wik07e]:

"Statistics is a mathematical science pertaining to the collection, analysis, interpretation or explanation and presentation of data. It is applicable to a wide variety of academic disciplines, from the physical and social sciences to the humanities. Statistics are also used for making informed decisions – and misused for other reasons – in all areas of business and government.
Statistical methods can be used to summarize or describe a collection of data. This is called *descriptive statistics*. In addition, patterns in the data may be modeled in a way that accounts for randomness and uncertainty in the observations and then used to draw inferences about the process or population being studied. This is called *inferential statistics*. Both descriptive and inferential statistics comprise applied statistics. There is also a discipline called mathematical statistics, which is concerned with the theoretical basis of the subject."

The objectives of this thesis realize metadata based *descriptive statistics* for the openSUSE Build Service. They can help users to make decisions, what software might be applicable for their special needs.

2.3 Browsing

Generally the term *browsing* means to have a look-around or to forage for something.

"Browsing is the electronic equivalent of wandering the library shelves looking for anything interesting and relevant that catches the eye and following references from one item, such as a book or journal, to another. [...]
The main attraction of browsing is that it gives the user direct control over the search process, as opposed to depending on information retrieval technology. It lets you see what is going on during the search process and allows you to make choices about what information is being included and what is being rejected. In the process you learn more about the subject area and what you are really looking for. Because browsing electronically imitates what people do in libraries it feels like a more natural process and can be less of a strain than using formal search tactics." [Bro07]

In computer science browsing means: Using an user interface that allows navigation of different objects.

In our special case, users have the possibility to browse the contents of the openSUSE Build Service. The objects they browse are the software projects and packages that the openSUSE Build Service hosts and provides for downloading. This way the users are being helped in finding software that they might like for different reasons such as Linux software that is most up to date, best rated or most popular.

3 Working Environment

In this chapter I describe the working environment in which this thesis was written. I describe the most important tools that were used and give a technical overview of the openSUSE Build Service.

3.1 Company, Team and Project

In the first section of this chapter I introduce the company, the team and the project where this thesis was created.

SUSE and Novell

SUSE was founded in late 1992 as a UNIX consulting group, which among other things regularly released software packages and printed UNIX/Linux manuals.

On November 4th 2003, Novell announced it would acquire SUSE Linux and finalized this in January 2004. Novells corporate technology strategist stated that Novell would not alter the way in which SUSE continues to be developed.

Nowadays SUSE is one of the most known Linux distributors in and around Europe, providing a complete and easy to install distribution for home users, as well as a wide range of Linux based enterprise products and services.

Internal Tools Team

Inside SUSE, the Internal Tools Team is responsible for creating and maintaining software, that supports the employees of SUSE with their daily business. During the writing of this thesis I was member of the Internal Tools Team.

openSUSE Project

The openSUSE project is a worldwide community program started in 2005 sponsored by Novell that promotes the use of Linux everywhere. Everyone can get involved and help making the distribution even better, or just use the results of this project. The openSUSE project homepage can be found at [Ope07a].

The main result of the openSUSE project is the openSUSE Linux distribution that can be downloaded for free from [Ope07d].

openSUSE Build Service

The openSUSE Build Service is an open platform for complete distribution development, that provides the infrastructure for development of the future openSUSE based distributions. It provides software developers with a tool to compile, release and publish their software as packages for the broad user audience.

Because of the increasingly advancement of the Build Service it is even possible to create an own Linux distribution. In addition users can build their software for other Linux distributions, such as Debian [Deb07], Fedora [Fed07], Ubuntu [Ubu07] and more to come. Open interfaces allow external services (e.g. SourceForge), web pages

and custom clients to integrate the Build Service into their systems by the use of its ressources.

At the moment the basic build functionality is already in place to create software packages from the source code. Three different clients are available: a web client, a command line client and a GUI client to interact with the Build Service. With the clients you can create projects and packages, upload source code files and instructions how this source code has to be compiled and some descriptive metadata.

A technical overview of the Build Service is given in chapter 3.3 starting on page 22.

3.2 Used Tools and Technologies

The following section highlights the used tools and technologies to accomplish this thesis.

3.2.1 Hardware

The workstation I used to develop the programming parts of this thesis is an intel pentium 4 dual core 3 GHz machine with 1 GB of RAM.

All production systems are hosted in the SUSE server room in Nuremberg. Altogether there are about 86 CPUs running for the Build Service. Most of them compile and build the software packages. This thesis affected the Build Service *frontend* and the *webclient* server, which run on the same machine: an AMD quad CPU machine with 4 GB of RAM.

3.2.2 Vim

To edit the source code of the openSUSE Build Service, I decided not to use a complete IDE like Eclipse [Ecl07] or Emacs [Wik07c]. I preferred using the very popular and powerful editor *vi* respectively the improved newer variant *vim* to be able to pull all the strings myself. Vim is a highly configurable text editor built to enable efficient text editing. It is an improved version of the vi editor distributed with most UNIX systems. See [Wik07g] and [Vim07a] for more information.

I additionally used the project plugin from [Vim07b] for vim, that enables efficient handling of many source files in big projects. The screenshot in figure 3.1 shows the vim project plugin in action.

3.2.3 Subversion

Subversion - usually abbreviated as SVN - is a revision control system which allows computer software to be developed by a distributed group of programmers. Subversion manages the concurrent access of multiple users to the same files - and helps to keep the code consistent even if many programmers changed the same source code file. To accomplish that, every change a user commits to the SVN

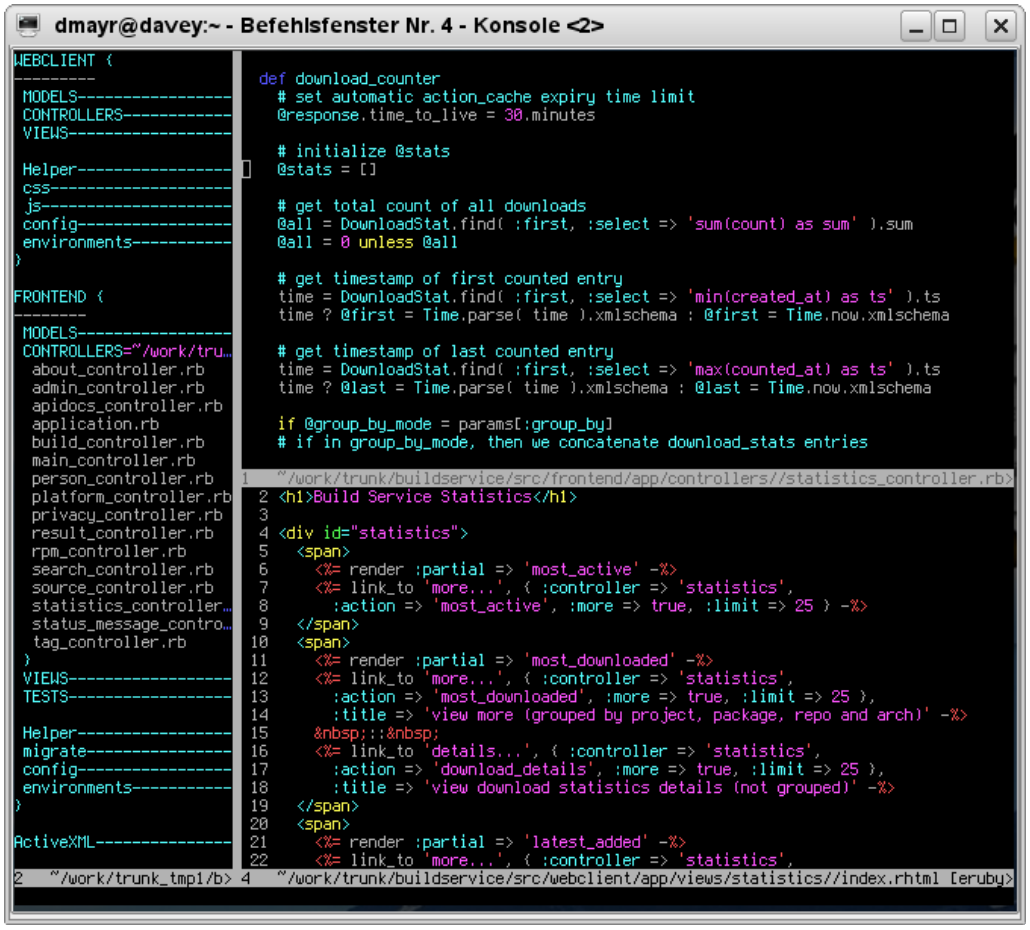


Figure 3.1: Screenshot of vim with the project plugin

repository gets an automatically increasing revision number, that identifies the change.

At SUSE SVN is used for almost all software projects. As the complete openSUSE Build Service source code is opensource, everybody can check it out from the official Novell Forge SVN server with the following command:

```
svn co https://forgesvn1.novell.com/svn/opensuse/trunk
```

More information about subversion can be found at Wikipedia on [Wik07f] and on the subversion homepage at [SVN07].

3.2.4 REST

REST is an acronym standing for Representational State Transfer. This is a term embossed in the Ph. D. dissertation by Roy Fielding [Roy00], where he describes an architectural style of networked systems.

"The web consists of resources. A resource is any item of interest. For example, the fictive website www.car-database.com may define a kaefer-1303 resource. Clients may access that resource with this URL:
`http://www.car-database.com/oldtimer/kaefer-1303`
A representation of the resource is returned (e.g. kaefer-1303.html). The representation places the client application in a specific state. The result of the client, traversing a hyperlink in kaefer-1303.html, is another resource which is accessed. The new representation places the client application into yet another state. Thus, the client application changes (transfers) the state with each resource representation.
REST uses standard HTTP operations like GET, PUT, DELETE to retrieve, upload and delete resources. It is not a standard, it is just an architectural style. But REST uses many established standards like: HTTP, URL, XML, HTML, GIF, JPG, PNG and many more."

All the Build Service API calls follow the REST addressing style for resources. The path to access a specific package of the Build Service by the webclient could be e.g. `/package/show/Apache/libapr2` where `Apache` is the project name and `libapr2` the package name.

3.2.5 MVC

MVC is an *architectural pattern* that describes how to split application code in three units: *model*, *view* and *controller*.

"Architectural patterns are software patterns that offer well-established solutions to architectural problems in software engineering. An architectural pattern expresses a fundamental structural organization schema for a software system. The schema consists of predefined subsystems and specifies their responsibilities and relations." [Arc07]

MVC was introduced by Trygve Reenskaug [Try79] and first implemented with the Smalltalk [Pet04] programming language.

> The *model* contains the "smart" domain objects that holds all the business logic and knows how to persist themselves to a database. It is responsible for the data and implements data storage and provision.
>
> The *view* is responsible for presenting data to users. In most MVC implementations the view (also called the presentation) is realized as

simple templates that are primarily responsible for inserting pre-built data e.g. in-between HTML tags.

The *controller* handles the incoming requests by manipulating the model and directing data to the view. [Rai07]

As the architectural overview of the openSUSE Build Service starting on page 23 shows, two of the three Build Service tiers are built with the Ruby on Rails application framework. Ruby on Rails follows the MVC design pattern very well and makes design and development of applications easy and clearly laid out.

3.2.6 Ajax

Ajax is an acronym for Asynchronous Javascript and XML and is a concept for data transmission between clients and servers, especially in the world wide web between web browsers and web servers. Ajax uses some well known standard techniques, to get more interactivity to web sites.

The most important advantage of Ajax is the ability to update parts of a webpage in the background without reloading the whole page. This allows web applications to react faster on user input. Additionally unnecessary server load, caused by generating and transmitting already sent parts of the web page, is avoided. This makes web applications more responsive and interactive.

In figure 3.2 you can see a standard HTTP request, as for example a simple request for a web page. The response from the web server replaces the complete page in the browser.

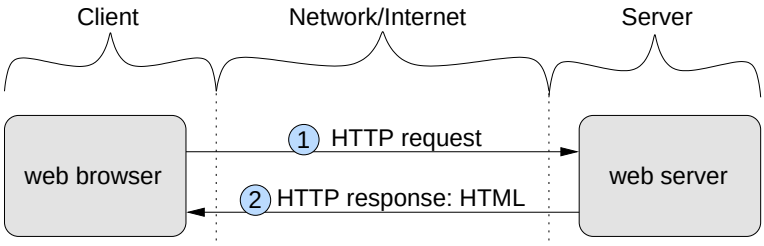


Figure 3.2: Standard HTTP request

Figure 3.3 illustrates an Ajax request, that could happen when a user clicks a button or link that is implemented as Ajax request. In step one, a JavaScript call to the browser embedded Ajax engine is made.

The Ajax engine itself is explained at [Aja07] as follows:
"Instead of loading a webpage, at the start of the session, the browser loads an Ajax engine [...]. This engine is responsible for both rendering the interface the user sees and communicating with the server on the user's behalf. The Ajax engine allows the user's interaction with the application to happen asynchronously – independent of communication with the server."

While processing step 2, the Ajax engine sends a request to the web server in the background. The web server answers the request with data usually in the XML format, but any other format works too (step 3). The browser receives the data and updates *defined parts* of the web site (inside the DOM, see glossary) by the use of JavaScript in step 4 rather than the whole page.

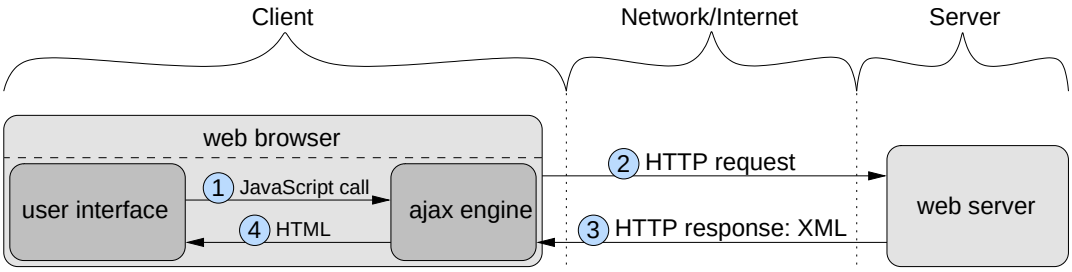


Figure 3.3: Ajax request

In this thesis I used Ajax for two reasons:

- to make the select box, where the user can choose how many statistic entries he would like to see at once (see page 49), a bit more responsive and
- at the rating stars, where the user can click to rate packages and projects, to update the stars without reloading the whole page – see figure 3.4 below.

Figure 3.4: Ajax updates rating stars and message box without reloading the whole page

3.2.7 Ruby

Ruby is a pure object oriented, single pass interpreted programming language. Its syntax was mainly inspired by Perl with Smalltalk-like object oriented features and also shares some features with Python, Lisp, Dylan and CLU. It was invented by Yukihiro Matsumoto from Japan, who started working on it in 1993 and released it to the public in 1995. Ruby was nearly unknown outside Japan until about the year 2000 because there was only japanese documentation available. Its main implementation is opensource software and thus available for free.

In Ruby everything is an object - even strings and numbers. Therefore it is possible to write things like this:

```
1 "this is a string object".length
2 # => 23
3 123.99.round
4 # => 124
```

Ruby is designed to be easy readable and quick learnable.

```
1 numbers = [ 1, 2, 3 ]
2 numbers.each do |number|
3   puts "This is line #{number}"
4 end
5 # => This is line 1
6 # => This is line 2
7 # => This is line 3
```

In Ruby all variables are dynamically typed, so the data types do not need to be declared. They are not known until execution time. The advantage of dynamic data typing is more flexibility and less work for the programmer.

Ruby was chosen as the programming language for the Build Service frontend and webclient, because of several reasons. The most important reason is that the web application framework Ruby on Rails (see section 3.2.8) is written in Ruby - the reason why Ruby on Rails was used is described later on. Furthermore Ruby is good for new developers to become acquainted with the code that is already present. Ruby also has other important features: dynamic created methods at runtime, instance methods and variables that only exist for only one instance of a class, rather than for all class instances that allow to implement things very efficient.

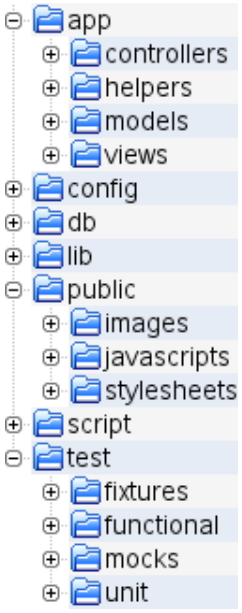
3.2.8 Ruby on Rails

Ruby on Rails - often called only Rails or RoR - is a sophisticated opensource web application framework released in 2004. Its main scopes are database driven web sites and web services, but it can also be used without a database.

Rails uses the powerful MVC architecture, explained in chapter 3.2.5 on page 13. This makes web development agile, easy, fast and clear. Figure 3.6 shows how the Rails components work together.

Besides the *don't repeat yourself* (DRY) principle, where Ruby on Rails tries to help avoiding repeated code fragments and helps re-using code, it follows the *convention over configuration* precept. This means there are meaningful defaults for most settings and the developer only needs to specify unconventional aspects of his application. The first visible convention is the predefined directory structure of a Rails application, outlined in figure 3.5.

Figure 3.5: Rails dir structure



The main application code resides in the subdirectory `app` which is splitted into directories for the *models*, *controllers* and *views*, following the MVC design pattern explained in section 3.2.5. The `config` directory contains settings for the database connection and other things like activeXML explained in section 3.3.3. The `public` directory contains static data like images, stylesheets and JavaScript code. The `test` directory contains code and *fixtures* for the testing framework explained in section 5.3 on page 67.

Ruby on Rails was chosen to be the application framework for the Build Service frontend and webclient, because it leads to suitable results rather quickly. The very first prototypes of the Build Service were built in less than two weeks, but nevertheless the code is very readable and easy to extend due to the well-elaborated concepts of Ruby on Rails.

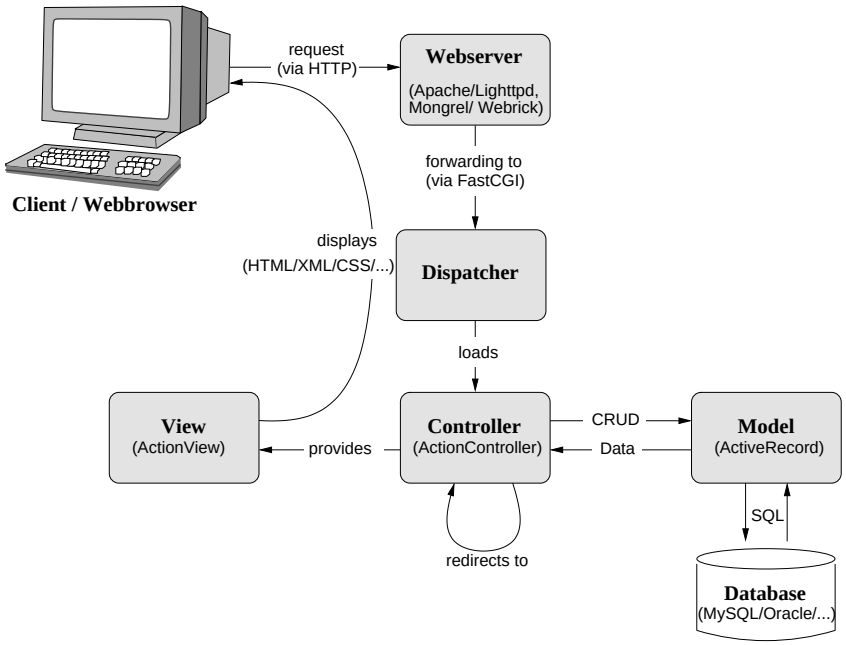


Figure 3.6: Ruby on Rails architecture

ActionPack - Controller and View Classes

The Action Pack documentation shows the following explanation what Action Pack is and what it is good for.

> "Action Pack splits the response to a web request into a *controller* part (performing the logic) and a *view* part (rendering a template). This two-step approach is known as an action, which will normally create, read, update, or delete (CRUD) some sort of *model* part before choosing either to render a template or redirecting to another action.
>
> Action Pack implements these actions as public methods on Action Controllers and uses Action Views to implement the template rendering.
>
> Action Controllers are then responsible for handling all the actions relating to a certain part of an application. This grouping usually consists of actions for lists and for CRUDs revolving around some model objects.
>
> Action View templates are written using embedded Ruby in tags mixed with HTML. To avoid cluttering the templates with code, a bunch of helper classes provide common behavior for forms, dates and strings. And it is easy to add specific helpers to keep the separation as the application evolves." [Act07b]

A short example should make it more clear – first the controller class, stored in a file named `app/controllers/message_controller.rb` :

```
1 class MessageController < ApplicationController
2   def show
3     @messages = [ 'message1', 'message2', 'message3' ]
4   end
5 end
```

Here is the view template that uses the prepared data from the controller, stored in a file named `app/view/message/show.rhtml` :

```
1 <html>
2 <head>
3   <title>Message overview</title>
4 </head>
5 <body>
6   <div>
7     <% @messages.each do |message| %>
8       <p><%= message %></p>
9     <% end %>
10  </div>
11 </body>
```

So when the client web browser requests the URL http://servername/messages/show it gets three messages displayed.

ActiveRecord - the Model Class

"Active Record connects business objects and database tables to create a persistable domain model where logic and data are presented in one wrapping. It's an implementation of the object relational mapping (ORM) pattern [...]." [Rai07]

ORM specifies the mapping of object-oriented data to relational data and vice versa. It maps Objects with their attributes and relations to table structures and columns inside databases and the other way round.

With the ActiveRecord class, it is possible to access database entries in a very easy object oriented way, without the need to worry about how the connection and updates to the database are made.

Assume you have two database tables like the ones in table 3.1 and table 3.2. Furthermore assume ActiveRecord classes like the ones in listing 3.1.

Then it is possible to do things like in listing 3.2.

id	company	name	first_name	street	city
1	LunaBOX	Mayr	David	Rollnerstrasse	Nürnberg
2	Minimag	Peter	Andreas	Ottisrieder Strasse	Haldenwang
3	NTA	Gruebler	Gerald	Seidenstrasse	Isny

Table 3.1: Customers example table

id	customer_id	date	amount
100	1	2006-01-13 09:08:52	12.18
101	1	2006-03-04 11:05:34	31.73
102	2	2006-12-26 13:15:58	23.43
103	2	2007-02-03 23:35:32	12.21
104	3	2007-03-17 17:28:52	25.10

Table 3.2: Bills example table

Listing 3.1: Simple ActiveRecord Model example

```
1 class Customer < ActiveRecord::Base
2   has_many :bills
3 end
4
5 class Bill < ActiveRecord::Base
6   belongs_to :customer
7 end
```

Listing 3.2: Access ActiveRecord Model Data example

```
1 lunabox = Customer.find_by_company 'LunaBOX'
2 lunabox.bills.each do |bill|
3   output  = bill.customer.company + ' ' + bill.customer.name + ' -> '
4   output += bill.date + ' - ' + bill.amount + ' EUR'
5   puts output
6 end
7 lunabox.street = 'Rollnerstrasse'
8 lunabox.city = 'Nuernberg'
9 lunabox.save
```

In line one a local object `lunabox` is gained from the dynamic method `find_by_company` of the `Customer` model. The method `find_by_company` is not defined in the ActiveRecord class, it is created at runtime by ActiveRecord because of the existing database table field `customer`. The statement `lunabox.bills` in line two returns an array of bills from the previous customer object. The lines up to number six iterate through this array of bills and display the following lines:

```
LunaBOX Mayr -> Fri Jan 13 09:08:52 0100 2006 - 12.18 EUR
LunaBOX Mayr -> Sat Mar 04 11:05:34 0100 2006 - 31.73 EUR
```

Line seven through nine change the street and city values of the selected customer and and saves it back to the database.

