Dimitar Zlatkov

Development of Web Community Application with Ruby on Rails framework

Bachelor Thesis based on the examination and study
regulations for the Bachelor of Engineering degree
programme Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg

Supervising examiner: Prof. Dr. Hans-Jürgen Hotop
Second examiner: Prof. Dr. Dieter Müller-Wichards
Day of delivery May 28th 2009

Dimitar Zlatkov

Title of the Bachelor Thesis

Development of Web Community Application with Ruby on Rails framework

Keywords

Ruby, Ruby on Rails, Web Application, Community, Internet, WWW, MVC Framework, HTML, MySQL, Mongrel

Abstract

This thesis discusses the development steps of a web-based community application that enables a user to log-in, communicate and interact with other users in a browser based environment. It describes in detail the implementation of an application front-end and back-end. The thesis has formulated and defined an approach to develop such a web application using the Ruby on Rails technology.

Dimitar Zlatkov

Thema der Bachelorarbeit

Entwicklung von Web-Community-Applikation mit Ruby on Rails Framework

Stichworte

Ruby, Ruby on Rails, Web Application, Community, Internet, WWW, MVC Framework, HTML, MySQL, Mongrel

Kurzzusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung einer web-basierten Community Platform, die es eingeloggten Benutzern erlaubt, in einer Browser-basierten Umgebung, miteinander zu interagieren. Es beschreibt detailliert die Implementation von Front- und Back-End einer Anwendung mit Hilfe der Ruby-on-Rails-Technologie.

Table of Contents

List of Figures

Chapter 1 – Introduction

1.1 The Problem

Nowadays, social web communities and networks have become part of everyday life of the Internet user. With the rapid growth of the World Wide Web, Internet communities have earned a wide popularity. Web communities have become a new way of communication between different people.

Many social clubs and organizations are still not adjusted to the modern way of Web 2.0 communication. Club members and company employees communicate between each other using telephone, mobile phone, instant messaging and mail services. These ways of communication can overcome long distances and are sufficient for a one-to-one communication. However, they can lead to additional costs, difficulties and misunderstandings when it comes to simultaneous communication between a group of people. Phones and mobile calls could spend sometimes a large amount from the club or organization's budget. Sending emails could be sometimes insufficient. For example, the club chairperson sends an email to all of the club members. Not every member is using the same mail service provider. Different providers can handle the email message differently, and some could handle it as Spam. In this way some club members will not receive the message.

From the above problems, the need was realized to develop a Web 2.0 application, which will provide more convenient, faster, cheaper and efficient way of communication between group of members. One of the most commonly used technologies to realize a web-based community application is the client-server architecture. The project implements a standard client-server web application, where the client sends a request for a resource located on the server, and the server application responds the required resource to the client.

1.2 The Web Community Application Overview

The Web Community application aims at supporting application users in communication and interaction with other application members by providing different functions and a user friendly interface.

The application interacts with the users through a website, which is always available via the Internet. The project is divided into two functional units – application front-end and application back-end.

Application Front-end

The front-end of the Web Community application is the user interface. The website page is the application interface, where the users can interact with the program. Club members or company employees have their user website account. They can log-in using their email address and application password. They can check their community mailbox, read and send messages. Application users can add club members as friends, communicate with them, share images and profiles with them. Additionally user can create albums, add images, link images to map, change profile, search other users, search addresses and locations on the World's map, save locations, etc.

Application Back-end

The application back-end consists of a relational database of different database tables, where the user data is stored and from where the user data is retrieved. When the community user submits a request, request values are checked and structured in the front-end and then sent to the back-end. The database handles these values and responds to the front-end to the user.

1.3 The Scope of this Report

The Objective

The objective of this project is to design and implement a prototype version of a web-based community platform using the Ruby on Rails technology . The application user interface will have to handle all the requests from the users, refine the input, send queries to the database, handle the response from it, refine the response and then send to the web interface to the users. This is the main task of the report, and it will focus on the development of a web-based application with Ruby on Rails framework.

The Target Audience

The target audience of this report are social network groups. Groups of people, who are members of the same club or are part of the same organization or company will largely benefit from the use of this application. The concept of social networking that has been introduced later in the chapters helps the application users to share their comments, images and profiles to other fellow members of the web community. These actions provide the perfect environment for an efficient user interaction. Logged-in as administrator user, the club chairperson can also use the community platform to aid the process of communication.

Normal everyday web users also benefit from the use of the web community application. These users can interact with the application by searching places of interest and addresses locations on the World's map.

1.4 The Structure of this Report

This report is divided into seven chapters. The thesis starts with an index page, followed by a list of figures. The report is written using the following chapter structure.

Chapter 1 is the Introduction chapter.

Chapter 2 presents the technologies needed to build up this application. This chapter introduces the reader to the Model-View-Controller architecture. The Ruby on Rails framework and the Ruby object-oriented programming language are also introduced.

Chapter 3 discusses the functional requirements for the Web Community application. This includes the user requirements which are mapped to the base functions of the application. Different possibilities for the realization are presented and discussed. The specific software to be used is presented and the reasons for choosing this software are pointed out. In conclusion the hardware requirements are discussed.

Chapter 4 examines in detail the design steps for realizing the Web Community application. An overview of the different implementation steps of the project is presented. The Model-View-Controller architecture and the relational database model of the application are discussed.

Chapter 5 describes the process of building up the application based on the system design architecture presented in chapter four. This chapter focuses on the realization of different parts of the Web Community application design. Some specific software techniques are investigated in detail.

Chapter 6 discusses the different techniques used for testing the Web Community application. Project testing is done to guarantee that the application is stable, reliable and safe from hacker attacks. Future project development, implementation of new parts, and extension of the application functionality is discussed as well.

Chapter 7 is the conclusion chapter of this report. A summary of the techniques used and design implementation is presented.

At the end of the report, the reader can find a References page followed by an Appendix. The CD with the source code and documentation of this project is also provided.

Chapter 2 - Software Tools Overview

This chapter will give a short explanation of some abbreviations and keywords, which will be used further in the report. The information included in this section aims to make reading this report easier to follow by the reader.

Web application

In software engineering, a web application or web-app is an application that is accessed with a web browser over a network such as the Internet. It is a computer software application, coded in a browser-supported language such as HTML and Javascript, and reliant on the web browser to render the application executable. Web applications are popular because the browser acts as a client, sometimes called a thin client. The ability to update and maintain Web applications without distributing and installing software on potentially thousands of client computers is a key reason for their popularity. Web applications are used to implement many different kind of tasks such as web mails and calendars, online shops and auctions, currency converters and etc.

Online Community

An online community or a virtual community is a group of people who primarily interact via communication media such as email, Internet social network service or instant messages rather than face to face, for social, professional, educational or other purposes. Nowadays virtual and online communities have become a supplemental form of communication between people who share common business interests, hobbies or friendship.

Web Framework

A web application framework is a software framework designed to support the development of dynamic websites, web applications and web services. Web application frameworks facilitate rapid application development by allowing the programmer to define a high-level description of the program. Many frameworks provide libraries for database access, template and session management, and often promote code reuse. One architectural pattern of a web framework is the Model View Controller (MVC) architecture, designed to separate the data model with business rules from the user interface. Most MVC frameworks follow a push-based architecture. These frameworks use actions that do the required processing, and then "push" the data to the view layer to render the results. Ruby on Rails and Spring MVC are good examples of this architecture. An alternative to this is pull-based architecture, sometimes also called "component-based". These frameworks start with the view layer, which can then "pull" results from multiple controllers as needed. In this architecture, multiple controllers can be involved with a single view. Figure 2.1 visualizes the MVC architecture.

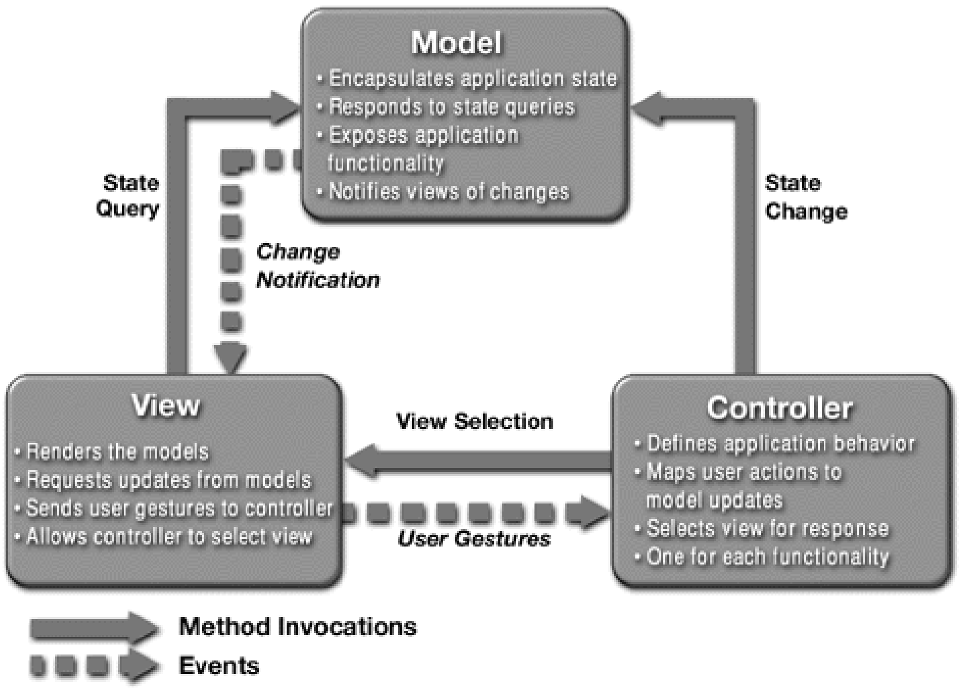

Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
State Change
Change Notification
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

Fig. 2.1 The Model View Controller architecture [7].

Don't Repeat Yourself

Don't Repeat Yourself or DRY is a process philosophy aimed at reducing duplication. The philosophy emphasizes that information should not be duplicated, because duplication increases the difficulty of change, may decrease clarity, and leads to opportunities for inconsistency. DRY code is created by data transformation, which allows the software developer to avoid copy and paste operations. DRY code usually makes large software systems easier to maintain and becomes increasingly important in applications that use multi-tier architectures.

Convention over Configuration

Convention over Configuration or CoC is a software design paradigm which seeks to decrease the number of decisions that software developer needs to make, gaining simplicity, but not necessarily losing flexibility. CoC essentially means a developer only needs to specify unconventional aspects of the application. For example, if there's a class User in the model, the corresponding table in the database is called users by default.

Representational state transfer

Representational state transfer or REST is a style of software architecture for distributed hypermedia systems such as the World Wide Web (WWW). REST refers in the strictest sense to a collection of network architecture principles which outline how resources are defined and addressed. Systems which follow REST principles are often referred to as "RESTful".

In the REST architecture, data and functionality are considered resources, and these resources are accessed using Uniform Resource Identifiers (URIs), typically links on the web. The resources are acted upon by using a set of simple, well-defined operations. The REST architecture is fundamentally a client-server architecture, and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture, clients and servers exchange representations of resources using a standardized interface and protocol. These principles encourages REST applications to be simple, lightweight, and have high performance. A RESTful web application typically map the four main HTTP methods: POST, GET, PUT, and DELETE respectively to database operations: create, retrieve, update, and delete, so-called CRUD actions.

Ruby

Ruby is a dynamic, open source script programming language with a focus on simplicity and productivity. It is object-oriented, platform-independent and supports multi-threading. In Ruby, every data type is an object, every function is a method. Every bit of information and code can be given their own properties and actions. Ruby has exception handling features, like Java or Python, which makes it easy to handle errors. The syntax of Ruby is broadly similar to Perl and Python. Since its public release in 1995 (Ruby 0.95), Ruby has drawn devoted coders worldwide. In 2006 (Ruby 1.8.6), Ruby achieved mass acceptance. The latest stable version of Ruby (Ruby 1.9.1) was released on 30 January 2009. It introduces a lot of changes and it is nearly two times faster compared to version 1.8.6. For more information about the Ruby programming language, please visit Ruby web page [5].

Ruby on Rails Framework

Ruby on Rails, RoR or just Rails is a web development framework written in Ruby language. It is designed to make programming web applications easier. Rails makes it possible for the programmer to write less code, while accomplishing more than many other languages and frameworks. Rails is organized around the Model, View, Controller architecture. MVC benefits for isolating the business logic from the user interface, keeping programming code DRY and making it clear where different types of code belong for easier maintenance of the application. For more information about Ruby on Rails, please visit the Rails web page [6]. Figure 2.2 shows a standard architecture of a RoR web application.

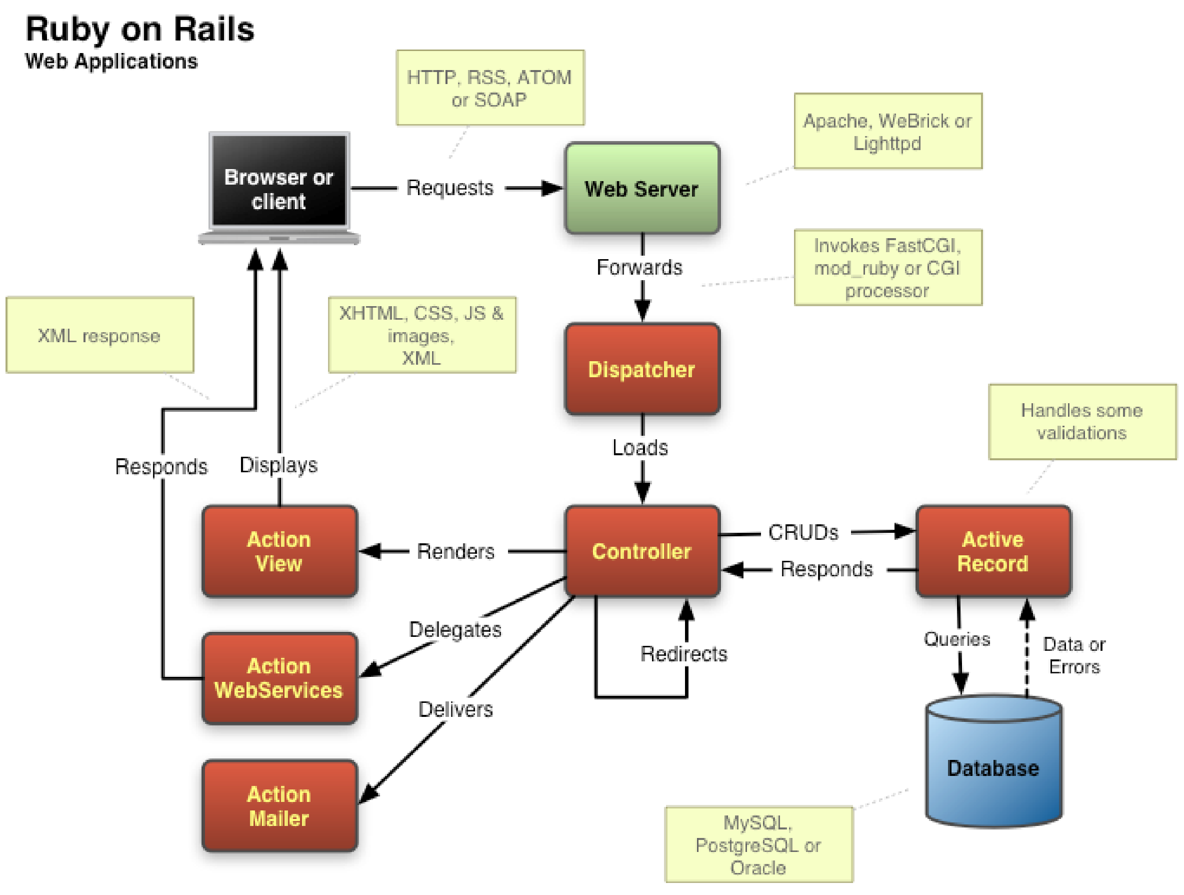


Fig. 2.2 Architecture of standard Ruby on Rails web application [8].

Models

A model represents the information data of an application and the rules to manipulate that data. In the case of Rails, models are primarily used for managing the rules of interaction with a corresponding database table. In most cases, one table in _the database will correspond to one model in the application. The bulk of the application’s business logic will be concentrated in the models.

Views

Views represent the user interface of the application. In Rails, views are often HTML files with embedded Ruby code that performs tasks related solely to the presentation of the data. Views handle the job of providing data to the web browser or other tool that is used to make requests from the application.

Controllers

Controllers provide the “glue” between models and views. In Rails, controllers are responsible for processing the incoming requests from the web browser, interrogating the models for data, and passing that data on to the views for presentation.

Ruby on Rails Components

Rails provides a full stack of components for creating web applications. They are the core classes of the Ruby on Rails framework.

Action Controller (ActionController::Base)

Action Controller is the component that manages the controllers in a Rails application. The Action Controller framework processes incoming requests to a Rails application, extracts parameters, and dispatches them to the intended action. An action is defined as a public method on the controller, which will automatically be made accessible to the web-server through Rails Routes. The two basic action archetypes used in Action Controllers are get-and-show and do-and-redirect. Most actions are variations of these themes. Services provided by Action Controller include session management, template rendering, and redirect management.

Action View (ActionView::Base)

Action View manages the views of a Rails application. It can create both HTML and XML output by default. Action View manages rendering templates, including nested and partial templates, and has built-in AJAX support. Action View templates can be written in three ways. Templates, using a mixture of embedded Ruby (ERb) code and HTML code, have an .html.erb file extension. Templates, using the Jim Weirich’s Builder::XmlMarkup library, have the .builder or .rxml file extension. Templates, using the ActionView::Helpers::PrototypeHelper::JavaScriptGenerator module, have .rjs file extension.

Active Record (ActiveRecord::Base)

Active Record is the object-relational mapping (ORM) layer supplied with Rails. It closely follows the standard ORM model: tables map to classes, rows to objects, and columns to object attributes. It provides database independence, basic CRUD functionality, advanced finding capabilities, and the ability to relate models to one another, among other services. Active Record objects don’t specify their attributes directly, but rather infer them from the table definition with which they are linked. Adding, removing, and changing attributes and their type are done directly in the database. Any change is instantly reflected in the Active Record objects.

Action Mailer (ActionMailer::Base)

Action Mailer is a framework for building e-mail services. Action Mailer can be used to send emails based on flexible templates, or to receive and process incoming email.

Active Resource (ActiveResource::Base)

Active Resource provides a framework for managing the connection between business objects a RESTful web services. It implements a way to map web-based resources to local objects with CRUD semantics.

Active Support (ActiveSupport::Base64)

Active Support is an extensive collection of utility classes and standard Ruby library extensions that are used in the Rails, both by the core code and by the application's code.

