Makumba and Ruby on Rails

Comparing two web development frameworks

S E B A S T I A N   Ö H R N

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2010

Makumba and Ruby on Rails

Comparing two web development frameworks

S E B A S T I A N   Ö H R N

Bachelor's Thesis in Engineering and Management (15 ECTS credits)
at the School of Industrial Engineering and Management
Royal Institute of Technology year 2010
Supervisor at CSC was Christian Bogdan
Examiner was Stefan Arnborg

URL: www.csc.kth.se/utbildning/kandidatexjobb/teknikmanagement/2010/
ohrn_sebastian_K10053.pdf

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc

Abstract

Web development frameworks are often used today to easily create and maintain large websites. Many different such frameworks exists, and many of them are also open source, meaning anyone can contribute to their codebase and improve them. The purpose of this thesis is to find areas of improvement for two web development frameworks, Makumba and Ruby on Rails. This is done by constructing a simple web application using both technologies. This application is analyzed based on a set of comparison criteria, also determined in this thesis. Based on the comparison, strengths and weaknesses of both frameworks can be identified, and act as grounds for determining future areas of improvement.

The results of the thesis indicate that the web development framework in most need of improvement is Makumba. The areas of improvement for Makumba lie in the tag structure, the lack of separation of application logic and presentation, interdependencies, and documentation. In Ruby on Rails, a possible area of improvement is to improve accessibility for beginners. Based on the findings made, it seems reasonable that Ruby on Rails has gained considerably more popularity than Makumba.

Referat

Idag används ofta s.k. ramverk för webbutveckling för att enkelt skapa och underhålla stora webbsajter. Det finns många olika sådana ramverk, varav många också har öppen källkod. Detta innebär att vem som helst kan bidra till deras källkod och förbättra den. Syftet med denna avhandling är att finna förbättringsområden för två ramverk för webbutveckling, Makumba och Ruby on Rails. Detta görs genom att skapa en enkel webbapplikation i båda teknologierna. Denna applikation analyseras sedan baserat på ett antal jämförelsekriterier, som också bestäms i denna avhandling. Baserat på jämförelsen så kan styrkor och svagheter hos båda ramverken identifieras, och sedan fungera som grund för framtida förbättringsområden.

Resultaten av denna avhandling indikerar att ramverket i störst behov av förbättring är Makumba. Förbättringsområdena finns i strukturen av taggarna, avsaknaden av separation mellan applikationslogik och presentation, beroenden, samt dokumentation. Ett möjligt förbättringsområde för Ruby on Rails är att öka tillgängligheten för nybörjare. Baserat på resultaten så är det rimligt att Ruby on Rails har blivit betydligt mer populärt än Makumba.

Contents

Chapter 1

Introduction

1.1 Background

The World Wide Web has grown to be a very important part of society, both privately and in corporate settings. Nowadays, a good looking and informative website is an essential component in the business strategy of almost any company. Traditionally, a website was a collection of HTML pages with static content, written entirely by hand. As the website grew, very much time had to be spent directly editing the various pages, sometimes making the exact same change in several places. Also, there was no separation of presentation and content, meaning that if a text writer was to update the text of a page, he would be working in the exact same file as those responsible for the actual page HTML code. The process of maintaining a large website quickly became tedious and time-consuming, therefore also costing a lot of money.

The solution to this problem required a whole new type of web pages to be invented: *dynamic* web pages. In these types of web pages, most content wasn't actually written inside the HTML file. Instead, it was fetched *dynamically* from a database, using technologies such as CGI, and later, PHP. This allowed text writers to work with their own tools, interacting only with the database, while web developers could take care of what they knew best — programming.

The concept of software *frameworks* had been known for some time, mainly from graphical user interface programming. Frameworks grouped up common tasks and made it easier and quicker to write a graphical user interface, by for example providing ready-made classes or implementable interfaces. The concept was found feasible for web development as well, and over the latest 15 years, many different *web development frameworks* have emerged, built on many different programming languages. This thesis will compare two such frameworks: Makumba and Ruby on Rails.

1.2 Problem description

Ruby on Rails is a modern web development framework (created in 2003) that has gained a lot of popularity, primarily because of claims of it being very productive. Some have, for example, stated that it's possible to "develop a web application at least ten times faster with Rails than ... with a typical Java framework". [40] If this was true for all applications, it would have tremendous effects on costs associated with web development. Questions have, however, been raised regarding Rails performance when used in a high-volume environment, the reason for this primarily being that Ruby (the language Rails is based on) is a relatively slow language. [18]

Makumba is a smaller[1] web development framework than, for example Ruby on Rails, designed as part of a PhD thesis in 2003. [1] The goals of Makumba are similar to those of Rails in that Makumba also aims to be simple to work with.

Both Makumba and Ruby on Rails are open souce projects, meaning anyone can get the source code of the frameworks and make changes to them. If the goal is to develop new features, instead of for example fixing bugs, there has to be some way of determining what features would be useful to add. This is what this thesis aims to investigate. It should be added that this thesis is based on an idea from the Makumba developer community.

1.3 Goals and purpose

The ultimate goal of this thesis is to identify areas of improvement for both Makumba and Ruby on Rails. However, there are many ways to reach that goal. I have chosen to include a practical part, where a comparable web application is developed in both frameworks, and then analyzed. The areas of improvement will then be derived from the findings of this analysis. The goals can be summarized as follows:

- Develop two comparable web applications using Makumba and Ruby on Rails
- Find suitable method of comparing these frameworks using the code created for the web application
- Analyze the resulting web applications using above method, in order to find strengths and weaknesses of the respective web development frameworks
- Based on this, identify possible areas of improvement for Makumba and Ruby on Rails in order to make them more effective

1.4 Questions

- What are suitable criteria for comparing two web development frameworks in this way?

[1]'Smaller' in the sense of the community surrounding it, and the number of sites using it.

- What are the strengths and weaknesses of Makumba and Ruby on Rails, with respect to the criteria defined?

- In what areas can Makumba and Ruby on Rails be improved, based on their respective strengths and weaknesses?

1.5 Delimitations

In order to avoid this thesis getting too long and "out of control", I have been forced to make some delimitations.

First, this thesis will only compare the two web development frameworks Makumba and Ruby on Rails. A couple of other frameworks are mentioned in the Theory part, but these are not part of the main comparison.

Second, the web application that is to be developed using both frameworks is a very simple one, with only a couple of different pages. However, it still demonstrates basic framework functionality like, for example, database interaction.

Third, this thesis will only compare the frameworks based on the set of criteria to be defined later. The reason for this being that they could possibly be compared in very many different ways, many of which would be downright unsuitable for the method I have chosen.

1.6 Shortcomings

I am aware that the approach I have chosen is not optimal. Programming the web application solely by myself makes the some of the outcome dependent on my own programming skills, which are by no means equal to those of an expert. I would say that my skills are close to the average programming skills of a third year M.Sc. university student, having done programming mostly as part of various courses in the subject. However, since the web application to be programmed is of such simple nature, I estimate that the possible difference in coding style between me and an expert programmer is small in this case.

To get more objective results, the task of programming the application the should probably have been done by several external parties, possibly with differing skill levels. Such a study has for example been done by Prechelt [34], where he compared seven different programming languages by defining a numerical problem to be solved and then have various people program the application. Statistical analyses were then made based on the performance of the different versions of the program. However, due to the time constraints involved in writing this thesis, an approach such as this was not estimated to be feasible.

1.7 Target group

The target group of this thesis is first and foremost developers working with Makumba or Ruby on Rails. However, anyone working with web development frameworks could find it interesting, as it shows a method that could potentially be applied to any combination of web development frameworks. Similar methods might also be applied to other types of software, making the 'secondary' target group very large.

Chapter 2

Theory

2.1 History

In the early days of the World Wide Web, all web sites were *static*. That is - they always showed the same content to every visitor. To change a web site one would have to edit the HTML file that represented the web page, which of course required access to the files on the web server. It is clear that having only static web sites severely limited the functionality of the World Wide Web, and hence, different methods for generating *dynamic* web sites were developed. Among the first were the *Common Gateway Interface* (CGI), a standard protocol for interfacing external applications with, for example, Web servers. This allowed programmers to write small programs that were executed in real-time when the site was visited.[28] Although popular, CGI had a number of problems, like it's platform dependence and lack of scalability. As a solution to those problems, the Java servlet technology was developed. Servlets are classes written in the cross-platform Java language, meaning the same application can be deployed to many different systems. [25]

Further improvements were made in the area, with the popular Apache HTTP Server gaining support for modules that enabled it to execute code dynamically. At around the same time, programming languages like PHP were being developed, with a specialization towards dynamic web site creation. The concept of libraries, collections of subroutines or classes used to develop software, was known since long, but the new area of dynamic web pages required entirely new tools to aid in for example HTML generation. This eventually evolved into entire frameworks that handle everything related to dynamic web pages in one place.