3.2.9 MySQL

MySQL is a wide spread relational opensource database management system (DBMS, see A.4 in the appendix). The choice for MySQL as data storage system for the Build Service frontend was already made before this thesis started, so its usage was obligatory.

3.3 Build Service – A Technical Overview

The basic idea of the openSUSE Build Service was already introduced in section 3.1 on page 9. In this section I give an overview of the openSUSE Build Service and how it works technically after clarifying some often used terms in this context.

3.3.1 Disambiguation

To avoid confusion and misunderstandings, I clarify some terms associated with the openSUSE Build Service.

3.3.1.1 Packages

Packages are the central content of the openSUSE Build Service. A package contains metadata like a name, title and description. Furthermore there are files belonging to a package. The most essential file of a package is the source code archive that should be built in the Build Service.

Depending on the type of binary package that should be built from the respective source code, there needs to be a special file that describes the requirements for building a binary software package. In case the binary package (see section 3.3.1.3) should be an *RPM* (see glossary), this special file is called *spec file*.

3.3.1.2 Projects

Projects within the openSUSE Build Service are containers for software *packages*, binary packages (see below) and other projects. Each project contains some descriptive metadata like a name, title and description and normally group packages of the same kind. A project can be seen as a workspace used to manage packages and build configuration. For example there are projects like `server:ftp` that contain packages for ftp-servers, `games:action` which is the home of all action game packages or the project Apache for packages related to the wide spread web server Apache. In every project at least one developer / packager is involved who maintains it.

3.3.1.3 Binary Packages

After the Build Service processed a software package, the result is a binary package that could be easily installed in a Linux system.

A binary package file contains the compiled binaries, config files, install instructions and maybe some scripts to get the software installed on a particular Linux distribution.

For the openSUSE Linux distribution these binary packages are called RPM packages and have the file name extension '.rpm'. The only other supported binary package type in the Build Service is DEB at the moment, primarily used for Debian Linux and compatible distributions like Ubuntu.

3.3.1.4 Repositories

A repository has two different, but related meanings in the Build Service:

A repository is

- a defined set of binary packages *used to build other* packages and
- a collection of binary packages as the result of a built project that can be added as an *installation source* for a Linux system.

3.3.2 Architecture

The openSUSE Build Service consists of three tiers: the *backend*, the *frontend* and the different *clients*.

The actual package building is done within the *backend*. On top of the backend a *frontend* is exposing the build service functionality as a web service (see A.3 in the appendix) to the public by an API. Several *clients* using the *frontend* API provide the user interface to the end-user. A mirror interface provides the built packages for download, mirroring and interfaces with installation tools.

This architecture makes the Build Service scale very well. If the load grows higher than the existing machines can handle, it is easy to add additional computing power to the Build Service.

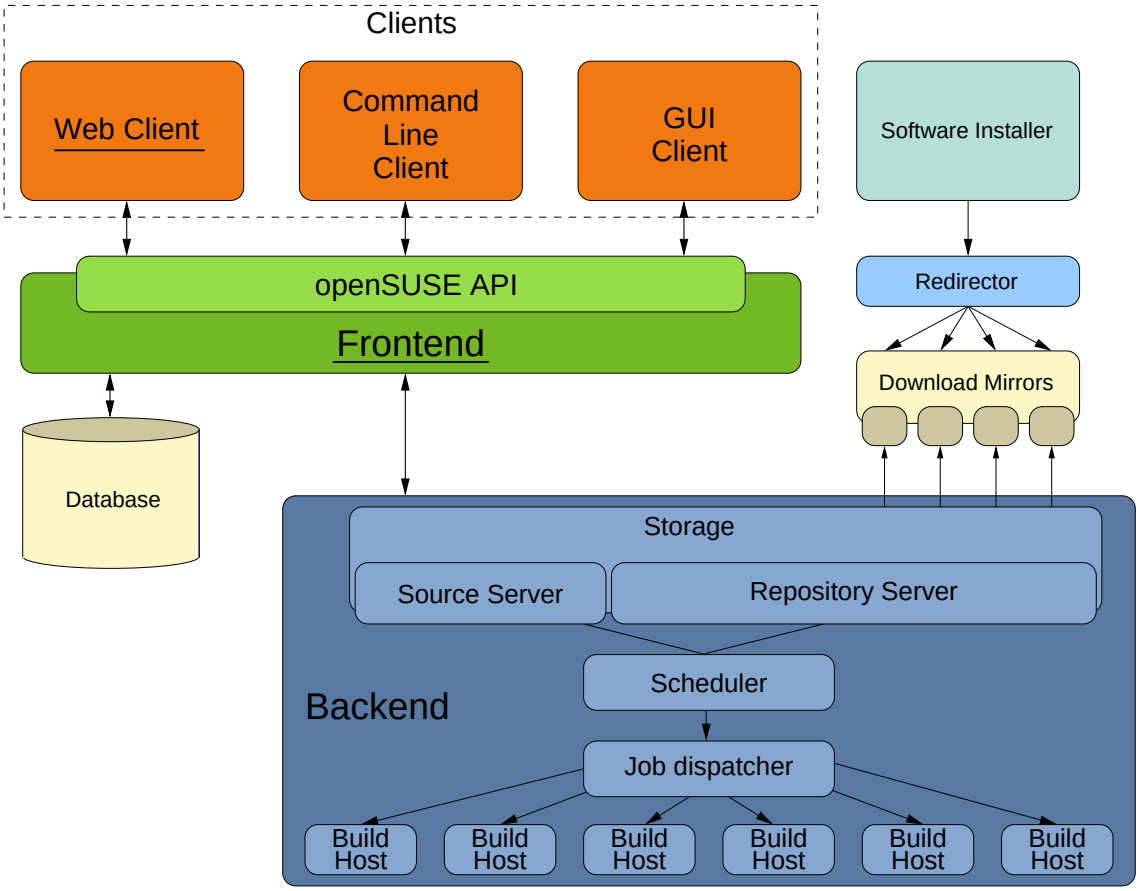


Figure 3.7: Build Service - Architectural Overview

Another big advantage of this architecture is the flexibility for the clients. At the moment three clients exist: a webclient for users who do not want to install any software beside their web browser, a command line client to operate the Build Service without graphical user interface and a graphical client that supports all the features that a locally running client can offer like fast sorting on client side.

Additionally the Build Service gains higher security from this three tier architecture. All commands and data from the clients have to go through the Build Service frontend, which requires users to be authenticated. All XML data is validated by the frontend, to guarantee that it conforms to the rules of the XML schemas. This way the backend can be sure it never gets corrupted data.

Another advantage of the three separated tiers is, that the Build Service is better maintainable. Every tier can be developed independent of the others, as long as the interfaces are clear and only changed in cooperation with the

other developers. Figure 3.7 shows an overview of the Build Service components.

3.3.2.1 Backend

The build backend is responsible for the actual building of packages. It acts on the source data uploaded through the frontend and creates packages according to the user-controlled build configuration.

Building packages is a complex process that goes far beyond just compiling the source code. It includes the creation of a safe environment for the packages to be built. As basis to be able to compile the packages all required dependencies are fulfilled. Another task is the decompression of the source code. In case there are patches available for this specific package, they are applied to the source code files. After the compilation procedure itself the corresponding files of the package will be picked up and bundled to binary a RPM or DEB package.

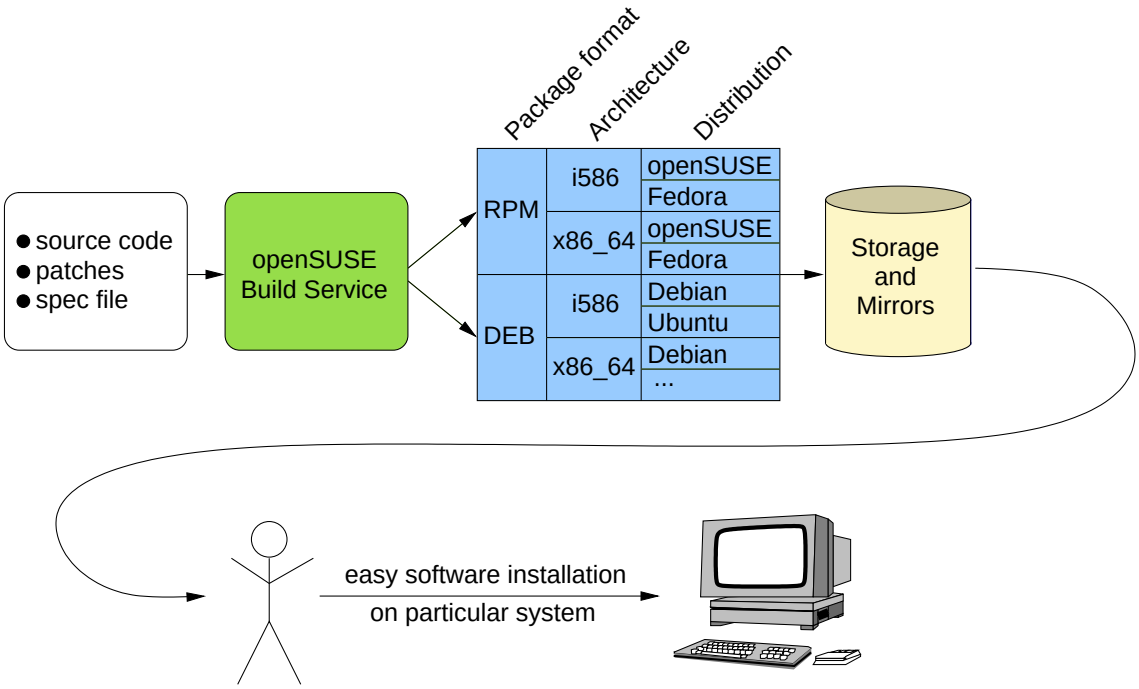


Figure 3.8: Build Service - overview from source code to end-user

Build processes are distributed over a set of build hosts to allow efficient building of large amounts of packages in parallel. A scheduler controls the distribution of the build jobs and of the build results to the storage subsystem, where they are made available through the mirror interface. The mirror interface allows other file servers (mirrors), which are spread all over the world to provide the downloadable

files for the users in reasonable speed, to get the files from the Build Service efficiently.

The backend is accessed by a similar REST-based API as the frontend. This API is not public, though. It is only directly accessible by the frontend and the mirror interface.

The Build Service backend contains also storage management for package data. The backend storage holds all data related to packages involved in building the packages. These are the sources for packages consisting of source code tarballs, patches and spec files, which act as input for the build process and the results of the build, i.e. the source and binary RPMs. The storage provides versioning and history for all uploaded data.

3.3.2.2 Frontend / API

The frontend, that implements the Build Service API, encapsulates the Build Service functions and provides an interface to them.

The frontend provides access to the package data within the backend through a REST-based HTTP interface. This allows to do load-balancing, mirroring of data, using proxies etc. in a simple way. Uploaded data always has to pass the frontend. Downloads can be redirected to other servers through the mirror interface without using the backend.

The frontend processes, prepares and routes data and requests from the user to the backend. The interface is provided as web service in the form of the openSUSE Build Service API accessible at [Ope07b] and documented at [Ope07c]. The user interfaces like the web client and the command line client access the Build Service through this API.

The API is implemented as REST-style web service based on XML resources. It uses standard HTTP operations like GET, PUT, DELETE to retrieve, upload and delete resources. These resources can be: source files, package files, meta information, control- or status information each in form of XML data.
POST operations are used to send commands to the frontend: triggering a rebuild, creation of templates for specfiles or similar operations which aren't represented as data files. By using standard HTTP operations a large collection of existing programming languages, frameworks and tools can be used to access the API in a simple way.

Access to the API requires authentication by the user. For this purpose all accesses have to go through a Novell iChain proxy (see [iCh07]) which checks the credentials of the user and makes sure that only requests of authenticated users are forwarded to the frontend accompanied by the required information about the user.

The frontend also maintains metadata of projects and packages as a set of XML files. These metadata describes the content and structure of the software to be built. This includes information like titles, descriptions and classifying tags visible to the end-user. Furthermore it includes technical data how to build the packages in a project. Target platform, used package repositories and other information that needs to be defined for the build environment of the software is also stored with the technical metadata. The frontend provides direct access to the data mentioned above and as well as the possibility to query data on basis of names, tags or other attributes.

For monitoring and controlling the build process, the frontend provides extensive status information about the state of package building and logs of the individual build jobs. Status information from build processes of different packages and targets are aggregated into project status information.

To ensure data consistency and security the frontend imposes some restrictions on the uploaded data. It checks XML schema validity (see A.1), user permissions and in the future also will apply user quotas.

The frontend is implemented using the Ruby on Rails framework. For access to the backend it uses a component called ActiveXML, which maps HTTP requests to XML resources to Ruby objects transparently.

3.3.2.3 Clients

The public API makes it possible to create specific clients to control the Build Service and provide a convenient interface to the end-user. Currently there are three clients, developed by SUSE:

webclient is the primary client of the Build Service, implemented as a Ruby on Rails web application. For a screenshot of the start page see figure 3.9. The webclient can be accessed with any browser at [Bui07]. This is the client affected by the work of this thesis. The webclient uses the ActiveXML framework to access the Build Service through the openSUSE frontend API. Functionality

to browse and view projects and packages is provided as well as an interface that manages source files and controls the build process. This includes status views and a live view of build logs.

osc is the openSUSE command-line tool, which allows using the Build Service without a graphical interface. Osc is written in python and used by many old hand developers and package maintainers because it makes interaction with the Build Service faster if you know exactly what you want. It is possible to check out sources and commit changes back to the server. The command line client is also able to perform builds of packages on the local system rather than in the Build Service, e.g. to test the setup on the local architecture before building it for all available architectures in the Build Service. It retrieves binary packages from the server required for building because of the need to resolve the package dependencies for the build process. The retrieved packages are installed to a local *chroot* environment in which the build process is started.

rich client is a graphical Build Service client, written in C++ and with the help of *Qt* (see [QtT07]). Its development lacks a bit compared to the other two clients. Currently viewing of project listing, project metada and some data about their packages works. For screenshots see [Ric07].

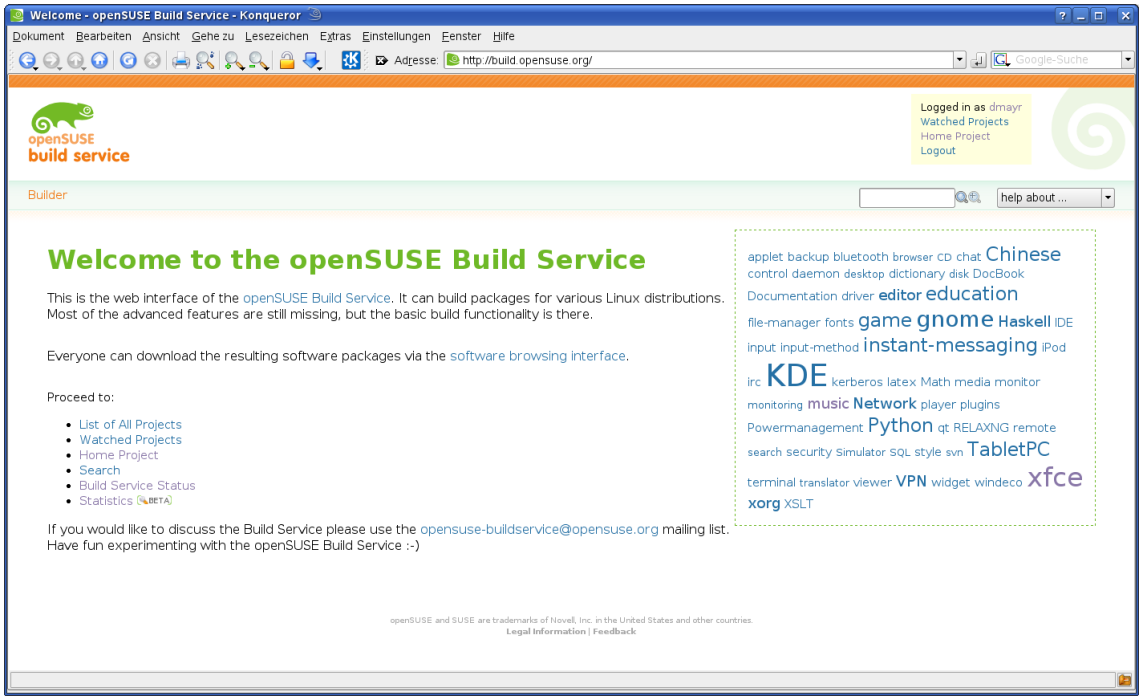


Figure 3.9: Build Service webclient startpage

3.3.3 ActiveXML

The clients and the Build Service API / frontend communicate via XML (see A.1) over a REST interface. To ease this communication, especially between the webclient and the frontend, the Build Service team developed a library called ActiveXML.

The main goal of ActiveXML is to provide a mapping between ruby objects and XML data similar to the object relational mapping of ActiveRecord as explained on page 19. In addition to the XML-to-object mapping, ActiveXML manages the HTTP connection between the webclient and the frontend by generating the appropriate URL for accessing the requested data.

ActiveXML provides an easy way to navigate through an XML document and access nested data by dynamically defining accessor methods for elements and attributes. Attribute accessors return the attribute string value, element accessors return an ActiveXML::Node object for the chosen element. This allows accessor calls to be chained as shown in the following example.

```
<project name="superkde">
  <title>SuperKDE</title>
  <description>SuperKDE is a heavily tuned version of KDE.</description>
  <person role="maintainer" userid="ernie"/>
  <repository name="kde4:factory">
    <path project="kde4" repository="factory"/>
    <arch>x86_64</arch>
    <arch>i386</arch>
    <arch>ppc</arch>
  </repository>
</project>
```

```
# assume that the preceding XML file is stored as string in the variable xmldata

axml_object = ActiveXML::Node.new( xmldata )
puts activeXML.name
# => "superkde"

puts activeXML.repository.name
# => "kde4:factory"

puts activeXML.repository.path.project
# => "kde4"

activeXML.repository.each_arch do |arch|
  puts arch
end
# => "x86_64"
# => "i386"
# => "ppc"
```

4 Build Service Browsing Interface

This chapter essentially is the main part of this thesis. First I will specify the *use cases* that describe what the result of this thesis should be. After that, the *design* of the metadata and statistics based software browsing system will be presented, followed by the *implementation* section, where I will describe how this was realized.

4.1 Specification

This section specifies the requirements of the thesis. After describing the initial situation of the openSUSE Build Service and the motivation why the enhancement through this thesis was necessary, I will highlight the objectives of this thesis. Afterwards use case analyses were made for all of these five objectives.

4.1.1 Initial Situation

When this diploma thesis was started, the Build Service could already be used to build software packages and had about 250 registered users and 5000 packages in about 400 projects.

To get an overview of all projects the only possible way was to go through the list of all 400 project names, which was very uncomfortable. In each project the user had to click on every single package in order to get the respective package information.

4.1.2 Motivation

The motivation of this thesis was to implement a user-friendly and easy to use interface for Build Service users to discover, browse and contribute to the most interesting software projects in a convenient way.

4.1.3 Objectives

Subject of this diploma thesis is to enhance the Build Service with browsing capabilities, so that any interested user or developer can find interesting software in the Build Service fitting to his needs. Also software project management participates from this improvement. In the following subsections, the desired objectives that should be browseable by the users are explained.

Newest Packages and Projects

A list of newly added packages and projects to the Build Service will give a quick overview of what is new and worth to have a look at. This helps users to recognize new packages or projects without browsing through the list of all of them.

Latest Updated Packages and Projects

Especially for users that always want to have the newest packages, it could be interesting to see what the latest updated packages are in a quick overview. If they find newer versions of packages they have already installed, they can decide to update.

Package and Project Rating

Users should get a possibility to rate packages and projects at a scale between one and five. The rating in general should be as simple as possible. Every package and project has a row of five stars next to its title, where a user can click the star which he thinks the package/project merits.

To avoid abuse of the rating system, a single user should not be allowed to rate the same package or project more than once. A user who rated a Build Service

item should be able to change the score afterwards. Only authenticated users are allowed to rate items. The actual rating score for an object should be displayed by the clickable stars itself.

Download Statistics of Packages and Projects

It is an interesting and useful information to know which are the most downloaded packages. Administrators and package maintainers of the Build Service benefit from download statistics because they can see how strong the demand for their packages is. Also for the users this data is interesting because they can see what are the most requested packages. This could support them in getting aware of new and interesting software. The results are also useful for project planning purposes in future.

Users should have the possibility to view download counters per project, per package, per repository or over the whole Build Service.

Because there is a central server where all software download requests are redirected to the geographically next located mirror, download counters can be collected easily. This redirector service is explained in detail in section 4.2.5.1 on page 44.

Activity Statistics of Packages and Projects

Another aspect of interest is the activity of a package or project in general to figure out which ones are the most active. This way maintainers could be encouraged to work on their packages and keep them up to date. Activity can be measured by taking the information on how often the package got updated during a specified range of time. Therefore it is necessary to conceive an algorithm for calculating a value for activity.

4.1.4 Use Case Analyses

To give a quick overview of the functional requirements of the planned features, use case analyses were made for all objectives and their results are shown in the following section.

4.1.4.1 Latest Added / Updated Packages

Because of the fact that the use cases for latest added and latest updated packages and projects are similar, they are subsumed here in one use case which can be seen in figure 4.1.

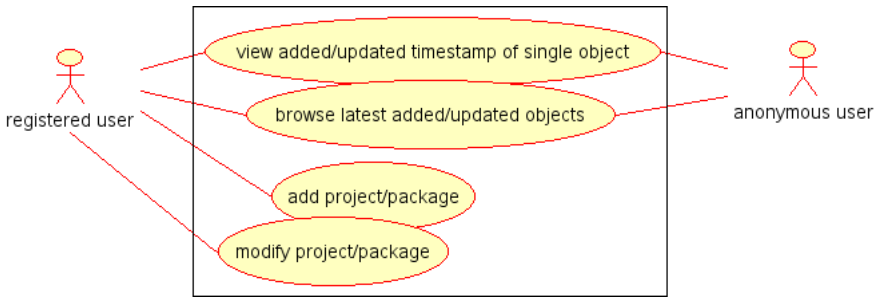


Figure 4.1: Use case analysis for latest added/updated projects/packages

View Added / Updated Timestamp of a Single Object

Every project and package page should display the timestamp when it was added and when it was updated last, so the user can see how old it is and when it was touched last in the Build Service.

Browse Latest Added / Updated Objects

The list of packages and projects should be browseable sorted by creation and modification timestamp. The length of list ought to be adjustable, so users can choose whether they want to see the top10 or e.g. top1000 of the latest updated packages.

Add Project / Package

Registered users can add projects and packages to the Build Service. Whenever a project or package is added, the creation timestamp of it must be set accordingly to the actual time and date.

Modify Project / Package

When a project or package is modified, the modification timestamp needs to be updated. The design chapter starting on page 38 explains what *modifying* a package or project means in detail.

4.1.4.2 Package and Project Rating

Figure 4.2 shows the use cases for rating of Build Service objects - projects and packages.