Ruby on Rails 2.1.x - Directory Structure

Rails framework assumes a specific run-time directory layout. Here is a structure of the top-level directories generated on rails project creation.

app: This directory organizes the application components.

app/controllers: The controllers sub-directory is where Rails looks to find controller classes.

app/helpers: The helpers sub-directory holds any helper classes used to assist the model, view, and controller classes.

app/models: The models sub-directory holds the classes that model and wrap the data stored in the application's database.

app/views: The views sub-directory holds the display templates to fill in with data from the application, convert to HTML, and return to the user's browser.

config: This directory contains the small amount of configuration code that an Rails application needs, including the database configuration (in database.yml), the Rails environment structure (in environment.rb), and routing of incoming web requests (in routes.rb). One can also tailor the behavior of the three Rails environments for test, development, and production with files found in the config/environments directory.

db: This directory holds the migration files of the application (in db/migrate) and the schema.rb

doc: This directory is where the application documentation will be stored when generated using rake doc:app.

lib: This directory contains application specific libraries and any kind of custom code that doesn't belong under controllers, models, or helpers.

log: Error logs go here. Rails creates scripts that help to manage various error logs. In this directory are the separate logs for the server (server.log) and each Rails environment (development.log, test.log, and production.log) stored.

public: Like the public directory for a web server, this directory has web files that don't change. It should be set as the DOCUMENT_ROOT of the used web server.

public/images: Sub-directory for the image files and graphics

public/javascripts: Sub-directory for the Javascript files.

public/stylesheets: Sub-directory for the CSS files

script: This directory holds scripts to launch and manage the various tools that come with Rails.

test: The tests written and those created by Rails go here. You'll see a sub-directory for fixtures (test/fixtures) functional (test/functional), integration (test/integration) and unit tests (test/unit).

tmp: Rails uses this directory to hold temporary files for intermediate processing.

vendor: Libraries provided by third-party vendors (such as security libraries, database utilities or plug-ins beyond the basic Rails distribution) go here.

Apart from these directories there are two files available in the root directory.

README: This file contains a basic detail about Rails application and description of the directory structure explained above.

Rakefile: This file is similar to Unix Makefile which helps with building, packaging and testing the Rails code. This will be used by rake utility supplied along with Ruby installation.

Chapter 3 - Project Analysis and Requirements

In this chapter the main requirements that the project has to fulfill are pointed out. The possible software solutions are presented. The software needed for the project implementation is discussed and reasons for choosing it is given. A discussion of the hardware requirements is made as well.

3.1 Project Analysis

The desire to a community is as old as humanity and online social networking sites do seem to solve a need that is different from simply using email, chat and blogging tools separately. Social network service focuses on building online communities of people who share interests and activities, or who are interested in exploring the interests and activities of others. Web based social networks provide a variety of ways for users to interact, such as e-mail and instant messaging services. Social networking has encouraged new ways to communicate and share information. The traditional way to interact face-to-face has been substituted by the interactive technology, which makes it possible for people to network with their peers from anywhere, at any time, in an online environment. Community websites are being used regularly by millions of people. In the past few years they have become an enduring part of everyday life.

In general, social networking services can be broken down into two broad categories: internal social networking (ISN) and external social networking (ESN) sites, such as Orkut, MySpace, Facebook, Twitter. Both types can increase the feeling of community among people. An ISN is a closed, private community that consists of a group of people within a club, company, association, society, education provider or organization. An ESN is open, public and available to all web users community. It can be small specialized community (i. e. community of people linked by a single common interest) or it can be large generic social networking site such as MySpace or Facebook. Based on the domain of application social networks can be divided into various types of networks and communities. Social communities can function as online meeting places for business and industry professionals. This allows individuals to be accessible from anywhere, at any time in an online environment and establish their real identity in a verifiable place. A professional network is used for the business to business marketplace. These networks improve the ability for people to advance professionally, by finding, connecting and networking with others. Business professionals can share experiences with others who have a need to learn from similar experiences. One example of social networking being used for business purposes is LinkedIn.com, which aims to interconnect professionals. List of popular social networking websites and communities is provided in the references [10].

Almost all social community applications have a set of features which are considered essential to qualify as a social networking service, namely: the ability to set up and customize a personal “profile”, an ability for members to comment, fine granular control of who sees what (privacy settings), ability to block an unwanted member, have own page of personal, blog like entries or notes and individual picture albums and ability to own, form or be member of a Group or Community within the network. Some additional features include the ability to create groups that share common interests or affiliations, ability to upload or stream live videos, and ability to hold discussions in forums. The users of a social community can be divided into three basic groups – visitors, normal users and super users. The visitor can only observe the community and view its content. They are not allowed to add new content or participate in discussions. Visitors are usually not visible to other community members. Normal users have the full access to all the community features. They have own page in the community and are visible to other members. The super users have extended control privileges.

This report focuses only on the first category of services – ISN web applications. Large generic social communities are out of the scope of the report and will not be point of discussion. The work describes in detail and focuses on the development of an internal social networking application, closed or private community. The project aims to reach a small, closed group of people, who are members of real life social organization, club or company. People of this community will share common interests. The application will make it easier to keep in touch with contacts around the world. It aims to improve the ability for people to advance professionally, by finding, connecting and networking with others.

In the following sub chapters the functional, software and hardware requirements for developing a social network/web community are discussed. In other words the project needs in terms of functionality, software and hardware are pointed out.

3.2 Project Functional Requirements

Functional requirements describe the desired functions of the project, they are the core of each project and they have an important role in the success of the project. This sub chapter starts with a discussion of the main functional needs and continues with explaining each of them in detail. As a small web community platform the project needs to implement some basic features. In order to achieve a social network functionality the application has to include user roles, profiles, semi-persistent public commentary on the profile, messaging system and a traversable publicly articulated social network displayed in relation to the profile. Including all these features in a web application transforms this application to a basic social network service. During the project development additional features can be added on to expand the application functionality. In the following section the main features are listed and discussed.

User Role. The project application requires different types of users, where user role corresponds to the privileges given to a certain user. Based on the application role users can be divided into two categories: superusers - administrators, and normal users. Superuser is a user with extended administrative privileges. He has control over the users in the social network. Superuser can delete, suspend or activate a normal user. Administrator user can access every user's profile, delete messages and photos with inappropriate content. Normal user is a user, member of the web community, who can modify only his/her profile and has no administrative privileges. A small community platform has to implement different user roles. A social network application needs to have administrator users, who have control over the network and can make decisions about users and their profile's content. Without user roles a small community can hardly exist and will be very difficult to maintain.

Profile. The web community application requires user profiles. A profile includes an identifiable handle - the person's name or nick name, information about that person (e.g. age, sex, location, interests, etc.). User profiles also include a photograph and information about last log-in. Profiles have unique URLs that can be visited directly. User has the ability to modify his/her web page, i.e. his/her profile. Community individual has the option to create new albums, add pictures to his/her album, post comment on them or link them to a location. A small community application has to have user profiles, because they represent the user in the social network. Profiles act as a presentation of user's identity in front of other community members and this is the reason why they have to be persistent in social network application.

Semi-persistent public comments. Participants can leave comments - guest book messages on other user profiles or photo comments for everyone to see. These comments are semi-persistent in that they are not ephemeral but they may disappear over period of time or upon removal. These comments are showed in the guest book section or in the album section of the user's profile and are reverse-chronological in display. Because of these comments, profiles are a combination of an individuals' self-expression and what others say about that individual. A small community needs to implement public comments feature, since comments form a way for the user to express his/her opinion in public. Public comments can reveal parts of member's identity and can be reason for communication between members, who do not know each other. The use of public comments contribute to the social network development and a web community project can only benefit from them.

Traversable, publicly articulated social network. Participants have the ability to list other profiles as “friends” or “contacts” or some equivalent. This generates a social community graph which may be directed (“attention network” type of social network where friendship does not have to be confirmed) or undirected (social network where the other person must accept friendship). This articulated social network is displayed on an individual's profile for all other users to view. Each node contains a link to the profile of the other person so that individuals can traverse the network through friends of friends. Becoming a friend with other community users, individual exposures his/her profile. In other words user's avatar, friends, albums and photos are visible and shared between members of the network who are linked as “friends”. Relations between users play an important role in communities, the user has the ability to distinguish between different members, to select individuals and allow them to see his/her profile. These are some of the main reasons why social network application has to implement relationships between users.

Messaging system. Communication is an important part of Web Community application. The social network requires a messaging system which allows users to communicate between each other at any time when they are logged in. User are allowed to compose, send and receive messages, as well as send and receive friendship invitations and requests. Using the messaging system, administrators has the ability to contact certain user or a group of users, provide important information about events or happenings. A small community needs to implement a messaging system, since the communication between users form the foundations of the communities. Without any messaging system community platform cannot exist.

3.3 Project Software Requirements

In this sub chapter the software needed for the implementation of the required social network functionality is discussed. Different types of software can be used for the implementation of the project. The following section focuses on some advantages and disadvantages of the possible software solutions, starting from Web content management system, through MVC frameworks like RoR, Zend and Cake, to Java programming.

Web content management system (WCMS or Web CMS) is content management system software, usually implemented as a web application, for creating and managing HTML content. It is used to manage and control a large, dynamic collection of web material such as HTML documents, images, audio and video files. A WCMS facilitates content creation, content control, editing, and many essential web maintenance functions. A Content Management System works by storing files and text into a database. When a web page is requested, the CMS system accesses the database and renders the web page. Because the data is separated from the code, changes to the data can be made using a web interface that requires no knowledge of HTML. Benefits of WCMS are that the administration is typically done via browser-based interfaces, it provides a web-site maintenance tool for non-technical administrators, many plug-ins and additional features exist for CMS applications. A lot of software companies, provide different content management systems, but not all of them are open source and free of charge. Some disadvantages are: Content Management systems are resource hungry, servers they run on need to be maintained by technicians, and content management requires more memory, CPU power and software maintenance; Content Management hosting is expensive since CMS require additional software (ASP, PHP or CGI) installed; Content Management systems need to be upgraded to changes in software - new versions of server software or office software will impact on the functionality of the CMS; Many CMS systems do not index properly on search engines.

Ruby on Rails is MVC framework designed for web application development. It offers higher developer productivity and less complexity. Because Rails is a complete solution, there are much fewer decisions to be made and it is much faster to get started. Some benefits of Rails are: Rails uses the dynamics of Ruby to bring back web application development closer to the productivity of productive non-web application development frameworks; Rails applications require a few lines of code to identify each database connection, as well as one line of configuration for each different type of routing; Rails embraces the DRY principle; Ruby on Rails includes all the components necessary to build complete web applications from the database forward (even including a pure-Ruby web server for those who wish to develop immediately without setting up a web server such as Apache), providing object-database linkage, a Model-View-Controller framework, unit and functional testing tools, out-of-the-box support for AJAX and CSS, support for multiple template systems, multi-environment deployments, support for automated local and remote deployments, inbound and outbound email support, web services support, etc. Ruby on Rails has many disadvantages. One potential constraint is that Ruby is relatively slow compared to other programming languages. Rails has a lot of techniques to compensate the slowness of Ruby in most scenarios, however those techniques involve more resources. Second constraint is the security. There are well established tools, libraries, and techniques in corporate environment that have leveraged C++/Java for a long time. Ruby on Rails is just too new and a bit immature to be able to challenge them yet. Another disadvantage is the lack of complete documentation online. Some of the generated rubydocs will just contain the name of the method, compared to php.net/ sample-function-name-here with lots of comments and tutorials or the javadocs from Sun. For more information please look at the documentation [9].

PHP is a popular scripting language originally designed for producing dynamic web pages, scales well, and offers many third party components. There are a number of MVC frameworks (including CakePHP and Zend), but none seem to be a standard. Another reported advantage of PHP over other languages is availability of a large pool of developers. This leads to very large community support. Like Ruby, PHP has very fast development cycle. PHP has no formal specification.

Java is platform independent object oriented programming language. Java is many times faster on code execution than other programming languages. It has the lowest hosting costs, and best scalability options, but the more complex solution. Developer speed and productivity varies drastically depending on which platforms, third party tools and development environments are chosen. Poor decisions have far reaching consequences. One famous open source application framework for the Java platform is Spring. The framework was first released under the Apache 2.0 license in June 2003. The current version of Spring is 2.5.6. Spring framework features its own MVC framework, which was not originally planned. Spring is not a single framework. Rather it is more of a collection of independent tools, packages, bind by its Core package. Web package provides a Model-View-Controller implementation for web applications. Spring framework is very well supported and documented. Spring enforces good programming practices like coding to interfaces, reducing coupling and allowing easy testability. Figure 3.1 shows the six modules on which the Spring framework is organized.

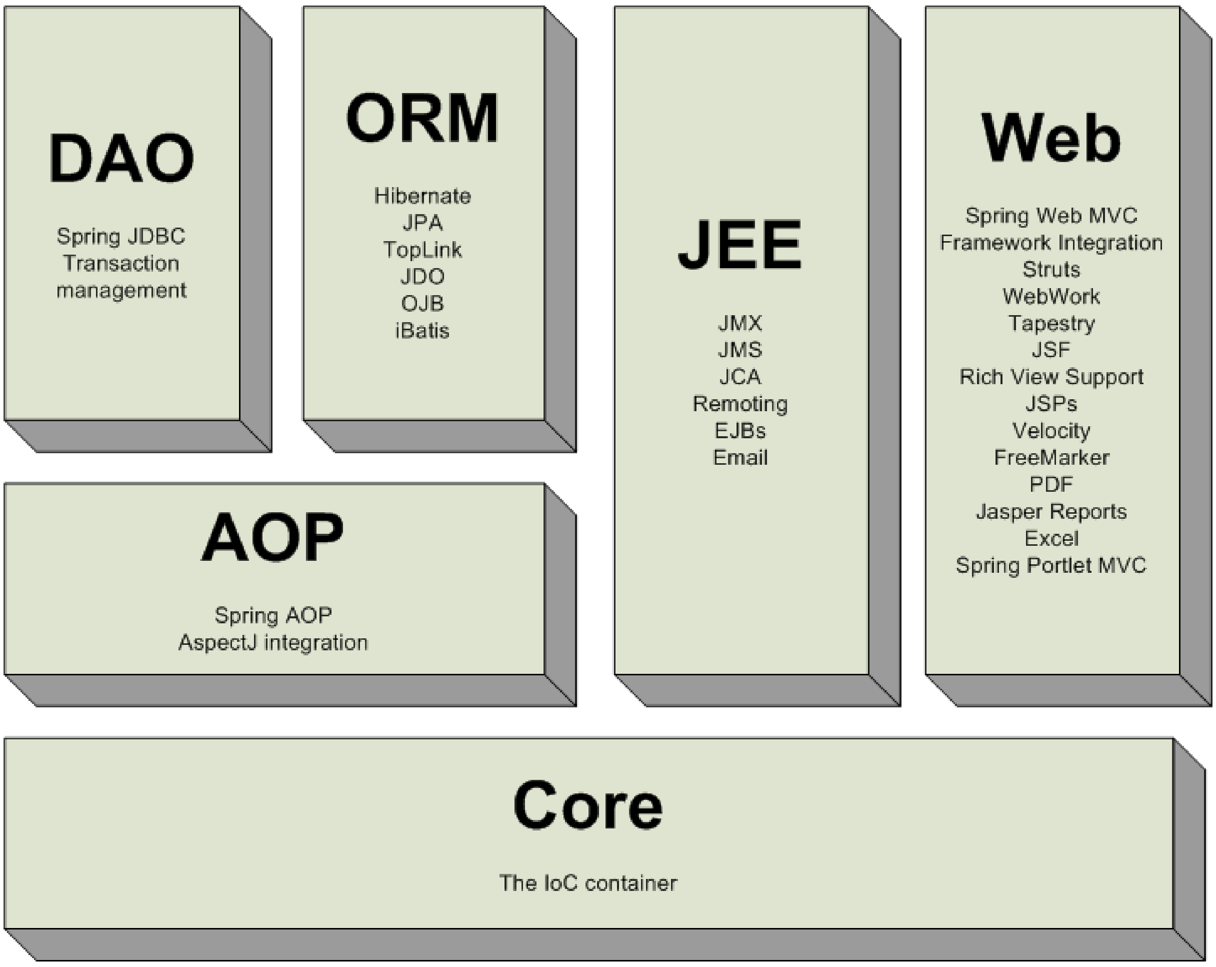


Fig. 3.1 Overview of the Spring Framework [12].

Some disadvantages of the Spring framework are that Spring distribution is over 150MB, Spring is complex, and it takes significant amount of time to learn Spring. Spring applications often use large amounts of XML. Considering Spring, means programmer will be spending a significant amount of time coding in XML, writing database configuration files. Moving application development from Java to XML has many significant drawbacks such as there is no tool available for editing XML, type safety is thrown away - typos are easy to make, but hard to find, compile time errors become a run-time errors.

In conclusion web application can be developed using different programming languages, systems, and frameworks. Each of the above explained techniques has an application spectrum, in which best programming performance and efficiency can be achieved. If the goal is to develop a large application which has to be frequently edited or updated by multiple authors, considering content management system is a logical choice. Large scale web applications running on multiple servers could be also developed using Java or PHP MVC framework. Java framework offers the best support for developing financial services, e.g. online banking web applications, where security features are critical requirements and cannot be compromised upon.

Ruby on Rails may not fit very well in an existing “enterprise” architecture with all kinds of rules and regulations. For example, certain large companies have policies such as using stored procedures to access databases and to use certain naming conventions for table and field names. Under such circumstances, Rails will begin to loose its magic since it relies on CoC and automatically reflects on the underlying table structure. Although security and scalability are downsides of Rails, the fast learning curve, the low complexity and the high developer productivity makes it very suitable for small up to middle-scaling web applications.

This report focuses only on the development of web community application with Ruby on Rails framework. It will not include any further information about the application development with other software frameworks or systems.

3.4 Project Hardware Requirements

This sub chapter discusses the hardware needed for the implementation of the project. In order to install and run the software and programs required for the project development, one needs to make sure that required hardware is up to date.

On the server side a minimum requirement to run web application is one web server. A web server is a computer that is responsible for accepting HTTP requests from clients (user agents such as web browsers), and serving them HTTP responses along with optional data contents, which usually are web pages such as HTML documents and linked objects (images, etc.). RoR application usually runs on Mongrel or Apache web server. Mongrel is an open-source HTTP library and web server for Ruby web applications. One popular configuration is to run Apache 2.2 as a load balancer using mod_proxy_balancer in conjunction with several Mongrel instances, with each Mongrel instance running on a separate port. This is something that can be configured very easily using the mongrel_cluster management utility. Apache can divide the incoming requests among the available Mongrel processes, and, with careful configuration, can even serve static content itself without having to delegate to Mongrel. This approach is called clustering and figure 3.2 shows a standard Mongrel cluster architecture.