2.2 Frameworks in general

In an article by Fayad and Schmidt, it is stated that "a framework is a reusable, 'semi-complete' application that can be specialized to produce custom applications".[9] The benefits of a framework include providing modularity, reusability, extensibility and inversion of control to the developer. This in turn improves the quality of the code as well as the productivity of the developer. The first frameworks were aimed for graphical user interface programming, like MacApp and Interviews. Later on came Microsoft Foundation Classes (MFC), which is still widely used today to create graphical applications on the Microsoft Windows platform. Among the first frameworks aimed at web application programming were Java EE (containing for example servlets described above) and WebObjects. Later on came more modern frameworks, like Ruby on Rails.

2.3 Other relevant technologies

REST

REST is an acronym for Representational State Transfer, and is an architecture style of networked systems, developed by Roy Thomas Fielding as part of a doctoral dissertation in 2000.[10] The Ruby on Rails team describe the main principles of REST as follows:

> - "Using resource identifiers (which, for the purposes of discussion, you can think of as URLs) to represent resources
> - Transferring representations of the state of that resource between system components.
>
> For example, to a Rails application a request such as this:
>
> `DELETE /photos/17`
>
> would be understood to refer to a photo resource with the ID of 17, and to indicate a desired action — deleting that resource." [32]

`DELETE` in this case is the type of HTML request being sent by the browser. Other examples of HTML requests are `GET`, `PUT`, and `POST`.

2.4 Current state of web development frameworks

JavaServer Pages

JavaServer Pages (JSP) is a technology maintained by Oracle (since their acquisition of Sun in January 2010) as part of their Java EE platform. JSP is an extension of the Java Servlet technology, and allows the developer to "easily create web content that has both static and dynamic components". [25] JSP pages also separate the user interface from content generation, a useful property allowing, for example, web designers to work on page design without exposing them to important application logic. Two types of text are used for static and dynamic content, respectively. For static content, any text-based format, like HTML or XML, can be used. For dynamic content, the XML-like tags of JSP are used. JSP pages are automatically compiled into servlets (by, for example, the Apache Tomcat servlet container), but the improved structure and better separation of presentation and content make them easier to use than ordinary servlets. [27]

JavaServer Faces

JavaServer Faces (JSF) is a more modern technology, building upon JSP but also including several new components. The technology includes a set of APIs for representing UI components and and managing their state, handling events and input validation, defining page navigation, and supporting internationalization and accessibility. JSF also includes a custom JSP tag library to include JSF content in a JSP environment. [26]

JSF is concentrated on the user interface part of the web application, and allows the developer to, with "minimal effort":

- Wire client-generated events to server-side application code
- Bind UI components on a page to server-side data
- Construct a UI with reusable and extensible components
- Save and restore UI state beyond the life of server requests [25]

The improvements that JSF provide over JSP is that it has an even clearer separation of presentation and behavior. For example, it is not possible using JSP technology alone too map HTTP requests to component-specific event handling, something that JSF can do.

PHP

The preface of the official PHP manual states that:

> "[PHP is a] widely-used Open Source general-purpose scripting language that is specially suited for Web development and can be embedded into HTML. Its syntax draws upon C, Java, and Perl, and is easy to learn. The main goal of the language is to allow web developers to write dynamically generated web pages quickly, but you can do much more with PHP." [13]

Judging by this description, PHP itself seems to share many qualities with actual web development frameworks, even though PHP is strictly speaking "only" a programming language. However, there are also several web development frameworks based on PHP. Two of the most used are CakePHP and Zend Framework.

CakePHP

CakePHP is a web development framework that is similar to Ruby on Rails in that it promotes rapid development and and uses an MVC design pattern (a closer description of Ruby on Rails follows below). Users of the framework don't have to "reinvent the wheel" but can get straight into coding what they really need — the logic of the application. [17]

Zend Framework

Zend Framework is another PHP based that promises to be simple and productive. It differs to many other web development frameworks because of its many different, loosely coupled, components. These components can in many cases be taken and used independently, but Zend Framework works just as well if you choose to build an application entirely based on the framework. [19] [20]

ASP and ASP.NET

ASP, or *Active Server Pages*, is a technology for creating dynamic web pages developed by Microsoft. First released in 1997, the technology has now been replaced by ASP.NET, introduced in as part of the .NET Framework in 2002. [7]

There are two different ways to compose web applications using ASP.NET: ASP.NET Web Forms and ASP.NET MVC. With Web Forms, user interface elements are used to build the interactive web applications. These have various properties, methods and events for the controls that are placed onto them, similar to what a desktop software application has. Specific application logic is placed in separate files. [35]

ASP.NET MVC is Microsoft's contender in the vast market of web frameworks based on the model-view-controller pattern. With version 1.0 released in 2009, it is a rather new technology, with the purpose of bringing in an alternative way of developing web applications using ASP.NET.

Restlet

Restlet is a framework that brings the REST architecture style to the Java environment. It consists of two parts, the Restlet API and the Noelios Restlet Engine. The Restlet API is a generic set of classes used for custom Restlet implementations. The Noelios Restlet Engine is a reference implementation, and the actual framework built on the Restlet API. [38]

There are several editions of Restlet available for different environments, such as Java SE/EE for the regular JVM or Servlet containers, GWT for use in Google Web Toolkit, or Android for use in mobile phones using the Android operating system. [37]

2.5 The Ruby Language

The Ruby programming language is a ”dynamic, open source programming language with a focus on simplicity and productivity.” [4] It has it’s origins in Japan, where the creator Yukihiro ”matz” Matsumoto took parts from his favorite programming languages (Perl, Smalltalk, Eiffel, Ada and Lisp), and blended them to form Ruby. One of the central ideas in Ruby is that everything is an object. In many other programming languages, numbers and other primitive types are not objects. In Ruby however, they are objects just like any other, and can be treated accordingly with regards to inheritance, class methods and other behavior commonly associated with objects.

The simplicity of the Ruby language is probably most easily demonstrated with an example. The following paragraph will show the classic ”Hello World” application implemented in Ruby, compared to it’s Java counterpart.

Listing 2.1: Ruby ”Hello World”

```
puts "Hello World!"
```

Listing 2.2: Java ”Hello World”

```
public class Hello {
  public static void main(String[] args) {
      System.out.println("Hello World!");
  }
}
```

As you can see, the Ruby example has no class definition, no braces, no main method, and just a call to the `puts` (put string) method instead of the longer `System.out.println()`. Parentheses are optional in Ruby function calls.

It gets even more interesting if we add some complexity to our program. The following example prints the above text five times, with an added counter at the end.

Listing 2.3: Ruby ”Hello World” loop

```
1.upto(5) {|i| puts "Hello World #{i}!"}
```

Listing 2.4: Java ”Hello World” loop

```
public class Hello2 {
  public static void main(String[] args) {
    for (int i=1; i<6; i++) {
      System.out.println("Hello World " + i + "!");
    }
  }
}
```

It is easy to see that the Ruby code more closely resembles natural language, which also goes hand in hand with the Ruby philosophy of keeping things natural.

2.6 Ruby on Rails

Ruby on Rails (or simply Rails) is a web development framework written in the Ruby language, and based on a *Model-View-Controller* (MVC) architecture. This separates the application's different parts and helps structure it more cleanly. Rails has a couple of key concepts that have influenced the development, called *DRY* and *convention over configuration.* Their respective meanings are explained below.

DRY

DRY stands for Don't Repeat Yourself and states that "every piece of knowledge in a system should be expressed in just one place". This means that one should never have to write the same code in two different places. Rails provides different tools to help achieve this, which will be described later in this essay.

Convention over configuration

Convention over configuration means that Rails has default values for almost everything related to the application already set, meaning that very little time is wasted in configuration files.[39] Because of this, it is possible to get a fully functional Rails application almost without writing any code at all. Of course, this is seldom enough for more advanced applications, which is why it is also easy to override the conventions with custom functionality.[41]

Agile development

Rails is also *agile.* According to the Agile Manifesto, agile development favors:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan [24]

Making web applications in Rails leads to all of the above, as it is lightweight with almost no configuration, quick to get started with, and easy to change.

2.7 Makumba

Makumba is a web development framework based on Java technology, specifically a JSP *tag-library.* The core ideas of Makumba are:

- Creating web-apps should be simple
- Code should be sustainable
- Everything is a query-fragment
- Breaking MVC is cool [22]

Like Ruby on Rails, Makumba aims to be easy and fast to work with. There is however, a difference in their view of the MVC architecture, as Ruby on Rails states that it is a pure MVC based framework, while Makumba says that breaking MVC is "cool". The reason for this is according the Makumba website "because [Makumba's] design makes it possible to write highly maintainable code without this pattern and all the indirection it brings in."[22] To achieve it's simplicity, Makumba relies on simple languages such as HTML and MQL (Makumba Query Language, based on SQL).