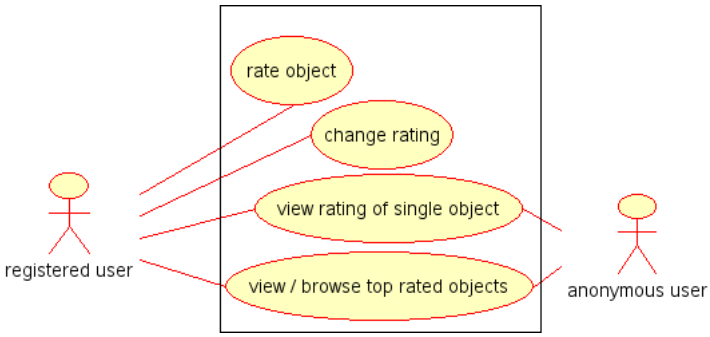


Figure 4.2: Use case analysis for rating

Rate Object

Every registered user should be able to rate a project or package. This requires a login name, a password and the user must be logged in. When he views a package or project, he simply should be able to click onto a row of five stars to rate the object.

Change Rating

If the user already rated an object, he should have the possibility to change the rating afterwards. Therefore it would be useful that the user can see his previous rating score somewhere nearby the rating stars.

View Rating of a Single Object

The user should see the actual average rating score as well as his own score at any point. The score should be visible numerical as well as embedded in

the rating stars figure. This should be accessible for all users – the authenticated and anonymous users, that have no credentials to log in to the Build Service.

Browse Top Rated Objects

All users should be able to see a list of the highest rated objects of the Build Service. The length of the list should be adjustable so you can see the top 10 or the top 200 objects if one wishes to. Packages and projects with less than three votes should not show up in this list. This should avoid objects with just one rating of five stars to lead the top of the list. Such few ratings are considered to be not very representative.

4.1.4.3 Most Downloaded Packages

Figure 4.3 shows the use cases for the most downloaded statistics in the Build Service.

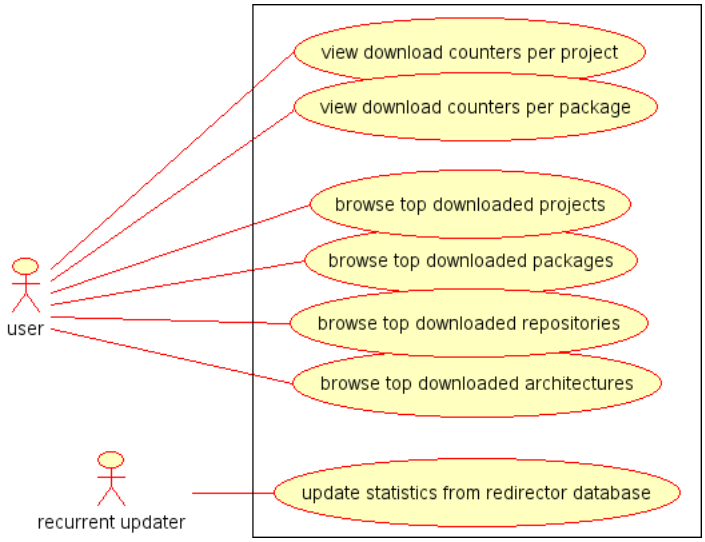


Figure 4.3: Use case analysis for download statistics

View Download Counters per Project and Package

If any user - logged in or not - visits a page with information for a project, he should see the accumulated download counters of all downloadable files that belong to all packages of this project.

It also should be possible to view detailed statistics for all packages of this project and all the files that have ever been downloaded from this project.

The same requirements apply to packages and their overview pages.

Browse Top Downloaded Projects, Packages, Repositories and Architectures

A very interesting aspect for the package maintainers in the Build Service is: how often was the own package downloaded? Thus all users should see the download counters sorted by download hits in a listing inside the Build Service webclient.

Update Statistics from Redirector Database

To have download counters always up to date, this needs to be automated. The downloads are counted per file by the openSUSE download redirector.

4.1.4.4 Most Active Packages / Projects

This section describes the use case for the most active objects in the openSUSE Build Service.

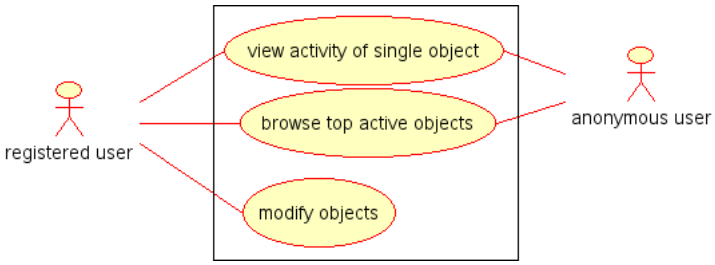


Figure 4.4: Use case analysis for activity of Build Service objects

View Activity of Single Object

On a page that is accessible for every user - logged in or not - it should be evident how active the shown object on that page is.

Browse Top Active Objects

Users should be able to browse the list of most active packages and projects. They ought to have the opportunity to decide how many of the top active objects they want to see.

Modify Objects

The term activity in this context is defined in section 4.2.4 on page 42. Activity of an object rises when it is modified. Modifications of a package involves:

- Changing package metadata like title, description or the upstream URL.
- Adding or changing source code or other files.
- Modifying the list of involved users/maintainers.
- Changing the list of repositories for which this package is built.

The activity of a project is the calculated by the average activity of all the packages that are in this project.

4.1.5 Technical Preconditions

Because the relevant parts of the Build Service for this thesis - the frontend and the webclient - are realized with the web application framework Ruby on Rails, it was implied to use it. This precondition comprehends the use of the more and more popular object oriented scripting language Ruby.

The same applies to the relational database management system MySQL, which was already used for data storage in the Build Service frontend as this thesis started.

For data exchange between the Build Service frontend and the webclient XML was obligatory.

4.2 Design

Given below is the section where the mentioned objectives of this thesis are designed and elaborated for the implementation that follows afterwards.

4.2.1 Newest Packages and Projects

To inform all users of the Build Service of new projects and packages, the latest added ones should be listed cumulated. Hence it is necessary to carry along the timestamp when the project/package was created.

Since all projects and packages are stored in a database, the tables have to be extended by one field named `created_at` with the type `datetime`. This way Rails sets the timestamp of the creation time to the current date and time when a new project or package is created fully automatically. This is one of the Rails specific features that results from the 'convention over configuration' principle mentioned earlier.

All that has to be made sure on the frontend side, is to make the timestamps available to the requesting clients in two forms:

- a single timestamp for one project or package
- a definable number of timestamps for projects or packages as a list

For this two separate actions `added_timestamp` and `latest_added` are needed, which are explained in table 4.1. The field labeled *xsd* refers to the listing section in the appendix starting on page 82, where all XML schema files are listed.

method	service	API path	xsd
GET	list of packages and projects latest added sorted by creation time	/statistics/latest_added/?limit=<limit>	B.3
GET	single timestamp when project or package was added	/statistics/added_timestamp/<project>/<package>	B.2

Table 4.1: Frontend interface for latest added projects/packages

For the `latest_added` function it should also be able to pass a limit parameter, to have the possibility to limit the count of results.

Figure 4.5 demonstrates the communication steps between the webclient and the frontend while getting the latest added objects.

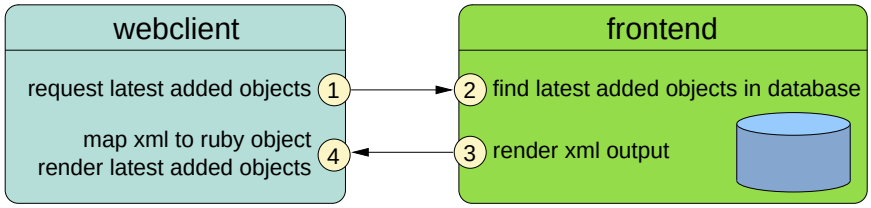


Figure 4.5: Communication between webclient and frontend while getting the list of latest added objects

In step one the webclient requests the list of latest added objects by accessing the frontend server path `/statistics/latest_added` . The frontend asks the database for an ordered list of the top newest objects in step two and sends this list in XML format back to the webclient in step three. The webclient maps the received XML data to ruby objects and renders the HTML page with the data.

4.2.2 Latest Updated Packages and Projects

To be able to show the latest updated projects and packages, an update timestamp also needs to be stored for projects and packages. The necessary steps to reach this are nearly identical to those described in section 4.2.1. The main differences are:

- the name of the database field name the must be `updated_at`
- the timestamp must be updated whenever the project/package changes:
  - the metadata (name, title, description) is changed
  - any file belonging to the package is removed, added or changed
  - the list of repositories it should be built for is changed

The frontend Build Service API needs new functions that are mentioned in table 4.2. The `latest_updated` function should also accept a limit parameter, to be able to limit the count of results. The communication cycle between the Build Service frontend and the webclient is basically the same as shown in figure 4.5.

method	service	API path	xsd
GET	list of packages and projects latest updated sorted by update time	/statistics/latest_updated/?limit=<limit>	B.5
GET	single timestamp when project or package was updated	/statistics/updated_timestamp/<project>/<package>	B.4

Table 4.2: Frontend interface for latest updated projects/packages

4.2.3 Package and Project Rating

To have some common feedback of the popularity of projects and packages, all registered Build Service users should be able to rate them within a range from one to five.

The user should be able to click on one of the five stars in figure 4.6 to rate a project or package.

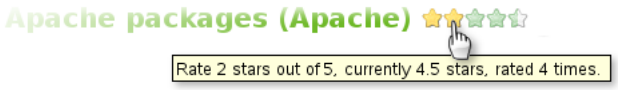


Figure 4.6: Rating stars next to the project header

Table 4.3 shows the API calls, that should be possible to get and set the ratings of projects and packages.

method	service	API path	xsd
GET	list of highest rated projects and packages sorted by score	/statistics/highest_rated?limit=<limit>	B.7
GET	rating of a specific project or package	/statistics/rating/<project>/<package>	B.6
PUT	rate a project or package	/statistics/rating/<project>/<package>	B.6

Table 4.3: Frontend interface for highest rated projects/packages

Because the user should be able to rate projects the same way as packages, we need a rating model that links to *both* other models - projects and packages - with *one* identifier. To achieve that, it is not sufficient to just carry along the referenced object id, it is also necessary to save the type of the referenced class. In this case we save the ID of an object as the attribute `object_id` and the type of an object as `object_type`. The object_type can be *package* or *project*.

Rails has already built in functions for this case: polymorphic associations (see [Pol07]). Polymorphic associations allow associations from one model to *multiple*

other model classes, or in other words they allow easier programming for n-to-many relationships.

Figure 4.7 shows the relation between the involved classes. The database table schema of the ratings table can be found in the appendix at B.13 on page 92.

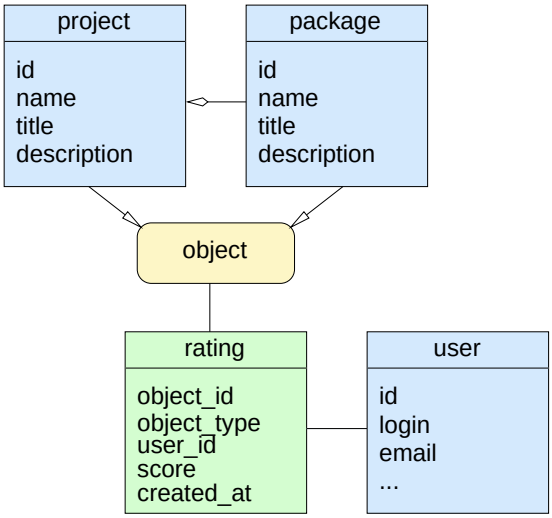


Figure 4.7: Simplified class diagram of involved classes for rating

The sequence diagram for the interaction of the involved parts while changing a rating can be seen in figure 4.8.

When the user calls a page in the webclient that displays a project or package, it requests the actual rating from the frontend via a HTTP GET request. The frontend looks up the average rating score of all ratings for the specified project or package in the database and returns it as an XML snippet.

In addition to the average rating score, it is also necessary to transmit the count of ratings and the score the user gave, if he already rated the same object earlier. This way that information can be displayed in the webclient too.

The webclient takes the score and displays it next to the header of the page. When the user clicks on the rating stars to express his assessment of the package or project, the webclient creates an XML snippet that contains the clicked score and sends it as HTTP PUT request to the frontend, which in turn saves the score in the database. After that the frontend gives the new score back to the webclient which displays the updated average score.

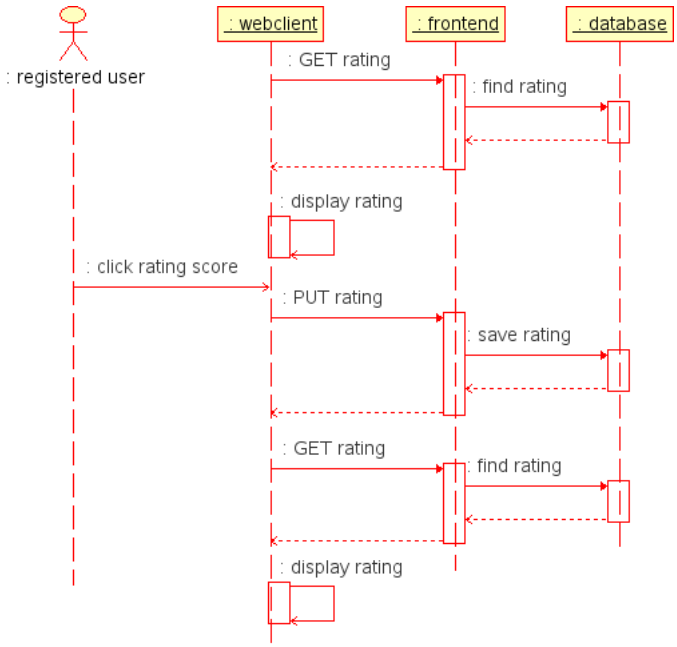


Figure 4.8: Sequence diagram for rating

4.2.4 Activity Statistics of Packages and Projects

To calculate the activity of a package, I assume the following:

- After creating a package, its activity is 100%.
- While time passes by without further actions regarding the package, activity decreases.
- Every time the package is modified, the activity of the package increases by a certain value, depending on how often the package was already updated.

Therefore we need to save the current value of activity at the moment of updating a package. I named this value activity_index. The actual activity value at any point later will be calculated ongoing, because the activity decreases permanently.

The formula to calculate the current activity is:

$$activity = activity\_index - \frac{days\_since\_update^{1.55}}{10}$$

This formula was ascertained empirical.

This results in a graph like the one visible in figure 4.9. It shows the devolution of activity exemplary for a package of the Build Service. The x-axis shows the time

in days and the y-axis the activity in percent. The green vertical lines visualize the points in time where the package was modified.

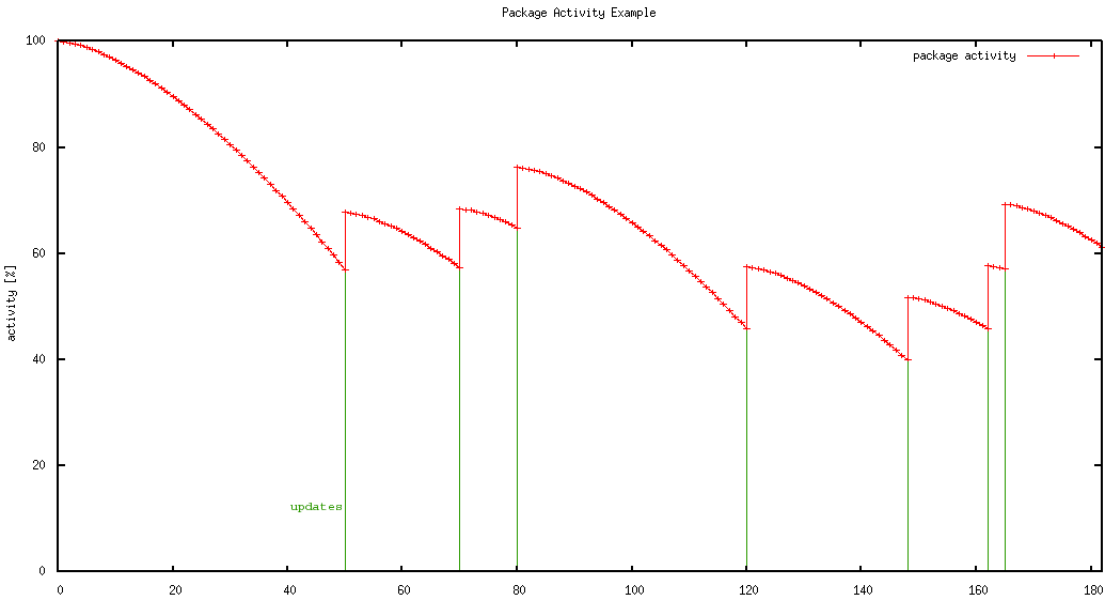


Figure 4.9: Typical gradient of package activity over approx. half a year

In table 4.4 you can see the API calls that are possible to ask the frontend for the most active objects of the Build Service. The `type` parameter can be either `packages` or `projects`. This way it can be chosen to get the list of the most active packages or the list of the most active projects.

method	service	API path	xsd
GET	list of most active packages or projects	/statistics/most_active?type=<type>&limit=<limit>	B.9
GET	activity in % of project or package	/statistics/activity/<project>/<package>	B.8

Table 4.4: Frontend interface for most active projects/packages

4.2.5 Download Statistics of Packages and Projects

The 'most downloaded packages and projects' of the Build Service is the only one of the five objectives, that receives its data from another software project outside the openSUSE Build Service: the openSUSE download redirector.

4.2.5.1 The openSUSE Download Redirector

To be able to serve all the software packages to the interested users with reasonable speed, the openSUSE download area has many download mirrors. Because we do not want that users have to select the mirrors manually, the openSUSE download redirector was written. The download redirector takes care of redirecting download requests transparently to a mirror geographically nearby, that hosts the requested file. This way, users only need to know one download URL for Build Service packages: `http://software.opensuse.org`. As a side effect, the redirector knows all files, that were requested by people who want to download. This information is used to feed a database table called redirect_stats (for schema see B.1).

As it is very important for the download redirector to be fast and reliable, the redirector has its own database where it stores the download statistics. Table 4.5 shows some example entries of this statistics table. Not using the same database as the Build Service frontend makes the redirector independent from the Build Service.

id	project	package	repository	arch	filename	filetype	version	release	count
17	Apache	apache2	openSUSE_10.2	i586	apache2	rpm	2.6.5	144.1	432
18	KDE4	kdelibs	openSUSE_10.2	i586	kdelibs	rpm	4.0.1	21.3	5379
19	ruby	FastCGI	SUSE_Linux_10.0	x86_64	FastCGI	rpm	2.4.0	3.1	143

Table 4.5: openSUSE download redirector statistics table excerpt

The download redirector parses the filenames and paths of all downloaded files and extracts the following information:

- project name
- package name
- repository name
- architecture
- base part of the filename
- version of the package
- release number of the package
- filename extension

This information is used to feed the database table `redirect_stats` every time a file is downloaded and the `count` field is increased by one. Because package names

can not be reliably extracted from the file and path name, the initial database entry with a count of zero is created by the backend, as the backend knows the full information of every downloadable file. An example path and filename that the redirector sees could look like figure 4.10.

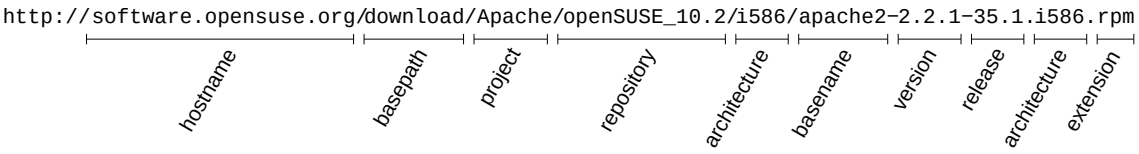


Figure 4.10: Download URL parts as parsed by the statistics module of the redirector

4.2.5.2 Transmission of Download Counters

To transfer the redirector counters to the Build Service, I decided to use XML. This decision was made, because it is the standard data exchange format in the whole Build Service and is predestined for this task. To read out the data from the redirector database, a small ruby script that writes a temporary XML file would be sufficient for the first step. The second step would be to send this file to the Build Service via an HTTP PUT request to the frontend server, what could be done by another ruby script or by the use of the external command line tool curl.

API Calls

Table 4.6 shows the API calls of the frontend to be able to request download counters in different ways and to update the counters. The `group_by` parameter specifies, how the results should be grouped and summarized. Possible values for `group_by` are: project, package, repo or arch. As with all objectives before, the limit parameter can be used to limit the count of results.

method	service	API path	xsd
GET	download counters for top downloaded files	/statistics/download_counter ?limit=<limit>	B.10
GET	summarized download counters for top downloaded projects, packages, repositories or architectures	/statistics/download_counter ?group_by=<group_by> &limit=<limit>	B.11
PUT	send download statistics, to update the download_counter database	/statistics/redirect_stats	B.12

Table 4.6: Frontend interface for most downloaded projects/packages

4.3 Implementation

In this section I highlight what was had to be done to gain the objectives pointed out in section 4.1.3 starting on page 31.

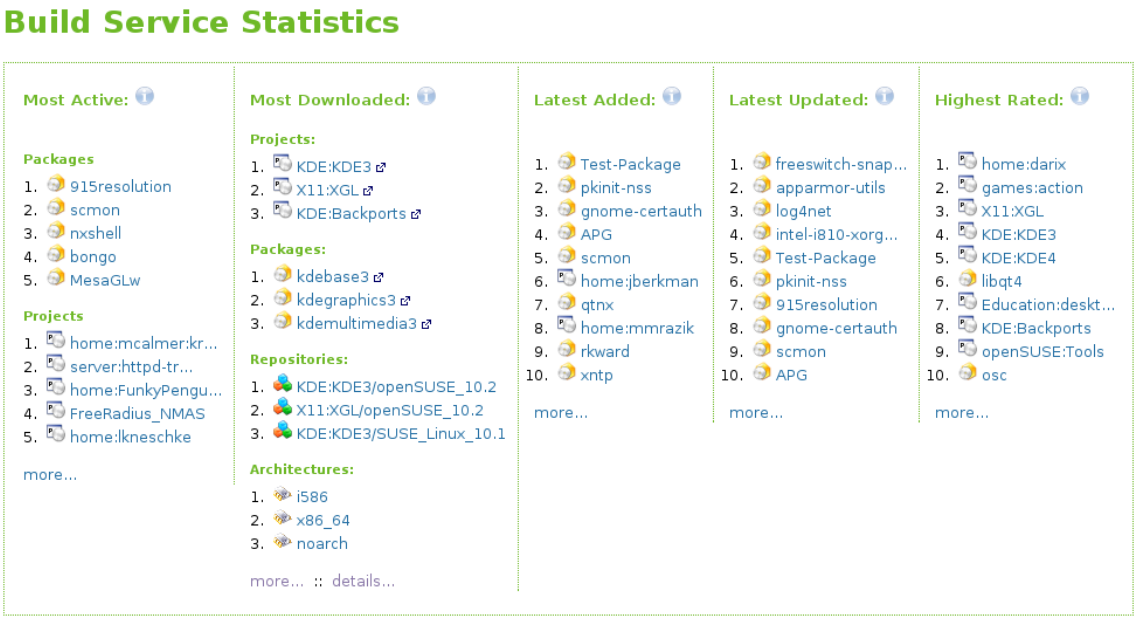


Figure 4.11: Screenshot of the statistics overview in the webclient

4.3.1 Common Aspects

First I explain some aspects of the implementation, that are common to all of the five objectives.