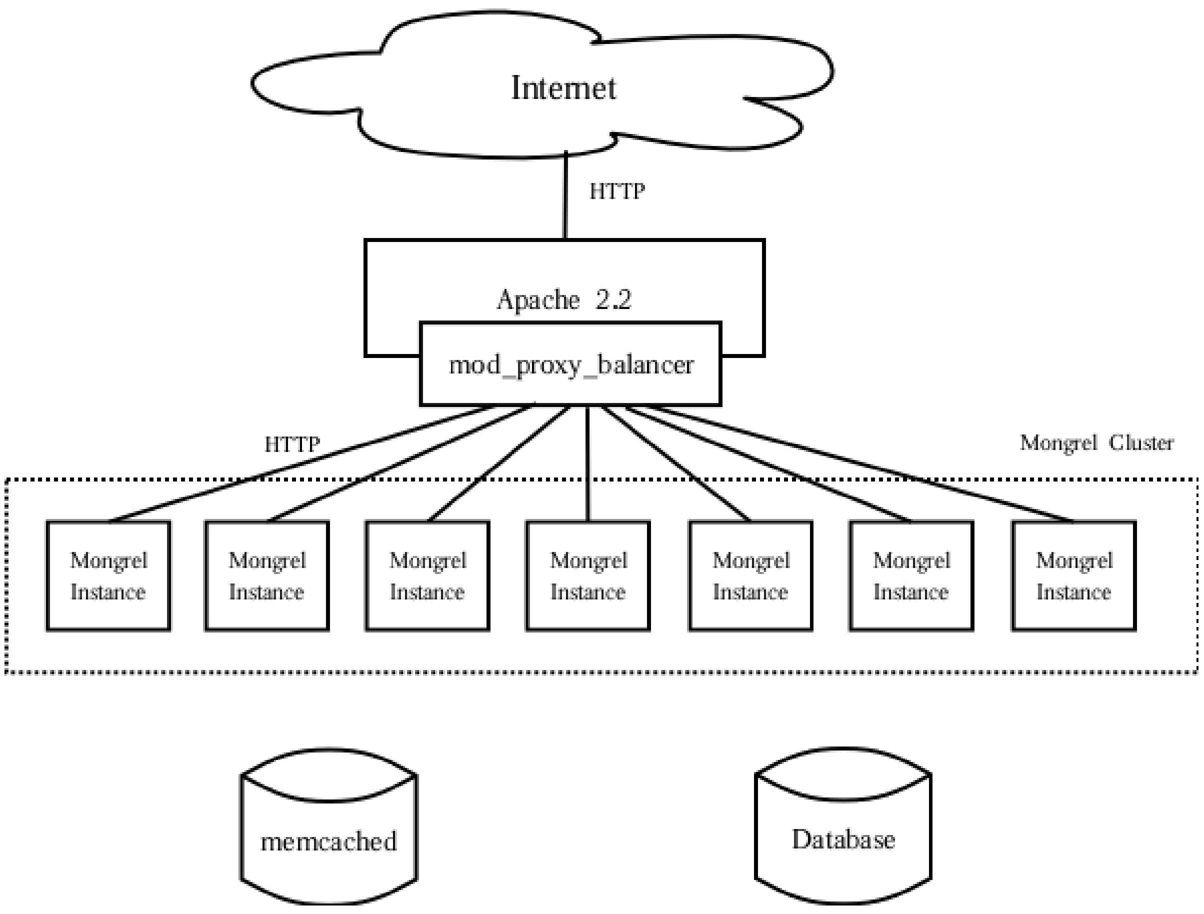


Fig. 3.2 Mongrel Cluster architecture using Apache 2.2 as load balancer [11].

Each Mongrel instance is self-sufficient and runs independent of the others duplicating all of the resources on each server. The database sits on a separate database server accessible by all other servers. Installation of a Relational Database Management System (RDBMS): MySQL is needed. MySQL is an open source RDBMS written in C and C++. It is used as a database component on many different system platforms such as Linux, Mac OS X, SunOS, Symbian, Solaris, MS Windows. MySQL is very popular for web applications. High-traffic web platforms such as Google, YouTube, Wikipedia, Flickr and Facebook use MySQL for data storage and logging user data.

On the developer side a minimum of one workstation is required. System requirements for installing and running all the developer tools are: Windows, Linux or Mac operating system, minimum of 512 MB RAM-Memory, Pentium 4-level processor or higher. An Ext3 or NTFS formatted file system partition with minimum 3 GB of free space plus adequate free space for the web site development is needed. A display at a minimum screen resolution of 1024 pixels x 768 pixels is required. For the developing, testing and deploying the web application project a router for Internet access is needed, as well as Internet connection, with 56 Kbps or faster connection between client computers and server. The local machine needs also an installed Software Development Kit (SDK) and MySQL client server.

Software and hardware requirements have an important role in the application development. Together with the functional requirements they build the foundation for every project. Requirements are considered by many experts to be the major non-management, non-business reason projects don't achieve the "magic triangle" of on-time, on-budget and high quality. Very few projects do an effective job of identifying and carrying through the project and all the requirements correctly. Many projects even fail due to requirements problems. This is the reason why developer needs to do a much better job on requirements if he wishes to develop quality software on-time and on-budget. Furthermore, requirements' errors compound during the project development. The earlier requirements problems are found, the less expensive they are to fix. Therefore, the best time to fix them is right when one is involved with gathering, understanding, and documenting them. In conclusion, it can be said that the success of the given task depends in the first place on the project planning as well as on the good analysis of the requirements and the possibilities for solving the problems.

Chapter 4 - Project Design

4.1 Web Application Three-Tier Model

The project follows a three-tier client-server software architecture model. The application is separated into three parts called “tiers”. Tiers are usually different machines. The three tiers include client tier, application tier and database tier. The client sends HTTP requests through the web browser. Web server services request by making queries and updates against the database server. Results are passed back to the web server, where user interface in HTML is generated. HTML template is returned to client browser. Figure 4.1 shows the Three-Tier architecture of the project.

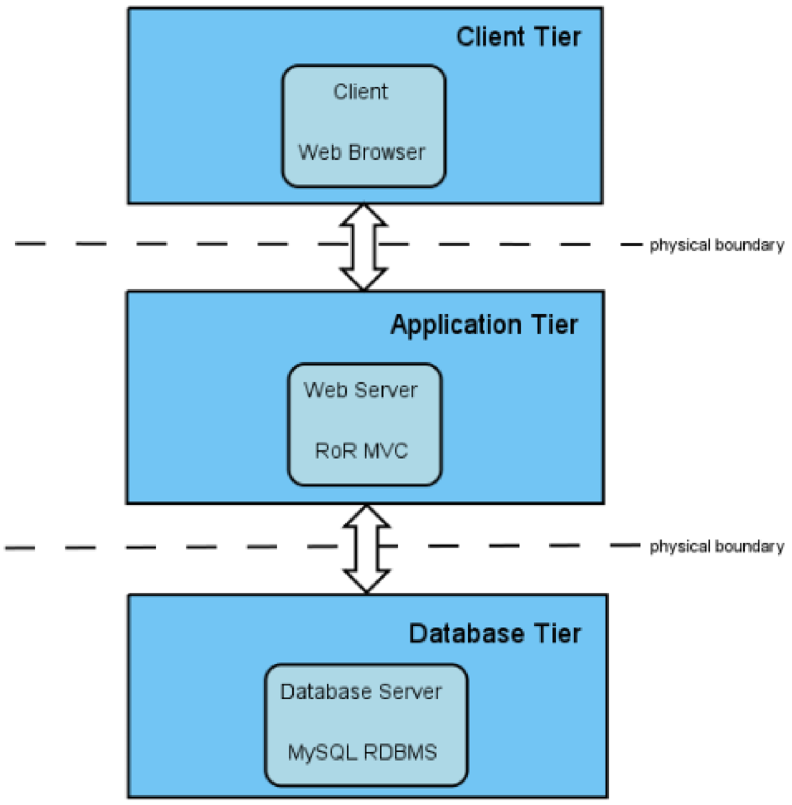


Fig. 4.1 Visual overview of a Three-tiered Web Application.

The client tier – the front-end of the application is a web browser. Each client has its own personal computer with installed operating system. A web browser runs on top of the operating system. Typical operating systems at this tier include various distributions of Linux such as Ubuntu, Red Hat and SUSE, Microsoft Windows and

Mac. Typical web browsers include Mozilla Firefox, Internet Explorer, SeaMonkey, Opera, Safari.

The Web Application Tier – the middle tier includes the application's business logic. It controls the application's functionality by performing detailed processing. In the MVC architecture of this application, Application-tier is used as the view, the controller and the model layer. Details about the design of these layers will be discussed in the next sub chapter – Web Community Application Architecture.

The Database Tier – the back-end of the application consists of MySQL RDBMS running on a database server/local server. This tier stores and retrieves the information. The database tier keeps data neutral and independent from business logic.

During the Web Community Application development, the front-end and back-end are located on the same local server.

After making it clear about the functional, software and hardware requirements of the project, and having understand the idea of the Model-View-Controller architecture, the next step is to design the web application architecture.

4.2 Web Community Application Architecture

The application is divided into application front-end, which is the user interface, from where the user interacts with the community platform, and application back-end, which is the database system of the platform.

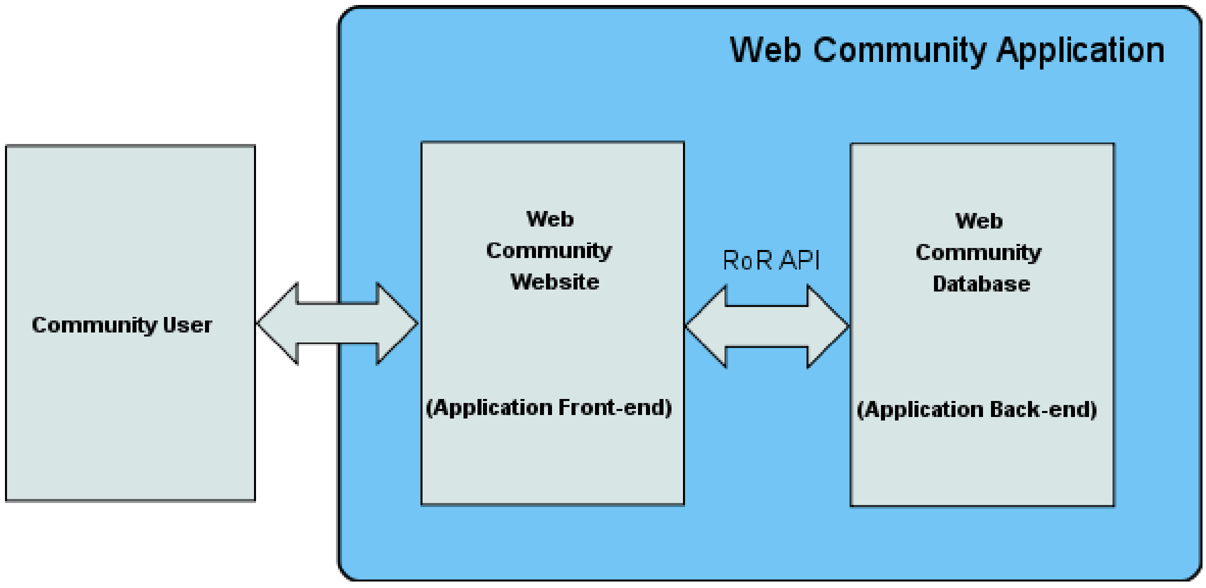


Fig. 4.2 High-level view of the Web Community application.

What does the Web Community User do?

Figure 4.3 shows the use case diagram of the potential actors and their actions. Use case describes the system's behavior as it responds to a request originated from outside of the system. In other words, a use case describes "who" can do "what" with the system in question.

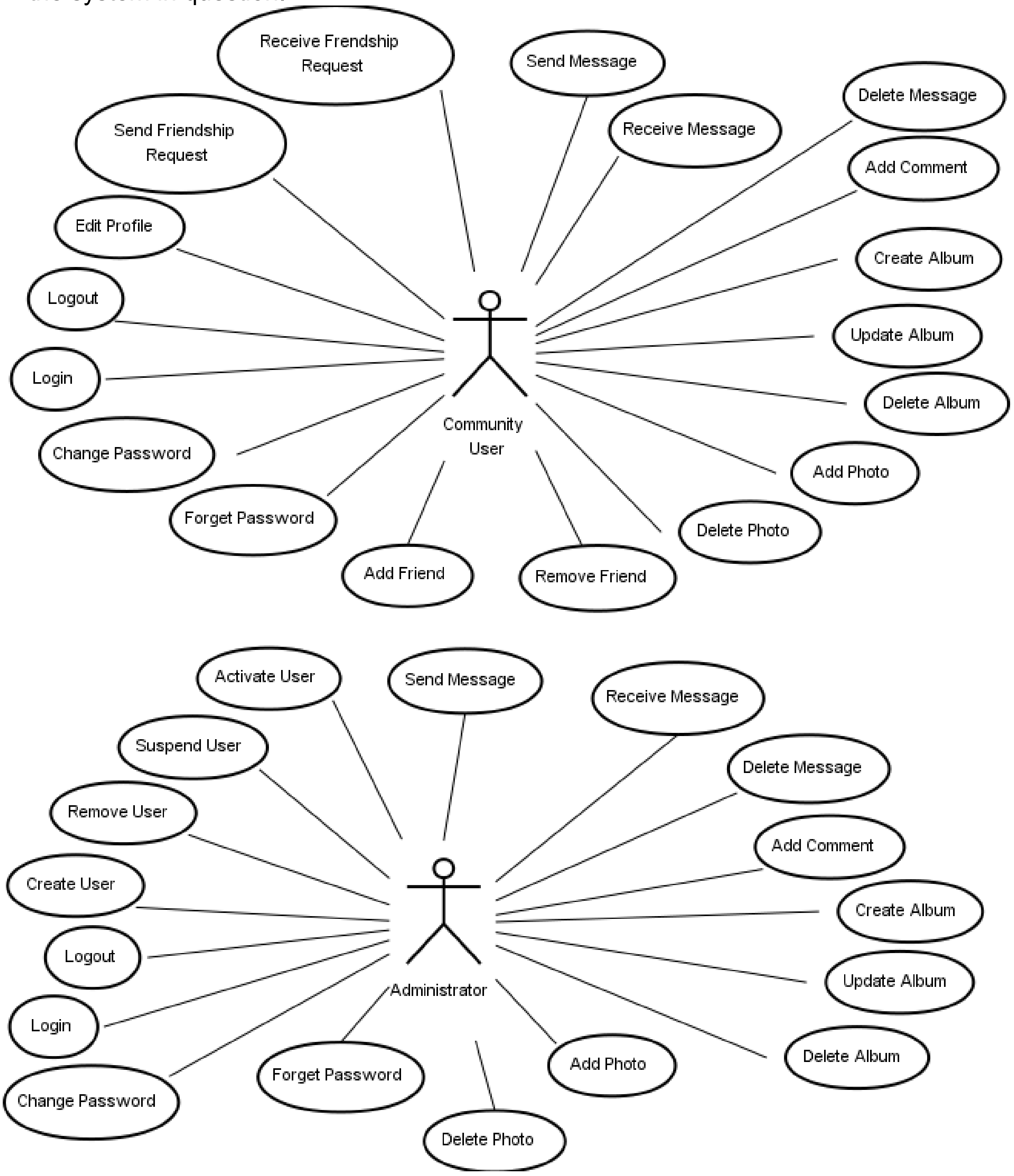


Fig. 4.3 Use case diagram of the Web Community application.

Once the functional requirements are partitioned into actions of requests and responds between users and application, the next step is to identify the business logic, presentation logic, control flow logic and model them as objects.
The following sub chapter introduces the general MVC structure design of the Web Community application and then we will go further into internal design of each layer of the MVC to decide which objects each layer needs.

4.3 Overview of the Application MVC Architecture

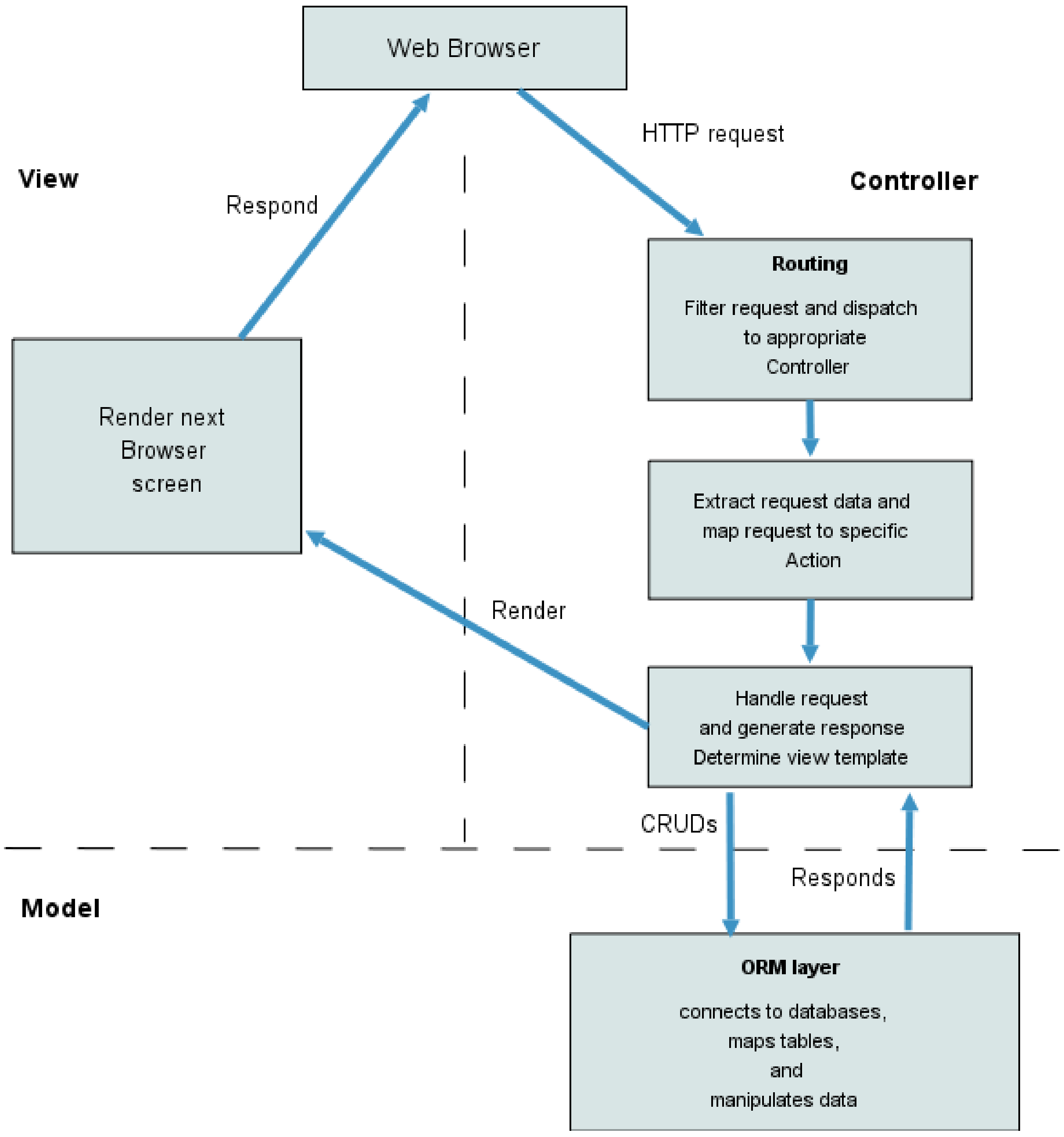

Web Browser
HTTP request
View
Respond
Controller
Routing
Filter request and dispatch to appropriate Controller
Render next Browser screen
Extract request data and map request to specific Action
Render
Handle request and generate response Determine view template
CRUDs
Responds
Model
ORM layer
connects to databases, maps tables, and manipulates data

Fig. 4.4 Model-View-Controller Architecture of the Web Community application.