As noted in the introduction to this thesis, Makumba was developed as part of a PhD thesis in 2003, where the goal was, amongst other things, to develop a system to support an *amateur community* — in this case a student organization. This has had some special implications on Makumba's design, since some factors become especially important when designing for an amateur community. One of these is *sustainability*: since the community is often run by volunteers, it has to be *self-sustainable*, meaning that new members can easily be integrated in the project and learn from the more skilled members. When these 'veterans' leave the community, the existing members should know enough to be able to completely take over their tasks. [1]

2.8 Comparison methods

There are several ways to compare different techniques in the software development domain. Over the years, many comparisons have been done between different programming languages. There are several aspects that can be compared: one could for example compare the different features of the languages, collect quantitative data on for example lines of code and number of classes, or compare the productivity based on number of bugs and speed of development. [8] This study will be more concentrated on the quantitative end of the spectrum, as well as some discussion on the features. The reason for this is that it a qualitative analysis requires a large enough project to analyze, something I have not been able to find (or, rather, not even looked for).

Motivation of comparison criteria

Written lines of code

Written lines of code is an intuitive and simple measurement of how expressive and concise a particular programming language is. Since the web application in this thesis is built with equal functionality and appearance, the number of written lines of code can be directly compared between the two web development frameworks. Fewer written lines means that the web development framework is better at either generating code "behind the scenes", or that the language it is built on is more expressive.

Number of different kinds of files (languages)

The simplicity of getting up and running with a web development framework will be largely dependent on how many different languages and file types the developer has to understand (if he doesn't already know them). Because of this, it is desirable to have a small number of different file types and languages involved in a web development framework. Also, all other things being equal, it is desirable to be based on a well known programming language instead of a smaller, more obscure one.[1]

Interdependencies between the files

According to the description of a framework provided earlier in this thesis, a framework should provide amongst other things *modularity* to the developer. This is what this criterion aims to measure. Having modularity means that the components of an object (a web application in this case) are loosely coupled and may be used in other contexts than the original. Modularity also means that it might be possible to use ready-made components in a system without having to configure them much. This means that little interdependency between the application files improves the modularity of the application, a desirable property.

Scaling of application size and interdependencies

This criterion measures if and how the results of the above points change when the application grows. Many projects will contain many more pages than the simple web application built for this thesis, and then this criterion becomes very important. In the optimal case, a large application will not contain disproportionally many written lines of code, or number of different kinds of files or languages. It will also not introduce many more interdependencies than a simple application would. This criterion will be discussed on a more speculative basis than the others, as the web application will still remain a simple one.

Team scaling

In larger software projects, there is seldom only one person involved. When creating and maintaining a website, there will probably be programmers, designers and

[1]This criterion could possibly be trumped by other criteria, for example the one regarding code intuitiveness. If a smaller and more obscure programming language produced code that was a lot more intuitive than that of a well known programming language, the prior would be preferred.

content writers, at the very least. This criterion aims to measure how well the web application fares when several people work with the application. Preferably, each 'role' should have their own distinct place to work within the application. Designers should not have to work in the same files that application logic programmers do. As with the criterion above, the discussion on team scaling will be more speculative, as no real measurements or observations will be made.

Code intuitiveness

Intuitive code leads to higher productivity. If less time is spent interpreting what various statements mean, or figuring out how to write them, more time can be spent writing actual code. As David Heinemeier Hansson says in his presentation "Pursuing beauty with Ruby on Rails":

> "Beauty leads to happiness,
> happiness leads to productivity,
> Thus
> Beauty leads to productivity" [15]

Beautiful code is, of course, almost always the same as intuitive code. However, measuring intuitiveness is hard to do in an objective way. Some people may think that a programming language is intuitive, just because they have worked with it for many years. A beginner on the other hand, may not find the same language intuitive at all. Because of this, I have chosen to do a subjective comparison, but have it backed by a set of HCI *usability goals*. According to Preece, Sharp and Rogers, these goals are:

- effective to use (effectiveness)
- efficient to use (efficiency)
- safe to use (safety)
- having good utility (utility)
- easy to learn (learnability)
- easy to remember how to use (memorability) [16]

Not all of these goals are suited for this comparison. For example, it is a reasonable assumption that all popular programming languages have a high effectiveness i.e. are good at doing what they are supposed to do. All other points, however, can be evaluated and yield discussable results.

Documentation

Every programmer knows that documentation is very important when learning a new language. It is simply not possible to just start programming without any sort of guidance. Also, even seasoned programmers need to consult API references to, for example, find out which methods are available for a certain class and what they are called. It's virtually impossible to know such things entirely by heart.

Because of this, this criterion is an important one. A well documented web development framework should preferably have an easy to follow, "getting started" type of guide, as well as several guides on more advanced topics. It should also have an API reference documenting anything the developer needs to know about the different components of the framework. Having a community of users available for consultation, be it via mailing lists or message boards, is also desirable.

2.9 Final comparison criteria

The comparison criteria are as follows:

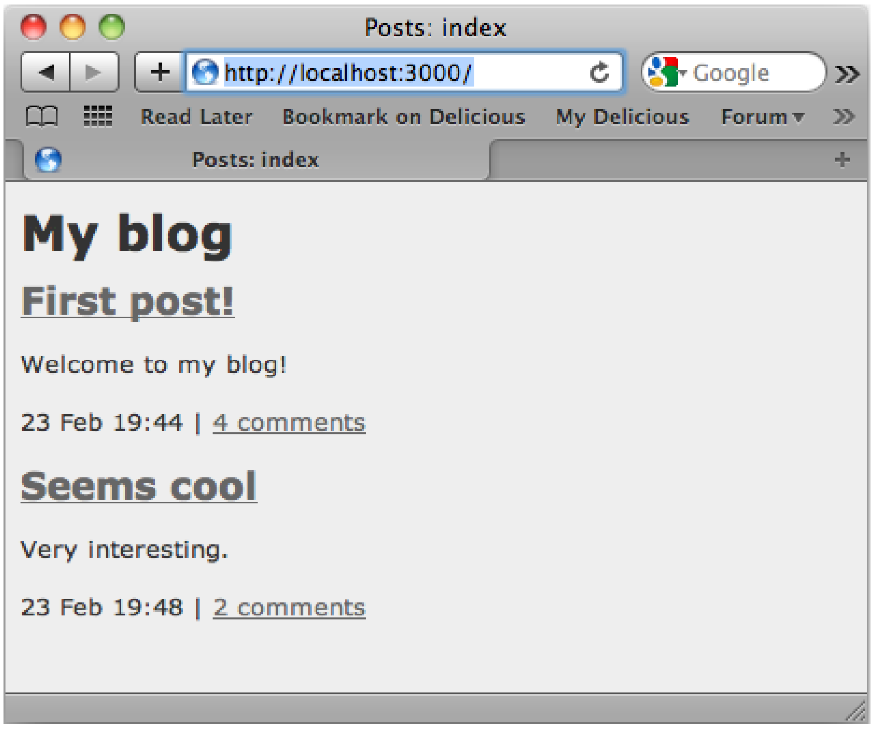
- Written lines of code
- Number of different kinds of files (languages)
- Interdependencies between the files
- Scaling of size and interdependencies
- Team scaling
- Code intuitiveness
- Documentation

Chapter 3

Description of the web application

The web application that will be developed in both Ruby on Rails and Makumba is a simple web log (commonly referred to as just 'blog'). The blog will follow the classic structure of blogs; the blog owner can make *posts* (each having a *title* and some *content*), and readers can *comment* on these posts, and sign with their *name*. To be able to edit, delete or create new posts, the application will also contain an *administration panel*. An image of the basic structure of the blog in displayed in figure 3.1. Clicking the post titles takes the visitor to the respective post, where the post is visible in full along with its comments, and a form to post a new comment (figure 3.2).

Figure 3.1: The blog index page.



My blog

First post!

Welcome to my blog!

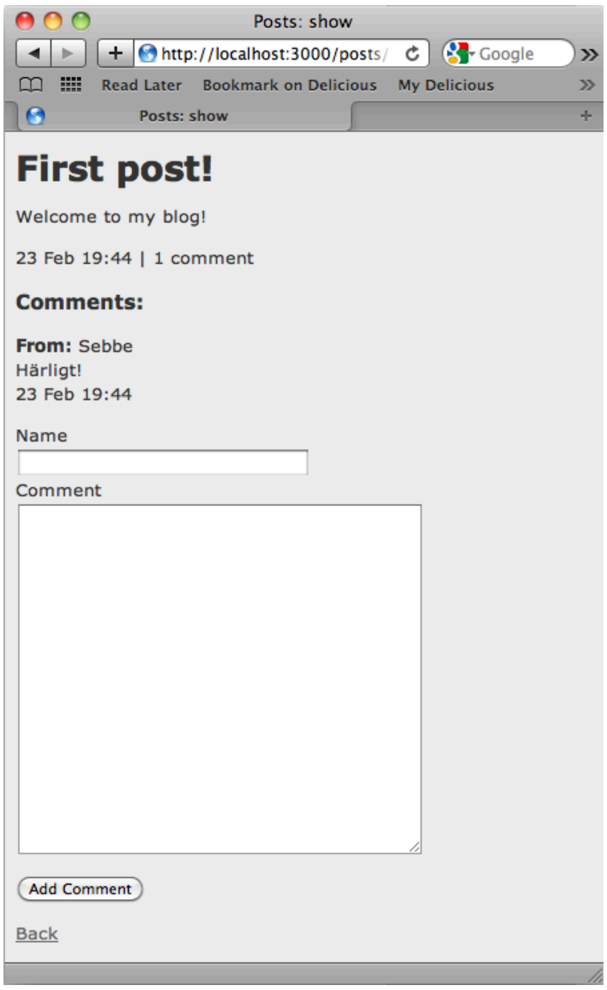
23 Feb 19:44 | 4 comments

Seems cool

Very interesting.