4.3.1.1 Controllers

To have all the Build Service statistics in one place, I decided to create a separate controller in the frontend part and in the webclient part with the meaningful name *statistics*. This way we get the address `http://api.opensuse.org/statistics` as base URL for all API/frontend functions and the webclient base address `http://build.opensuse.org/statistics`.

4.3.1.2 Database Migrations

For changing the database structure, Rails provides a special mechanism called *migrations*. With migrations Rails manages different versions of database structures and makes it easy to change between them.

"Migrations can manage the evolution of a schema used by several physical databases. It is a solution to the common problem of adding a field to make a new feature work in a local database, but being unsure of how to push that change to other developers and to the production server. With migrations, you can describe the transformations in self-contained classes that can be checked into version control systems and executed against another database that might be one, two, or five versions behind." [Mig07a]

This includes many benefits:

- while working in a team of developers: if one person makes a schema change, the other developers just need to update by running `rake migrate`
- on production servers: run `rake migrate` while rolling out a new application release, to bring the database up to date as well.
- while using multiple machines: if developing on both a desktop and a laptop, or in more than one location, migrations can help you keep them all synchronised.

Furthermore Rails uses migrations to abstract the database structure definitions from the underlying DBMS implementation. More on the Rails migration concept can be found at [Mig07a] and [Mig07b].

Migration files are located in a directory called `db/migrate/` and have a fixed naming convention: they start with a triple-digit number (the version number of the database structure), followed by an underscore and some descriptive words joined by underscore signs, e.g. `010_add_download_counter.rb` . To apply a migration to the database, it is sufficient to call `rake migrate` in the Rails installation base directory.

4.3.1.3 ActiveXML Models

As mentioned earlier, communication between the frontend and the webclient is done via XML. To ease this, the ActiveXML class, which is explained on page 29, was written by the Build Service team. All ActiveXML models have to be configured

for the webclient, so that the webclient knows how to communicate with the Build Service frontend and where to get or write model data. For the current ActiveXML implementation, this is done in the configuration file `webclient/config/environment.rb` and looks like listing B.14 on page 92 in the appendix, where only the essential excerpt is shown.

4.3.1.4 Web Client Integration

As the integration of the statistics in the webclient is similar for most of the objectives. This following explanation of the procedure is capable for all five objectives. Starting at section 4.3.2, I basically address only the Build Service frontend scopes.

At first I created new ActiveXML models in the webclient subdirectory `app/models`. This might be a file named `most_active.rb` for the model of the most active objects of the Build Service with the following content:

```
1 class MostActive < ActiveXML::Base
2 end
```

This model knows where to get data from the Build Service frontend because of the ActiveXML configuration mentioned just before. In this case the important configuration lines could look like this:

```
1 map.connect :mostactive, 'rest:///statistics/most_active?:type&:limit',
2   :specific => 'rest:///statistics/activity/:project/:package'
```

This tells the webclient: 'If you need data from the MostActive model, ask the frontend server at the path `/statistics/most_active` and append the parameters'. The webclient can then use the new model in a way like that: `@objects = MostActive.find( :limit => 10, :type => packages )` to get to 10 top active packages. The result is received as XML data and wrapped in a ActiveXML::Node object. Then the webclient forwards the data to the view part which renders the list to be viewable in the web browser.

4.3.1.5 Read-Only Pages for Projects and Packages

At the moment, only registered users can see more than the start page of the Build Service. To browse packages and projects, users need to log in. This needs to be changed in the future and is already work in progress, but is a long-term subject. In the meantime, I decided to add read only pages inside the Build Service, to provide

informative pages about projects and packages that do not require to be logged in. This pages do not display any buttons or links that can change anything of the Build Service. They are just informative, show the package/project title, name, description, download URL and all five types of statistics for the respective package or project.

Figure 4.12: Package statistics on a separate read only page

The present pages for projects and packages can be retrieved at the webclient path `/packages/show/?project=<projectname>&package=<packagename>` and `/projects/show/?project=<projectname>` . For the read only pages just mentioned, I use the methodname *view* instead of *show*.

4.3.1.6 Separate Page for more Statistics

To be able to display more than just the few entries per objective as shown in figure 4.11, I built in a link to display more items at once. Figure 4.13 shows that for the most active projects and packages. The right side in the red box shows the clipping of a new page, that gets loaded when the *more...* link is clicked. There you can choose in the upper area, how many items should be displayed.

The small form that realizes the selection is implemented in two ways:

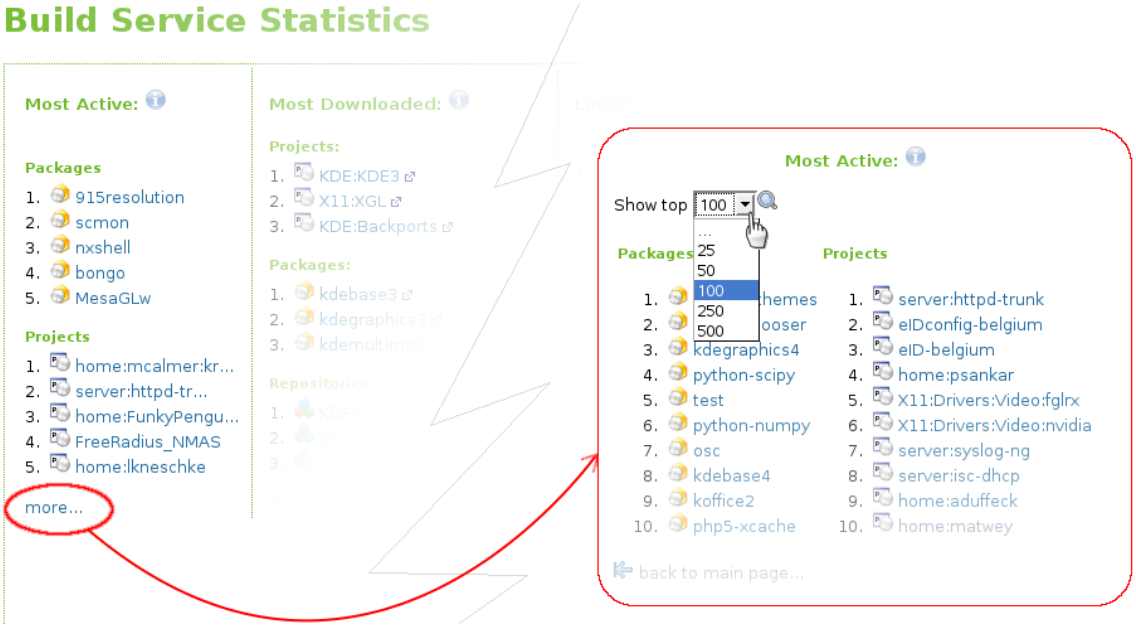


Figure 4.13: All statistics link to a view, where more items can be displayed

1. as standard HTML form element that sends the selection via an HTTP GET request to the server and receives a complete new page with the new information and

2. additionally as Ajax request, that reloads only the relevant part of the page (the list with projects and packages) in the background as explained earlier on page 14 where I introduced the Ajax technology.

The code for that is shared between all statistics pages by the use of a helper which can be seen at B.19 in the appendix. Helper functions can be shared between different views, so there is nothing unnecessarily duplicated.

4.3.1.7 XML Validation

To be sure the Build Service frontend gets correct data, all sent XML files that are received by the frontend are validated (see appendix A.1) against the appropriate XML schema. This is done by the external tool `xmllint` with the XML schema files that can be found in the appendix starting on page 82 up to page 90.

4.3.2 Newest Packages and Projects

As already stated in the design chapter 4.2.1, there is not much more to do than to extend the projects and packages database table by one field named `created_at`. As soon as this field exists, the Ruby on Rails magic fills it automatically with the actual timestamp while creating a new project or package.

The database migration file for adding the new field to the project and package table looks like the following listing:

Listing 4.1: Migration for adding created_at timestamps

```
1  class AddCreateAndModifiedTimeStamps < ActiveRecord::Migration
2
3    def self.up
4      add_column :projects, :created_at, :datetime
5      add_column :packages, :created_at, :datetime
6    end
7
8    def self.down
9      remove_column :projects, :created_at
10     remove_column :packages, :created_at
11   end
12
13 end
```

The frontend methods to send single timestamps and a list of timestamps are called `added_timestamp` and `latest_added`. They are imprinted in the listings section B.16 and B.15 starting at page 92.

For the view part of the implementation in the frontend I used RXML views, because they are the simplest way to return XML data with Rails. RXML views use the Rails built in *Builder* to render XML data.

"Builder is a freestanding library that lets express structured text, such as XML, in code. A Builder template, which is in Rails a file with an `.rxml` extension, contains Ruby code that uses the Builder library to generate XML." [Dav05]

Here is a simple Builder template that outputs the timestamp when a package was added to the Build Service in XML:

```
1 xml.instruct!
2 xml.latest_added do
3     xml.package(
4       :name => @package.name,
5       :project => @package.project.name,
6       :created => @package.created_at.xmlschema
7     )
8 end
```

This leads to the following XML output:

```
<latest_added>
  <package project="home:dmayr" name="x11vnc" created="2005-01-01T00:00:02"/>
</latest_added>
```

Notice how Builder has taken the names of methods and converted them to XML tags. When using `xml.latest_added` , it created a tag called `<latest_added>` whose subelements were set from the subsequent hash. [Dav05]

The created RXML views for this thesis are listed in section B.17 and B.18 in the appendix.

The following two listings show examples for the returned XML data from the frontend of a single project (listing 4.2) and for the top newest objects (listing 4.3).

Listing 4.2: XML data for a single package timestamp

```
<?xml version="1.0" encoding="UTF-8"?>
<latest_added>
  <project name="home:dmayr" created="2005-01-01T00:00:01+01:00"/>
</latest_added>
```

Listing 4.3: XML data for latest added objects

```
<?xml version="1.0" encoding="UTF-8"?>
<latest_added>
  <project name="testproject" created="2007-03-26T15:57:49+02:00"/>
  <package name="libgpod" project="dmayr" created="2007-03-11T19:12:14+01:00"/>
  <package name="apache2" project="Apache" created="2007-03-09T23:08:58+01:00"/>
  <project name="home:tlitsch" created="2007-03-09T19:20:05+01:00"/>
  <package name="x11-input" project="X11" created="2007-03-09T17:39:14+01:00"/>
</latest_added>
```

The generic implementation process to integrate the functions in the webclient is already described in section 4.3.1.4. For this one objective I point it out anyway.

On the webclient side, a new model `latestadded` needs to be defined, which can be done with just two lines of code, as listing 4.4 shows:

Listing 4.4: Webclient model for the latest added objects

```
class LatestAdded < ActiveXML::Base
end
```

This model knows how to get data from the frontend because of the ActiveXML configuration mentioned in 4.3.1.3, which is imprinted in the listing section on page 92.

After that, the webclient can call `LatestAdded.find( :limit => 10 )` to get an array of the 10 newest objects sorted by creation time. The webclient statistics controller calls this and forwards the data to the view part of the webclient, which renders the list of latest added packages and projects as HTML and delivers it to the client web browser.

To get a specific timestamp for just one package or project, the webclient calls `LatestAdded.find( :specific, :project => @project, :package => @package ).package.created` and gets XML data as shown in listing 4.2.

Figure 4.14 shows what the webclient displays in the detailed view of the top newest projects and packages. The tooltip displayed when the mouse cursor remains some seconds over an entry shows the period of time that passed since the object was added to the Build Service.

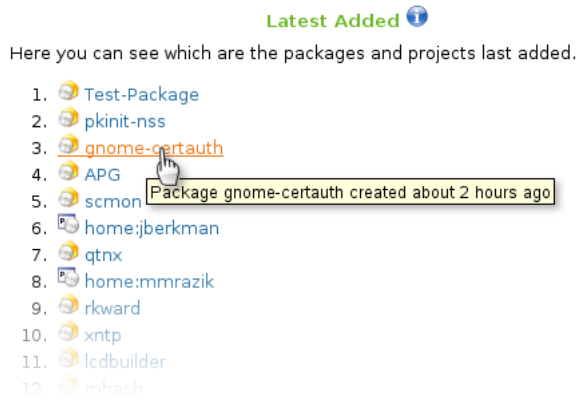


Figure 4.14: Screenshot of latest added projects/packages in webclient

4.3.3 Latest Updated Packages and Projects

Rails has also another automatism for timestamps: if a database field of an ActiveRecord model is named `updated_at`, Rails updates the timestamp every time a model instance is updated and saved.

The database migration file looks nearly the same as the migration of the previous section (see listing 4.1) except of the field name, that needs to be `updated_at` instead of `created_at`.

In addition to the automatic timestamp updates, the timestamp of packages needs to be updated when files are added and the timestamp of projects needs to be updated when the list of build repositories is changed. This can be gained by just calling

the `save` method on the appropriate model object after adding files to a package or build repositories to a project.

The XML data of the latest updated objects is analogous to the latest added objects described in the section before and therefore not listed here anymore. Figure 4.15 shows the detailed view of the latest updated projects and packages in the webclient.

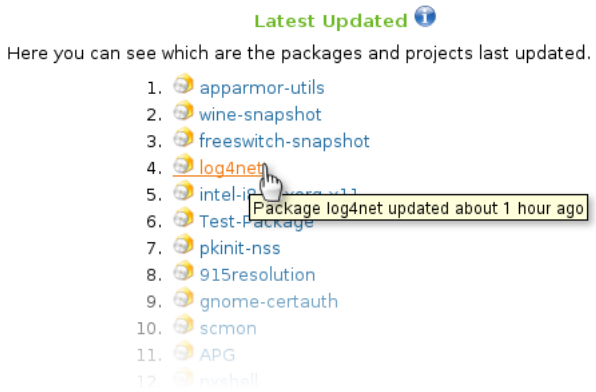


Figure 4.15: Screenshot of latest updated projects/packages in webclient

4.3.4 Package and Project Rating

For the rating objective is something more to do than for the previous objectives. Data for this one is not generated automatically, but must be gathered from the Build Service users.

To be able to receive ratings from the users in the frontend, a method needs to be created that accepts XML data according to the XML schema B.6 via an HTTP PUT request. To avoid making it more complicated than necessary, I decided to use the same method that sends single ratings of projects or packages, which is just called `rating`. Inside this method it is checked, whether it's an GET or PUT request and branched accordingly. For the source code, see listing B.21 on page 96. The PUT part first checks if the user already rated the object in question and if he has, the rating is just updated. If the user did not yet rate the object, a new rating entry is created and saved.

The method `highest_rated`, which sends rating scores from the frontend to the client can also be seen in listing B.21. The data the frontend returns for a single rating score is shown in listing 4.5 below. The XML data for a list of highest rated objects is shown in listing 4.6.

Listing 4.5: XML data for a single package rating score

```
<rating project="home:dmayr" package="x11vnc" count="251" user_score="4">2.3</
    rating>
```

Listing 4.6: XML data for the highest rated objects

```
<?xml version="1.0" encoding="UTF-8"?>
<highest_rated>
  <project score="4.7592" name="KDE:KDE3" count="41"/>
  <project score="4.6345" name="openSUSE:Tools" count="15"/>
  <package score="4.4970" project="openSUSE:Tools" name="osc" count="81"/>
  <project score="4.0294" name="FATE" count="53"/>
</highest_rated>
```

Figure 4.16: Screenshot of highest rated projects/packages in webclient

The model part implements the polymorphic association addressed in the design chapter mentioned earlier at page 40. In this case, the model classes look like the following listing:

Listing 4.7: Models for rating

```
class Package < ActiveRecord::Base
  has_many :ratings, :as => :object, :dependent => :destroy
end

class Project < ActiveRecord::Base
  has_many :ratings, :as => :object, :dependent => :destroy
end

class Rating < ActiveRecord::Base
  belongs_to :projects, :class_name => "Project", :foreign_key => "object_id"
  belongs_to :packages, :class_name => "Package", :foreign_key => "object_id"
end
```

An example for the polymorphic associations may look like the relations example in figure 4.17. Figure 4.16 shows the result of the highest rated packages and projects in the webclient.

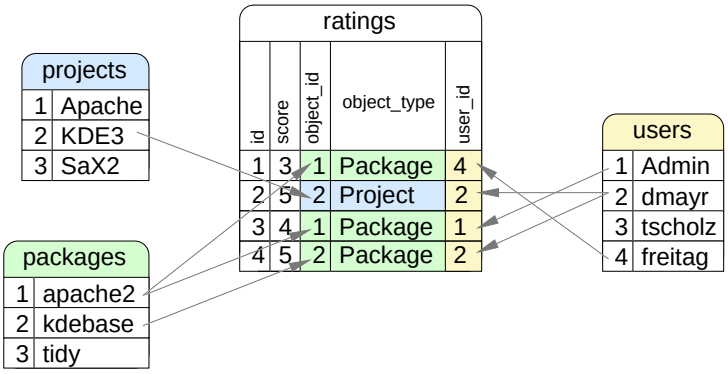


Figure 4.17: Simplified example of rating table relations

4.3.5 Activity Statistics of Packages and Projects

For the users it is very interesting, how active projects and packages are. If a project is no more active, it will not be developed any more further and will not get enhanced. This might be a reason for many users to not install inactive software. By calculating and displaying activity of projects and packages, maintainers and developers could get encouraged to work on their packages and keep them up to date.

4.3.5.1 Activity of Packages

I already explained the activity algorithm in the design chapter 4.2.4 on page 42. After some tests with real data, it turned out that implementing this algorithm in ruby is much too slow. The cause for this is, that if we want a listing of the most active packages, we always need to recalculate the activity of all packages – which can be a huge number – in order to sort them. A much faster approach is to let the MySQL database calculate the activity values.

Because the algorithm is needed at several places, I wrote a small method for the packages model class, that just returns a SQL snippet with the activity algorithm as string.

```
1 def activity_algorithm
2   # the algorithm (sql snippet) we use for calculating activity of packages
3   '@activity:=(' +
4     'pac.activity_index - POWER(' +
5       'TIME_TO_SEC( TIMEDIFF( NOW(), pac.updated_at ))/86400, 1.55' +
6     ') / 10' +
7   ')'
8 end
```

This snippet could be inserted at every place where it is needed:

```
@packages = Package.find :all,
  :from => 'packages pac, projects pro',
  :conditions => 'pac.project_id = pro.id',
  :select => 'pac.*, pro.name AS project_name,' +
    "(#{Package.activity_algorithm}) AS act_tmp," +
    'IF( @activity<0, 0, @activity ) AS activity_value'
```

The activity_index attribute of each package needs to be updated every time the package itself is updated. This is done in the package model by the following method:

```
def update_timestamp
  # the value we add to the activity, when the object gets updated
  activity_addon = 10
  activity_addon += Math.log( self.update_counter ) if update_counter > 0
  new_activity = activity + activity_addon
  new_activity = 100 if new_activity > 100

  self.activity_index = new_activity
  self.created_at ||= Time.now
  self.updated_at = Time.now
  self.update_counter += 1
end
```

4.3.5.2 Activity of Projects

Activity of projects is generated by calculating the average activity of all contained packages. This is done in the project model, see the following listing:

```
def activity
  # get all packages including activity values
  @packages = Package.find :all,
    :from => 'packages pac, projects pro',
    :conditions => "pac.project_id = pro.id AND pro.id = #{self.id}",
    :select => 'pro.*,' +
      "( #{DbPackage.activity_algorithm} ) AS act_tmp," +
      'IF( @activity<0, 0, @activity ) AS activity_value'

  # count packages and sum up activity values
  project = { :count => 0, :sum => 0 }
  @packages.each do |package|
    project[:count] += 1
    project[:sum] += package.activity_value.to_f
  end

  # calculate and return average activity
  return project[:sum] / project[:count]
end
```

The integration into the weblient was already described on page 48.

Figure 4.18 shows the list of most active projects and packages in the webclient.

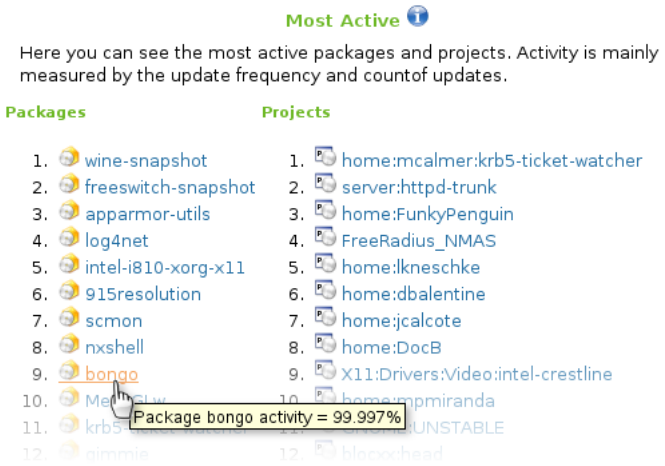


Figure 4.18: Screenshot of most active projects/packages in webclient

4.3.6 Download Statistics of Packages and Projects

The download statistics objective is an exceptional case: it gets its data from outside the Build Service, from the openSUSE download redirector that was already introduced in section 4.2.5.1 on page 44.

Because the redirector uses its own database, the statistics must be transferred somehow to the Build Service frontend. As data transfer format it was chosen to use XML because it is easy parseable and already used all over the Build Service. To be able to validate the XML file a XSD schema was written for the statistics file, see listing B.12 on page 90.

4.3.6.1 Generating Download Statistics XML File

To generate the XML file with all the download counters, I decided to create an external ruby script. This script uses the ActiveRecord class mentioned on page 19 to access the redirector database and the XMLBuilder to create the XML output. The listing of the script is imprinted on page 97 and is regularly called by the cron daemon, which is explained more detailed in chapter 5.2.2.

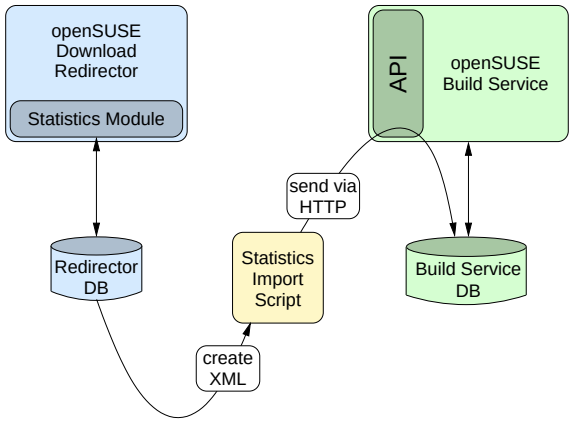


Figure 4.19: Separate databases for the download redirector and the Build Service

As stated earlier in section 4.3.1.7, the frontend validates the generated XML data with a schema file. This schema file can be seen in listings section B.12.

Figure 4.19 shows the way of the download counters from the openSUSE redirector database to the Build Service.

4.3.6.2 Import of the Download Counters

As the Build Service frontend has an open API via HTTP(s), it was an easy decision to send the statistics XML file via HTTP to the frontend. To achieve this, a little command line tool called `curl` was used. The command line to send the XML file to the frontend looks like the following:

```
curl -X PUT -u user:pass http://api.opensuse.org/statistics/redirect_stats
  -T statistics.xml
```

The redirector collects only raw strings for the project, package, repository and architecture names in its database. But in the frontend it would be desirable to have them as IDs, referencing the respective objects in the other tables as foreign keys. Therefore it is necessary to resolve the ID of a name coming from the XML file and write it into the frontend database instead of the raw name. Figure 4.22 shows the flow chart of the procedure that achieves that.

The resulting table entries of the example redirector statistics mentioned before could look like the lines in table 4.7.