A user sends a HTTP request to the controller through the web browser. The controller receives the request, handles the request with corresponding control logic for this specific request, and invokes a view to respond to the user browser. The model has different objects to store different data type and objects which have business rules to process the data. When receiving request data from a browser, data is checked for invalid input value, refined and sent to the back-end. The response data from the back-end is as well refined and sent to the web browser. The view is responsible for generating the user interface, normally based on data in the model. The view includes different HTML templates and layouts, their dynamic values are the values stored in the value objects. The object state and value can be updated.

4.4 Implementation of the Application MVC Architecture

4.4.1 The View Layer

The Ruby on Rails technology for handling user view in this web community application are rhtml templates. Rhtml templates are a mixture of HTML and embedded Ruby (ERb). They are used to generate HTML pages. In general, this application uses ERb for presentation of the dynamic contents and logic for presenting the view.

There are some considerations in the view layer

- View has only presentation logic
- Embedded Ruby code is kept to minimum
- Make use of the view template helpers
- Make use of view partial templates and layouts
- Friendly user interface for the view

It is decided that the controller layer is responsible for the control logic, and the model layer manages the business rules. So that the view layer will focus only on the presentation logic. Besides, Rails comes with a bunch of built-in helper methods and modules, which will be used to simplify templates and encapsulate the complex presentation logic.

Many pages in the Web Community application share the same tops, tails, and sidebars. Same functionality appears in many places. In order to avoid duplication layouts, shared partials and components are used.

The Web Community application makes use of two layout styles. One is used for logged-in users, it has a top with a navigation menu and search bar, a central place, where all the templates are loaded, a sidebar on the right, and a footer on the bottom. Figure 4.5 shows the logged-in layout template. A second layout for the not logged-in/ not registered users is implemented. It has a top with a log-in form and a central template place. Figure 4.6 shows the not logged-in layout template.

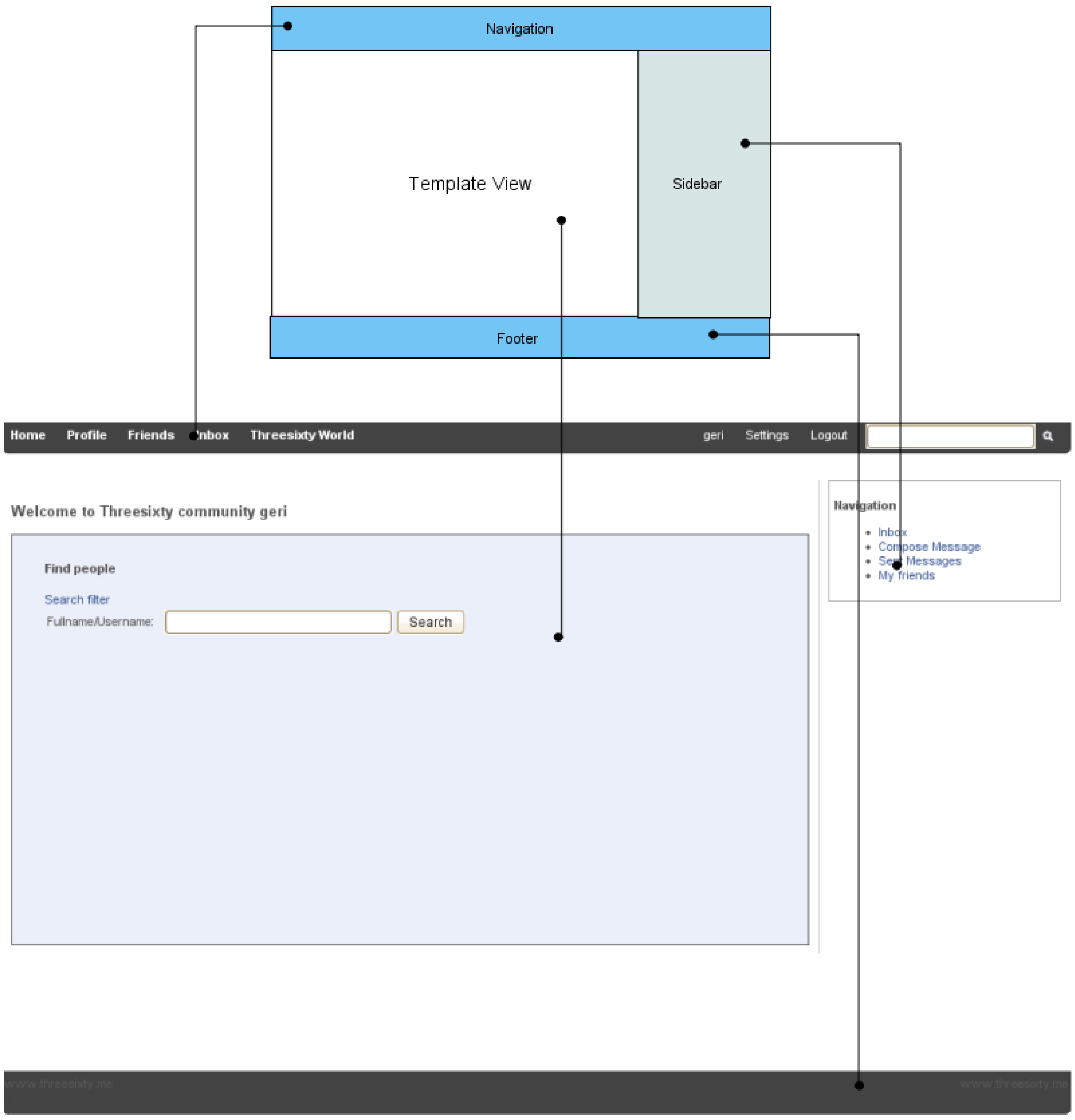


Fig. 4.5 Web Community application logged-in layout template view.

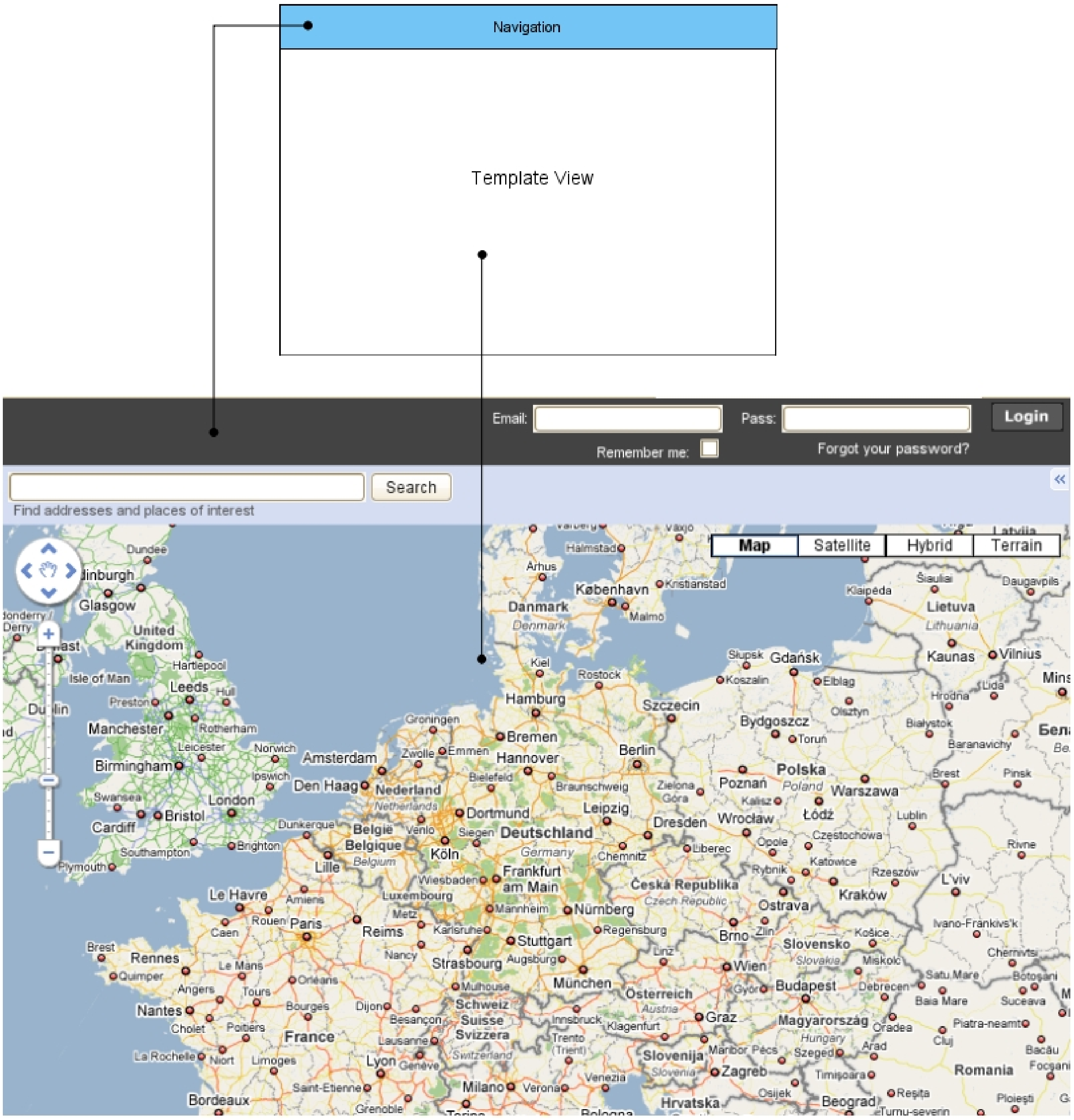


Fig. 4.6 Web Community application not logged-in layout template view.

In conclusion we can say that using this design, the View layer is totally separated from others layers. The layout can be easily modified in future development. The design encourages developer roles. If developer team becomes bigger in future, designers can work separately on the templates and views, and the developing period can be shortened.

4.4.2 The Model Layer

The model layer encapsulates the data objects and business logic of the application. It is the object-relational mapping (ORM) layer of the application. The project closely follows the standard ORM model: tables map to classes, rows to objects and columns to object attributes. Figure 4.7 shows the model layer realized as a class diagram.

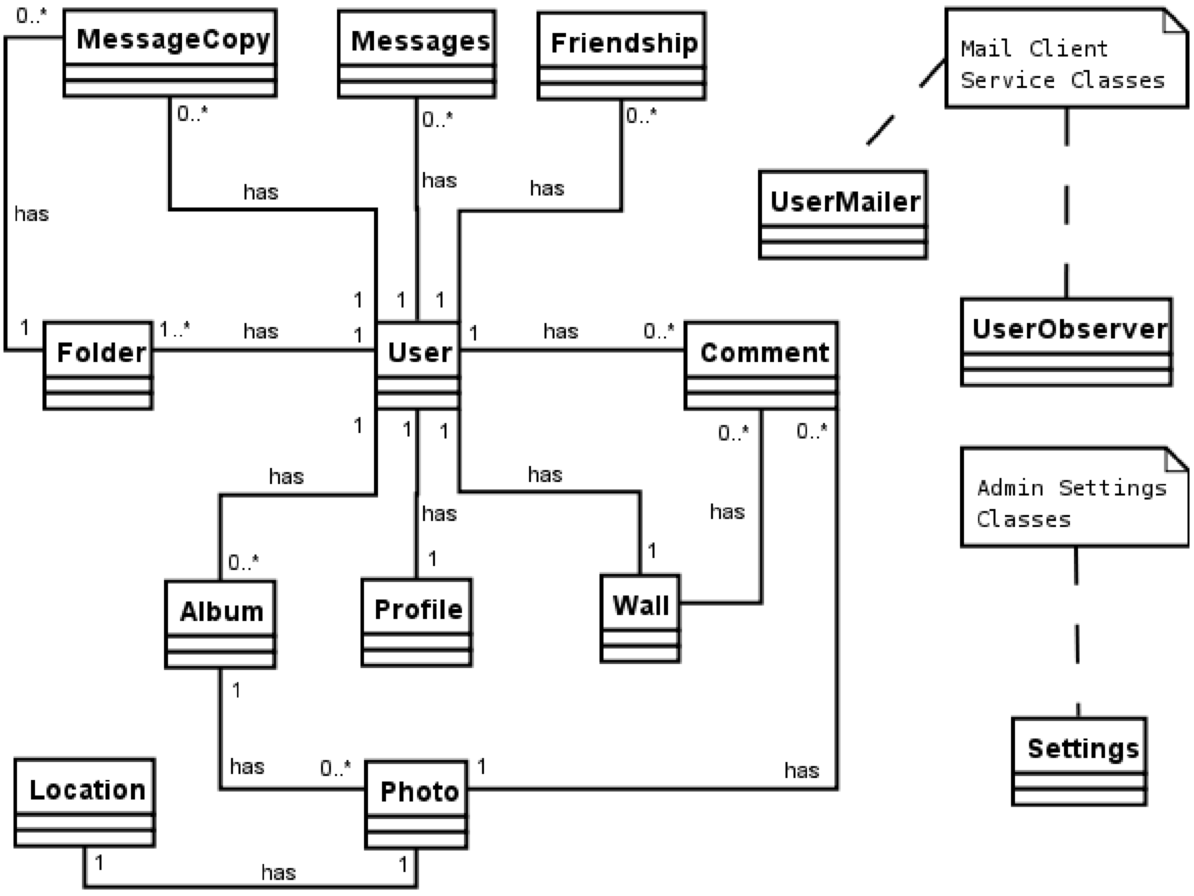


Fig. 4.7 Model layer class diagram of the Web Community Application.

<u>User class</u> is an important class of the web community project. It includes all the business logic for user account authorization and authentication. The User class includes methods for creating new users and building new accounts, searching users of the community and finding mutual friends, resetting user's password. Only signed in users are allowed to use the community platform. A User object will be created for each logged-in user. The User class has *login*, *name*, *email*, *crypted_password*, *salt*,

and *state* attributes. In the next chapter more information about process of logging in, password generation and user state will be provided.

<u>UserMailer class</u> allows the application to send email messages to user's private mail account. The class includes methods for creating notification emails after registering a community account, after activating a community account, and after requesting forgotten password or password change.

<u>UserObserver class</u> watches the user account. It has methods for delivering emails after user account is created or activated. The *after_create()* and *after_save()* methods use UserMailer class to send an email to user's personal email account.

<u>Settings class</u> include attributes for the current space available on server, as well as number of signed in users, maximum albums available per user, maximum photos available per album, maximum number of messages per user. Settings class attributes can be modified only by the administrator and are used for limiting community user's account.

<u>Comment class</u> implements polymorphic association, which allows comments to be added to multiple models. Comments are used for the community user's wall, where people can leave short messages, and for the user's photos, where people can comment on a photo. The Comment class includes attributes such as *comment*, *user_id*, *commentable_id* and *commentable_type*. In next chapter an example how the comments work will be provided.

<u>Folder class</u> provides the mailbox system for the community platform. An Folder object with *name* attribute “Inbox” is created when new community user is added.

<u>Message, MessageCopy class</u> provide the messaging system for the mailbox. Message class includes method for creating message copies to all the message recipients. After creating a Message object, MessageCopy objects are created for all the recipients.

<u>Profile class</u> includes many different attributes about the community user's profile. A Profile object is created when new user is added. Its attributes can be modified by the community user. Each user has a profile, which provides some personal information about the user to other community members.

<u>Wall class</u> represents the user profile's wall (guest book) and includes references to user comments. A Wall object is created when new user is added to the community application. User can modify his/her wall, this includes also deleting comments from his/her it.

<u>Album class</u> is used for the community user's picture albums. An Album object with *album_name* “Profile Album” is created when new user is added. Community User is allowed to create additional Album objects and to change object attributes.

Photo class is used for the user's photos. It includes methods for resizing images, as well as finding next and previous picture in an album. Photo class makes use of Rmagick image processing library for resizing photos. User is allowed to create, update and delete Photo objects.

Location class objects are used to store a photo location. User is allowed to create one Location object per Photo object. Location object is used by the application to locate an image on the map.

In conclusion we can say the design of the model layer is optimized for future expansion of the application. Separating different tasks in the model layer can help to divide the development process to different developer roles in this model layer. All the business logic of the project is encapsulated in the model. Model methods are used to refine request data, send the request to the back-end, receive the response from the database, refine it and respond to the client's view.

4.4.3 The Controller Layer

The controller layer of the MVC is used for the control logic of the application. The control logic works with both the view layer and the model layer. For the view layer, the controller is responsible for selecting the next view. For the model layer, the controller will use the business value objects and executes the business logic processes.

At its simplest, the web application accepts an incoming request from a web browser, processes it, and sends a response back. The controller acts as a single interface for all incoming HTTP requests from the clients. The information is encoded in the request URL, a subsystem called “routing” is used to determine what should be done with that request. The web application determines the name of the controller that handles this particular request, along with a list of any other request parameters. Typically one of these additional parameters identifies the action to be invoked in the target controller. Once the controller is identified, a new instance of the controller class is created, and its process method is called, passing in the request details and a response object. The controller then calls a method with the same name as the action. The action method in the controller will need to execute the business logic processing, to update the business value objects and to decide the next page to return to the browser.

The *config/routes.rb* file contains all routing information for the application. The Routing component draws a map that connects external URLs to the internals of the web community application. The *routes.rb* file is processed from top to bottom when a request comes in. The request will be dispatched to the first matching route. If there is no matching route, an HTTP status 404 – Not Found error message, is returned to the caller. In order to achieve the desired application functionality RESTful, Named and Nested Routes were used. A look at the *config/routes.rb* file will provide the reader a short explanation about the different types of routes. Figure 4.8 shows a table of the used routes in the Web Community project, along with the matching controller, action and HTTP verb.