23 Feb 19:48 | 2 comments

Figure 3.2: A page for displaying a specific post.



First post!

Welcome to my blog!

23 Feb 19:44 | 1 comment

Comments:

From: Sebbe
Härligt!
23 Feb 19:44

Name

Comment

Chapter 4

Comparison

4.1 Written lines of code

Makumba

When starting a Makumba project, it is recommended to use Apache Maven, a command line tool for "building and managing any Java-based project". [12] By using Maven together with the Makumba *archetype*, the latest version of Makumba is automatically downloaded and a sample project is generated. The file structure then looks as follows:

```
src
  main/
    resources/
      Makumba.conf
      dataDefinitions/
        general/
          Person.mdd
    webapp/
     index.jsp
     META-INF/
     WEB-INF/
       web.xml
```

When continuing to build upon this project, one has to create new files in the correct places. For example, in order for the blog application to work, two new data definitions (Post and Comment) are needed in the `src/main/resources/dataDefinitions/` folder. The different pages accessible from the web are placed in `src/main/webapp/` or in subfolders to that folder.

In Makumba, all code has to be written by hand. It is possible to reuse some code by using the `include` mechanism provided by the JSTL[1], but the savings from using this technique are rather limited.

[1]JavaServer Pages Standard Tag Library

The total number written lines of code for the complete Makumba blog application are summarized in the table below.

Framework	**Written lines of code**
Makumba	138 lines

Ruby on Rails

Similar to Makumba, Rails has a command line tool available for starting a project. The difference is that while Maven is a third party tool, the Rails command line tool is a part of the framework itself. The command is called simply `rails` and running this command with the project name as an argument yields the basic Rails folder structure seen in listing A.2 of appendix A.

It is obvious that Rails generates quite a lot more files than Makumba, but this is not a negative thing. In Rails, a lot of code can be automatically generated using the `script/generate` command utility. The results of this is shown when the number of written lines of the complete Rails blog application is compared to its Makumba counterpart:

Framework	**Written lines of code**
Makumba	138 lines
Ruby on Rails	66 lines

4.2 Number of different kinds of files (languages)

Makumba

The blog application for Makumba is built using two kinds of files, MDD files and JSP files.

MDD

MDD stands for *Makumba Data Definition*, and is a "simple text file that contain the description of an entity, i.e. its fields but also validation rules and query functions." [23]

The *field definitions* are what describe the entity. These have the form of

```
fieldname = [attributes] fieldtype [parameters] [; description]
```

where `fieldtype` is one of several predefined types, like `int`, `date`, `char` or `text`. `fieldtype` can also be a *pointer* to a field in another MDD file, then taking the form of either `ptr` or `set`, depending on the relationship. The MDD file describing a post in the Makumba blog application is displayed below.

Listing 4.1: Post.mdd

```
title = not null char[50]
content = not null text
```

The interpretation of this is that each post can have a title with at most 50 characters of length, and an unlimited amount of text content. Neither the title nor the content field can be empty.

A comment belongs to a specific post, and this is reflected in the comment data definition:

Listing 4.2: Comment.mdd

```
commenter = not null char[50]
body = not null text
references = ptr general.Post
```

Just like posts, each comment has two text fields (though comments have a *commenter* instead of a title). The difference is that, in the third line, each comment also *references* a post. This sets up the one-to-many relationship between a post and its comments. Again, like posts, none of the text fields can be empty.

JSP

JSP, or JavaServer Pages, is the technology that Makumba uses to display dynamic content on the web pages. JSP files follow the structure of HTML files, but have added custom, "XML-like tags that encapsulate the logic that generates the content for the page." [27] Makumba is implemented using it's own library of such tags. The JSP page for the blog index page is displayed below.

Listing 4.3: index.jsp

```
<%@ taglib uri="http://www.makumba.org/presentation" prefix="mak" %>
<jsp:include page="/layout/header.jsp" />
<title>My blog</title>
</head>
<h1>My blog</h1>

<mak:list from="general.Post p">
  <h2>
    <a href="showPost.jsp?id=<mak:value expr='p' />">
      <mak:value expr="p.title" />
    </a>
  </h2>
    <p>
      <mak:value expr="p.content" />
    </p>
  <mak:value expr="p.TS_create" format="yyyy-MM-dd" />
  &nbsp;|&nbsp;
  <mak:list from="general.Comment c" where="c.references = p" groupBy="p">
    <a href="showPost.jsp?id=<mak:value expr='p' />#comments">
```

```
      <mak:if test="count(c) = 0">0 comments</mak:if>
      <mak:value expr="count(c)" /> comments
    </a>
  </mak:list>
</mak:list>
</body>
</html>
```

Aside from the some standard HTML tags, such as `<h1>` and `<a>`, this page shows a couple of the Makumba-specific tags. The topmost `<mak:list>` tag fetches all of the posts from the database, and whatever is inside of the tag will be repeated for each of the database records. In this case, the posts are displayed together with the date they were created and how many comments they have. Clicking the title takes the user to the `showPost.jsp` page, where the post is visible together with its comments. The specific data from each post is fetched using the `<mak:value>` tag. This tag requires an `expr` attribute, where the name of the data field is entered.

Java code

Aside from the MDD and JSP files, Makumba also allows for placing more advanced data processing (*business logic*) in ordinary Java files. These files are then executed when the form corresponding to the logic class is submitted. This was however not something that was needed for the simple blog application built for this essay, but Java knowledge might be needed for a more advanced Makumba project.

Ruby on Rails

Most of Rails' back end files, like the *models* and *controllers* in the MVC pattern, are written in Ruby code. The files handling the display of content, the *views*, are written by default using the ERb (Embedded Ruby) template language.[2] ERb is similar to JSP in that it allows mixing static code like HTML with another language responsible for generating dynamic content, in this case Ruby.

The model file for the post in Rails is displayed below.

Listing 4.4: post.rb

```
class Post < ActiveRecord::Base
    validates_presence_of :title, :content
    has_many :comments
end
```

This model describes that each post must have a title and some content, and that each post can have many comments. The `has_many :comments` declaration is actually optional, but enables some shortcuts for accessing the comments belonging to a specific post.

[2]"By default" means that ERb is what comes with the default installation of Rails. It is however possible to use other template languages for views.

The files in Rails are seldom written entirely by hand, however. Instead, they are generated from Rails' command line tool, located in `./script/generate`. For example, running the command `script/generate scaffold post title:string body:text` produces the output seen in listing A.1 of appendix A. This automatically creates the foundations for a model, a controller and several views for showing, editing, and adding new posts. The command also creates a *database migration*, taking care of adding the correct data fields to the database. This is in contrast to Makumba, where the field declarations exist in the model file itself. Migrations are an important part of Rails, and will be discussed later.

As written above, the views in Rails MVC system are written using the Embedded Ruby (ERb) language. The view for the blog index page is shown below.

Listing 4.5: index.html.erb

```
<h1>My blog</h1>

<% @posts.each do |post| %>
  <h2>
    <%= link_to post.title, post %>
  </h2>
  <p>
    <%=h post.content %>
  </p>
  <%= post.created_at.to_s(:short) %>
  |
  <% link_to(post_path(post, :anchor => "comments")) do %>
    <%= pluralize(post.comments.count, 'comment') %>
  <% end %>
  <% if admin? %>
  |
  <%= link_to "Edit", edit_post_path(post) %>
  |
  <%= link_to "Destroy", post_path(post),
    :confirm => "Are you sure?", :method => :delete %>
  <% end %>
<% end %>
<% if admin? %>
<br />
<br />
<%= link_to "New post", new_post_path %>
<br />
<%= link_to "Log out", logout_path %>
<% end %>
```

The structure of this file is very similar to the corresponding Makumba page, differing mostly in syntax. The `@post` variable is supplied by the post controller and contains, in this case, all the posts in the database. The `<% @posts.each do |post| %>` command, a Ruby *block*, iterates over these posts and gets the content for each of them. The `link_to` function creates links dynamically, taking the name of the link and the object to link to as arguments. `admin` is another variable that is defined in the application controller, and it equates to true when the user has entered a correct password, making the edit, delete, and new post links visible.

Comparison

As described above, a Makumba developer needs to work with at least two different kinds of files, MDD files and JSP files. Java files with business logic are optional, but might be needed for more advanced applications. All files need to be created manually by the developer.

In Rails, files are written in the Ruby language, with the exception of the view files, which are written by default using ERb (based strongly on Ruby). It is a rare case, however, that the developer writes these files entirely by hand. A base is usually generated by the command line tool that comes with Rails, as described above. This base is then modified and extended to fit the application.