The two figures 4.20 and 4.21 show the webclient pages that display the most downloaded Build Service items. Figure 4.20 displays the top downloaded projects, pack-

id	project_id	package_id	repo_id	arch_id	filename	filetype	version	release	count
413	321	3124	32	4	apache2	rpm	2.6.5	144.1	432
927	911	9473	32	4	kdelibs	rpm	4.0.1	21.3	5379
229	249	5113	12	6	FastCGI	rpm	2.4.0	3.1	143

Table 4.7: Frontend download statistics table excerpt

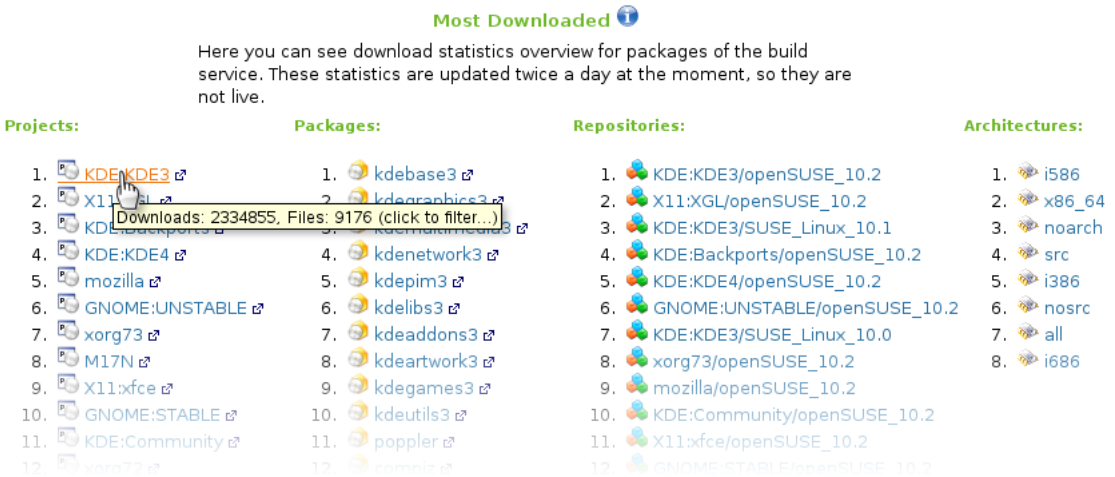


Figure 4.20: Screenshot of most downloaded projects/packages in webclient

ages, repositories and architectures. Mouse-over tooltips show the exact counter value and the count of files.

Figure 4.21 appears, when the user clicks any entry of the previous screenshot. It displays download details where you can see all the contained files with filename, version, release and so on. This detailed list can be filtered by project, package, repository and architecture or any combination of it. In this view, any of the listed items can be clicked to filter by it. This is shown by a small filter icon in the header of the table. This icon can be used to remove the filter of a certain column again.

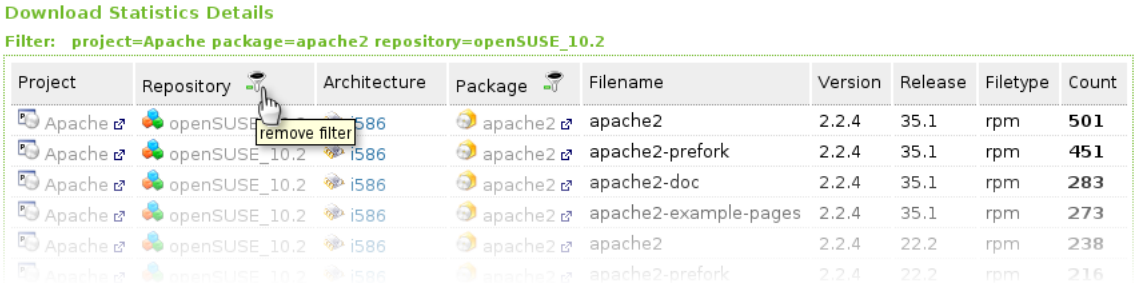


Figure 4.21: Screenshot of detailed download statistics with active filter

Figure 4.22 shows the program flow chart, how the download counters from the openSUSE download redirector are imported by the frontend.

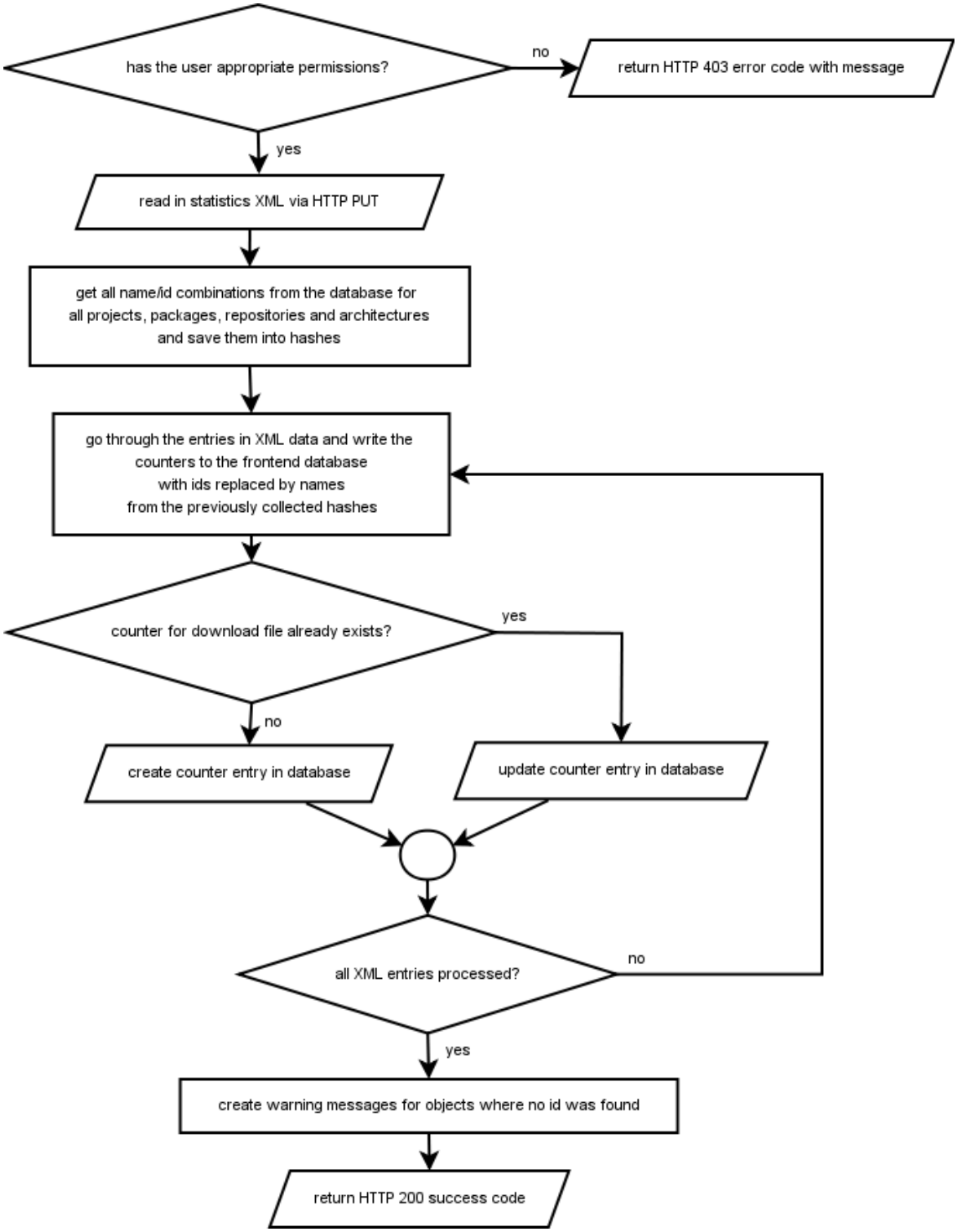


Figure 4.22: Program flow chart of the download statistics import in the frontend

5 Runtime Experience

In this chapter some experiences that occurred during runtime of the implemented statistic features are pointed out.

5.1 Performance

While developing the statistic features I never had the same amount of data in the database as in production systems, where the quantity of data grows every day. Because of that it turned out that some functions did not perform satisfactory enough with real world data masses. This make improvements necessary that I describe in this section.

5.1.1 XML Stream Parser for Download Statistics Import

The first version of the importer script for download statistics did not perform very well, it was way too slow and consumed oodles of memory. For importing an XML file with e.g. 20 MB, the pristine method took about 800 MB of RAM. The reason for this waste was, that it read in the entire XML file and built an ActiveXML object for every single entry, whereof about 200,000 were in the XML file at that time.

The solution for this problem was to implement the XML import in the Build Service frontend as an XML stream parser. One possibility would have been to use a SAX (see [Wik07d]) parser. Rails provides a light SAX-like parser implementation that is quite fast and memory economical called `StreamListener`.

"SAX parsers scan the document once and invoke event methods to as they encounter each atomic XML components and each boundary of a large XML component [...]. Consequently, documents that require simple processing can be of unlimited size, be-

cause memory resources depend only on the activity needed to translate one stream of input data to a second stream of output data." [Str07]

For this, the class `StreamHandler` (see listing B.24) was written. It includes the Rails built in StreamListener and implements the methods `tag_start` and `text`, which are called every time the parser reads in an XML tag or the content of an XML tag.

The class `StreamHandler` is used in the statistics_controller by the `redirect_stats` method, where the data is sent as PUT request to the frontend in XML format, as follows:

```
1 data = request.raw_post
2 streamhandler = StreamHandler.new
3 REXML::Document.parse_stream( data, streamhandler )
```

This improvement made the import process using only about 30 MB of RAM memory independent of the XML data size.

5.1.2 Caching Frontend Output

Because it is not necessary to have most of the statistics in real time, a performance speedup could be achieved by caching the frontend XML output for a period of time after generation. The Rails framework provides three different methods for caching output:

page caching Page caching is an caching-approach where the entire action output is stored as an HTML file that the web server can serve without going through the Action Pack. This can be as much as 100 times faster than going through the process of dynamically generating the content. Unfortunately, this speedup is only available to stateless pages where all visitors are treated the same.

action caching Action caching is similar to page caching by the fact that the entire output of the response is cached, but unlike page caching, every request still goes through the Action Pack. The key benefit of this is that filters run before the cache is served. This allows authentication and other restrictions to decide whether someone is allowed to see the cache or not.

fragment caching Fragment caching is used for caching various blocks within templates without caching the entire action as a whole. This is useful when certain elements of an action change frequently or if they depend on complicated state while other parts rarely change or can be shared amongst multiple parties.

To use Rails caching generally, it needs to be turned on first.

```
# config/environment.rb
Base.perform_caching = true
```

However, the built in caching techniques have some disadvantages:

Parameters of requests are ignored In the original Rails caching implementation GET parameters of requests are just ignored. Rails treated for example the request `/statistics/most_active?type=packages&limit=1000` the same way as the request `/statistics/most_active?type=projects&limit=5` . As we are using parameters, e.g. for limiting the number of or the type of request, parameters are essentially.

Cached items must be manually expired To invalidate a cached page, action or fragment, Rails needs to be told when it should expire the particular cached item and rebuild it new.

Caching did not work correctly Somehow the original Rails caching did not work correctly all time. Maybe there exist implementation errors that are fixed now, but during the preparation of this writing I had some not comprehensible errors that were very annoying.

After some research it turned out that there already exists a Rails caching system that fits all our needs, see [Act07a]. It pays attention to parameters of the request and expires cached items automatically after a predetermined period of time.

To use it, one first have to copy the file `action_cache.rb` into the `lib/` directory inside the Rails directory structure. Then it needs to be loaded in `config/environment.rb` with the command as follows:

```
# config/environment.rb
require 'action_cache'
```

Afterwards it needs to be declared, which actions output should be cached:

```
# app/controllers/statistics_controller.rb
caches_action :highest_rated, :most_active, :latest_added, :latest_updated,
  :download_counter
```

Last thing to do here is to set the time to live for each affected action inside the action code itself:

```
@response.time_to_live = 10.minutes
```

5.2 Deployment

In this chapter I show how the developed code is carried to the frontend and webclient production systems and how it is activated.

5.2.1 Automated Deployment with Capistrano

For the deployment of the frontend and webclient we used capistrano (see [Cap07c]).

Capistrano is a utility which can be used to automate the deployment of applications. It can execute commands in parallel on multiple machines and automates the process of making a new version of an application available on one or more web servers, including supporting tasks such as changing databases.

"Capistrano is written in the Ruby language and is distributed using the RubyGems [Rub07] distribution channel. It is an outgrowth of the Ruby on Rails web application framework, but has also been used to deploy web applications written using other frameworks." [Wik07a]

Application deployment is one of those things that becomes more and more complicated as the scale of an application increases. With just one single box running a database and the application, it's quite simple. But when it is necessary to put the database on a different server and then separating the web servers from the app servers and eventually splitting your database into master and slave servers, it gets more and more complex. It may come to the point where administrators almost don't want to deploy their applications manually any more. [Cap07a]

"Capistrano is a standalone utility that can also integrate nicely with Rails. You simply provide Capistrano with a deployment 'recipe' that describes your various servers and their roles and voila! You magically have single-command deployment. It even allows you to roll a bad version out of production and revert back to the previous release.
It should be stated that the concepts that Capistrano uses and encourages are not specific to Rails, or even Ruby. They are common-sense practices that are applicable to a variety of environments. In fact, you'll find that there is very little that is Rails-specific about Capistrano, aside from the fact that it is in Rails that it found its genesis. No matter where you are or what environment you are using, Capistrano can probably help ease your deployment pains." [Cap07b]

The capistrano homepage with more information can be found at [Cap07c].

5.2.2 Download Statistics Import via Cronjob

To update the download counters in the Build Service frontend and feed it with actual data, there are two scripts:

create_stats_xml.rb to get statistics from the opensuse redirector database and write them into an XML file (see B.22 on page 97 for the source code)

send_stats_to_api.sh to send this xml file to the Build Service frontend where it is stored in its own database

These two scripts have to run regularly. Due to the low demand for being up to date I decided to run them just twice a day: at midnight and at noon.

In order to get some feedback (see below) from these scripts, a wrapper script that calls them and adds some text and timing information was written:

```
#!/bin/bash

echo -e "\nget statistics from the opensuse redirector: ..."
time ~/bin/create_stats_xml.rb

echo -e "\nsend statistics to the Build Service api: ..."
time ~/bin/send_stats_to_api.sh
```

This wrapper script is now called by cron twice a day. After it ran, the output is sent via email to a given address. The following line in the crontab of the user 'opensuse' executes the regular calls:

```
00 01,13 * * * ~/bin/update_download_stats 2>&1 | mail -s "download
  statistics import result" dmayr@suse.de
```

The resulting mail looks like this:

```
get statistics from the opensuse redirector: ...

redirector statistics written as XML to redirect_stats.xml, size is 67420 Kb.

real    4m50.896s
user    4m42.870s
sys     0m4.528s

send statistics to the Build Service api: ...

<?xml version="1.0" encoding="UTF-8"?>
<status code="ok">
```

```
13   <summary>Ok</summary>
14   <details></details>
15 </status>
16
17 real     77m0.789s
18 user     0m0.020s
19 sys      0m0.224s
```

5.3 Testing

5.3.1 Rails Testing Framework

Rails has a sophisticated testing framework built in. It allows automated testing of controller and model classes. This helps developers to check constantly whether the application still behaves as planned during development or not. Furthermore it is possible to develop test-driven: first write tests that define how the application should behave and then write the code to meet the test requirements.

There are three sorts of tests available:

unit tests to test the models. Usually the CRUD functions (create, read, update and delete) and all custom model methods are tested with unit tests.

functional tests to test the controllers and their methods.

integration tests to test the cooperation of multiple controllers with sessions and routing in mind.

For the Build Service I implemented only functional tests in the frontend. This is sufficient to ensure that they work as suspected due to the simplicity of the models and the lack of session depending functionality.

Rails uses a separate database for testing. To have a defined starting point for every test, the Rails framework clears this database every time a test run is started and fills it with a defined set of test data. This initial test data setup is defined in so called fixtures. Such a fixture file could look like listing 5.1, which is defined in the *YAML* format. This way it is assured that previous tests do not influence the result of current tests.

Listing 5.1: Example YAML test fixture for users

```
1 admin:
2   id: 1
3   name: admin
4   email: admin@suse.de
5
6 dmayr:
7   id: 2
8   name: dmayr
9   email: dmayr@suse.de
```

5.3.2 Implemented Tests

Because of the importance of the Build Service *frontend*, I implemented tests for the most important statistics functions.

The following test methods are implemented:

test_latest_added tests the latest_added function and traverses therefore the following steps:

- get `/statistics/latest_added`
- response was successful?
- response contains the XML element `project` ?
- response contains latest added object according to the fixture data?

test_latest_updated tests the latest_updated function and traverses therefore the following steps:

- get `/statistics/latest_updated`
- response was successful?
- response contains the XML element `project` ?
- response contains latest updated object according to the fixtures?

test_download_counter tests the download_counter function and traverses therefore the following steps:

- get `/statistics/download_counter`
- response was successful?
- response contains the XML element `download_counter` with the subelement `count` ?

- response contains `count` element with attributes for the most downloaded file according to the fixtures?

test_download_counter_group_by tests the download_counter function where results are grouped by type and traverses therefore the following steps:

- get `/statistics/download_counter?group_by=project`
- response was successful?
- response contains the XML element `download_counter` with the sub-element `count` ?
- response contains `count` element with attributes for the most downloaded file according to the fixtures?
- get /statistics/download_counter?
  group_by=arch& project=Apache&package=apache2
- response was successful?
- response contains the XML element `download_counter` with the sub-element `count` ?
- response contains `count` element with attributes for the most downloaded file in the apache2 package inside the Apache project?

test_most_active tests the most_active function and traverses therefore the following steps:

- get `/statistics/most_active?type=packages`
- response was successful?
- response contains the XML element `most_active` with the sub-element `package` ?
- response contains `package` element with attributes for the most active package according to the fixtures?
- get `/statistics/most_active?type=projects`
- response was successful?
- response contains the XML element 'most_active' with the sub-element `project` ?

- response contains `project` element with attributes for the most active project according to the fixtures?

test_highest_rated tests the highest_rated function and traverses therefore the following steps:

- get `/statistics/highest_rated`
- response was successful?
- response contains the XML element `highest_rated` with the sub-element `project` ?
- response contains `project` element with attributes for the highest rated project according to the fixtures?

In listing 5.2 you can see the extract of the file `test/functional/statistics_controller_test.rb` that tests the `download_counter` method of the Build Service frontend. The full listing of all implemented tests is imprinted in the appendix starting on page 99.

Listing 5.2: Example functional test for download counters

```
1  class StatisticsControllerTest < Test::Unit::TestCase
2
3    fixtures :projects, :packages, :download_stats
4
5    def test_download_counter_group_by
6      prepare_request_with_user @request, 'tom', 'thunder'
7
8      # without project- & package-filter
9      get :download_counter, {
10     'group_by' => 'project'
11     }
12     assert_response :success
13     assert_tag :tag => 'download_counter', :child => { :tag => 'count' }
14     assert_tag :tag => 'download_counter', :attributes => { :all => 9302 }
15     assert_tag :tag => 'count', :attributes => {
16       :project => 'Apache', :files => '9'
17     }, :content => '8806'
18
19     # with project- & package-filter
20     get :download_counter, {
21       'project' => 'Apache', 'package' => 'apache2', 'group_by' => 'arch'
22     }
23     assert_response :success
24     assert_tag :tag => 'download_counter', :child => { :tag => 'count' }
25     assert_tag :tag => 'download_counter',
26       :attributes => { :all => 9302 }
27     assert_tag :tag => 'count', :attributes => {
28       :arch => 'x86_64', :files => '6'
29     }, :content => '5537'
30   end
31
32 end
```

6 Conclusion and Outlook

In this chapter I will review the parts of the system that I have implemented and give a brief outlook on the further development of the project.

At the end of the writing of this thesis the openSUSE Build Service had about 1200 users, nearly 9000 packages included in about 700 projects.

On the one hand the separation into the three tiers - the client(s), the frontend and the backend - the basic architecture of the openSUSE Build Service gives high flexibility and scalability. But on the other hand development may be a bit more complicated, because changes that concern more than one tier need to be made concurrently by all involved developers and need to be implemented in all the affected tiers at the same time.

Development with Ruby on Rails is clear, easy and transparent. Sometimes it becomes misleading when non-obvious dynamical created methods come into effect. This issue is faced with the possibility to write tests for the application, that can be run everytime while development and runtime. It is even possible to develop test-driven, which means first writing tests and then the code to fulfill the test requirements.

Ruby on Rails is a well elaborated, consistent and practical application development framework, which is a joy to use and which makes it easier to develop, test, deploy and maintain web applications than anything else I have used so far.

Ruby allows to write short, precise and maintainable code, a very important aspect when many developers are involved in a project.

The statistics facilitate to browse the Build Service content in complete new level. It delivers very useful information to users of the Build Service and the Build Service team. They represent the activity of the involved users, usage of the results, popularity of packages and acceptance of the whole Build Service.

Especially end-users benefit from the implemented enhancements. They can now find software easier, which is interesting in many different aspects. But also the Build Service team and SUSE as company benefits from the statistics as they reflect the acceptance, popularity and usage of the whole system very well. Project managers inside SUSE can use the information to determine how active package maintainers are and which packages are the real demanded ones.

Outlook

The Build Service will be opened entirely to the public soon, so no manual approval steps of the user registration will be necessary in the near future. This will give us a broader user base in the future and also much more load on our servers and storage systems. It remains to be seen how well the system behaves with this challenge, but as the basic concepts are well deliberated no bigger problems are expected.

As further improvement for the Build Service statistics it would be nice to have some kind of history or graph for the download counters, so the progression of download requests over a period of time is visible. To achieve this, the round robin database `rrdtool` [Oet07] and its tools would be a good solution. The `rrdtool` can produce nice graphs like the example in figure 6.1.

For the other statistic objectives of this thesis it would also be nice to be able to see how e.g. rating of a package or its activity changes over time.

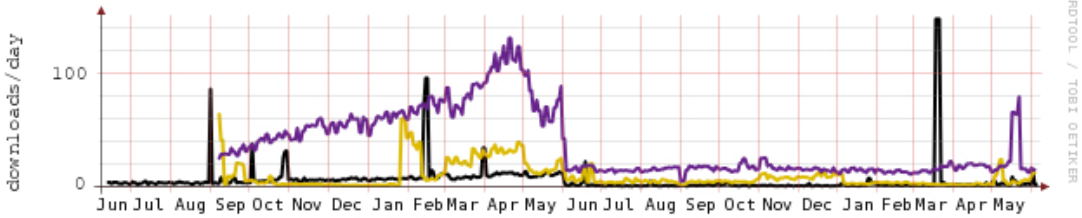


Figure 6.1: RRDtool example how download statistics graph could look like

To improve the performance of the download statistics import even more, it would be good for the future to use another XML parser as for example *libXML for Ruby*. As [Lib07] shows, this parser, written in C, should be much faster than any Ruby implementation. In addition, the SQL queries in the Build Service frontend to get the download counters needs to be improved and optimized.

For the future the openSUSE Build Service will become more and more important. In the long-term, it will replace the system that is currently responsible for the complete building of the official openSUSE Linux distribution.