HTTP verb	URL	Controller	Action	Used for
GET	/users	Users	index	Home page
GET	/users/:id	Users	show	Show user profile
GET	/users/:user_id/friends	Friends	index	Show user friends
PUT	/users/:user_id/friends/:id	Friends	update	Accept friendship request
POST	/users/:user_id/friends/:id	Friends	create	Create friendship request
DELETE	/users/:user_id/friends/:id	Friends	destroy	Delete friend, decline friendship request
GET	/users/:user_id/albums	Albums	index	Show user albums
GET	/users/:user_id/albums/new	Albums	new	HTML form for new album
GET	/users/:user_id/albums/:id/edit	Albums	edit	HTML form for edit album
PUT	/users/:user_id/albums/:id	Albums	update	Update user album
DELETE	/users/:user_id/albums/:id	Albums	destroy	Delete user album
GET	/users/:user_id/albums/:album_id/photos/:id	Photos	show	Show user photo
GET	/users/:user_id/albums/:album_id/photos/:id/edit	Photos	edit	HTML form for edit photo
PUT	/users/:user_id/albums/:album_id/photos/:id	Photos	update	Update user photo
DELETE	/users/:user_id/albums/:album_id/photos/:id	Photos	destroy	Delete user photo
GET	/profiles/:id/edit	Profiles	edit	Edit user profile
PUT	/profiles/:id	Profiles	update	Update user profile
GET	/mailbox & /inbox	Mailbox	index	User Inbox
POST	/mailbox/delete_messages	Mailbox	delete_messages	Delete selected messages
GET	/messages/:id	Messages	show	Show user

				message
GET	/messages/:id/reply	Messages	reply	Reply to user message
GET	/messages/:id/forward	Messages	forward	Forward user message
GET	/sent	Sent	index	Show all sent messages
GET	/sent/:id	Sent	show	Show sent message
POST	/sent/delete_messages	Sent	delete_messages	Delete selected messages
GET	/sent/new	Sent	new	HTML form for creating new message
POST	/sent	Sent	create	Creates new message
GET	/maps	Maps	index	Show map
GET	/admin/users	admin/Users	index	Admin home
GET	/admin/users/new	admin/Users	new	HTML form for adding new user
POST	/admin/users/:id/suspend	admin/Users	suspend	Suspend user
POST	/admin/users/:id/unsuspend	admin/Users	unsuspend	Activate user
DELETE	/admin/users/:id/purge	admin/Users	purge	Delete user
GET	/admin/settings/edit	admin/Settings	edit	HTML form for editing application settings
PUT	/admin/settings	admin/Settings	update	Update application settings
GET	/session/new & /login	Session	new	HTML form for login

Fig. 4.8 Routes table of the Web Community application.

After routing has determined which controller to use for a request, the controller is responsible for making sense of the request and producing the appropriate output. The controller ensures the model data is available to the view, so it can display this data to the user, and the controller is responsible for saving and updating user data in the model.

4.5 Back-end - Database Design

After introducing the MVC structure design of the Web Community application, it is time to have closer look on the database design. Developing with Rails provides a convenient way of creating, altering and managing databases and tables by using migrations. Migrations are sub classes of ActiveRecord::Migration class and can be found in the *db/migrate* directory. Creating a model class in Rails application, automatically generates a create table migration class with two class methods: up – for performing the required transformations and down – for reverting them.

For the back-end of the Web Community project a MySQL RDBMS is used. The design of database is strongly connected to the design and the business rules of the application’s model layer. The configuration of the application's database is done in *config/database.yml* file.

Data stored and retrieved from tables on the database server are as follows.

Users Table

This table holds all the information entered by the administrator user at the time of registration.

- **id** – this field is the primary key of the users table
- **login** – this field holds the community nick name of the user
- **name** – this field holds the full name of the user
- **email** – this field holds the email address of the user, used for log-in authentication
- **crypted_password** – this field holds the user crypted password, stored as 40-character string of hex digits
- **salt** – this field holds the password salt, stored as 40-character string of hex digits
- **remember_token** – this field holds the remember me value, stored as 40-character string of hex digits
- **remember_token_expires_at** – the field holds the remember me expire date and time, stored as datetime type
- **activation_code** – this field holds a 40-character string activation code, generated when new user is created
- **activated_at** – this field holds the date and time, when user activated his/her account, stored as datetime type
- **state** – this field holds the user state, stored as varchar(255) type
- **deleted_at** – this field holds the date and time, when user has been deleted, stored as datetime type
- **reset_code** – this field holds the 40-character string reset code for password change
- **created_at** – this field holds the date and time, when user has been created, stored as datetime type

- **updated_at** – this field holds the date and time, when user has been updated, stored as datetime type

Profiles Table

This table holds all the personal information about the community user.

- **id** – the field is the primary key of profiles table
- **user_id** – the field is the foreign key to users table
- **first_name** – the field holds the first name of the community user
- **last_name** – the field holds the second name of the community user
- **birthday** – the field holds the the birthday of the community user, stored as date type
- **gender** – the field holds the gender of the community user
- **hometown** – the field holds the home town/city of the community user
- **homecountry** – the field holds the home country of the community user
- **city** – the field holds the current living town/city of the community user
- **zipcode** – the field holds the zip code of user's address
- **country** – the field holds the current living country of the community user
- **region** – the field holds the current living region of the community user
- **status** – the field holds the user status
- **school** – the field holds the user's school
- **skype** – the field holds the user's skype account
- **interests** – the field holds the user's interests
- **about_me** – the field holds information about the user
- **here_for** – the fields holds the user's community intentions
- **website** – the field holds the user's website
- **created_at** – this field holds the date and time, when profile has been created, stored as datetime type
- **updated_at** – this field holds the date and time, when profile has been updated, stored as datetime type

Albums Table

This table holds the community user albums – album name and album location.

- **id** – the field is the primary key of albums table
- **user_id** – the field is the foreign key to users table
- **album_name** – the field holds the album name
- **location** – the field holds the album location
- **created_at** – the field holds the date and time, when the album is created
- **updated_at** – the field holds the date and time, when the album is updated

Photos Table

This table holds the user photos. Each photo is represented by photo image, photo thumb and photo small.

- **id** – the field is primary key of photos table
- **user_id** – the field is foreign key to users table
- **album_id** – the field is foreign key to albums table
- **parent_id** – the field holds the parent id (the id of photo image) for photo thumb and photo small
- **size** – the field holds the photo size
- **width** – the field holds the photo width
- **height** – the field holds the photo height
- **content_type** – the field holds the photo, image type
- **filename** – the field holds the file name of the photo
- **thumbnail** – the thumbnail type of the photo (NULL for photo image, thumb for photo thumb and small for photo small)
- **description** – the field holds the photo description, entered by the user
- **location** – the field holds the photo location, entered by the user
- **created_at** – the field holds the date and time, when photo has been added
- **updated_at** – the field holds the date and time, when photo has been updated

Friendships Table

This table holds the community user friendships and friendship requests.

- **id** – the field is the primary key of friendships table
- **user_id** – the field is the foreign key to users table
- **friend_id** – the field holds the friend user id
- **status** – the field holds the friendship status (requested, pending, accepted)
- **created_at** – the field holds the date and time, when friendship has been requested
- **updated_at** – the field holds the date and time, when friendship has been updated

Walls Table

This table holds the community user profile wall.

- **id** - the field is primary key of walls table
- **user_id** – the field is the foreign key to users table
- **title** - the field holds the wall's title
- **created_at** – the field holds the date and time, when wall has been created
- **updated_at** – the field holds the date and time, when wall has been updated

Comments Table

This table holds the community user comments for user wall profile and user photos.

- **id** – the field is primary key of comments table
- **title** – the field holds the comment's title

- **comment** – the field holds the comment's text
- **user_id** – the field is foreign key to users table
- **commentable_id** – the field holds the commentable id (wall id or photo id)
- **commentable_type** – the field holds the commentable type (Wall or Photo)
- **created_at** – the field holds the date and time, when comment has been created
- **updated_at** – the field holds the date and time, when comment has been updated

Folders Table

This table holds the community user messages folders.

- **id** – the field is primary key of folders table
- **user_id** – the field is foreign key to users table
- **parent_id** – the field holds the id of the parent folder
- **name** – the field holds the folder name
- **created_at** – the field holds the date and time, when folder has been created
- **updated_a**t – the field holds the date and time, when folder has been updated

Messages Table

This table holds the community user messages.

- **id** – the field is primary key of messages table
- **author_id** – the field is foreign key to users table
- **subject** – the field holds the message subject
- **body** – the field holds the message text
- **removed** – the field holds the removed flag, stored as boolean.
- **created_at** – the field holds the date and time, when message has been created
- **updated_at** – the field holds the date and time, when message has been updated

MessageCopies Table

This table holds the community user received messages.

- **id** – the field is primary key of message_copies table
- **recipient_id** – the field is foreign key to users table
- **message_id** – the field is foreign key to messages table
- **folder_id** – the field is foreign key to folders table
- **read_at** – the field holds the date and time, when message has been read
- **created_at** – the field holds the date and time, when message has been created
- **updated_at** – the field holds the date and time, when message has been updated

Locations Table

This table holds the locations for added to map photos

- **id** – the field is primary key of locations table
- **photo_id** – the field is foreign key to photos table
- **name** – the field holds the location description
- **address** – the field holds the full address location
- **city** – the field holds the location city
- **country** – the field holds the location country
- **state** – the field holds the location state
- **zip** – the field holds the location zip code
- **lat** – the field holds the location latitude, stored as decimal
- **lng** – the field holds the location longitude, stored as decimal
- **created_at** – the field holds the date and time, when the location has been created
- **updated_at** – the field holds the date and time, when the location has been updated

Settings Table

This table holds application server information and constants, user by the administrator user to restrict and manage community users.

- **id** – the field is primary key of settings table
- **space** – the field holds the absolute available application space in megabytes
- **user_space** – the field holds the available user space in megabytes
- **photo_size** – the field holds the average photo size of the application
- **max_photos** – the field holds the maximum photos available per album
- **max_albums** – the field holds the maximum album available per user
- **max_messages** – the field holds the maximum messages available per user
- **created_at** – the field holds the date and time, when setting has been created
- **updated_at** – the field holds the date and time, when setting has been updated

SchemaMigrations Table

This table holds information about the current schema migration version. The table has just one column and one row. When the database is migrated (roll back or update), the version number is read, the migration code then looks at all migration files in *db/migrate* directory, finds the current version and migrates the database to previous or newer version.

- **version** – the field is primary key of schema_migrations table

Figure 4.10 show the database design model of the web community application.

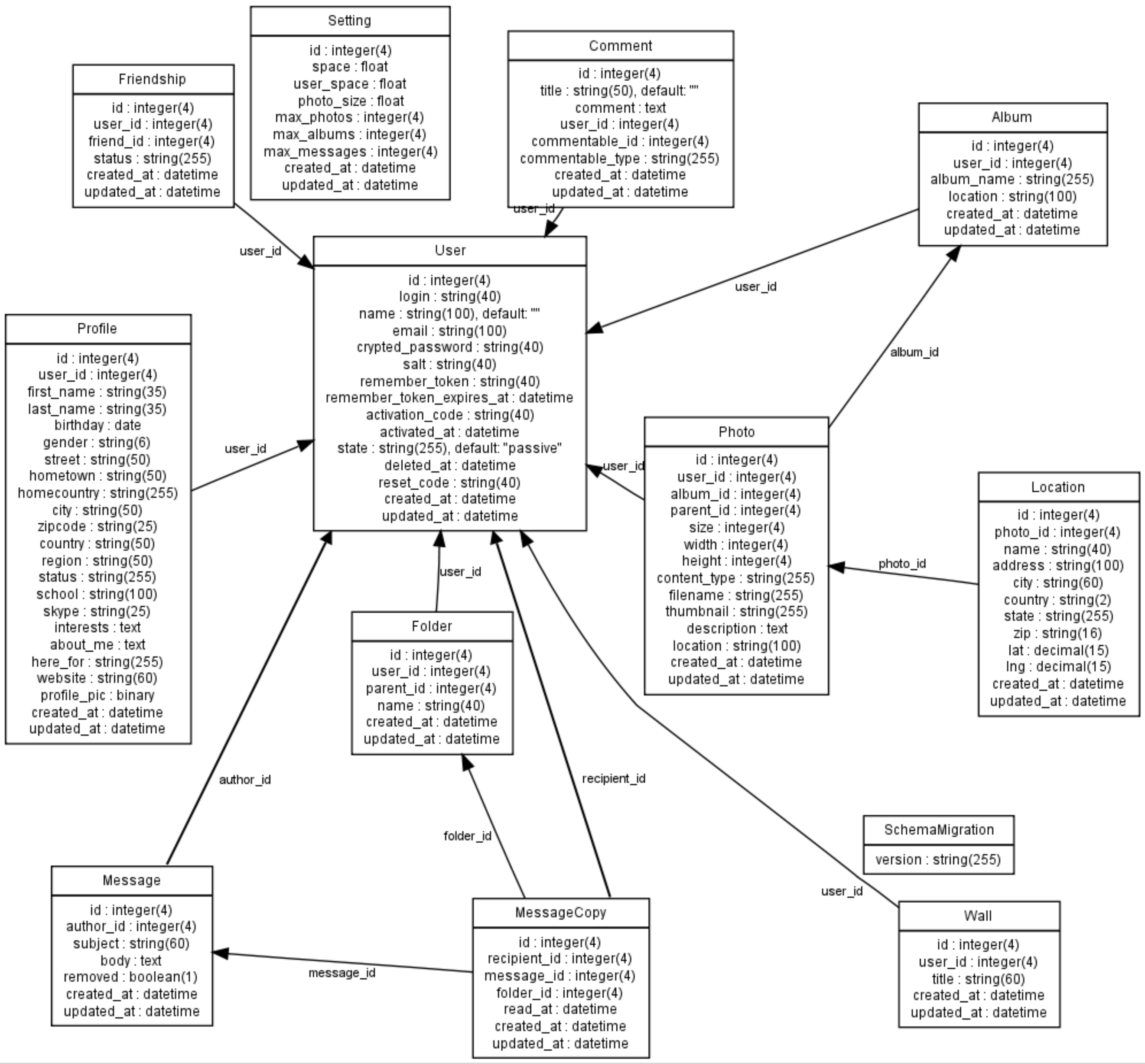


Fig. 4.9 Database design model of the Web Community application.

Finally, it should be stated, that the current database model is not a final one. During the Web Community project development, changes in database tables and table attributes could be committed.

Chapter 5 – System Realization

5.1 Realization of Log-in

The log-in/start page provides two text fields on top right corner used for user authentication. User is authenticated by email account and password. Figure 5.1 shows the start page of the Web Community application.

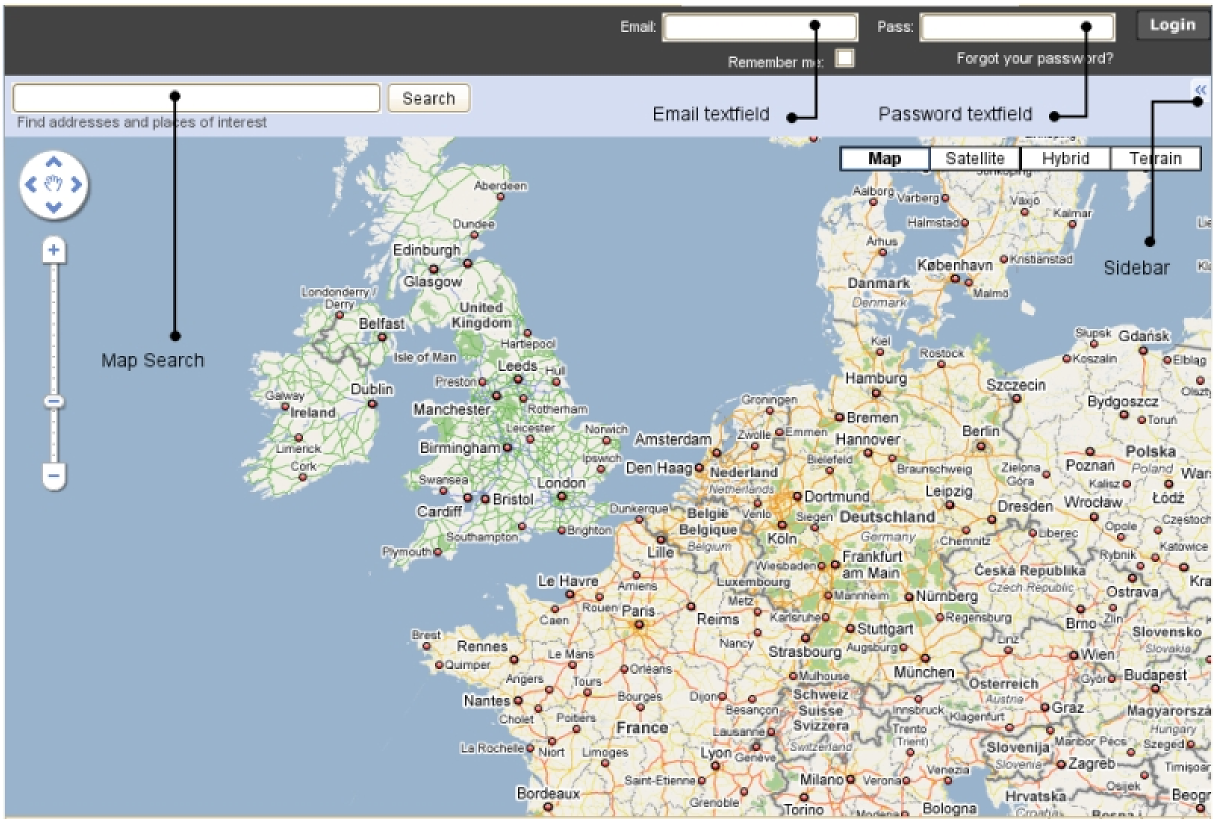


Fig. 5.1 Start page of the Web Community application.

A non registered user has the ability to use the map, find addresses and places of interest. More information about the realization and integration of Google Maps, as well as the implementation of the search engine and the map sidebar will be given further in this chapter.

The next step is to have a look at how the User authentication is implemented. The SessionsController is the controller that handles the actual log-in/logout functionality on the site. After submitting a valid email address and password, SessionsController calls the create method for creating new session. A cookie is used to store the session id on the user's web browser, but before the cookie is stored, the email and password are checked. This is done by the User model method *authenticate(email, password)*.

```
def self.authenticate(email, password)
  return nil if email.blank? || password.blank?
  u = find_in_state :first, :active, :conditions => {:email => email}
  u && u.authenticated?(password) ? u : nil
end
```

The *authenticate(email, password)* method takes the email and password parameters from the log-in form on the start page. It first checks if email or password are empty and returns a nil object if one of them is blank. Then it get the first user from the database where state is active and email matches, and saves it in an object. At the end, if object is not nil and object password matches the password parameter, user object is returned. The *authenticated?(password)* method takes the user password, encrypts it and returns true if equal to the user *crypted_password* attribute in the Users database table.

```
def authenticated?(password)
 crypted_password == encrypt(password)
end
```

How is the authentication system of the Web Community application realized?