Makumba and Rails have different approaches when it comes to the style of the languages for generating dynamic content. The Makumba tag library strongly resembles ordinary HTML tags, which will make them instantly familiar to anyone that has worked with that language before. In Rails, the ERb language instead looks more like actual Ruby code (also suggested by the full name, Embedded Ruby). This means that Rails could be considered to have a more unified language structure, with all parts of the application being written in Ruby or a language strongly based on Ruby. On the other hand, it might be a little bit harder for a beginner (knowing only HTML) to understand a Rails view than it would be for him to understand a JSP page with Makumba tags.

Although the Ruby language is getting increasingly popular, Java is still indisputably a more popular language. This can for example be seen in the TIOBE Programming Community index, approximating the market share of different programming languages. In April 2010, Java had about 18 % market share, while Ruby had a little more than 2 %. [3]

4.3 Interdependencies between files

Makumba

In Makumba, some interdependency can be seen, for example in the JSP files. All links to other pages need to be hard coded, meaning that, should the developer want to change the name of a file, all references to that file would need their name changed as well. It is possible to circumvent this by assigning the path to a variable using JSTL, but Makumba itself has no solution to this problem.

Makumba also has interdependency between the web application and the database belonging to it. Consider the case where the developer wants to add a field to one of his data models. By adding the field to the MDD file, a new column is added to the database. The rows that were already in the database get a null value for this column. In order to change this, the developer would have to manually enter the SQL code for setting the appropriate value for the already existing rows. If several databases exist, for example one for development and one in the production environment, this would have to be done once in every database. Even though the SQL command would be a simple one, and would only have to be entered once for every database (and every field added), it still adds some complexity to maintaining the application as a whole.

Ruby on Rails

Rails makes use of *routing*, a technology that it uses both for connecting incoming HTTP requests to the code in the applications controllers, and for generating URLs without hard-coding them. [32] The routes are defined in the `routes.rb` file, located in the `/config` subfolder to the application root. The `routes.rb` file for the Rails blog application is displayed below.

Listing 4.6: routes.rb

```
ActionController::Routing::Routes.draw do |map|
  map.resources :posts, :has_many => :comments
  map.resources :sessions

  map.login "login", :controller => "sessions", :action => "new"
  map.logout "logout", :controller => "sessions", :action => "destroy"
end
```

The first two lines of the files (omitting the enclosing block) declare RESTful routes for posts and sessions, and comments are declared as a nested resource to posts. As described in the Theory part, the REST architecture thinks of URLs as identifiers to a resource, and the type of action to be taken on a resource is dependent on the request type.

What this means is that, in Rails, a RESTful route creates a mapping between HTTP verbs (`GET`, `POST`, `PUT`, or `DELETE`), controller actions, and CRUD (create, read, update, and delete) operations in a database.

The last two lines in the route configuration file define *named routes* for logging in and out of the blog administration system. They do this by assigning the login and logout path to the `new` and `destroy` action of the `sessions` controller.

By defining routes for all resources in the application, Rails automatically generate helper methods for accessing the path of these resources. For example, by defining posts as a RESTful resource, a method called `posts_path` is generated, that maps to either the index of posts (for a `GET` request) or adding a new post (`POST` request). These helper methods are then used to, for example, avoid hard coding links, like in the `index.html.erb` view shown earlier:

```
<%= link_to "Edit", edit_post_path(post) %>
```

Rails also has a solution to the interdepency problem that Makumba has with its database, described above. Whenever the developer wants to interact with the database, Ruby files called migrations are used. The official Ruby on Rails guides explain the purpose of migrations very well:

> "Migrations are a convenient way for you to alter your database in a structured and organised manner. You could edit fragments of SQL by hand but you would then be responsible for telling other developers that they need to go and run it. You'd also have to keep track of which changes need to be run against the production machines next time you deploy.
>
> Active Record tracks which migrations have already been run so all you have to do is update your source and run `rake db:migrate`. Active Record will work out which migrations should be run. It will also update your `db/schema.rb` file to match the structure of your database.
>
> Migrations also allow you to describe these transformations using Ruby. The great thing about this is that (like most of Active Record's functionality) it is database independent: you don't need to worry about the precise syntax of `CREATE TABLE` any more that you worry about variations on `SELECT *` (you can drop down to raw SQL for database specific features). For example you could use SQLite3 in development, but MySQL in production." [31]

The migration responsible for creating the posts table is displayed below.

Listing 4.7: Migration for creating posts table

```ruby
class CreatePosts < ActiveRecord::Migration
  def self.up
    create_table :posts do |t|
      t.string :title
      t.text :content

      t.timestamps
    end
  end

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

The migration is a class containing two class methods, `up` and `down`. `up` describes what to do when applying the transformations, and `down` describes what to do when reverting them. In this case, a table is created with a string column called `title`, and a text column called `content`. Timestamps for when the row was created and last updated are also added. To revert these changes, the entire table is dropped.

A migration can preferably be used when adding a field to the database. This avoids having to use raw SQL code in the database for filling null values, and also makes sure the code will only have to be written once. To add a Boolean value called `posted` to the post model, a migration could be generated by using the Rails command line tool like so: `script/generate migration AddPostedToPosts posted:boolean`. This automatically generates a migration for adding and removing the Boolean column called `posted`. To take care of the already existing posts in the database, the developer would have to edit the migration somewhat, the final result shown below.

Listing 4.8: Migration for adding posted column to post table

```
class AddPostedToPosts < ActiveRecord::Migration
  def self.up
    add_column :posts, :posted, :Boolean
    Post.update_all ["posted = ?", true]
  end

  def self.down
    remove_column :posts, :posted
  end
end
```

Only the fourth line must the developer write by hand, all the rest is generated by the command line tool.

4.4 Scaling of size and interdependencies

Makumba and Ruby on Rails both scale similarly as the web application grows. For each new page, they both need a new file responsible for displaying the content to the user (if it is a normal web page, a JSP or html.erb file, respectively). In Rails, a method will probably also need to be added in the controller corresponding to the page. If more advanced logic is needed, Java business logic might be needed in the Makumba web application. However, for most applications, these files are needed rarely enough to conclude that they will not cause the size of the application to scale badly.

As discussed above, Makumba has no way of linking to other sites dynamically. This might potentially cause problems if the developer wants to change the name on one of the pages, as the link to that page would have to be changed on every page that it exists. There are however ways around that problem, such as ”find and replace”-scripts that can scan multiple files at once. [6]

All in all, it is clear that scaling of size or interdependencies will not be a problem for Makumba or Rails.

4.5 Team scaling

Makumba

Since Makumba does not strictly follow the model-view-controller design pattern, application logic is not entirely separated from presentation. Consider this excerpt from the Makumba JSP page for showing a specific post:

Listing 4.9: Excerpt 1 from showPost.jsp

```
<mak:object from="general.Post p" where="p=$id">
```

This tag fetches the post to be shown by comparing its object identifier value (`p`) to the value of the page parameter `id` (supplied as part of the page URL) in the `where` attribute of the tag. This is typical page logic that would belong in a controller in a model-view-controller design. However, since Makumba doesn't entirely follow the MVC pattern, this statement has to go inside the actual page, the same page where web designers work on their own types of problems.

The same pattern can be seen later in the same page, where all comments belonging to the post are fetched:

Listing 4.10: Excerpt 2 from showPost.jsp

```
<mak:list from="general.Comment c">
  <p>
    <mak:if test="c.references=p">
      <b>From:</b> <mak:value expr="c.commenter" /><br />
      <mak:value expr="c.body" /><br />
      <mak:value expr="c.TS_create" format="yyyy-MM-dd" />
    </mak:if>
  </p>
</mak:list>
```

The problem here can be seen on line 3, `<mak:if test="c.references=p">`. This is also typical application logic, sorting out all comments that don't have a reference to the post currently shown.

Ruby on Rails

Rails is based on the model-view-controller architecture, meaning that presentation and application logic are clearly separated. This allows designers to work on their own parts of the web application, without risking to interfere with the back end. An example of this is shown below.

Listing 4.11: Excerpt from show.html.erb for posts

```
<% @post.comments.each do |c| %>
  <p>
```

```
    <b>From:</b>
    <%=h c.commenter %><br />
    <%=h c.body %><br />
    <%= c.created_at.to_s(:short) %>
  </p>
<% end %>
```

This is the same excerpt as shown in listing 4.10, only this time in Rails. The difference is that this excerpt does not contain any logic at all. Instead, the comments to be shown are fetched automatically by accessing `@post.comments`, an array containing all comments belonging the `@post` variable (empty if there are none). This is made possible by the association done in the models for posts and comments (see listing 4.4), a post `has_many :comments`, and a comment `belongs_to :post`.

The post to be contained in the `@post` variable is also not determined in this page, but in the posts controller, as can be seen in the following listing:

Listing 4.12: Excerpt from show method in posts controller