List of Figures

List of Tables

List of Listings

A Technologies

A.1 XML

The Extensible Markup Language, mostly abbreviated as XML, is a general purpose markup language. Its primary purpose is to share and transport data across different information systems, particularly via the Internet. XML is a free and open standard recommended by the World Wide Web Consortium [W3C07].

A simple XML example file could look like this:

```
<? xml version="1.0" encoding="UTF-8" standalone="no" ?>
<package name="kdelibs" project="KDE">
  <title>KDE libraries</title>
  <description>Base libraries of the KDE framework</description>
</package>
```

An XML document is called *well-formed* when it complies to the syntactic rules for XML. When a well-formed XML document additionally follows the rules of a particular *XML schema*, it is called a *valid* XML document. An XML schema is a language for describing the structure and constrains XML documents. XML schemata itself are also described in XML.

A simple XML schema, to be able to validate the above XML sample, could look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">
  <xs:element name="package">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="title" type="xs:string" />
        <xs:element name="description" type="xs:string" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:string"/>
      <xs:attribute name="project" type="xs:string"/>
    </xs:complexType>
  </xs:element>
```

```
14 </xs:schema>
```

XML is the essential data transmission format used all over the Build Service. All communication between the three tiers – the backend, the frontend and the different clients – is done via XML. Additionally there exist XML schemata, which are used to validate all XML data received by the Build Service frontend.

A.2 Web Applications

A web application uses a website as the frontend interface for an application running on a central server. On the client side a standard web browser is sufficient to use a web application.

Web applications are getting more and more popular because nearly every computer has a web browser installed, so on the client side normally nothing needs to be done to be able to use web applications. Because web application software runs on central servers, they can be updated easily and without touching the client side.

The Build Service provides three *clients* to access it (besides using the web service API itself), see 3.3.2.3. The one client that is addressed with this thesis is a *web application*, the openSUSE Build Service webclient.

A.3 Web Services

A web service can be defined as a software application, that is addressable with an URI, communicates via standard HTTP or HTTPS protocol and sends/receives data as XML snippets. Web services are primarily designed to interact with other software applications in a standardized way.

There are many examples for public web services:

Google the well known internet search engine provides an open interface to use its search functions and other services in other applications. For more information see [Goo07]

Flickr provides an open platform for everyone to upload and share photos with an open web service API. For more information on their web service see [Fli07]

Amazon provides developers with direct access to Amazons technology platform. This helps third party suppliers offering book ordering services or e.g. historical book pricings integrated in their own services. For details see [Ama07].

eBay offers an API for developers to design their own applications to interact with eBay. See [eBa07].

The Build Service *frontend*, which implements the API for the Build Service, is also a web service. This is described on page 26.

A.4 DBMS

"A database management system (DBMS) is a complex set of software programs that controls the organization, storage and retrieval of data in a database." [Wik07b]

The DBMS manages user requests so that users and other programs are free from having to understand where the data is physically located on storage media. Furthermore database management systems take care of concurrent access from multiple clients, data consistency and data integrity. To interact with a DBMS usually slight various dialects of SQL are used.

Popular examples for DBMS are Oracle, MySQL and DB2. The Build Service uses the MySQL DBMS for data storage in the frontend tier – see the technical overview of the Build Service in section 3.3.

B Listings

Listing B.1: Redirector database table schema

```
CREATE TABLE `redirect_stats` (
  `id` int(11) NOT NULL auto_increment,
  `project` varchar(255) default NULL,
  `repository` varchar(255) default NULL,
  `arch` varchar(10) default NULL,
  `filename` varchar(255) default NULL,
  `filetype` varchar(10) default NULL,
  `version` varchar(255) default NULL,
  `release` varchar(255) default NULL,
  `count` int(11) default '0',
  `package` varchar(255) default NULL,
  `created_at` timestamp NULL default CURRENT_TIMESTAMP,
  `counted_at` timestamp NULL default NULL,
  PRIMARY KEY  (`id`),
  KEY `project` (`project`),
  KEY `package` (`package`),
  KEY `repository` (`repository`),
  KEY `arch` (`arch`),
  KEY `filename` (`filename`),
  KEY `filetype` (`filetype`),
  KEY `version` (`version`),
  KEY `release` (`release`)
) ENGINE=MyISAM AUTO_INCREMENT=1 ;
```

Listing B.2: XML schema for timestamp when project/package was added

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="latest_added">
    <xs:annotation>
      <xs:documentation>
        Package or project with timestamp when it was added.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:choice minOccurs="1" maxOccurs="1">
        <xs:element ref="package" minOccurs="1" maxOccurs="1" />
        <xs:element ref="project" minOccurs="1" maxOccurs="1" />
      </xs:choice>
    </xs:complexType>
  </xs:element>

  <xs:element name="package">
```

```
19      <xs:complexType>
20        <xs:attribute name="name" type="xs:string" use="required" />
21        <xs:attribute name="project" type="xs:string" use="required" />
22        <xs:attribute name="created" type="xs:dateTime" use="required" />
23      </xs:complexType>
24    </xs:element>
25
26    <xs:element name="project">
27      <xs:complexType>
28        <xs:attribute name="name" type="xs:string" use="required" />
29        <xs:attribute name="created" type="xs:dateTime" use="required" />
30      </xs:complexType>
31    </xs:element>
32
33  </xs:schema>
```

Listing B.3: XML schema for latest added projects/packages

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4    <xs:element name="latest_added">
5      <xs:annotation>
6        <xs:documentation>
7          List of packages and projects latest added to the openSUSE build
8          service ordered by creation timestamp.
9        </xs:documentation>
10     </xs:annotation>
11     <xs:complexType>
12       <xs:choice minOccurs="0" maxOccurs="unbounded">
13         <xs:element ref="package" minOccurs="0" maxOccurs="unbounded" />
14         <xs:element ref="project" minOccurs="0" maxOccurs="unbounded" />
15       </xs:choice>
16     </xs:complexType>
17   </xs:element>
18
19   <xs:element name="package">
20     <xs:complexType>
21       <xs:attribute name="name" type="xs:string" use="required" />
22       <xs:attribute name="project" type="xs:string" use="required" />
23       <xs:attribute name="created" type="xs:dateTime" use="required" />
24     </xs:complexType>
25   </xs:element>
26
27   <xs:element name="project">
28     <xs:complexType>
29       <xs:attribute name="name" type="xs:string" use="required" />
30       <xs:attribute name="created" type="xs:dateTime" use="required" />
31     </xs:complexType>
32   </xs:element>
33
34 </xs:schema>
```

Listing B.4: XML schema for timestamp when project/package was updated

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```
3
4   <xs:element name="latest_updated">
5     <xs:annotation>
6       <xs:documentation>
7         Package or project with timestamp of the last update.
8       </xs:documentation>
9     </xs:annotation>
10    <xs:complexType>
11      <xs:choice minOccurs="1" maxOccurs="1">
12        <xs:element ref="package" minOccurs="1" maxOccurs="1" />
13        <xs:element ref="project" minOccurs="1" maxOccurs="1" />
14      </xs:choice>
15    </xs:complexType>
16  </xs:element>
17
18  <xs:element name="package">
19    <xs:complexType>
20      <xs:attribute name="name" type="xs:string" use="required" />
21      <xs:attribute name="project" type="xs:string" use="required" />
22      <xs:attribute name="updated" type="xs:dateTime" use="required" />
23    </xs:complexType>
24  </xs:element>
25
26  <xs:element name="project">
27    <xs:complexType>
28      <xs:attribute name="name" type="xs:string" use="required" />
29      <xs:attribute name="updated" type="xs:dateTime" use="required" />
30    </xs:complexType>
31  </xs:element>
32
33 </xs:schema>
```

Listing B.5: XML schema for latest updated projects/packages

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4   <xs:element name="latest_updated">
5     <xs:annotation>
6       <xs:documentation>
7         List of packages and projects latest updated in the openSUSE build
8         service ordered by the updated timestamp.
9       </xs:documentation>
10    </xs:annotation>
11    <xs:complexType>
12      <xs:choice minOccurs="0" maxOccurs="unbounded">
13        <xs:element ref="package" minOccurs="0" maxOccurs="unbounded" />
14        <xs:element ref="project" minOccurs="0" maxOccurs="unbounded" />
15      </xs:choice>
16    </xs:complexType>
17  </xs:element>
18
19  <xs:element name="package">
20    <xs:complexType>
21      <xs:attribute name="name" type="xs:string" use="required" />
22      <xs:attribute name="project" type="xs:string" use="required" />
23      <xs:attribute name="updated" type="xs:dateTime" use="required" />
```

```
    </xs:complexType>
  </xs:element>

  <xs:element name="project">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required" />
      <xs:attribute name="updated" type="xs:dateTime" use="required" />
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Listing B.6: XML schema for the rating score of a single project/package

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

  <xs:annotation>
    <xs:documentation>
      This schema describes the format for ratings of objects
      (packages/projects) of the opensuse build service.
    </xs:documentation>
  </xs:annotation>


  <xs:complexType name="rating" mixed="true">
    <xs:annotation>
      <xs:documentation>
        This element contains the rating score and some (optional) attributes
        to identify the rated object.
      </xs:documentation>
    </xs:annotation>
    <xs:attribute name="project" type="xs:string" />
    <xs:attribute name="package" type="xs:string" />
    <xs:attribute name="count" type="xs:nonNegativeInteger" />
    <xs:attribute name="user_score" type="scoreInteger">
      <xs:annotation>
        <xs:documentation>
          This is the value which the currently logged in user gave
          this project/package.
        </xs:documentation>
      </xs:annotation>
    </xs:attribute>
  </xs:complexType>

  <xs:element name="rating" type="rating" />

  <xs:simpleType name="scoreInteger">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="5"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

Listing B.7: XML schema for the highest rated projects/packages

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="highest_rated">
    <xs:annotation>
      <xs:documentation>
        List of the highest rated packages and projects of the
        openSUSE build service ordered by rating.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="package" minOccurs="0" maxOccurs="unbounded" />
        <xs:element ref="project" minOccurs="0" maxOccurs="unbounded" />
      </xs:choice>
    </xs:complexType>
  </xs:element>

  <xs:element name="package">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required" />
      <xs:attribute name="project" type="xs:string" use="required" />
      <xs:attribute name="count" type="xs:nonNegativeInteger" use="required">
        <xs:annotation>
          <xs:documentation>
            Count of votes / ratings for this package.
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="score" type="scoreFloat" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:element name="project">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required" />
      <xs:attribute name="count" type="xs:nonNegativeInteger" use="required">
        <xs:annotation>
          <xs:documentation>
            Count of votes / ratings for this project.
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="score" type="scoreFloat" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:simpleType name="scoreFloat">
    <xs:restriction base="xs:float">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="5"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

Listing B.8: XML schema for activity of a single project/package

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="activity">
    <xs:annotation>
      <xs:documentation>
        Package or project with activity value.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:choice minOccurs="1" maxOccurs="1">
        <xs:element ref="package" minOccurs="1" maxOccurs="1" />
        <xs:element ref="project" minOccurs="1" maxOccurs="1" />
      </xs:choice>
    </xs:complexType>
  </xs:element>

  <xs:element name="package">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required" />
      <xs:attribute name="project" type="xs:string" use="required" />
      <xs:attribute name="activity" type="xs:float" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:element name="project">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required" />
      <xs:attribute name="activity" type="xs:float" use="required" />
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Listing B.9: XML schema for most active projects/packages

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="most_active">
    <xs:annotation>
      <xs:documentation>
        List of most active packages or projects in the openSUSE build
        service ordered activity.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:choice>
        <xs:sequence>
          <xs:element ref="package" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:sequence>
          <xs:element ref="project" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
      </xs:choice>
    </xs:complexType>
```

```
21   </xs:element>
22
23   <xs:element name="package">
24     <xs:complexType>
25       <xs:attribute name="name" type="xs:string" use="required" />
26       <xs:attribute name="project" type="xs:string" use="required" />
27       <xs:attribute name="activity" type="activityFloat" use="required" />
28       <xs:attribute name="update_count" type="xs:nonNegativeInteger" use="required"
             >
29         <xs:annotation>
30           <xs:documentation>
31             Count of updates a package already had.
32           </xs:documentation>
33         </xs:annotation>
34       </xs:attribute>
35     </xs:complexType>
36   </xs:element>
37
38   <xs:element name="project">
39     <xs:complexType>
40       <xs:attribute name="name" type="xs:string" use="required" />
41       <xs:attribute name="activity" type="activityFloat" use="required" />
42       <xs:attribute name="packages" type="xs:nonNegativeInteger" use="required">
43         <xs:annotation>
44           <xs:documentation>
45             Count of packages this project has.
46           </xs:documentation>
47         </xs:annotation>
48       </xs:attribute>
49     </xs:complexType>
50   </xs:element>
51
52   <xs:simpleType name="activityFloat">
53     <xs:restriction base="xs:float">
54       <xs:minInclusive value="0"/>
55       <xs:maxInclusive value="100"/>
56     </xs:restriction>
57   </xs:simpleType>
58
59 </xs:schema>
```

Listing B.10: XML schema for a list of top downloaded files

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4    <xs:element name="download_counter">
5      <xs:annotation>
6        <xs:documentation>
7          Download counter - top build service downloads.
8        </xs:documentation>
9      </xs:annotation>
10     <xs:complexType>
11       <xs:sequence>
12         <xs:element ref="count" minOccurs="0" maxOccurs="unbounded" />
13       </xs:sequence>
14       <xs:attribute name="first" use="required" type="xs:dateTime">
```

```
15        <xs:annotation>
16          <xs:documentation>
17            Timestamp of first counted download (of all counters).
18          </xs:documentation>
19        </xs:annotation>
20      </xs:attribute>
21      <xs:attribute name="last"  use="required" type="xs:dateTime">
22        <xs:annotation>
23          <xs:documentation>
24            Timestamp of last counted download (of all counters).
25          </xs:documentation>
26        </xs:annotation>
27      </xs:attribute>
28      <xs:attribute name="sum"   use="required" type="xs:nonNegativeInteger">
29        <xs:annotation>
30          <xs:documentation>
31            Sum of all counted downloads of the selected objects
32            (project/package/repo/arch).
33          </xs:documentation>
34        </xs:annotation>
35      </xs:attribute>
36      <xs:attribute name="all"   use="required" type="xs:nonNegativeInteger">
37        <xs:annotation>
38          <xs:documentation>
39            Sum of all counted downloads (of the whole build service).
40          </xs:documentation>
41        </xs:annotation>
42      </xs:attribute>
43    </xs:complexType>
44  </xs:element>
45
46  <xs:element name="count">
47    <xs:complexType mixed="true">
48      <xs:attribute name="project"      type="xs:string" use="required" />
49      <xs:attribute name="repository"   type="xs:string" use="required" />
50      <xs:attribute name="package"      type="xs:string" use="required" />
51      <xs:attribute name="architecture" type="xs:string" use="required" />
52      <xs:attribute name="filename"     type="xs:string" use="required" />
53      <xs:attribute name="filetype"     type="xs:string" use="required" />
54      <xs:attribute name="version"      type="xs:string" use="required" />
55      <xs:attribute name="release"      type="xs:string" use="required" />
56    </xs:complexType>
57  </xs:element>
58
59 </xs:schema>
```

Listing B.11: XML schema for a list of top downloaded projects

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4   <xs:element name="download_counter">
5     <xs:annotation>
6       <xs:documentation>
7         Download counter - top build service downloads.
8       </xs:documentation>
9     </xs:annotation>
```

```
10      <xs:complexType>
11        <xs:sequence>
12          <xs:element ref="count" minOccurs="0" maxOccurs="unbounded" />
13        </xs:sequence>
14        <xs:attribute name="first" use="required" type="xs:dateTime">
15          <xs:annotation>
16            <xs:documentation>
17              Timestamp of first counted download (of all counters).
18            </xs:documentation>
19          </xs:annotation>
20        </xs:attribute>
21        <xs:attribute name="last"  use="required" type="xs:dateTime">
22          <xs:annotation>
23            <xs:documentation>
24              Timestamp of last counted download (of all counters).
25            </xs:documentation>
26          </xs:annotation>
27        </xs:attribute>
28        <xs:attribute name="sum"   use="required" type="xs:nonNegativeInteger">
29          <xs:annotation>
30            <xs:documentation>
31              Sum of all counted downloads (of the whole build service).
32            </xs:documentation>
33          </xs:annotation>
34        </xs:attribute>
35      </xs:complexType>
36    </xs:element>
37
38    <xs:element name="count">
39      <xs:complexType mixed="true">
40        <xs:attribute name="files"     type="xs:nonNegativeInteger" use="required">
41          <xs:annotation>
42            <xs:documentation>
43              Sum of all different files that are in this container
44              (project, package, repo or arch).
45            </xs:documentation>
46          </xs:annotation>
47        </xs:attribute>
48        <xs:attribute name="project" type="xs:string" use="optional" />
49        <xs:attribute name="package" type="xs:string" use="optional" />
50        <xs:attribute name="repo"    type="xs:string" use="optional" />
51        <xs:attribute name="arch"    type="xs:string" use="optional" />
52      </xs:complexType>
53    </xs:element>
54
55  </xs:schema>
```

Listing B.12: XML schema for redirector statistics import format

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3    elementFormDefault="qualified">
4
5
6    <xs:element name="redirect_stats">
7      <xs:annotation>
8        <xs:documentation>
```

```
9         Detailed download statistics from redirector.
10      </xs:documentation>
11    </xs:annotation>
12    <xs:complexType>
13      <xs:sequence>
14        <xs:element ref="project" minOccurs="0" maxOccurs="unbounded" />
15      </xs:sequence>
16    </xs:complexType>
17  </xs:element>
18
19  <xs:element name="project">
20    <xs:complexType mixed="true">
21      <xs:sequence>
22        <xs:element ref="package" minOccurs="0" maxOccurs="unbounded" />
23      </xs:sequence>
24      <xs:attribute name="name" use="required" />
25    </xs:complexType>
26  </xs:element>
27
28  <xs:element name="package">
29    <xs:complexType mixed="true">
30      <xs:sequence>
31        <xs:element ref="repository" minOccurs="0" maxOccurs="unbounded" />
32      </xs:sequence>
33      <xs:attribute name="name" use="required" />
34    </xs:complexType>
35  </xs:element>
36
37  <xs:element name="repository">
38    <xs:complexType mixed="true">
39      <xs:sequence>
40        <xs:element ref="arch" minOccurs="0" maxOccurs="unbounded" />
41      </xs:sequence>
42      <xs:attribute name="name" use="required" />
43    </xs:complexType>
44  </xs:element>
45
46  <xs:element name="arch">
47    <xs:complexType mixed="true">
48      <xs:sequence>
49        <xs:element ref="count" minOccurs="0" maxOccurs="unbounded" />
50      </xs:sequence>
51      <xs:attribute name="name" use="required" />
52    </xs:complexType>
53  </xs:element>
54
55  <xs:element name="count" >
56    <xs:complexType mixed="true">
57      <xs:attribute name="filename" use="required" type="xs:string" />
58      <xs:attribute name="filetype" use="required" type="xs:string" />
59      <xs:attribute name="version" use="required" type="xs:string" />
60      <xs:attribute name="release" use="required" type="xs:string" />
61      <xs:attribute name="created_at" use="required" type="xs:dateTime" />
62      <xs:attribute name="counted_at" use="required" type="xs:dateTime" />
63    </xs:complexType>
64  </xs:element>
65
```

```
</xs:schema>
```

Listing B.13: SQL database table for ratings

```
CREATE TABLE `ratings` (
  `id` int(11) NOT NULL auto_increment,
  `score` int(11) default NULL,
  `object_id` int(11) default NULL,
  `object_type` varchar(255) default NULL,
  `created_at` datetime default NULL,
  `user_id` int(11) default NULL,
  PRIMARY KEY  (`id`),
  KEY `object` (`object_id`),
  KEY `user` (`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Listing B.14: Essential excerpt of webclient/config/environment.rb, where the ActiveXML models relevant for the statistics are configured

```
ActiveXML::Base.config do |conf|
  conf.setup_transport do |map|
    map.default_server :rest, "#{FRONTEND_HOST}:#{FRONTEND_PORT}"

    # Statistics

    map.connect :latestadded, 'rest:///statistics/latest_added?:limit',
      :specific => 'rest:///statistics/added_timestamp/:project/:package'

    map.connect :latestupdated, 'rest:///statistics/latest_updated?:limit',
      :specific => 'rest:///statistics/updated_timestamp/:project/:package'

    map.connect :downloadcounter, 'rest:///statistics/download_counter' +
      '?:project&:package&:arch&:repo&:group_by&:limit'

    map.connect :rating, 'rest:///statistics/rating/:project/:package',
      :all => 'rest:///statistics/highest_rated?:limit'

    map.connect :mostactive, 'rest:///statistics/most_active?:type&:limit',
      :specific => 'rest:///statistics/activity/:project/:package'

    map.connect :globalcounters, 'rest:///statistics/global_counters',
      :all => 'rest:///statistics/global_counters'

  end
end
```

Listing B.15: Frontend controller part, to return latest_added packages and projects

```
def latest_added

  packages = DbPackage.find :all,
    :from => 'db_packages pac, db_projects pro',
    :select => 'pac.name, pac.created_at, pro.name AS project_name',
    :conditions => 'pro.id = pac.db_project_id',
    :order => 'created_at DESC, name', :limit => @limit
```

```
8   projects = DbProject.find :all,
9     :select => 'name, created_at',
10    :order => 'created_at DESC, name', :limit => @limit
11
12  list = []
13  projects.each { |project| list << project }
14  packages.each { |package| list << package }
15  list.sort! { |a,b| b.created_at <=> a.created_at }
16
17  @list = list[0..@limit-1]
18 end
```

Listing B.16: Frontend controller part, to return the added_timestamp of a package or project

```
1 def added_timestamp
2   @project = DbProject.find_by_name( params[:project] )
3   @package = DbPackage.find( :first, :conditions =>
4     [ 'name=? AND db_project_id=?', params[:package], @project.id ]
5   ) if @project
6 end
```

Listing B.17: Frontend view, to to build latest_added XML data

```
1 xml.instruct!
2
3 xml.latest_added do
4   @list.each do |item|
5
6     ### item is a package
7     if item.instance_of? DbPackage
8       xml.package(
9         :name => item.name,
10        :project => item.project_name,
11        :created => item.created_at.xmlschema
12      )
13    end
14
15    ### item is a project
16    if item.instance_of? DbProject
17      xml.project(
18        :name => item.name,
19        :created => item.created_at.xmlschema
20      )
21    end
22
23  end
24 end
```

Listing B.18: Frontend view, to to build added_timestamp XML data

```
1 xml.instruct!
2
3 xml.latest_added do
4
5   if @package
```

```
6
7     xml.package(
8       :name => @package.name,
9       :project => @package.db_project.name,
10      :created => @package.created_at.xmlschema
11    )
12
13  elsif @project
14
15    xml.project(
16      :name => @project.name,
17      :created => @project.created_at.xmlschema
18    )
19
20  end
21
22 end
```

Listing B.19: Webclient Helper for the statistic views