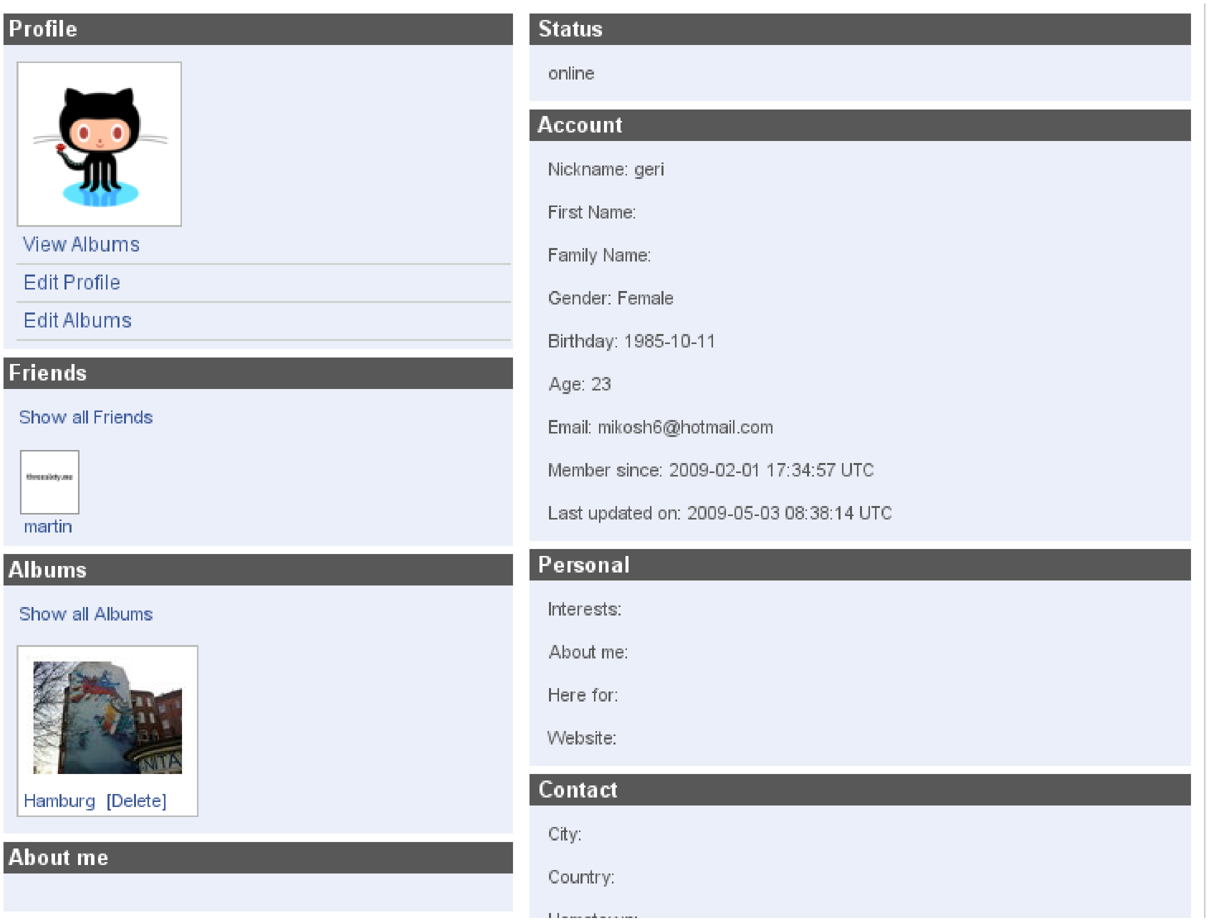
The authentication system is realized in such a way that the Users database table does not save the user password itself. The Users table has two columns *crypted_password* and *salt*, and each User object has a *crypted_password* attribute and a *salt* attribute. When new user is created, a salt is generated using the *make_token* method and user's password is encrypted using the *encrypt(password)* method. Both methods make use of the *secure_digest(*args)* method, which runs a SHA1 (Secure Hashing Algorithm) digest on the received arguments and returns a 40-character string of hexadecimal digits. In this way the saved values for the *crypted_password* and *salt* in the Users table are a 40-character hex strings.

```
def make_token
 secure_digest(Time.now, (1..10).map{ rand.to_s })
end

def encrypt(password)
  self.class.password_digest(password, salt)
end

def secure_digest(*args)
 Digest::SHA1.hexdigest(args.flatten.join('--'))
end

def password_digest(password, salt)
  digest = REST_AUTH_SITE_KEY
  REST_AUTH_DIGEST_STRETCHES.times do
   digest = secure_digest(digest, salt, password, REST_AUTH_SITE_KEY)
  end
  digest
end
```

The *password_digest(password, salt)* method makes the crypted password more secured by using the two site keys REST_AUTH_SITE_KEY and REST_AUTH_DIGEST_STRETCHES, which are randomly generated numbers. The site keys gives additional protection against a dictionary attack. Without site key, if database were to be compromised the users' passwords will be vulnerable to a hacker attack. With a site key, an attacker needs access to both site's code and site's database in order to mount an offline dictionary attack.

5.2 Realization of User Profile

After log-in, user is able to see and modify his/her profile. The profile page include information about the user, user's friends and user's albums. The profile page serves also as a presentational page, it presents the user to the community. Figure 5.2 shows the profile page of the Web Community application.


Profile

View Albums
Edit Profile
Edit Albums

Friends

Show all Friends

martin

Albums

Show all Albums

Hamburg [Delete]

About me

Status

online

Account

Nickname: geri

First Name:

Family Name:

Gender: Female

Birthday: 1985-10-11

Age: 23

Email: mikosh6@hotmail.com

Member since: 2009-02-01 17:34:57 UTC

Last updated on: 2009-05-03 08:38:14 UTC

Personal

Interests:

About me:

Here for:

Website:

Contact

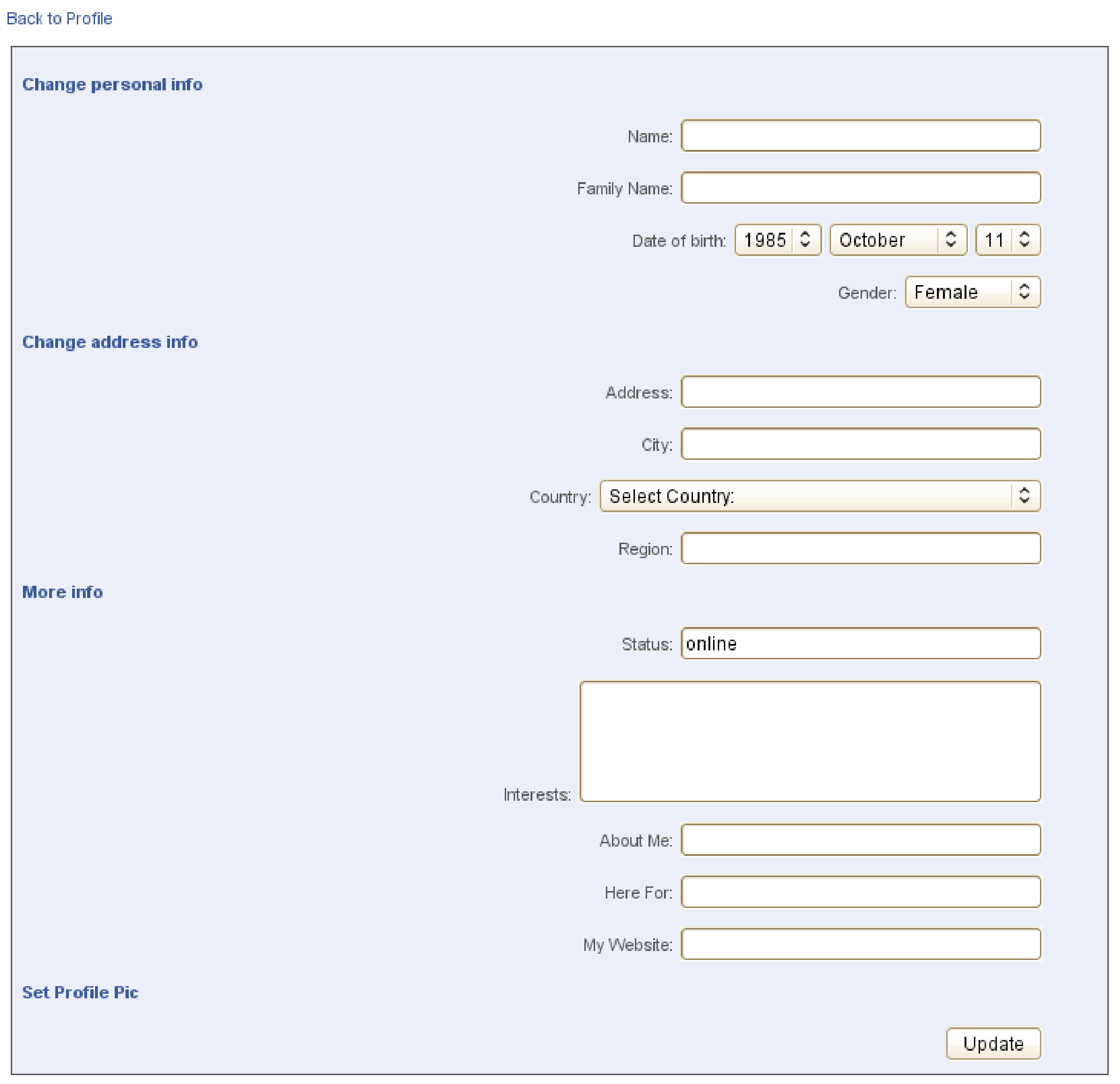
City:

Country:

Fig. 5.2 Profile page of the Web Community application.

The profile page is separated into several sections. The Account, Personal and Contact sections are managed by the ProfilesController, user's profile attributes are saved in the Profiles database table. The Profile section contains the user profile image, links to view and edit user albums, and link to edit user's profile. The Friends and Albums sections contain links to user's friends, respectively user's albums.

Editing the profile is done by clicking on the “Edit Profile” link. The edit profile page provides a form for editing all the user information. Edit profile form is divided into four sections – “Change personal info”, “Change address info”, “More info” and “Set Profile Pic”. In “Change personal info” section user can edit information about his/her date of birth, gender, first and family name. The “Change address info” section includes information about user's address, city, country and region. The “More info” section contains information about user's interests, hobbies, user's website and others. The last link navigates user to his/her Profile album, where user can change his/her profile image. Clicking the “Update” button, updates the user's profile information. Figure 5.3 show the edit profile page of the Web Community application.


Back to Profile

Change personal info

Name:	
Family Name:	
Date of birth:	1985 October 11
Gender:	Female

Change address info

Address:	
City:	
Country:	Select Country:
Region:	

More info

Status:	online
Interests:	
About Me:	
Here For:	
My Website:	

Set Profile Pic

Fig. 5.3 Edit profile page of the Web Community application.

Uploading images to user profile

Community user can create new albums and upload images to albums. The Web Community application uses “attachment_fu” plug-in by Rick Olson. Attachment_fu facilitates the image file uploads in the application. For image manipulation and resizing, as well as generation of thumbnails, the Rmagick image processor is used. RMagick is an interface, using the Ruby programming language and the ImageMagick® and GraphicsMagick image processing libraries. More information is available on [13].

The upload photo form is part of the edit album page. The photo upload form consist of text field and browse button, where user can browse images on his/her personal computer or other external storage devices. The “Update” button uploads the selected from user image. Figure 5.4 shows the edit album page of the Web Community application.

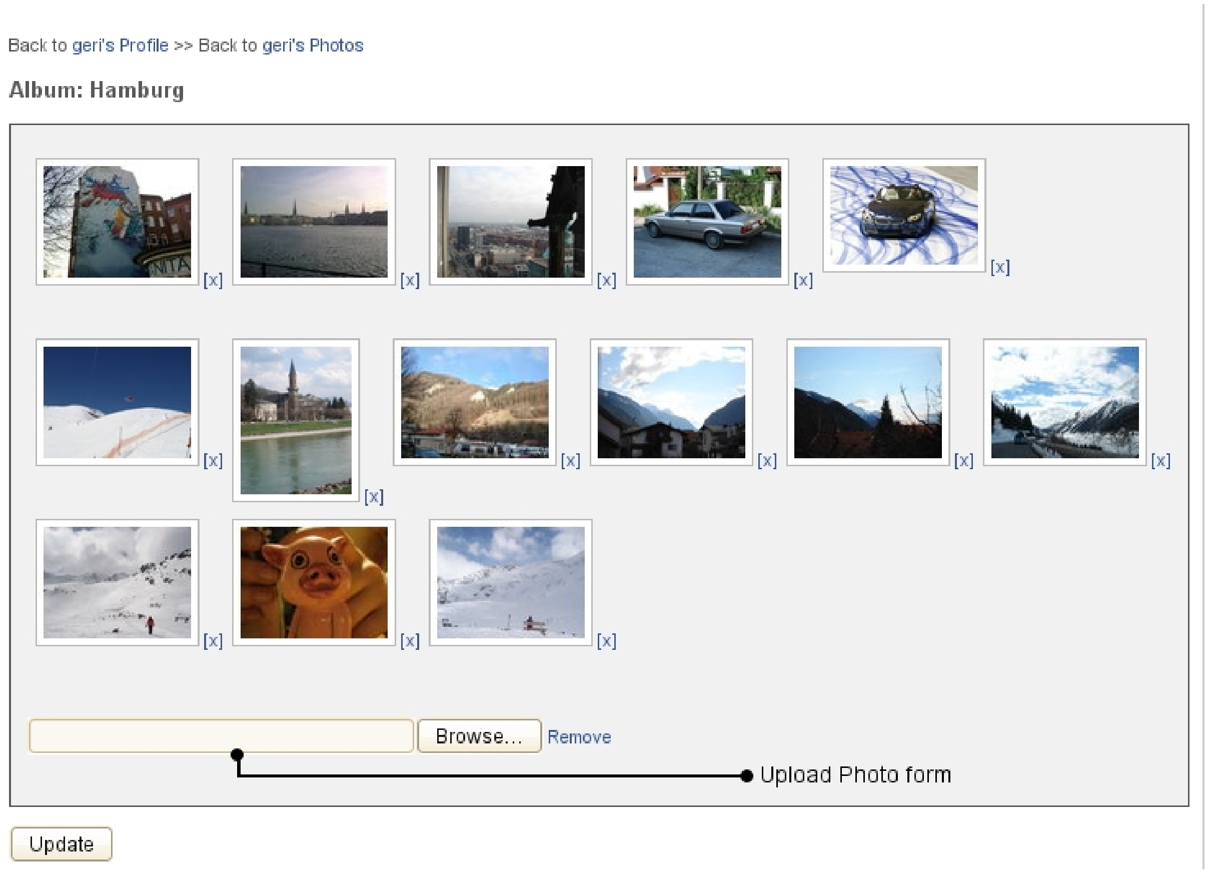


Fig. 5.4 Edit album page of the Web Community application.

The Photo model of the application make use of two methods available from the “attachment_fu” plug-in. The following code snippet shows part of model class Photo, available in *app/models* directory.

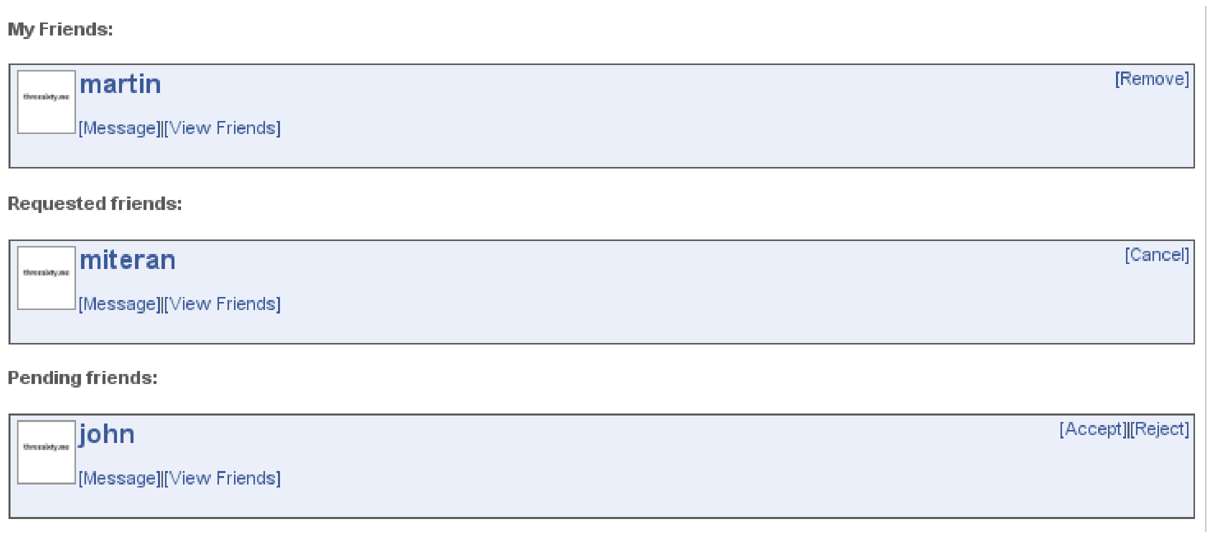
```
has_attachment :content_type => :image,
 :size => 0.megabyte..3.2.megabytes,
 :storage => :file_system,
 :processor => 'Rmagick',
 :resize_to => '450x450',
 :thumbnails => {:thumb => 'thumb: 90x90>', :small => 'crop: 30x30>'}

validates_as_attachment
```

The used parameters for the *has_attachment* method are *content_type*, *size*, *storage*, *processor*, *resize_to* and *thumbnails*. The *content_type* specifies the type of uploading file, *image* option allows all standard types of images to be uploaded. The *size* defines the maximum and minimum size of the file. The *storage* specifies the storage system, where the files are stored. Using the *file_system* parameter saves the resized images in a directory on the server. The *processor* parameter defines the used image library for image processing. The Web Community application uses the Rmagick image library. The *resize_to* parameter specifies the size to which the images will be resized. Application uses a “450x450” pixel, the image will be resized to 450 pixel on the longer side. In other words if the image width is bigger than the image height, image width will be resized to 450 pixel, and the image height will be resized proportionally. The *thumbnails* parameter of the *has_attachment* method takes two sub-parameters - *thumb* and *small*. When an image is uploaded, it will be resized, two additional images will be generated – thumb with 90x90 pixel and small with 30x30 pixel. Looking in the database table Photos on each image upload three new records are been added. The first is the actual image resized with filename - *image_filename* and id - *image_id*. The second is the small image with filename - *image_filename_small* and parent_id - *image_id.* The third is the thumb image with filename - *image_filename_thumb* and parent_id - *image_id.*

The *validates_as_attachment* method prevents files outside of the valid range (size range) from being saved.

5.3 Realization of Social Network

Social network functionality is one of the core features of the application. The Web Community user can add other users – friends to his or her profile. The friends index page allows the user to see his or her current friends, requested friends, and pending friends. Figure 5.5 shows the friends index page of the Web Community application.


My Friends:

martin [Remove]
[Message][View Friends]

Requested friends:

miteran [Cancel]
[Message][View Friends]

Pending friends:

john [Accept][Reject]
[Message][View Friends]

Fig. 5.5 Friends index page of the Web Community application.

By clicking the friend's name or profile picture, the user is linked to his or her friend (current, requested or pending) profile. User has the option to send messages to friends and view friends of his or her friend. User can also remove current friend from his or her network, cancel a friendship request, accept or reject a pending friend.