```
@post = Post.find(params[:id])
```

Discussion

The results of the evaluation of this criterion indicate that Ruby on Rails is better than Makumba at separating presentation from logic. Designers shouldn't have to be exposed to (and risk destroying) parts of the inner workings of a web application. Conversely, logic programmers should be kept away from the pages that describe how the content is displayed.

There are several reasons why this is good. First, application logic programming and presentation programming are two different programming styles. When programming logic, one is generally not concerned with how the data is displayed, only that the correct data is sent to the correct place. When designing however, it is the look of the interface that is important, not the data inside it. Because of this, separating these two programming styles makes sense.

Second, logic and presentation programming are often not done simultaneously. Often, an initial design draft is done first, and actual data is connected later. Separating these parts makes them time independent of each other and enables them to be developed separately.

Third, in teams where there are several individuals with deep but "thin" knowledge about some subject, this kind of separation allows them to have a designated workplace, without distractions from statements they don't really understand. The teams in question could for instance contain web designers that know HTML and CSS well but hardly any programming concepts, and programmers that know barely anything about HTML.[3] If such a team were to use Makumba, web designers would

[3]This specific team composition is more of a special case than anything, as most programmers

have some logic statements in their code that they would possibly not understand, and because of this could risk destroying them. Also, if the programmers had to edit or add new such statements, they would have to do so in a HTML file, and risk tampering with the design in an unwanted way. Version control could also be a problem since the two would sometimes work in the same file.

Fourth, the code is ultimately more readable if presentation is separated from logic, as it is easier to read business logic if it isn't mixed with presentation, and vice versa. [5]

4.6 Code intuitiveness

Makumba

By looking at the core ideas of Makumba, presented in the introduction to this essay, it is clear that Makumba aims to present simple, readable and maintainable code to those that want to work with it, or simply understand it. In this part, it will be investigated how well Makumba achieves this goal.

Makumba JSP tags

The blog index page, seen in listing 4.3 on page 21, is taken to be a good representation of how a basic Makumba JSP page might look. The first Makumba tag, `<mak:list from="general.Post p">` , looks rather simple at first but actually contains quite a lot of information. `mak:` is the Makumba tag prefix and `list from="general.Post p"` tells the reader that a list is generated from the all the records of type general.Post. The `p` at the end is the name of the pointer to each record.

This `p` is actually very significant when trying to interpret what the tag does. If the `p` wasn't present, most readers would probably interpret the tag as simply displaying a list of every record from general.Post, similar to what a SQL `SELECT * FROM post_table` statement would return. This is however not how `mak:list` works. As said above, the `p` is a pointer to each record, meaning that whatever is between the start and end tag will be executed once for every record in the list, with `p` being the way of accessing each specific record.

The `mak:list` tag performs rather well when it comes to most of the usability goals presented in section 2.8. For example, the tag is efficient and has good utility, because it is not unnecessarily long and supports all the major functions one would expect from such a tag. It is also rather easy to learn and remember how to use. However, because of the issue raised earlier about the letter `p` signifying the pointer to each record, the tag lacks a little when it comes to safety. Safety involves "protecting the user from dangerous conditions and undesirable situations". [16] If a

today have at least come in contact with some HTML, and many web designers also know how to code. Separation of business logic from presentation logic, as the design idiom is commonly called, is still something worth striving for, however.

reader misses this small letter and, as a result, interprets the tag to work differently, that could be seen as an undesirable situation.

This problem could potentially be solved by giving the pointer a more verbose name (`p` is only an arbitrary name), such as `current_post`. This solution has its own problem however. If the pointer is referenced many times, the code will be a lot longer, meaning that the efficiency of the tag decreases. Also, even after changing the pointer name, nothing in the tag explicitly tells the reader that the name is in fact a pointer. The name is actually declared as part of the `from` attribute, which is not very intuitive. Moving the pointer name assignment to its own attribute might be a better solution, making the tag look like `<mak:list from="general.Post" pointer="p">`. As said above however, this decreases the efficiency of the tag, so one will have to decide whether security or efficiency is more important in the design.

The `mak:value` tags show up in several places when data needs to be fetched from the database records. These tags are almost always relatively short and easy to understand, as they should be considering the basic but important functionality they provide.

Another simple tag is the `mak:if` tag, that takes a `test` attribute and displays its contents if the test returns `true`. Anyone familiar with basic programming if-clauses will probably understand this tag, although some of the operators are different from those of the large programming languages, like `=` for equals (instead of `==`) and `<>` for not equals (instead of `!=`). This might limit the memorability of the tag, as programmers could possibly have problems remembering these specific operators.

Makumba forms

Forms are an essential part of all dynamic web applications, and the blog application presented in this essay is no exception. Makumba provides several JSP tags for building and working with forms, as can be seen from the following excerpt from the page for editing posts:

Listing 4.13: Excerpt from editPost.jsp

```
<mak:object from="general.Post p" where="p=$id">
<mak:editForm object="p">
  <mak:action>
    showPost.jsp?id=<mak:value expr='p' />
  </mak:action>
  Title: <mak:input field="title" /><br/>
  Content: <mak:input field="content" /><br/>
  <input type=submit value="Save changes">
</mak:editForm>
```

The post that is to be edited is first fetched with the `mak:object` tag, described in the previous section. A form is then generated using the `mak:editForm` tag, with the pointer to the object supplied.

The presence of the `mak:action` tag shows that the Makumba developers have taken measures to improve the readability of the code. Every form needs an `action` attribute, which describes where to send the data. If this had to be included as a part of the main form tag, the tag would contain several nested tags and would be hard to read as consequence. Instead, using the `mak:action` tag it is possible to move the action attribute assignment to its own tag. This increases efficiency as it gets easier and quicker to get an overview of the form and where its data is sent.

The `mak:input` tags are very simple, and contain in many cases just one `field` attribute describing what data field they belong to. Based on the type of this field, the type of HTML input to use is then automatically determined.

Makumba Data Definitions

The Makumba MDDs are, for most applications, short and easy to read and understand. A good example is the data definition for posts in the Makumba blog:

Listing 4.14: Post.mdd

```
title = not null char[50]
content = not null text
```

As long as the reader is familiar with the term *null* (a reasonable assumption, considering its commonness in the software development domain), this file needs almost no explanation, and can hardly be misinterpreted. Adding things like custom validations and non-standard field types such as sets increases the complexity somewhat, but the MDDs can still be considered intuitive, as can be seen in the example below.

Listing 4.15: Post.mdd with added validation and field types

```
title = not null char[50]
content = not null text
tags = set general.Tag

length(title) {4..? } : "Title must be at least
                               four characters."
```

One interesting feature of the validations is that the custom error messages (the string at to the right of the colon) also work as documentation for the validation function, if they are formulated well.

Ruby on Rails

Being based on Ruby, a language with a focus on simplicity and productivity, it is easy to draw the conclusion that Rails probably has the same qualities. It remains to be seen whether this is the case in practice.

ERb HTML files

Just as with Makumba, the blog index page shown in listing 4.5 on page 23 is taken to be a good representation of a basic Rails ERb view.

All ERb tags are enclosed within `<% %>` symbols, or variants of them. Without any other characters after the percent signs, the tags simply embed standard Ruby statements. If the statements return something that is to be printed in the page, like the `link_to` function does, an `=` sign is needed after the first percent sign. The `=h` sign escapes HTML tag characters for security purposes.

Since these tags contain only 2-4 characters, they are very short and therefore efficient. However, in being so short they also have to compromise a bit of safety. When building the web application in Ruby on Rails for example, I found myself forgetting `=` signs after the percent signs several times. This had the effect of not displaying any output, as well as doing so *silently*, meaning it was relatively hard to find the cause of the problem.

The Rails tag for displaying all posts, `<% @posts.each do |post| %>` accomplishes the same thing as its Makumba counterpart, but looks radically different. Here, the `.each` statement clearly indicates that whatever is inside the tag is done once for *each* record. The pointer name is also clearly separated from the rest of the statement by using "pipes" (`| |`), making it easy to distinguish, although there is still nothing that explicitly states that it is a pointer name. Realizing that is part of understanding the basic semantics of the Ruby language, one could argue.

The `<% @posts.each do |post| %>` tag performs very well when judged on the basis of the usability goals presented earlier in this thesis. It is efficient because of its conciseness, safe because of its readability, has good utility when combined with an appropriate controller action, and is easy to learn and remember due to its similarity to natural language.

When outputting content from an object in Rails, the structure of the ERb language enable the statements to be very short and easy to understand. The statement to output the content from a post in the blog is simply `<%=h post.content %>`. As described earlier, the percent symbols signify the start of a Ruby statement, while the equality signs signify that the result of the statement is to be outputted to the page. The `h` enables HTML encoding of certain characters, for security purposes. All this information is described using only three characters, and the eyes of the reader are immediately drawn to the body of the statement, which describes the most important information — what field is to be displayed. The result of this is that once again, Rails demonstrates its power in generating short but readable code. The tag accomplishes all of the usability goals, with the minor exception being the safety issues involved in missing the `=` tag, explained earlier.

Ruby on Rails forms

Thanks to Rails' model-view-controller structure, form definitions can be made very concise and readable, and largely generated automatically. Rails also support reusing the same code for an "edit" and a "new" form. Consider the following example:

Listing 4.16: _form.html.erb