```
1  module StatisticsHelper
2
3    def statistics_limit_form( action, title='' )
4      out = ''
5      out << start_form_tag( nil, :method => :get )
6      out << statistics_limit_select( "#{title} " )
7      out << hidden_field_tag( 'more', params[:more] )
8      out << hidden_field_tag( 'package', @package ) if @package
9      out << hidden_field_tag( 'project', @project ) if @project
10     out << hidden_field_tag( 'repo', @repo) if @repo
11     out << hidden_field_tag( 'arch', @arch) if @arch
12     out << image_submit_tag( 'system-search' )
13     out << image_tag( 'rotating-tail.gif', :border => 0, :style => 'display: none;'
           , :id => 'spinner' )
14     out << end_form_tag
15     out << observe_field( :limit, :update => action,
16       :url => { :action  => action, :more => true,
17         :project => @project, :package => @package,
18         :arch => @arch, :repo => @repo
19       },
20       :with => "'limit=' + escape(value)", :loading => "Element.show('spinner')",
21       :complete => "Element.hide('spinner')"
22     )
23     return out
24   end
25
26   def statistics_limit_select( left_text='', right_text='' )
27     out = ''
28     out << "#{left_text}"
29     out << select_tag( 'limit', options_for_select(
30       [['...',10],[25,25],[50,50],[100,100],[250,250],[500,500]])
31       )
32     out << javascript_tag( "document.getElementById('limit').focus();" )
33     out << "#{right_text}"
34     return out
35   end
36
```

```
37   def link_to_package_view( name, project, title='', length=15 )
38     link_to image_tag( 'package', :border => 0 ) + " #{shorten_text(name, length)}"
         ,
39       { :action => 'view', :controller => 'package',
40         :package => name, :project => project },
41       :title => "Package #{name} #{title}"
42   end
43
44   def link_to_project_view( name, title='', length=15 )
45     link_to image_tag( 'project', :border => 0 ) + " #{shorten_text(name, length)}"
         ,
46       { :action => 'view', :controller => 'project',
47         :project => name },
48       :title => "Project #{name} #{title}"
49   end
50
51   def link_to_mainpage
52     link_to image_tag( 'start', :border => 0 ) + ' back to main page...',
53       :controller => 'statistics'
54   end
55
56 end
```

Listing B.20: General webclient Helper

```
1  # Methods added to this helper will be available to all templates in the
       application.
2  module ApplicationHelper
3
4    def link_to_project project
5      link_to project, :controller => "project", :action => :show,
6        :project => project
7    end
8
9    def link_to_package project, package
10     link_to package, :controller => "package", :action => :show,
11       :project => project, :package => package
12   end
13
14   def shorten_text( text, length=15 )
15   text = text[0..length-1] + '...' if text.length > length
16     return text
17   end
18
19   def focus_id( id )
20     javascript_tag(
21       "document.getElementById('#{id}').focus();"
22     )
23   end
24
25   def focus_and_select_id( id )
26     javascript_tag(
27       "document.getElementById('#{id}').focus();" +
28       "document.getElementById('#{id}').select();"
29     )
30   end
31
```

```
32   def min_votes_for_rating
33     MIN_VOTES_FOR_RATING
34   end
35
36 end
```

Listing B.21: Frontend controller methods that care for rating

```
1  def rating
2    @package = params[:package]
3    @project = params[:project]
4
5    begin
6      object = DbProject.find_by_name @project
7      object = DbPackage.find :first, :conditions =>
8        [ 'name=? AND db_project_id=?', @package, object.id ] if @package
9      throw if object.nil?
10   rescue
11     @package = @project = @rating = object = nil
12     return
13   end
14
15   if request.get?
16
17     @rating = object.rating( @http_user.id )
18
19   elsif request.put?
20
21     # try to get previous rating of this user for this object
22     previous_rating = Rating.find :first, :conditions => [
23       'object_type=? AND object_id=? AND user_id=?',
24       object.class.name, object.id, @http_user.id
25     ]
26     data = ActiveXML::Base.new( request.raw_post )
27     if previous_rating
28       # update previous rating
29       previous_rating.score = data.to_s.to_i
30       previous_rating.save
31     else
32       # create new rating entry
33       begin
34         rating = Rating.new
35         rating.score = data.to_s.to_i
36         rating.object_type = object.class.name
37         rating.object_id = object.id
38         rating.user_id = @http_user.id
39         rating.save
40       rescue
41         render_error :status => 400, :errorcode => "error setting rating",
42           :message => "rating not saved"
43         return
44       end
45     end
46     render_ok
47
48   else
49     render_error :status => 400, :errorcode => "invalid_method",
```

```
50        :message => "only GET or PUT method allowed for this action"
51    end
52  end
53
54
55  def highest_rated
56    ratings = Rating.find :all,
57      :select => 'object_id, object_type, count(score) as count,' +
58        'sum(score)/count(score) as score_calculated',
59      :group => 'object_id, object_type',
60      :order => 'score_calculated DESC'
61    ratings = ratings.delete_if { |r| r.count.to_i < min_votes_for_rating }
62    @ratings = ratings[0..@limit-1]
63  end
```

Listing B.22: Script to create download statistics XML file from the redirector database

```
1  #!/usr/bin/env ruby
2  # this script generates a xml file from the 'redirect_stats'-table
3  # of the opensuse download redirector
4
5  #-------------------------------------
6  # CONFIG
7  db_host  = 'downloadserver.suse.de'
8  db_user  = 'dbuser'
9  db_pass  = 'verysecret'
10 db_name  = 'redirector'
11 db_table = 'redirect_stats'
12 filename = 'redirect_stats.xml'
13 #-------------------------------------
14
15 require 'rubygems'
16 require_gem 'activesupport'
17 require_gem 'activerecord'
18
19 # connect to database
20 ActiveRecord::Base.establish_connection(
21   :adapter  => 'mysql',
22   :host     => db_host,
23   :username => db_user,
24   :password => db_pass,
25   :database => db_name
26 )
27
28 # define model for statistics entries
29 class RedirectStats < ActiveRecord::Base ; end
30
31 # get all entries from database
32 begin
33   db_stats = RedirectStats.find :all
34 rescue
35   puts "ERROR while getting redirect_stats from database. Abort."
36   exit false
37 end
38
```

```
39 # build nested hash with counters
40 stats = {}
41 db_stats.each do |s|
42   stats[ s.project ] ||= {}
43   stats[ s.project ][ s.package ] ||= {}
44   stats[ s.project ][ s.package ][ s.repository ] ||= {}
45   stats[ s.project ][ s.package ][ s.repository ][ s.arch ] ||= []
46   stats[ s.project ][ s.package ][ s.repository ][ s.arch ] << s
47 end
48
49 # initialize xml-builder, send result to the xml_output variable
50 xml = Builder::XmlMarkup.new( :target => xml_output='', :indent => 2 )
51
52 # generate xml
53 xml.instruct!
54 xml.redirect_stats do
55   stats.each_pair do |project_name, project|
56
57     xml.project( :name => project_name ) do
58       project.each_pair do |package_name, package|
59
60         xml.package( :name => package_name ) do
61           package.each_pair do |repo_name, repo|
62
63             xml.repository( :name => repo_name ) do
64               repo.each_pair do |arch_name, arch|
65
66                 xml.arch( :name => arch_name ) do
67                   arch.each do |counter|
68
69                     xml.count(
70                       counter.count,
71                       :filename   => counter.filename,
72                       :filetype   => counter.filetype,
73                       :version    => counter.version,
74                       :release    => counter.release,
75                       :created_at => counter.created_at.xmlschema,
76                       :counted_at => counter.counted_at.xmlschema
77                     )
78
79                   end # each_counter
80                 end # 'arch' xml tag
81
82               end # each_arch
83             end # 'repository' xml tag
84
85           end # each_repo
86         end # 'package' xml tag
87
88       end # each_package
89     end # 'project' xml tag
90
91   end # each_project
92 end # outer 'redirect_stats' xml tag
93
94 # write xml output to file
95 file = File.new( filename, 'w+' )
```

```
96 file << xml_output
97 puts "redirector statistics written as xml file to #{file.path}, size is #{file.
       stat.size/1024} Kb."
98 file.close
```

Listing B.23: Functional tests implemented in the frontend

```
1  require File.dirname(__FILE__) + '/../test_helper'
2  require 'statistics_controller'
3
4  class StatisticsController; def rescue_action(e) raise e end; end
5  class StatisticsControllerTest < Test::Unit::TestCase
6
7    fixtures :db_projects, :db_packages, :download_stats, :repositories, :
         architectures
8
9    def setup
10     @controller = StatisticsController.new
11     @request    = ActionController::TestRequest.new
12     @response   = ActionController::TestResponse.new
13   end
14
15   def test_latest_added
16     prepare_request_with_user @request, 'tom', 'thunder'
17     get :latest_added
18     assert_response :success
19     assert_tag :tag => 'latest_added', :child => { :tag => 'project' }
20     assert_tag :tag => 'project', :attributes => {
21       :name => "kde4",
22       :created => "2008-04-28T05:05:05+02:00",
23     }
24   end
25
26   def test_latest_updated
27     prepare_request_with_user @request, 'tom', 'thunder'
28     get :latest_updated
29     assert_response :success
30     assert_tag :tag => 'latest_updated', :child => { :tag => 'project' }
31     assert_tag :tag => 'project', :attributes => {
32       :name => "kde4",
33       :updated => "2008-04-28T06:06:06+02:00",
34     }
35   end
36
37   def test_download_counter
38     prepare_request_with_user @request, 'tom', 'thunder'
39     get :download_counter
40     assert_response :success
41     assert_tag :tag => 'download_counter', :child => { :tag => 'count' }
42     assert_tag :tag => 'download_counter', :attributes => { :sum => 9302 }
43     assert_tag :tag => 'count', :attributes => {
44       :project => 'Apache',
45       :package => 'apache2',
46       :repository => 'SUSE_Linux_10.1',
47       :architecture => 'x86_64'
48     }
49     assert_tag :tag => 'count', :content => '4096'
```

```
50   end
51
52   def test_download_counter_group_by
53     prepare_request_with_user @request, 'tom', 'thunder'
54     # without project- & package-filter
55     get :download_counter, { 'group_by' => 'project' }
56     assert_response :success
57     assert_tag :tag => 'download_counter', :child => { :tag => 'count' }
58     assert_tag :tag => 'download_counter', :attributes => { :all => 9302 }
59     assert_tag :tag => 'count', :attributes => {
60       :project => 'Apache', :files => '9'
61     }, :content => '8806'
62     # with project- & package-filter
63     get :download_counter, {
64       'project' => 'Apache', 'package' => 'apache2', 'group_by' => 'arch'
65     }
66     assert_response :success
67     assert_tag :tag => 'download_counter', :child => { :tag => 'count' }
68     assert_tag :tag => 'download_counter',
69       :attributes => { :all => 9302 }
70     assert_tag :tag => 'count', :attributes => {
71       :arch => 'x86_64', :files => '6'
72     }, :content => '5537'
73   end
74
75   def test_most_active
76     prepare_request_with_user @request, 'tom', 'thunder'
77     # get most active packages
78     get :most_active, { :type => 'packages' }
79     assert_response :success
80     assert_tag :tag => 'most_active', :child => { :tag => 'package' }
81     assert_tag :tag => 'package', :attributes => {
82       :name => "x11vnc",
83       :project => "home:dmayr",
84       :update_count => 0
85     }
86     # get most active projects
87     get :most_active, { :type => 'projects' }
88     assert_response :success
89     assert_tag :tag => 'most_active', :child => { :tag => 'project' }
90     assert_tag :tag => 'project', :attributes => {
91       :name => "home:dmayr",
92       :packages => 1
93     }
94   end
95
96   def test_highest_rated
97     prepare_request_with_user @request, 'tom', 'thunder'
98     get :highest_rated
99     assert_response :success
100  end
101 end
```

Listing B.24: StreamHandler part of the frontend statistics controller to import the download statistics

```
1
2  # StreamHandler for parsing incoming download_stats / redirect_stats (xml)
3  class StreamHandler
4    include StreamListener
5
6    attr_accessor :errors
7
8    def initialize
9      @errors = []
10     # build hashes for caching id-/name- combinations
11     projects = DbProject.find :all, :select => 'id, name'
12     packages = DbPackage.find :all, :select => 'id, name, db_project_id'
13     repos  =  Repository.find :all, :select => 'id, name, db_project_id'
14     archs = Architecture.find :all, :select => 'id, name'
15     @project_hash = @package_hash = @repo_hash = @arch_hash = {}
16     projects.each { |p| @project_hash[ p.name ] = p.id }
17     packages.each { |p| @package_hash[ [ p.name, p.db_project_id ] ] = p.id }
18     repos.each { |r| @repo_hash[ [ r.name, r.db_project_id ] ] = r.id }
19     archs.each { |a| @arch_hash[ a.name ] = a.id }
20   end
21
22   def tag_start name, attrs
23     case name
24     when 'project'
25       @@project = @project_hash[ attrs['name'] ]
26     when 'package'
27       @@package = @package_hash[ [ attrs['name'], @@project ] ]
28     when 'repository'
29       @@repo = @repo_hash[ [ attrs['name'], @@project ] ]
30     when 'arch'
31       unless @@arch = @arch_hash[ attrs['name'] ]
32         # create new architecture entry (db and hash)
33         arch = Architecture.new( :name => attrs['name'] )
34         arch.save
35         @arch_hash[ arch.name ] = arch.id
36         @@arch = @arch_hash[ arch.name ]
37       end
38     when 'count'
39       @@count = {
40         :filename => attrs['filename'],
41         :filetype => attrs['filetype'],
42         :version => attrs['version'],
43         :release => attrs['release'],
44         :created_at => attrs['created_at'],
45         :counted_at => attrs['counted_at']
46       }
47     end
48   end
49
50   def text( text )
51     text.strip!
52     return if text == ''
53     unless @@project and @@package and @@repo and @@arch and @@count
54       @errors << { :project => @@project, :package => @@package,
55         :repo => @@repo, :arch => @@arch, :count => @@count }
56       return
57     end
```

```
58
59      # try to find existing entry in database
60      ds = DownloadStat.find :first, :conditions => [
61        'db_project_id=? AND db_package_id=? AND repository_id=? AND ' +
62        'architecture_id=? AND filename=? AND filetype=? AND ' +
63        'version=? AND download_stats.release=?',
64        @@project, @@package, @@repo, @@arch,
65        @@count[:filename], @@count[:filetype],
66        @@count[:version], @@count[:release]
67      ]
68      if ds
69        # entry found, update it if necessary ...
70        if ds.count.to_i != text.to_i
71          ds.count = text
72          ds.counted_at = @@count[:counted_at]
73          ds.save
74        end
75      else
76        # create new entry - we do this directly per sql statement, because
77        # that's much faster than through ActiveRecord objects
78        DownloadStat.connection.insert "\
79        INSERT INTO download_stats ( \
80          `db_project_id`, `db_package_id`, `repository_id`, `architecture_id`,\
81          `filename`, `filetype`, `version`, `release`,\
82          `counted_at`, `created_at`, `count`\
83        ) VALUES(\
84          '#{@@project}', '#{@@package}', '#{@@repo}', '#{@@arch}',\
85          '#{@@count[:filename]}',    '#{@@count[:filetype]}',\
86          '#{@@count[:version]}',     '#{@@count[:release]}',\
87          '#{@@count[:counted_at]}', '#{@@count[:created_at]}',\
88          '#{text}'\
89        )", "Creating DownloadStat entry: "
90      end
91    end
92  end
```

Listing B.25: The complete frontend statistics controller

```
1
2   require 'rexml/document'
3   require "rexml/streamlistener"
4
5   class StatisticsController < ApplicationController
6
7
8     before_filter :get_limit, :only => [
9       :highest_rated, :most_active, :latest_added, :latest_updated,
10      :latest_built, :download_counter
11    ]
12
13    caches_action :highest_rated, :most_active, :latest_added, :latest_updated,
14      :latest_built, :download_counter
15
16    validate_action :redirect_stats => :redirect_stats
17
18
19    # StreamHandler for parsing incoming download_stats / redirect_stats (xml)
```

```
20  class StreamHandler
21    include REXML::StreamListener
22
23    attr_accessor :errors
24
25    def initialize
26      @errors = []
27      # build hashes for caching id-/name- combinations
28      projects = DbProject.find :all, :select => 'id, name'
29      packages = DbPackage.find :all, :select => 'id, name, db_project_id'
30      repos  =  Repository.find :all, :select => 'id, name, db_project_id'
31      archs = Architecture.find :all, :select => 'id, name'
32      @project_hash = @package_hash = @repo_hash = @arch_hash = {}
33      projects.each { |p| @project_hash[ p.name ] = p.id }
34      packages.each { |p| @package_hash[ [ p.name, p.db_project_id ] ] = p.id }
35      repos.each { |r| @repo_hash[ [ r.name, r.db_project_id ] ] = r.id }
36      archs.each { |a| @arch_hash[ a.name ] = a.id }
37    end
38
39    def tag_start name, attrs
40      case name
41      when 'project'
42        @@project_name = attrs['name']
43        @@project_id = @project_hash[ attrs['name'] ]
44      when 'package'
45        @@package_name = attrs['name']
46        @@package_id = @package_hash[ [ attrs['name'], @@project_id ] ]
47      when 'repository'
48        @@repo_name = attrs['name']
49        @@repo_id = @repo_hash[ [ attrs['name'], @@project_id ] ]
50      when 'arch'
51        @@arch_name = attrs['name']
52        unless @@arch_id = @arch_hash[ attrs['name'] ]
53          # create new architecture entry (db and hash)
54          arch = Architecture.new( :name => attrs['name'] )
55          arch.save
56          @arch_hash[ arch.name ] = arch.id
57          @@arch_id = @arch_hash[ arch.name ]
58        end
59      when 'count'
60        @@count = {
61          :filename => attrs['filename'],
62          :filetype => attrs['filetype'],
63          :version => attrs['version'],
64          :release => attrs['release'],
65          :created_at => attrs['created_at'],
66          :counted_at => attrs['counted_at']
67        }
68      end
69    end
70
71    def text( text )
72      text.strip!
73      return if text == ''
74      unless @@project_id and @@package_id and @@repo_id and @@arch_id and @@count
75        @errors << {
76          :project_id => @@project_id, :project_name => @@project_name,
```

```
77            :package_id => @@package_id, :package_name => @@package_name,
78            :repo_id => @@repo_id, :repo_name => @@repo_name,
79            :arch_id => @@arch_id, :arch_name => @@arch_name, :count => @@count
80          }
81          return
82        end
83
84        # lower the log level, prevent spamming the logfile
85        old_loglevel = DownloadStat.logger.level
86        DownloadStat.logger.level = Logger::ERROR
87
88        # try to find existing entry in database
89        ds = DownloadStat.find :first, :conditions => [
90          'db_project_id=? AND db_package_id=? AND repository_id=? AND ' +
91          'architecture_id=? AND filename=? AND filetype=? AND ' +
92          'version=? AND download_stats.release=?',
93          @@project_id, @@package_id, @@repo_id, @@arch_id,
94          @@count[:filename], @@count[:filetype],
95          @@count[:version], @@count[:release]
96        ]
97        if ds
98          # entry found, update it if necessary ...
99          if ds.count.to_i != text.to_i
100           ds.count = text
101           ds.counted_at = @@count[:counted_at]
102           ds.save
103         end
104       else
105         # create new entry - we do this directly per sql statement, because
106         # that's much faster than through ActiveRecord objects
107         DownloadStat.connection.insert "\
108         INSERT INTO download_stats ( \
109           `db_project_id`, `db_package_id`, `repository_id`, `architecture_id`,\
110           `filename`, `filetype`, `version`, `release`,\
111           `counted_at`, `created_at`, `count`\
112         ) VALUES(\
113           '#{@@project_id}', '#{@@package_id}', '#{@@repo_id}', '#{@@arch_id}',\
114           '#{@@count[:filename]}',    '#{@@count[:filetype]}',\
115           '#{@@count[:version]}',     '#{@@count[:release]}',\
116           '#{@@count[:counted_at]}', '#{@@count[:created_at]}',\
117           '#{text}'\
118         )", "Creating DownloadStat entry: "
119       end
120
121       # reset the log level
122       DownloadStat.logger.level = old_loglevel
123     end
124   end
125
126
127
128
129   def index
130     text =  "This is the statistics controller.<br />"
131     text += "See the api documentation for details."
132     render :text => text
133   end
```

```
134
135
136   def highest_rated
137     # set automatic action_cache expiry time limit
138     response.time_to_live = 10.minutes
139
140     ratings = Rating.find :all,
141       :select => 'object_id, object_type, count(score) as count,' +
142         'sum(score)/count(score) as score_calculated',
143       :group => 'object_id, object_type',
144       :order => 'score_calculated DESC'
145     ratings = ratings.delete_if { |r| r.count.to_i < min_votes_for_rating }
146     @ratings = ratings[0..@limit-1]
147   end
148
149
150   def rating
151     @package = params[:package]
152     @project = params[:project]
153
154     begin
155       object = DbProject.find_by_name @project
156       object = DbPackage.find :first, :conditions =>
157         [ 'name=? AND db_project_id=?', @package, object.id ] if @package
158       throw if object.nil?
159     rescue
160       @package = @project = @rating = object = nil
161       return
162     end
163
164     if request.get?
165
166       @rating = object.rating( @http_user.id )
167
168     elsif request.put?
169
170       # try to get previous rating of this user for this object
171       previous_rating = Rating.find :first, :conditions => [
172         'object_type=? AND object_id=? AND user_id=?',
173         object.class.name, object.id, @http_user.id
174       ]
175       data = ActiveXML::Base.new( request.raw_post )
176       if previous_rating
177         # update previous rating
178         previous_rating.score = data.to_s.to_i
179         previous_rating.save
180       else
181         # create new rating entry
182         begin
183           rating = Rating.new
184           rating.score = data.to_s.to_i
185           rating.object_type = object.class.name
186           rating.object_id = object.id
187           rating.user_id = @http_user.id
188           rating.save
189         rescue
190           render_error :status => 400, :errorcode => "error setting rating",
```

```
191             :message => "rating not saved"
192           return
193         end
194       end
195       render_ok
196
197     else
198       render_error :status => 400, :errorcode => "invalid_method",
199         :message => "only GET or PUT method allowed for this action"
200     end
201   end
202
203
204   def download_counter
205     # set automatic action_cache expiry time limit
206     response.time_to_live = 30.minutes
207
208     # initialize @stats
209     @stats = []
210
211     # get total count of all downloads
212     @all = DownloadStat.find( :first, :select => 'sum(count) as sum' ).sum
213     @all = 0 unless @all
214
215     # get timestamp of first counted entry
216     time = DownloadStat.find( :first, :select => 'min(created_at) as ts' ).ts
217     time ? @first = Time.parse( time ).xmlschema : @first = Time.now.xmlschema
218
219     # get timestamp of last counted entry
220     time = DownloadStat.find( :first, :select => 'max(counted_at) as ts' ).ts
221     time ? @last = Time.parse( time ).xmlschema : @last = Time.now.xmlschema
222
223     if @group_by_mode = params[:group_by]
224     # if in group_by_mode, then we concatenate download_stats entries
225
226       # generate parts of the sql statement
227       case @group_by_mode
228       when 'project'
229         from = 'db_projects pro'
230         select = 'pro.name as obj_name'
231         group_by = 'db_project_id'
232         conditions = 'ds.db_project_id=pro.id'
233       when 'package'
234         from = 'db_packages pac, db_projects pro'
235         select = 'pac.name as obj_name, pro.name as pro_name'
236         group_by = 'db_package_id'
237         conditions = 'ds.db_package_id=pac.id AND ds.db_project_id=pro.id'
238       when 'repo'
239         from = 'repositories repo, db_projects pro'
240         select = 'repo.name as obj_name, pro.name as pro_name'
241         group_by = 'repository_id'
242         conditions = 'ds.repository_id=repo.id AND ds.db_project_id=pro.id'
243       when 'arch'
244         from = 'architectures arch'
245         select = 'arch.name as obj_name'
246         group_by = 'architecture_id'
247         conditions = 'ds.architecture_id=arch.id'
```