How does the social network work?

All the friendship functionality is done by the FriendsController class, which can be found in *app/controllers* directory. A typical scenario is when the community user log-ins and starts playing around with the application. First the user finds a known friend using the searching machine and sends a friendship request. How the searching machine works is explained later in this chapter. After the friendship request is sent, the requested user receives a message in his or her mailbox. This first step is done by the *create* action in FriendsController class.

```
  def create
   @message = current_user.sent_messages.build(:to => params[:friend_id],
   :subject => "Friendship Requested", :body => params[:friendship][:body])
   @user = User.find(current_user)
   @friend = User.find(params[:friend_id])
   params[:friendship1] = {:user_id => @user.id, :friend_id => @friend..id,    :status =>
'requested'}
   params[:friendship2] = {:user_id => @friend.id, :friend_id => @user.id,     :status =>
'pending'}
   @friendship1 = Friendship.create(params[:friendship1])
   @friendship2 = Friendship.create(params[:friendship2])
         ...
 end
```

A Message object is created, the user and the friend are found and saved in User objects, and two Friendship objects are created – one for the user with *status* attribute “requested” and one for the friend with *status* attribute “pending”. Every new relationship add two new records (rows) in the Friendships database table with *user_id*, *friend_id* and *status*. As one can see in the *friendship1* and *friendship2* parameters have the *user_id* and *friend_id* values substituted.

The next step is when the possible new friend sees the friendship invitation, he or she has two possibilities - either to accept or to reject the friendship request. In case of rejecting it the *destroy* action in FriendsController class is called.

```
  def destroy
    @user = User.find(params[:user_id])
    @friend = User.find(params[:id])
    @friendship1 =
@user.friendships.find_by_user_id_and_friend_id(params[:user_id],params[:id]).destroy
    @friendship2 =
@friend.friendships.find_by_user_id_and_friend_id(params[:id],params[:user_id]).destroy
    redirect_to user_friends_path
  end
```

The users are first found in the Users database table and saved in instance variables *@user* and *@friend.* Next, the two friendships are found and deleted. In case of accepting the friendship request, the *update()* method in FriendsController class is called.

```
  def update
    @user = User.find(current_user)
    @friend = User.find(params[:id])
    params[:friendship1]={:user_id => @user.id, :friend_id => @friend.id, :status =>
'accepted'}
    params[:friendship2]={:user_id => @friend.id, :friend_id => @user.id, :status =>
'accepted'}
    @friendship1 = Friendship.find_by_user_id_and_friend_id(@user.id, @friend.id)
    @friendship2 = Friendship.find_by_user_id_and_friend_id(@friend.id, @user.id)
    if @friendship1.update_attributes(params[:friendship1]) &&
      @friendship2.update_attributes(params[:friendship2])
      flash[:notice] = 'User sucessfully accepted!'
      redirect_to user_friends_path(@user)
    else
      redirect_to user_path(current_user)
    end
  end
```

It finds the user and friend in the Users database table, creates the new friendship parameters with status 'accepted', finds the two friendships in the Friendships table and updates them with the new parameters.

5.4 Realization of Messaging System

Web community user can compose, send and receive messages from other users. The mailbox system makes use of the Folders, Messages and MessageCopies database tables. Three controllers are used to handle the messaging system functionality.

Community user can compose and send messages to other users. The sent new page is showed on figure 5.6. The compose HTML form has text fields for recipient, message subject and message body. After pressing the “Send” button, the *create()* method in SentController is called.

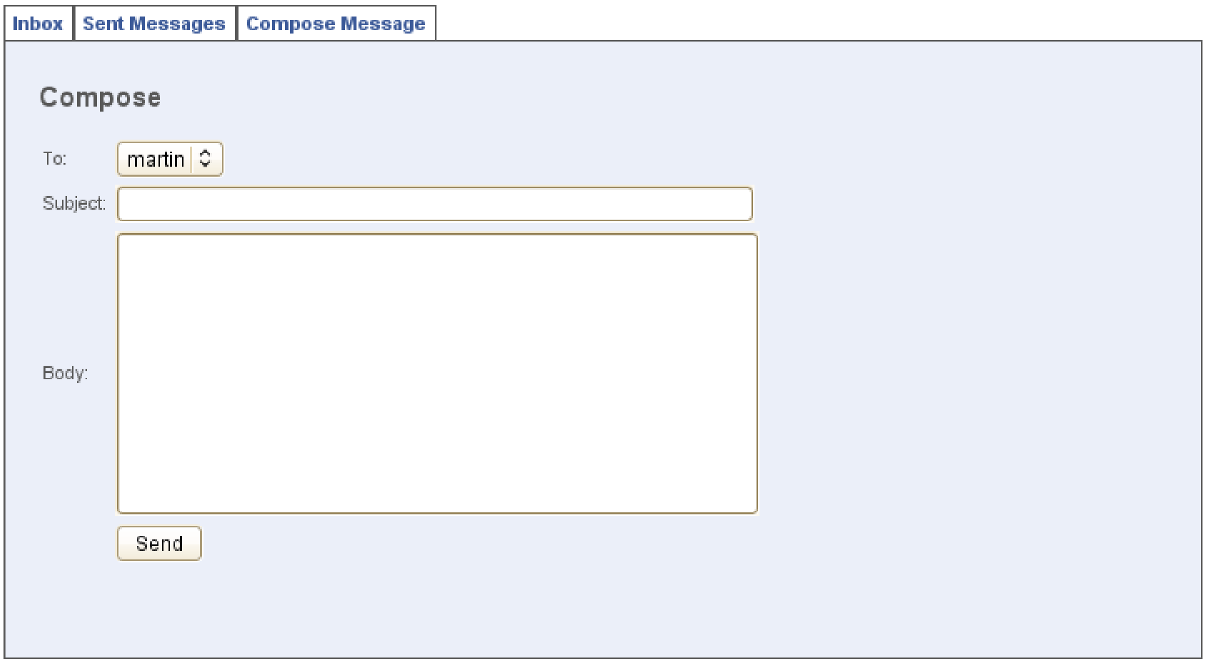


Fig. 5.6 Sent new page of the Web Community application.

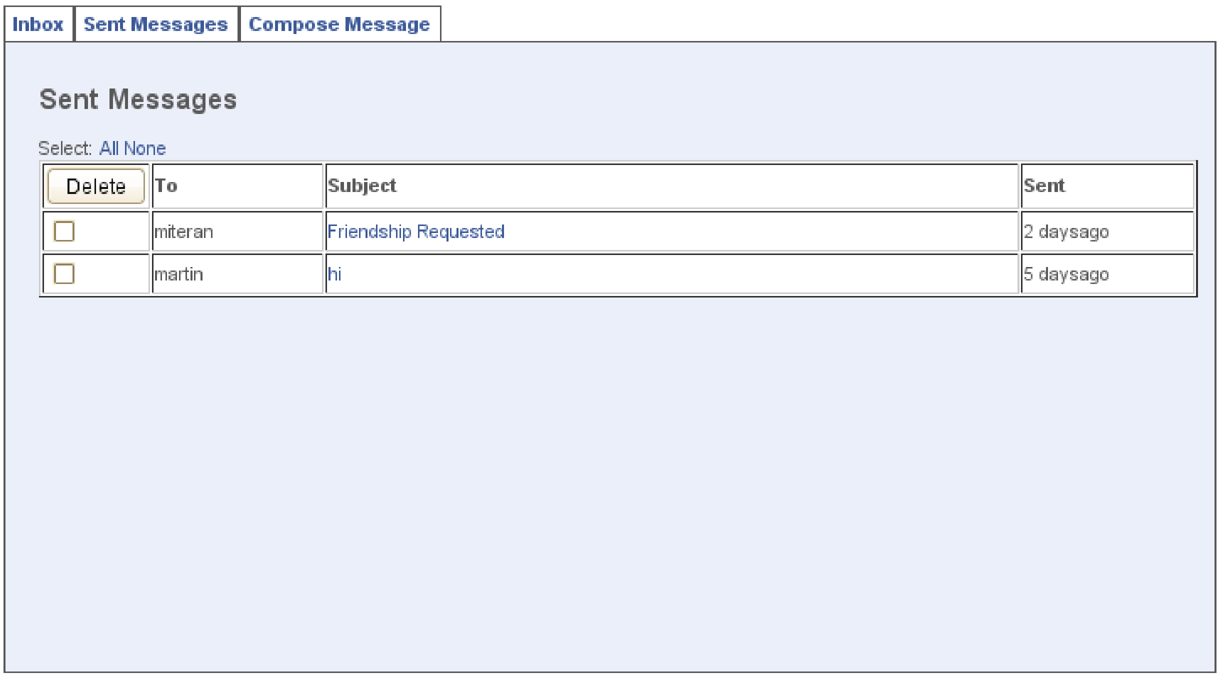
Before new message is created and stored in the Messages database table, the Message model, in *app/models* directory, validates the message by using the validation helper methods. The following code snippet shows the validation of presence and length of the :subject and :body Message attributes.

```
validates_presence_of    :subject, :message => "Subject is missing!"
validates_length_of     :subject,   :within => 1..60, :message => "Subject should be 1 to 60 characters long!"
validates_presence_of    :body, :message => "Message body is missing!"
validates_length_of     :body,    :within => 1..450, :message => "Body should be 1 to 450 characters long!"
```

After the validation the *prepare_copies()* method in Message model is called. The method is responsible for building copies for each recipient and link them to recipients Inbox folder id.

```
def  prepare_copies
  return if to.blank?
  to.each do |recipient|
    recipient = User.find(recipient)
    message_copies.build(:recipient_id => recipient.id, :folder_id =>  recipient.inbox.id,
         :created_at => Time.now)
  end
end
```

After message copies are built, message is created and saved in Messages table. Web Community member can see and read his or her sent messages using the sent page. Sent page shows a list of all the sent messages by the user. Figure 5.7 shows the sent page of the Web Community application.



Sent Messages

Select: All None

Delete	To	Subject	Sent
☐	miteran	Friendship Requested	2 daysago
☐	martin	hi	5 daysago

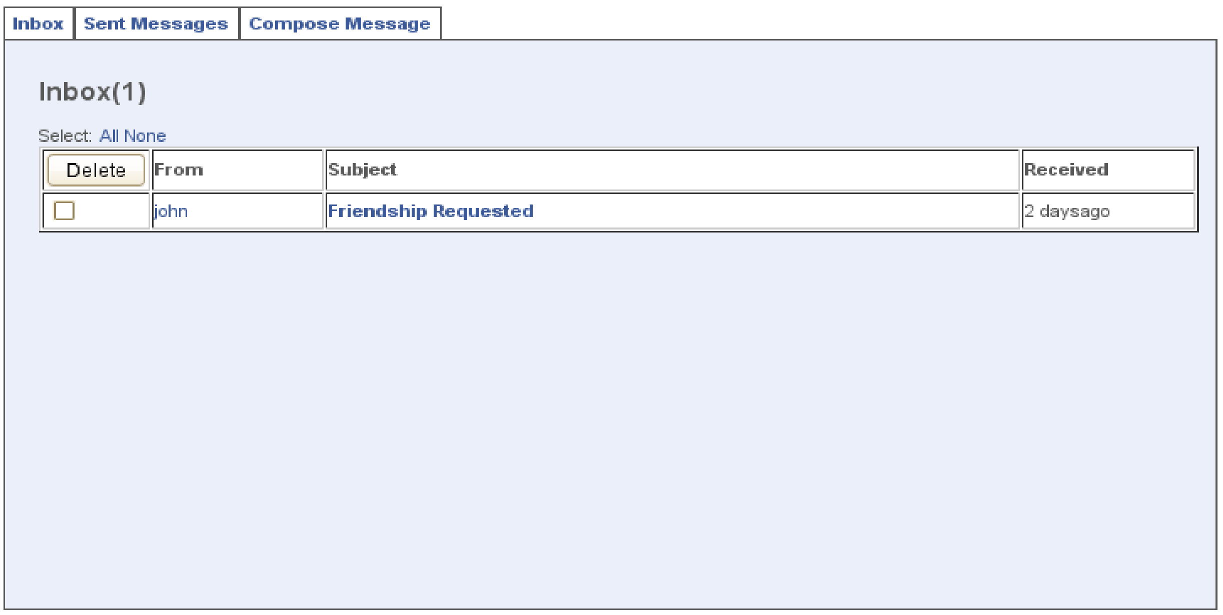
Fig. 5.7 Sent page of the Web Community application.

User has the option to select or unselect all sent messages. The user can delete sent messages by selecting the desired message and pressing the “Delete” button. This will call the *delete_messages* method in SentController.

```
def delete_messages
  if params[:message_ids]
      @messages = current_user.sent_messages.find(params[:message_ids])
      for message in @messages
        message_copy = MessageCopy.find_by_message_id(message.id, :first)
        if message_copy !=  nil
          message.removed = 1
          message.save
        else
          message.destroy
        end
      end
    end
    redirect_to :action => "index"
  end
```

The *message_ids* parameter is an array containing all the selected message ids. All the selected messages are found in the Messages table and stored in *@messages* instance variable. For each message in *@messages*, a message copy is searched, if found the message *removed* attribute is set to “1” and message is saved, else the message is deleted. The reason of not deleting the message directly is that all the received messages the user has, are instances of MessageCopies. Since the MessageCopies table does not have the message *body* and *subject* attributes, it stores only the *message_id*, *recipient_id* and *folder_id*, the original message record in Messages database table is needed. The *removed* message attribute is set “1”, if the message was deleted by the author, but it is still present in the recipient's Inbox.

User can check his or her received messages using the Inbox page. The MailboxController is responsible for showing and deleting received messages – objects of MessageCopies database table, from the Inbox folder. Figure 5.8 shows the Inbox page of the web community application.



Delete	From	Subject	Received
☐	john	Friendship Requested	2 daysago

Fig. 5.8 Inbox page of the Web Community application.

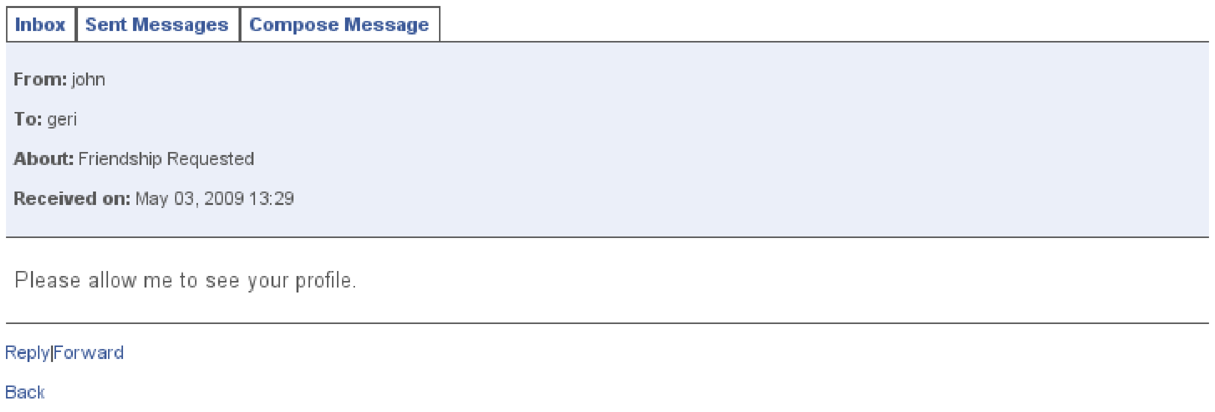
User can also select and unselect all messages. The “Delete” button calls the *delete_messages* method from the MailboxController.

```
def delete_messages
  @folder ||= current_user.folders.find(params[:id])
  if params[:message_ids]
    @messages = @folder.messages.find(params[:message_ids])
    for received_message in @messages
      sent_message = Message.find(received_message.message_id)
      received_message.destroy
      if (sent_message.removed == true)
        sent_message.destroy unless
        @folder.messages.find_by_message_id(sent_message.id)
          end
        end
      end
      redirect_to inbox_path
```

```
        end
```

The *delete_messages* method here first finds and saves the user's Inbox folder into a *@folder* instance variable. The Inbox folder contains all the received messages. This time *@messages* represents an array of MessageCopies objects. For each message in *@messages* the original message is found first, then the received message(the MessageCopy) is deleted. If the sent message *removed* flag is true, in other words – the author has deleted this message, and the are no other recipients, who have this message saved, the sent message is deleted from Messages database table.

User can read each message, reply to it or forward it. By clicking the message subject in the Inbox page the *show* action in MessagesController is called. The MessagesController handles the *show*, *reply* and *forward* actions. Figure 5.9 shows the show message page of the Web Community application.



From: john

To: geri

About: Friendship Requested

Received on: May 03, 2009 13:29

Please allow me to see your profile.

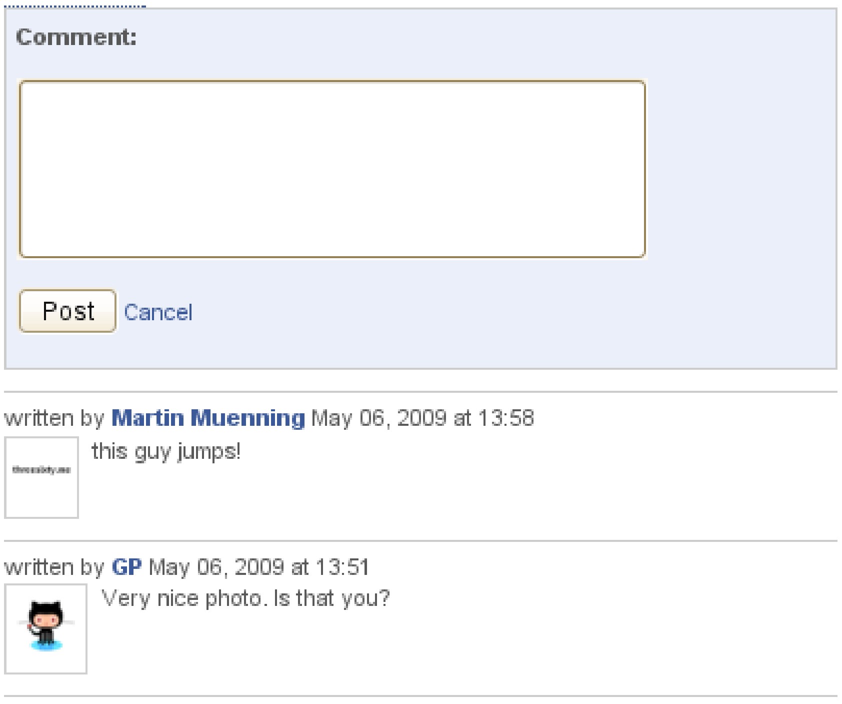
Fig. 5.9 Show message page of Web Community application.