```
<% form_for @post do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :content %><br />
    <%= f.text_area :content %>
  </p>
  <p>
    <%= f.submit "Save" %>
  </p>
<% end %>
```

This is a *partial template*, indicated by the underscore in the file name. Because the `form_for` method takes `@post` as an argument, the post controller can tailor this form to fit both new posts and existing posts. For the form for adding new post , `@post` is set to `Post.new` in the `new` method of the controller. For the form responsible for editing posts, `@post` is instead set to `Post.find(params:id])`, getting the correct post from the database. This form in then included in both the `new.html.erb` and `edit.html.erb` files, using the statement `<%= render :partial => "form" %>`.

`form_for` is a built in *helper method*, that is able to determine very much information automatically, due to posts being declared as a RESTful resource in this application. [30] This makes using forms in Rails very efficient and safe, as most logic is done "under the hood". This simple approach is also easy to learn and remember.

Rails models

Just like in Makumba, the models in a Rails application are almost always short and easily readable. They are the part of the Rails application where the code most closely resembles natural langauge, as can be seen in the following example.

Listing 4.17: post.rb

```
class Post < ActiveRecord::Base
    validates_presence_of :title, :content
    has_many :comments
```

```
end
```

Due to the fact that field definitions are done in the command line tool, models can be kept short and only contain validations and associations.

Comparison

This is perhaps the criterion where there is the biggest difference between Makumba and Ruby on Rails. Since Makumba has chosen their tags to resemble HTML closely, they are limited in the design of the tags. Tags need to have the form of `<tagname attribute=value>` which is a rather long and sometimes unintuitive style, as the example with `mak:list` shows. Resembling HTML is of course sometimes a positive thing, the advantage being that it is rather easy to get started if you know mostly HTML.

A topic that comes up several times in the above investigation, is the one about safety versus efficiency. In both Makumba and Rails, there are some problems with safety, mostly from small characters that are easy to miss. Expanding these characters, however, would decrease efficiency as the tag itself would then be bigger. There is no right way to solve this problem, as it is down to personal preference whether safety or efficiency is better. More experienced coders will likely prefer efficiency to safety, as they are less prone to making errors, and the opposite is true to beginners.

4.7 Documentation and other available help

Makumba

Virtually all available documentation on the Makumba web development framework can be found on the official website, `http://www.makumba.org`. At the time of writing, the following documentation was available:

- ”Quick start” guide, describing how to install and get started with a basic Makumba application
- Information on how to configure Makumba using the configuration file
- ”Usage” section with more in depth information on certain subjects, such as:
  - Data definitions
  - Displaying data
  - Creating and editing data
  - Web 2.0 features (AJAX)
- ”Documentation” section, listing all available file types, validations, query language functions, JSP tags, the request lifecycle, and more.

Not all parts of Makumba are documented, however. For example, the "access control" page of the usage section only displays a "Coming soon" message, and the page titled "layout & navigation" clearly shows that documenting Makumba is a work in progress, with text such as "explain about error.jsp, ..." [21]

Makumba also has a mailing list available, linked to from the official website. Though certainly not the most active one — historical records indicate that the number of posts per month was between 0 and 20 — I was able to get answers to my questions in between an hour and five days. [36]

Ruby on Rails

The official documentation on Rails can be found on *Rails Guides*, a user-maintained subsection of the official Rails website. This site contains a "Getting Started with Rails" guide, and more in-depth guides covering several subjects regarding models, views and controllers. There is also a "Digging Deeper" section with guides on subjects such as internationalization, testing, security, debugging, and more. [33] In addition to this, a complete API is available, documenting all classes and methods existing in the Rails framework. [29]

Rails has a large community of developers. The official Rails mailing list, *Ruby on Rails: Talk* had around 20,000 members as of May 2010. The historical number of posts per month on this mailing lists ranged from around 2,000 to 5,000. [14] Apart from the official mailing list, several forums exist, such as Rails Forum with around 25,000 members and 116,000 total posts as of May 2010. [11]

Because of its large user base, there are also much information to be found on non-official sites such as blogs, as well as several written books specifically about Rails.

Comparison

The conclusions that can be drawn from the above investigation, is that Rails has a massive amount of documentation and sources of help available, while the features of Makumba are not fully documented, and the only source of private help is the official mailing list. This is not surprising, given that Rails has gained a lot of recognition since its creation, while Makumba has not.

Because of Makumba's background as a tool for supporting amateur communities (mentioned in the introduction), the lack of documentation could be seen as an intentional choice. This is mentioned in an article by Bogdan and Mayer, where the following is stated:

> "As there are not many resources dedicated to Makumba on the Internet, [the new members] have to resort to their fellow Tech Committee members, thus being forced into socialization, and thus fueling peer learning." [2]

This might be a good thing in this particular case, but if Makumba aims to be used for more than just amateur communities, it needs to be fully documented.

Chapter 5

Results

Based on the comparison done, several areas of improvement can now be identified for Rails and Makumba. The findings indicate that the web development framework in most need of improvement is Makumba. In its current implementation, as a JSP tag library, Makumba is rather limited in the structure of its tags. This results in making some tags rather long and cluttered, compared to the corresponding tags in Rails. On the upside, this implementation leads to Makumba tags being easier to understand for someone knowing only HTML, who might have problems interpreting Ruby.

Makumba also suffers from its deviation from the model-view-controller design pattern. This makes it hard to separate application logic from presentation logic, which might cause issues when working in a team, and in general decreases code readability (see section 4.5).

Interdependency is also something that can be seen in Makumba web applications. Migrations in Rails are a neat way of structuring database alterations, and Makumba could certainly benefit from a similar technology, which would decrease interdependency in that respect. Rails' use of routing removes the need for hard-coding links, and for large applications Makumba could also benefit from such a technology.

Finally, Makumba needs to be fully documented and provide extensive guides on many different subjects. A good way of finding out what to write could be to look at what is available in the Ruby on Rails documentation, and write similar documents for Makumba.

A possible area of improvement for Rails is to improve its accessibility for beginners. This might be hard because of its dependence on the Ruby language, however. If Ruby continues to gains more recognition, more people will probably find it easier to get started with Rails.

Chapter 6

Discussion

The results of this thesis seems to indicate that a large project might encounter problems if it is developed using Makumba. There is, however, an example of a large real world project that has been using Makumba and not encountered any major problems. The project, the intranet of a European-wide student organization, had around 2000 dynamic web pages at the end of 2008, and is maintained by a group of voluntary students. This group has had over 10 active members at any given time since 2002, with about one third of the group being renewed each year. In spite of the relatively large project and many new members each year, the intranet has been actively maintained, and several new features have been added every year. [2]

The reasons why the project has been so successful, despite the problems of Makumba identified in this thesis, are probably too many to cover fully here. Some discussion on the subject could however still be interesting. The project mentioned above is a typical "amateur community", a setting that Makumba was designed for from the beginning. Amateur communities are different from other software projects since they are often maintained by volunteers, and must be self-sustainable. This means that a framework that is good for building and maintaining ordinary websites might not also be good for amateur communities, and vice versa.

In this case, Makumba's simple design makes it easy to integrate new members in the project, something that is more important for an amateur community than, for example, separating logic from presentation. If the intranet mentioned above was built using a technology such as Ruby on Rails, it is possible that the project would 'die' eventually, as existing members quit the project and possible new members find the code base inaccessible.

The question that the Makumba developers should ask themselves is *"Is Makumba only supposed to be a tool for amateur communities?"*. If the answer is yes, then the success story from the student organization mentioned earlier should prove that Makumba is already rather well suited for projects in amateur communities. Some improvements could probably be made based on the results of this thesis, as long as their implications on Makumba projects' self-sustainability are considered.

If the answer is no, then Makumba has several areas where it could be improved, as presented in this thesis. However, if the self-sustainability of Makumba projects is to be kept somewhat intact, the implications of these new features will also need to be carefully considered. It might simply not be possible for Makumba to provide an appropriate amount of self-sustainability while also being well suited for ”normal” web application development.

Chapter 7

Conclusion

Ruby on Rails is a framework that has gained a lot more popularity than Makumba. The reasons for this could be several, but some of them can be seen in this thesis. Because of its foundations in the Ruby language, the code in Rails can get very short and readable. The design philosophies of Rails, like "convention over configuration", make it possible to develop powerful applications very quickly. The MVC pattern that it is based on also gives every piece of code a good place to sit, without cluttering up other parts of the application.

Makumba has a long way to go to be able to reach popularity akin to that of Rails, but has a few concepts that might be worth holding on to. The similarities of Makumba tags to HTML tags make it easy for beginners to understand and start developing with Makumba, but are something of a double-edged sword since they also limit the overall tag structure. Some kind of compromise would have to be made to make improvements in this area.

Chapter 8

Future work

The findings in this thesis could be the grounds for many different investigations. Of course, the comparison method applied in this thesis could be done on infinitely many combinations of web development frameworks, as well as many other types of software. A more interesting approach would probably be to finish the more objective method mentioned in the introduction to this thesis, where the task of programming the application is delegated to several other parties.

A whole other type of investigation could more closely examine the areas of improvement determined in this thesis. More work is needed to determine exactly how Makumba and Ruby on Rails should improve the areas found suitable for improvement.

Appendix A

Code

Listing A.1: Output from script/generate scaffold post title:string body:text command