```
248       else
249         @cstats = nil
250         return
251       end
252
253       # execute the sql query
254       @stats = DownloadStat.find :all,
255         :from => 'download_stats ds, ' + from,
256         :select => 'ds.*, ' + select + ', ' +
257           'sum(ds.count) as counter_sum, count(ds.id) as files_count',
258         :conditions => conditions,
259         :order => 'counter_sum DESC, files_count ASC',
260         :group => group_by,
261         :limit => @limit
262
263     else
264     # we are not in group_by_mode, so we return full download_stats data
265
266       # get objects
267       prj = DbProject.find_by_name params[:project]
268       pac = DbPackage.find :first, :conditions => [
269         'name=? AND db_project_id=?', params[:package], prj.id
270       ] if prj
271       repo = Repository.find :first, :conditions => [
272         'name=? AND db_project_id=?', params[:repo], prj.id
273       ] if prj
274       arch = Architecture.find_by_name params[:arch]
275
276       # return immediately, if any object is invalid / not found
277       return if not prj  and not params[:project].nil?
278       return if not pac  and not params[:package].nil?
279       return if not repo and not params[:repo].nil?
280       return if not arch and not params[:arch].nil?
281
282       # create filter, if parameters given & objects found
283       filter = ''
284       filter += " AND ds.db_project_id=#{prj.id}" if prj
285       filter += " AND ds.db_package_id=#{pac.id}" if pac
286       filter += " AND ds.repository_id=#{repo.id}" if repo
287       filter += " AND ds.architecture_id=#{arch.id}" if arch
288
289       # get download_stats entries
290       @stats = DownloadStat.find :all,
291         :from => 'download_stats ds, db_projects pro, db_packages pac, ' +
292           'architectures arch, repositories repo',
293         :select => 'ds.*, pro.name as pro_name, pac.name as pac_name, ' +
294           'arch.name as arch_name, repo.name as repo_name',
295         :conditions => 'ds.db_project_id=pro.id AND ds.db_package_id=pac.id' +
296           ' AND ds.architecture_id=arch.id AND ds.repository_id=repo.id' +
297           filter,
298         :order => 'ds.count DESC',
299         :limit => @limit
300
301       # get sum of counts
302       @sum = DownloadStat.find( :first,
303         :from => 'download_stats ds',
304         :select => 'sum(count) as overall_counter',
```

```
305         :conditions => '1=1' + filter
306       ).overall_counter
307     end
308   end
309
310
311   def redirect_stats
312
313     # check permissions
314     unless permissions.set_download_counters
315       render_error :status => 403, :errorcode => "permission denied",
316         :message => "download counters cannot be set, insufficient permissions"
317       return
318     end
319
320     # get download statistics from redirector as xml
321     if request.put?
322       data = request.raw_post
323
324       # parse the data
325       streamhandler = StreamHandler.new
326       logger.debug "download_stats import starts now ..."
327       REXML::Document.parse_stream( data, streamhandler )
328       logger.debug "download_stats import is finished."
329
330       if streamhandler.errors
331         logger.debug "prepare download_stats warning message..."
332         err_count = streamhandler.errors.length
333         dayofweek = Time.now.strftime('%u')
334         logfile = "log/download_statistics_import_warnings-#{dayofweek}.log"
335         msg  = "WARNING: #{err_count} redirect_stats were not imported.\n"
336         msg += "(for details see logfile #{logfile})"
337
338         f = File.open logfile, 'w'
339         streamhandler.errors.each do |e|
340           f << "project: #{e[:project_name]}=#{e[:project_id] or '*UNKNOWN*'}  "
341           f << "package: #{e[:package_name]}=#{e[:package_id] or '*UNKNOWN*'}  "
342           f << "repo: #{e[:repo_name]}=#{e[:repo_id] or '*UNKNOWN*'}  "
343           f << "arch: #{e[:arch_name]}=#{e[:arch_id] or '*UNKNOWN*'}\t"
344           f << "(#{e[:count][:filename]}:#{e[:count][:version]}:"
345           f << "#{e[:count][:release]}:#{e[:count][:filetype]})\n"
346         end
347         f.close
348
349         logger.warn "\n\n#{msg}\n\n"
350         render_ok msg # render_ok with msg text in details
351       else
352         render_ok
353       end
354
355     else
356       render_error :status => 400, :errorcode => "only_put_method_allowed",
357         :message => "only PUT method allowed for this action"
358       logger.debug "Tried to access download_stats via '#{request.method}' - not
              allowed!"
359       return
360     end
```

```
361   end
362
363
364   def most_active
365     # set automatic action_cache expiry time limit
366     response.time_to_live = 30.minutes
367
368     @type = params[:type] or @type = 'packages'
369
370     if @type == 'projects'
371       # get all packages including activity values
372       @packages = DbPackage.find :all,
373         :from => 'db_packages pac, db_projects pro',
374         :conditions => 'pac.db_project_id = pro.id',
375         :select => 'pac.*, pro.name AS project_name,' +
376           "( #{DbPackage.activity_algorithm} ) AS act_tmp," +
377           'IF( @activity<0, 0, @activity ) AS activity_value'
378       # count packages per project and sum up activity values
379       projects = {}
380       @packages.each do |package|
381         pro = package.project_name
382         projects[pro] ||= { :count => 0, :sum => 0 }
383         projects[pro][:count] += 1
384         projects[pro][:sum] += package.activity_value.to_f
385       end
386       # calculate average activity of packages per project
387       projects.each_key do |pro|
388         projects[pro][:activity] = projects[pro][:sum] / projects[pro][:count]
389       end
390       # sort by activity
391       @projects = projects.sort do |a,b|
392         b[1][:activity] <=> a[1][:activity]
393       end
394       # apply limit
395       @projects = @projects[0..@limit-1]
396
397     elsif @type == 'packages'
398       # get all packages including activity values
399       @packages = DbPackage.find :all,
400         :from => 'db_packages pac, db_projects pro',
401         :conditions => 'pac.db_project_id = pro.id',
402         :order => 'activity_value DESC',
403         :limit => @limit,
404         :select => 'pac.*, pro.name AS project_name,' +
405           "( #{DbPackage.activity_algorithm} ) AS act_tmp," +
406           'IF( @activity<0, 0, @activity ) AS activity_value'
407     end
408   end
409
410
411   def activity
412     @project = DbProject.find_by_name params[:project]
413     @package = DbPackage.find :first, :conditions => [
414       'name=? AND db_project_id=?', params[:package], @project.id ] if @project
415   end
416
417
```

```
418   def latest_added
419     # set automatic action_cache expiry time limit
420     response.time_to_live = 5.minutes
421
422     packages = DbPackage.find :all,
423       :from => 'db_packages pac, db_projects pro',
424       :select => 'pac.name, pac.created_at, pro.name AS project_name',
425       :conditions => 'pro.id = pac.db_project_id',
426       :order => 'created_at DESC, name', :limit => @limit
427     projects = DbProject.find :all,
428       :select => 'name, created_at',
429       :order => 'created_at DESC, name', :limit => @limit
430
431     list = []
432     projects.each { |project| list << project }
433     packages.each { |package| list << package }
434     list.sort! { |a,b| b.created_at <=> a.created_at }
435
436     @list = list[0..@limit-1]
437   end
438
439
440   def added_timestamp
441     @project = DbProject.find_by_name( params[:project] )
442     @package = DbPackage.find( :first, :conditions =>
443       [ 'name=? AND db_project_id=?', params[:package], @project.id ]
444     ) if @project
445   end
446
447
448   def latest_updated
449     # set automatic action_cache expiry time limit
450     response.time_to_live = 5.minutes
451
452     packages = DbPackage.find :all,
453       :from => 'db_packages pac, db_projects pro',
454       :select => 'pac.name, pac.updated_at, pro.name AS project_name',
455       :conditions => 'pro.id = pac.db_project_id',
456       :order => 'updated_at DESC, name', :limit => @limit
457     projects = DbProject.find :all,
458       :select => 'name, updated_at',
459       :order => 'updated_at DESC, name', :limit => @limit
460
461     list = []
462     projects.each { |project| list << project }
463     packages.each { |package| list << package }
464     list.sort! { |a,b| b.updated_at <=> a.updated_at }
465
466     @list = list[0..@limit-1]
467   end
468
469
470   def updated_timestamp
471     @project = DbProject.find_by_name( params[:project] )
472     @package = DbPackage.find( :first, :conditions =>
473       [ 'name=? AND db_project_id=?', params[:package], @project.id ]
474     ) if @project
```

```
475   end
476
477
478   def global_counters
479     @users = User.find( :first ,
480       :select => 'count(id) AS count', :conditions => 'state=2'
481     ).count
482     @repos = Repository.find( :first , :select => 'count(id) AS count' ).count
483     @projects = DbProject.find( :first , :select => 'count(id) AS count' ).count
484     @packages = DbPackage.find( :first , :select => 'count(id) AS count' ).count
485   end
486
487
488   def latest_built
489     # set automatic action_cache expiry time limit
490     response.time_to_live = 10.minutes
491
492     # TODO: implement or decide to abolish this functionality
493   end
494
495
496   def get_limit
497     @limit = 10 if (@limit = params[:limit].to_i) == 0
498   end
499
500
501   def randomize_timestamps
502
503     # ONLY enable on test-/development database!
504     # it will randomize created/updated timestamps of ALL packages/projects!
505     # this should NOT be enabled for prodution data!
506     enable = false
507     #
508
509     if enable
510
511       # deactivate automatic timestamps for this action
512       ActiveRecord::Base.record_timestamps = false
513
514       projects = DbProject.find(:all)
515       packages = DbPackage.find(:all)
516
517       projects.each do |project|
518         date_min = Time.utc 2005, 9
519         date_max = Time.now
520         date_diff = ( date_max - date_min ).to_i
521         t = [ (date_min + rand(date_diff)), (date_min + rand(date_diff)) ]
522         t.sort!
523         project.created_at = t[0]
524         project.updated_at = t[1]
525         if project.save
526           logger.debug "Project #{project.name} got new timestamps"
527         else
528           logger.debug "Project #{project.name} : ERROR setting timestamps"
529         end
530       end
531
```

```
532       packages.each do |package|
533         date_min = Time.utc 2005, 6
534         date_max = Time.now - 36000
535         date_diff = ( date_max - date_min ).to_i
536         t = [ (date_min + rand(date_diff)), (date_min + rand(date_diff)) ]
537         t.sort!
538         package.created_at = t[0]
539         package.updated_at = t[1]
540         if package.save
541           logger.debug "Package #{package.name} got new timestamps"
542         else
543           logger.debug "Package #{package.name} : ERROR setting timestamps"
544         end
545       end
546
547       # re-activate automatic timestamps
548       ActiveRecord::Base.record_timestamps = true
549
550       render :text => "ok, done randomizing all timestams."
551       return
552     else
553       logger.debug "tried to execute randomize_timestamps, but it's not enabled!"
554       render :text => "this action is deactivated."
555       return
556     end
557
558   end
559
560
561 end
```

C Bibliography

[Act07a] "Action Cache Plugin website," 2007, [accessed 24-July-2007]. [Online]. Available: http://blog.craz8.com/action-cache-plugin

[Act07b] "Action Pack – On rails from request to response," 2007, [accessed 9-July-2007]. [Online]. Available: http://ap.rubyonrails.com/

[Aja07] "Adaptive path – Ajax: a new approach to web applications," 2007, [accessed 16-June-2007]. [Online]. Available: http://www.adaptivepath.com/publications/essays/archives/000385.php

[Ama07] "Amazon Webservices," 2007, [accessed 16-June-2007]. [Online]. Available: http://www.amazon.com/gp/browse.html?node=3435361

[Arc07] "Architectural Patterns," 2007, [accessed 9-July-2007]. [Online]. Available: http://www.answers.com/topic/architectural-pattern

[Bro07] "Gaels training manuals," 2007, [accessed 14-April-2007]. [Online]. Available: http://gaels.lib.strath.ac.uk/forensic/module1/unit1_4/other4.html

[Bui07] "openSUSE Build Service Startpage," 2007, [accessed 15-April-2007]. [Online]. Available: http://build.opensuse.org

[Cap07a] "Capistrano: Automating Application Deployment," 2007, [accessed 1-July-2007]. [Online]. Available: http://manuals.rubyonrails.com/read/book/17

[Cap07b] "Capistrano Online Book," 2007, [accessed 23-June-2007]. [Online]. Available: http://manuals.rubyonrails.com/read/book/17

[Cap07c] "Capistrano Homepage," 2007, [accessed 8-April-2007]. [Online]. Available: http://www.capify.org/

[Dav05] Dave Thomas, *Agile Web Development with Rails*. The Pragmatic Programmers, 2005.

[Deb07] "Debian Linux Homepage," 2007, [accessed 7-June-2007]. [Online]. Available: http://debian.org/

[eBa07] "eBay Webservices," 2007, [accessed 12-June-2007]. [Online]. Available: http://developer.ebay.com/

[Ecl07] "Eclipse IDE Homepage," 2007, [accessed 1-June-2007]. [Online]. Available: http://www.eclipse.org/

[Fed07] "Fedora Linux Homepage," 2007, [accessed 19-June-2007]. [Online]. Available: http://fedoraproject.org/

[Fli07] "Flickr Webservices," 2007, [accessed 20-May-2007]. [Online]. Available: http://flickr.com/services

[Goo07] "Google Webservices," 2007, [accessed 13-June-2007]. [Online]. Available: http://code.google.com/apis

[iCh07] "Novell iChain Product Homepage," 2007, [accessed 5-April-2007]. [Online]. Available: http://www.novell.com/en-us/products/ichain/index.html

[Lib07] "LibXML for Ruby Homepage," 2007, [accessed 18-June-2007]. [Online]. Available: http://libxml.rubyforge.org/

[Mig07a] "ActiveRecord / Migration API Documentation," 2007, [accessed 15-April-2007]. [Online]. Available: http://api.rubyonrails.com/classes/ActiveRecord/Migration.html

[Mig07b] "Rails Migrations," 2007, [accessed 23-April-2007]. [Online]. Available: http://www.recentrambles.com/pragmatic/view/51

[Nat04] National Information Standards Organization, "Understanding Metadata," NISO Press, 2004.

[Oet07] T. Oetiker, "RRDtool," 2007, [accessed 1-July-2007]. [Online]. Available: http://oss.oetiker.ch/rrdtool/

[Ope07a] "openSUSE website," 2007, [accessed 2-April-2007]. [Online]. Available: http://www.opensuse.org/

[Ope07b] "openSUSE Build Service API," 2007, [accessed 13-April-2007]. [Online]. Available: https://api.opensuse.org

[Ope07c] "openSUSE Build Service API Documentation," 2007, [accessed 21-April-2007]. [Online]. Available: https://api.opensuse.org/apidocs

[Ope07d] "openSUSE download," 2007, [accessed 21-May-2007]. [Online]. Available: http://download.opensuse.org/

[Pet04] Peter W. Lount, *What is Smalltalk*, 16 Aug. 2004, http://www.smalltalk.org/smalltalk/whatissmalltalk.html.

[Pol07] "Rails Wiki: polymorphic associations," 2007, [accessed 3-May-2007]. [Online]. Available: http://wiki.rubyonrails.org/rails/pages/UnderstandingPolymorphicAssociations

[QtT07] "Trolltech qt product homepage," 2007, [accessed 24-June-2007]. [Online]. Available: http://trolltech.com/products/qt

[Rai07] "Ruby on Rails API Documentation," 2007, [accessed 21-April-2007]. [Online]. Available: http://api.rubyonrails.org/

[Ric07] "Build Service RichClient Screenshots," 2007, [accessed 10-July-2007]. [Online]. Available: http://jarpack.net/images/nbs/

[Roy00] Roy Thomas Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Doctoral dissertation, University of California, Irvine, 2000.

[Rub07] "RubyGems Manuals," 2007, [accessed 1-July-2007]. [Online]. Available: http://rubygems.org/read/book/1

[Sou07] "Source Forge Homepage," 2007, [accessed 14-April-2007]. [Online]. Available: http://sourceforge.net

[Str07] "Generic xml stream parser api," 2007, [accessed 21-June-2007]. [Online]. Available: http://www.mulberrytech.com/Extreme/Proceedings/html/2004/Gorman01/EML2004Gorman01.html

[SVN07] "Subversion Homepage," 2007, [accessed 19-June-2007]. [Online]. Available: http://subversion.tigris.org/

[Try79] Trygve Reenskaug, *Thing-Model-View-Editor an Example from a planningsystem*, 12 1979, http://heim.ifi.uio.no/ trygver/1979/mvc-1/1979-05-MVC.pdf.

[Ubu07] "Ubuntu Linux Homepage," 2007, [accessed 30-June-2007]. [Online]. Available: http://www.ubuntu.com/

[Vim07a] "Vim Homepage," 2007, [accessed 16-June-2007]. [Online]. Available: http://www.vim.org

[Vim07b] "Vim Project Plugin," 2007, [accessed 21-May-2007]. [Online]. Available: http://www.vim.org/scripts/script.php?script_id=69

[W3C07] "W3C Homepage," 2007, [accessed 27-April-2007]. [Online]. Available: http://www.w3.org/

[Wik07a] "Capistrano — Wikipedia, The Free Encyclopedia," 2007, [accessed 1-July-2007]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Capistrano&oldid=140465284

[Wik07b] "DBMS — Wikipedia, The Free Encyclopedia," 2007, [accessed 1-July-2007]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Database_management_system&oldid=143431413

[Wik07c] "Emacs — Wikipedia, The Free Encyclopedia," 2007, [accessed 1-July-2007]. [Online]. Available: http://en.wikipedia.org/wiki/Emacs?oldid=142270680

[Wik07d] "SAX — Wikipedia, The Free Encyclopedia," 2007, [accessed 1-July-2007]. [Online]. Available: http://de.wikipedia.org/wiki/Simple_API_for_XML?oldid=34051357

[Wik07e] "Statistics — Wikipedia, The Free Encyclopedia," 2007, [accessed 1-July-2007]. [Online]. Available: http://en.wikipedia.org/wiki/Statistics?oldid=143370619

[Wik07f] "Subversion — Wikipedia, The Free Encyclopedia," 2007, [accessed 1-July-2007]. [Online]. Available: http://en.wikipedia.org/wiki/Subversion_%28software%29?oldid=143349092

[Wik07g] "Vim — Wikipedia, The Free Encyclopedia," 2007, [accessed 1-July-2007]. [Online]. Available: http://en.wikipedia.org/wiki/Vim_%28text_editor%29?oldid=142905941

D Curriculum vitae

David Mayr was born at the 28th of April 1979 in Obergünzburg, Germany.

school education

1985-1987 primary school Börwang

1987-1990 primary school Haldenwang

1990-1999 secondary school Marianum Buxheim near Memmingen

community service

1999-2000 Stiftsklinik, Bad Grönenbach

vocational training

08.2000-01.2003

Fachinformatiker/ FR Systemintegration, Systemhaus Abele Informatik

professional activity

02.2003-09.2003

Systemhaus Abele Informatik as Fachinformatiker

scholastic

09.2003-02.2007

nta FH (university of applied sciences) Isny, branch of computer sciences

additional qualification

since 06.2005

LPI Certification Level 1, Linux Professionals Institute

practice semester

10.2006-03.2007

SUSE Linux Products GmbH, Research & Development, Nuremberg

E Media

The printed version of this thesis, delivered the 11th of July in 2007, contains a data Compact Disc (CD). This CD contains three directories:

- The directory `source_code` contains the complete code of the openSUSE Build Service. The relevant subdirectories are:
  - `buildservice/src/frontend/(app)`
  - `buildservice/src/webclient/(app)`
  - `buildservice/src/common/lib/activexml/`
  - `tools/download-stats/`
- The directory `thesis_source` contains the LaTeX source code of this thesis.
- The directory `thesis_pdf` contains the final PDF version of this thesis.

F Glossary

Ajax

Asynchronous Javascript and XML is a concept for data transmission between clients and servers.

API

Application programmable interface.

chroot

A chroot on Linux/Unix operating systems is an operation that changes the root director. This provides a convenient way to sandbox an untrusted, untested or otherwise dangerous program.

Compile Farm

A set of networked computers, whose sole job it is to compile source code to binaries.

Cron

Cron is a time-based scheduling service in Unix and Unix-like operating systems. A cronjob is a scheduled task in cron.

CRUD

Create, read, update and delete - the most common actions with data records.

Curl

Curl is a command line tool for transferring files with URL syntax, supporting FTP, FTPS, HTTP, HTTPS and many more. The main purpose and use for curl is to automate unattended file transfers or sequences of operations.

DBMS

Database management system.

DDL

Data Definition Language - a subset of SQL.

DEB

The Debian software package format used by Debian Linux, Ubuntu and other distributions.

DOM

Document Object Model is a way to refer to XML or HTML elements as objects.

DQL

Data Query Language - a subset of SQL.

DRY

Abbreviation for “Don’t repeat yourself”, a main principle for Ruby on Rails.

Fixture

Fixtures is another word for ‘sample data’ in the context of automated Ruby on Rails tests. Fixtures allow to populate the testing database with predefined data before the tests run.

GET

The HTTP GET method requests data over the HTTP protocol by specifying the ressource via an URL.

GUI

Graphical User Interface.

HTTP

Hyper Text Transfer Protocol.

iChain

Novell iChain is a Web-based federated single sign-on solution that provides simplified yet secure access to resources without the need to modify existing web applications.

IDE

Integrated Development Environment.

Linux

Free and open source Unix-like computer operating system.

Metadata

Metata can be defined as data over data, data that describes other data.

MVC

Model-View-Control – an application design pattern.

NTA

Naturwissenschaftlich-Technische Akademie (academy of natural sciences).

Open Source Software

Open source software is computer software whose source code is available under a license that permits users to use, change and improve the software, and to redistribute it in modified or unmodified form.

ORM

Object-relational mapping.

Package

In the context of Linux, open source and the Build Service, a package contains all that is necessary for a specific software application to be easy installed on a computer running Linux. For details see page 22.

POST

The HTTP POST method sends commands and parameters over the HTTP protocol in the request body.

PUT

The HTTP PUT method sends data over the HTTP protocol in the request body.

Qt

Qt is a cross-platform application development framework from Trolltech, widely used for the development of GUI programs for Linux, Mac OS and Windows.

R&D

Abbreviation for Research and Development (department).

REST

Representational State Transfer, an architectural style for designing web services.

RoR

Ruby on Rails, a web application framework.

RPM

RPM Package Manager (originally Red Hat Package Manager) is a software package management system for Linux used by many distributions like SUSE, RedHat, Fedora, Mandrake and others.

Ruby

An object oriented scripting language.

SAX

Simple API for XML is a serial access parser API for XML.

SourceForge

A very popular Project hosting service for free and open source software.

SQL

Structure Query Language.

SVN

Subversion is a popular revision control system.

URI

Uniform Resource Identifier.

URL

Uniform Resource Locator.

W3C

The World Wide Web Consortium (W3C) is the main international standards organization for the World Wide Web (W3).

XHR

XMLHttpRequest (XHR) is an API that can be used by JavaScript to transfer XML to and from a web server.

XML

Extensible Markup Language.

XSD

XML Schema Definition.

YAML

A human-readable data serialization format.

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the definition; numbers in roman refer to the pages where the entry is used.