```
      exists  app/models/
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/posts
      exists  app/views/layouts/
      exists  test/functional/
      exists  test/unit/
      create  test/unit/helpers/
      exists  public/stylesheets/
      create  app/views/posts/index.html.erb
      create  app/views/posts/show.html.erb
      create  app/views/posts/new.html.erb
      create  app/views/posts/edit.html.erb
      create  app/views/layouts/posts.html.erb
      create  public/stylesheets/scaffold.css
      create  app/controllers/posts_controller.rb
      create  test/functional/posts_controller_test.rb
      create  app/helpers/posts_helper.rb
      create  test/unit/helpers/posts_helper_test.rb
       route  map.resources :posts
  dependency  model
      exists    app/models/
      exists    test/unit/
      exists    test/fixtures/
      create    app/models/post.rb
      create    test/unit/post_test.rb
      create    test/fixtures/posts.yml
      create    db/migrate
      create    db/migrate/20100417132057_create_posts.rb
```

Listing A.2: Initial Rails project folder structure

```
./README
./Rakefile
```

```
./app
  ./app/controllers
    ./app/controllers/application_controller.rb
  ./app/helpers
    ./app/helpers/application_helper.rb
  ./app/models
  ./app/views
    ./app/views/layouts
./config
  ./config/boot.rb
  ./config/database.yml
  ./config/environment.rb
  ./config/environments
    ./config/environments/development.rb
    ./config/environments/production.rb
    ./config/environments/test.rb
  ./config/initializers
    ./config/initializers/backtrace_silencers.rb
    ./config/initializers/inflections.rb
    ./config/initializers/mime_types.rb
    ./config/initializers/new_rails_defaults.rb
    ./config/initializers/session_store.rb
  ./config/locales
    ./config/locales/en.yml
  ./config/routes.rb
./db
  ./db/seeds.rb
./doc
  ./doc/README_FOR_APP
./lib
  ./lib/tasks
./log
  ./log/development.log
  ./log/production.log
  ./log/server.log
  ./log/test.log
./public
  ./public/404.html
  ./public/422.html
  ./public/500.html
  ./public/favicon.ico
  ./public/images
  ./public/images/rails.png
  ./public/index.html
  ./public/javascripts
    ./public/javascripts/application.js
    ./public/javascripts/controls.js
    ./public/javascripts/dragdrop.js
    ./public/javascripts/effects.js
    ./public/javascripts/prototype.js
  ./public/robots.txt
  ./public/stylesheets
./script
  ./script/about
  ./script/console
  ./script/dbconsole
  ./script/destroy
  ./script/generate
  ./script/performance
    ./script/performance/benchmarker
    ./script/performance/profiler
  ./script/plugin
  ./script/runner
  ./script/server
./test
  ./test/fixtures
  ./test/functional
  ./test/integration
  ./test/performance
    ./test/performance/browsing_test.rb
  ./test/test_helper.rb
  ./test/unit
./tmp
  ./tmp/cache
  ./tmp/pids
  ./tmp/sessions
  ./tmp/sockets
./vendor
  ./vendor/plugins
```

Bibliography

[1] Cristian Bogdan. *IT Design for Amateur Communities.* PhD thesis, Royal Institute of Technology, 2003.

[2] Cristian Bogdan and Rudolf Mayer. Makumba: the role of the technology for the sustainability of amateur programming practice and community. In *C&T '09: Proceedings of the fourth international conference on Communities and technologies*, pages 205–214, New York, NY, USA, 2009. ACM.

[3] TIOBE Software BV. *TIOBE Programming Community Index for April 2010.* `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, April 2010.

[4] The Ruby Community. *Ruby Programming Language.* `http://www.ruby-lang.org/en/`, April 2010.

[5] Paragon Corporation. *Separation of Business Logic from Presentation Logic in Web Applications (ASP.NET and PHP).* `http://www.paragoncorporation.com/ArticleDetail.aspx?ArticleID=21`, July 2003.

[6] Liam Delahunty. *Linux - Search and replace over multiple files.* `http://www.liamdelahunty.com/tips/linux_search_and_replace_multiple_files.php`, April 2010.

[7] Greg DeMichillie. *Microsoft ASP.NET 2.0 Next Stop on Microsoft Web Development Roadmap.* `http://www.directionsonmicrosoft.com/sample/DOMIS/update/2004/08aug/0804a2nsow.htm`, July 2004.

[8] Michael English and Patrick McCreanor. *Exploring the Differing Usages of Programming Language Features in Systems Developed in C++ and Java.* `http://www.ppig.org/papers/21st-english.pdf`, March 2010.

[9] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, 1997.

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

[11] Rails Forum. *Rails Forum - Ruby on Rails Help and Discussion Forum.* `http://www.railsforum.com/`, May 2010.

[12] The Apache Software Foundation. *What is Maven?* `http://maven.apache.org/what-is-maven.html`, April 2010.

[13] The PHP Group. *PHP: Preface - Manual.* `http://www.php.net/manual/en/preface.php`, May 2010.

[14] Google Groups. *Ruby on Rails: Talk.* `http://groups.google.com/group/rubyonrails-talk/about`, May 2010.

[15] David Heinemeier Hansson. *Pursuing beauty with Ruby on Rails.* `http://media.rubyonrails.org/presentations/pursuitofbeauty.pdf`, May 2010.

[16] Yvonne Rogers Helen Sharp and Jenny Preece. *Interaction design: beyond human-computer interaction.* John Wiley & Sons, Ltd., 2nd edition, 2007.

[17] Cake Software Foundation Inc. *What is CakePHP? Why Use it?* `http://book.cakephp.org/view/8/What-is-CakePHP-Why-Use-it`, May 2010.

[18] Reuven Lerner. *Can Rails Scale? Twitter Raises Some Questions.* `http://ostatic.com/blog/can-rails-scale-twitter-raises-some-questions`, May 2008.

[19] Zend Technologies Ltd. *Zend Framework & MVC Introduction.* `http://framework.zend.com/manual/en/learning.quickstart.intro.html`, May 2010.

[20] Zend Technologies Ltd. *Zend Framework: About.* `http://framework.zend.com/about/overview`, May 2010.

[21] Makumba. *Makumba: Layout Howto.* `http://www.makumba.org/page/LayoutHowto`, May 2010.

[22] Makumba. *Makumba: Main.* `http://www.makumba.org`, March 2010.

[23] Makumba. *Quick Start.* `http://www.makumba.org/page/QuickStart`, April 2010.

[24] The Agile Manifesto. *Manifesto for Agile Software Development.* `http://agilemanifesto.org/`, March 2010.

[25] Sun Microsystems. *The J2EE(TM) 1.4 Tutorial.* `http://java.sun.com/j2ee/1.4/docs/tutorial/doc/`, 2005.

[26] Sun Microsystems. *JavaServer Faces Overview.* `http://java.sun.com/javaee/javaserverfaces/overview.html`, April 2010.

[27] Sun Microsystems. *JavaServer Pages Overview.* `http://java.sun.com/products/jsp/overview.html`, Aprl 2010.

[28] University of Illinois at Urbana-Champaign. *CGI: Common Gateway Interface.* `http://hoohoo.ncsa.illinois.edu/cgi/intro.html`, March 2010.

[29] Ruby on Rails. *Rails Framework Documentation.* `http://api.rubyonrails.org/`, May 2010.

[30] Ruby on Rails. *Rails Framework Documentation: form_for.* `http://api.rubyonrails.org/classes/ActionView/Helpers/FormHelper.html#M001604`, May 2010.

[31] Ruby on Rails guides. *Migrations.* `http://guides.rubyonrails.org/migrations.html`, April 2010.

[32] Ruby on Rails guides. *Rails Routing from the Outside In.* `http://guides.rubyonrails.org/routing.html`, April 2010.

[33] Ruby on Rails guides. *Ruby on Rails Guides.* `http://guides.rubyonrails.org/`, May 2010.

[34] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[35] Pauld D. Sheriff. *Introduction to ASP.NET and Web Forms.* `http://msdn.microsoft.com/en-us/library/ms973868.aspx`, November 2001.

[36] SourceForge.net. *makumba-user mailing list.* `http://sourceforge.net/mailarchive/forum.php?forum_name=makumba-user`, May 2010.

[37] Noelios Technologies. *Restlet - Features.* `http://www.restlet.org/about/features`, May 2010.

[38] Noelios Technologies. *Restlet - Introduction.* `http://www.restlet.org/about/introduction`, May 2010.

[39] Dave Thomas and David Heinemeier Hanson. *Agile Web Development with Rails.* The Pragmatic Programmers LLC., 2nd edition, 2007.

[40] Bill Walton and Curt Hibbs. *Rolling with Ruby on Rails Revisited.* `http://oreilly.com/ruby/archive/rails-revisited.html`, December 2006.

[41] Rails Wiki. *Basic Tenets of Ruby on Rails.* `http://wiki.rubyonrails.org/getting-started/overview/tenets`, April 2010.