User can reply to or forward the message. The “Reply” link will call the *reply* action in MessagesController, which will generate a new compose message form with a *recipient* - “John”, *subject* - “Re: Friendship Request” and *body* - “> Please allow me to see your profile.”. The “Forward” link will call the *forward* action in MessagesController, which will generate new compose message form with *subject* - “Fwd: Friendship Requested” and *body* - “> Please allow me to see your profile.”.

5.5 Realization of Comments System

A community user can comment on his or her wall and on his or her friends' walls. User can also comment on his or her photos and on his or her friends' photos. Since both walls and photos have comments and the application uses only one Comments table, the Comment model implements a polymorphic relationship. In other words a single model can be associated to an arbitrary number of other model types. If the user adds a wall comment, a new record with be created in the Comments database table with the *comment* text, *user_id* – the current user, *commentable_id* – the wall id and *commentable_type* – Wall. On the other hand if the user comments on a photo, a new record will be created in the Comments database table with *comment* text, *user_id* – current user, *commentable_id* – the id of the photo and *commentable_type* – Photo. Using the polymorphic association keyword: commentable, there is no need to create additional table columns in Comments table such as *wall_id* or *photo_id*. The application makes use of the two *commentable_id* and *commentable_type* columns to distinguish between different types of comments. In this way adding more models, which have many comments will not be a problem.

The comment system functionality is realized by the CommentsController in *app/controllers* directory. Community user can add new comment to walls or photos, and he or she can delete comments from his or her wall or photos. Figure 5.10 shows the comments page of the Web Community application, which includes the add new comment form.



written by **Martin Muenning** May 06, 2009 at 13:58

this guy jumps!

written by **GP** May 06, 2009 at 13:51

Very nice photo. Is that you?

Fig. 5.10 Comments partial page of the Web Community application.

By pressing the “Post” button, POST request is generated, the *create* action in CommentsController is called, comment text is first validated, then comment object is created and saved in the Comments database table.

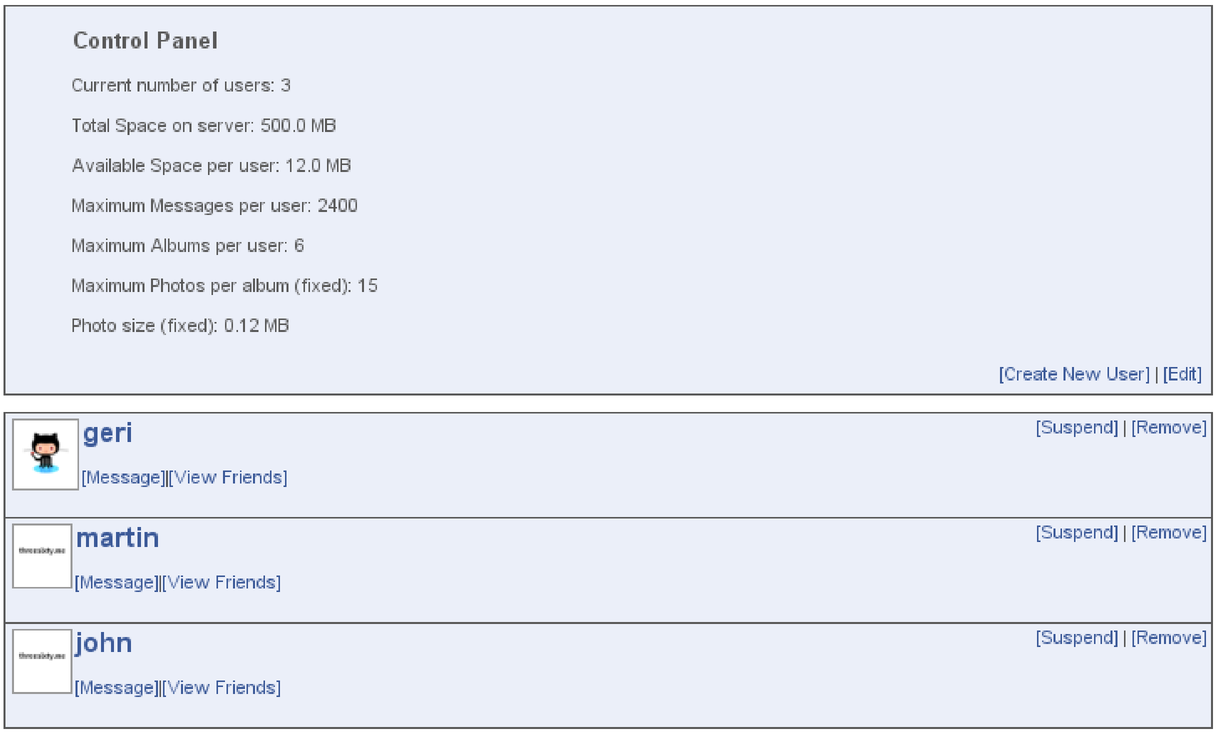
5.6 Realization of Administrator User

The administrator user is super user, who has access to the application settings, stored in Settings database table, and can control other members of the web community application. For the realization of the administrator user, the application uses an *admin* namespace with routes to admin UsersController and admin

SettingsController. The code snippet shows the declaration of *admin* namespace in *routes.rb* file, stored in *config* directory.

```
map.namespace :admin do |admin|
  admin.resources :users, :member => { :suspend   => :post,
                                      :unsuspend => :post,
                                      :purge     => :delete }
  admin.resource :settings, :collection => {:calculate => :any}
end
```

The *admin.resources :users* and *admin.resource :settings* create RESTful routes. RESTful routes provide a mapping between the HTTP verbs and Admin::UsersController, respectively Admin::SettingsController actions. Figure 5.11 shows the administrator user page of the Web Community application. The page has “Control Panel” section, where the administrator can see the current application settings. Application settings can be edited by pressing the “Edit” link in the bottom-right corner of the “Control Panel”. Under the “Control Panel” section administrator can see and control all the community users.



Control Panel

Current number of users: 3

Total Space on server: 500.0 MB

Available Space per user: 12.0 MB

Maximum Messages per user: 2400

Maximum Albums per user: 6

Maximum Photos per album (fixed): 15

Photo size (fixed): 0.12 MB

[Create New User] | [Edit]

geri [Suspend] | [Remove]
[Message][View Friends]

martin [Suspend] | [Remove]
[Message][View Friends]

john [Suspend] | [Remove]
[Message][View Friends]

Fig. 5.11 Administrator user page of the Web Community application.

The User model in the application acts as a finite state machine. The Web Community application uses the AASM (Acts As State Machine) library for Ruby to add the finite state machine functionality. The following code snippet defines the user states.

```
aasm_column :state
```

```
aasm_initial_state :initial => :passive
aasm_state :passive
aasm_state :pending, :enter => :make_activation_code
aasm_state :active,  :enter => :do_activate
aasm_state :suspended
aasm_state :deleted, :enter => :do_delete
```

The column *state* in Users database table is used to save the current user state. User can have five different states, where passive state is used only when new user is created. The *:enter* option executes the given action when user enters in the state. Figure 5.12 shows a state diagram of User model.

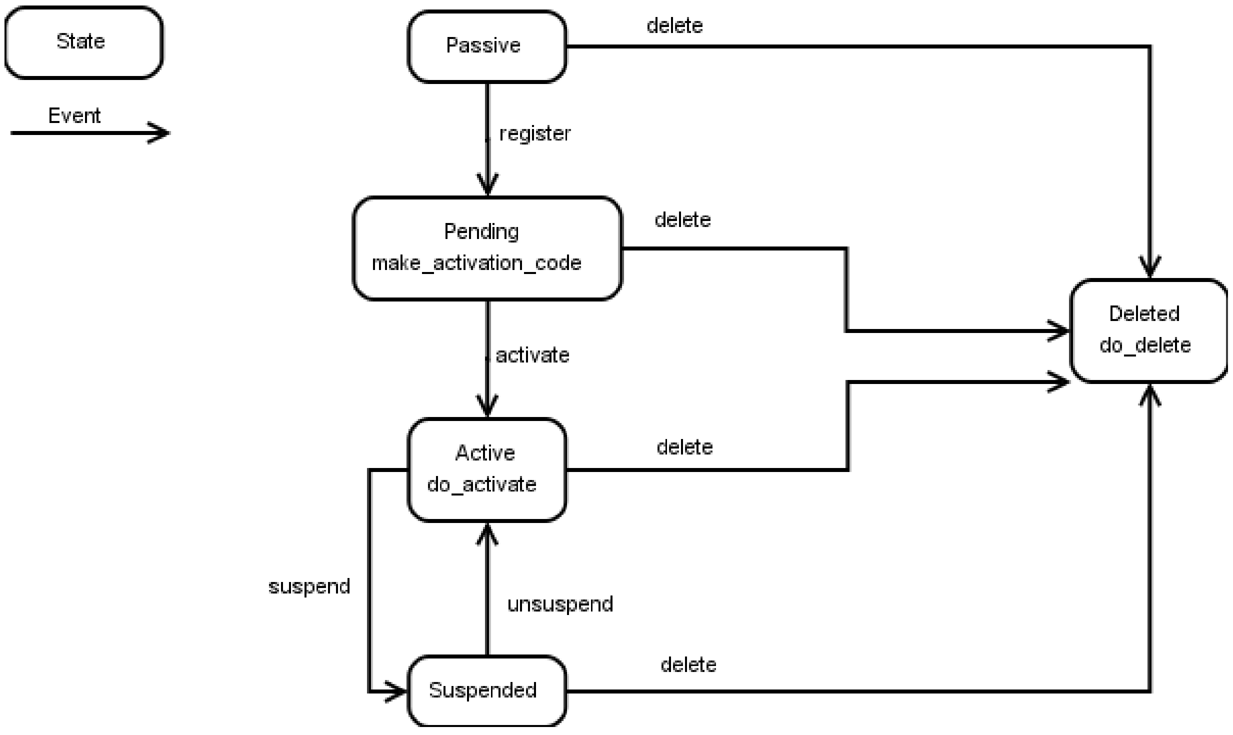


Fig. 5.12 State diagram of User model in the Web Community application.

The state transitions are defined by the following code snippet.

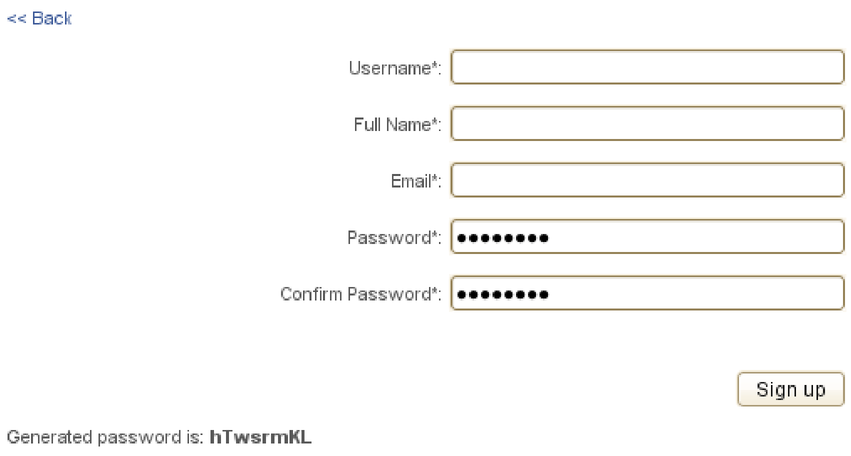
```
 aasm_event :activate do
  transitions :from => :pending, :to => :active
end
aasm_event :suspend do
  transitions :from => [:passive, :pending, :active], :to => :suspended
end
aasm_event :delete do
  transitions :from => [:passive, :pending, :active, :suspended], :to => :deleted
end
aasm_event :unsuspend do
  transitions :from => :suspended, :to => :active,  :guard => Proc.new {|u| !
u.activated_at.blank? }
  transitions :from => :suspended, :to => :pending, :guard => Proc.new {|u| !
u.activation_code.blank? }
```

```
    transitions :from => :suspended, :to => :passive
  end
```

Now lets see how a new community user is created and controlled by the administrator. By pressing the “Create New User” link, the administrator calls the *new* action in Admin::UsersController.

```
def new
  @random_password = User.random_password(8)
  @user = User.new
end

def self.random_password(size = 8)
  chars = (('a'..'z').to_a + ('0'..'9').to_a + ('A'..'Z').to_a) - %w(i o 0 1 l 0 O)
  (1..size).collect{|a| chars[rand(chars.size)] }.join
end
```

The *new* action generates a random password using the *random_password(size)* method in User model, creates new empty User object and renders the new user page. The *random_password(size)* method takes alphabets, numbers from 0 to 9, capital letter alphabets and adds them to an array. Then removes 'I', 'o', '0', '1', 'l', '0', 'O' and stores the new array. Finally eight random characters are chosen from the array and returned as a string. Figure 5.13 shows the new user page of the Web Community application.



Username*:	
Full Name*:	
Email*:	
Password*:	••••••••
Confirm Password*:	••••••••

Generated password is: **hTwsrmKL**

Fig. 5.13 New user page of the Web Community application.

Administrator should enter user name, full name and email in the text fields provided, the password and confirm password fields are filled out automatically. After pressing the “Sign up” button, the *create* action in Admin::UsersController is called.

```
def create
  @user = User.new(params[:user])
  @user.register! if @user && @user.valid?
  ...
end
```

New user object is created using the :user parameters, initial user *state* is passive. User is registered and state transition is done – from passive to pending. User activation code is generated using the *make_activation_code()* method in User model and stored in the *activation_code* column in Users database table. An email with the user activation code is sent to user's email account. After user clicks the activation link, a state transition is done from pending to active.

How does the email notification work?

The class UserObserver in *app/models* directory observes the User model, it has two methods *after_create(user)* and *after_save(user)*. Each time a user is saved or new user is created, the *after_save*, respectively *after_create* method is called.

```
def after_create(user)
  user.reload
  UserMailer.deliver_signup_notification(user)
end

def after_save(user)
  user.reload
  UserMailer.deliver_activation(user) if user.recently_activated?
  UserMailer.deliver_reset_notification(user) if user.recently_reset?
end
```

The UserMailer class in *app/models* is responsible for email message creation and delivery. The application use email notifications when new user is created, activated or user password is reseted.

The administrator user can suspend, activate or remove by one click any user of the Web Community application. Once the “suspend” link is clicked, the *suspend* action in Admin::UsersController is called, a state transition from active state to suspended state is done. In suspended state user can not access his or her account, but the account is present in the database and can be activated by the administrator. Once the “remove” link is clicked, the *purge* action in Admin::UsersController is called, a state transition from current user state to deleted state is done, user account is deleted and can not be recovered.

5.7 Realization of Navigation Bar and Search Engine

When logged-in, the community user can navigate through the pages using the navigation bar, positioned on the top of the page. Figure 5.14 shows the navigation bar of the community platform. The partial template can be found in *app/views/shared/_navigation.html.erb* file.

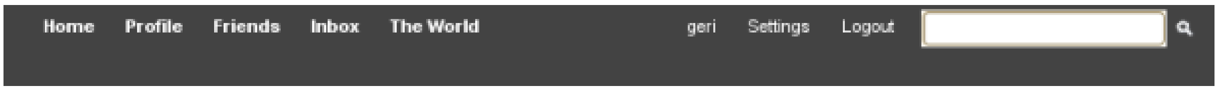
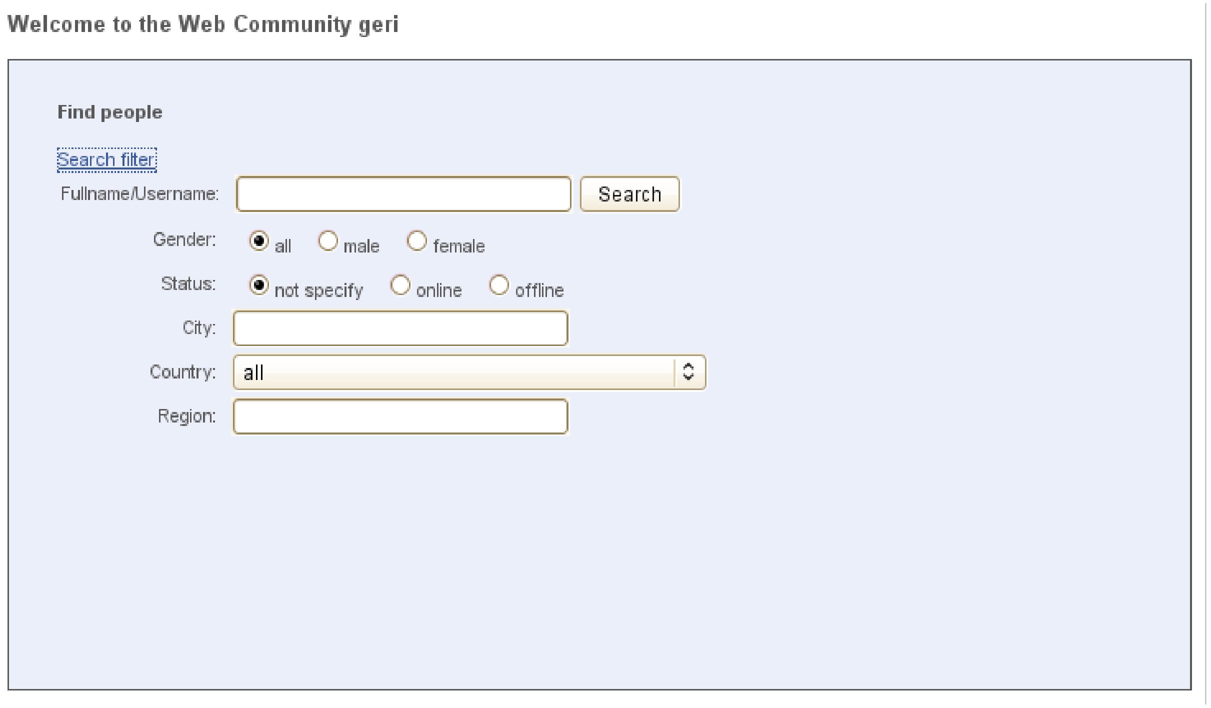


Fig. 5.14 Navigation bar of the Web Community application.

The navigation bar contains links to home page (“Home”), show profile page (“Profile”), show friends page (“Friends”), inbox page (“Inbox”), show map page (“The World”). In right corner links to home page (user's login name used), settings (“Settings”) and logout link (“Logout”) can be found. The search form in the right corner uses the *search(search)* method in User model.

```
def self.search(search)
  if(search)
    find(:all, :conditions => ['login LIKE ? OR name LIKE ?', "%#{search}%", "%#{search}%"])
  end
end
```

The method takes the search parameters entered by the user, finds all users in Users database table, who have login like or name like the searched name. The home page of the Web Community application includes an extended search form, where user can search other users by user name, full name, gender, status, city, country and region. Figure 5.15 shows the home page of the Web Community platform.


Welcome to the Web Community geri

Find people

Search filter

Fullname/Username:	
Gender:	[✓ all] male female
Status:	[✓ not specify] online offline
City:	
Country:	all
Region:	

Fig. 5.15 Home page of the Web Community application.

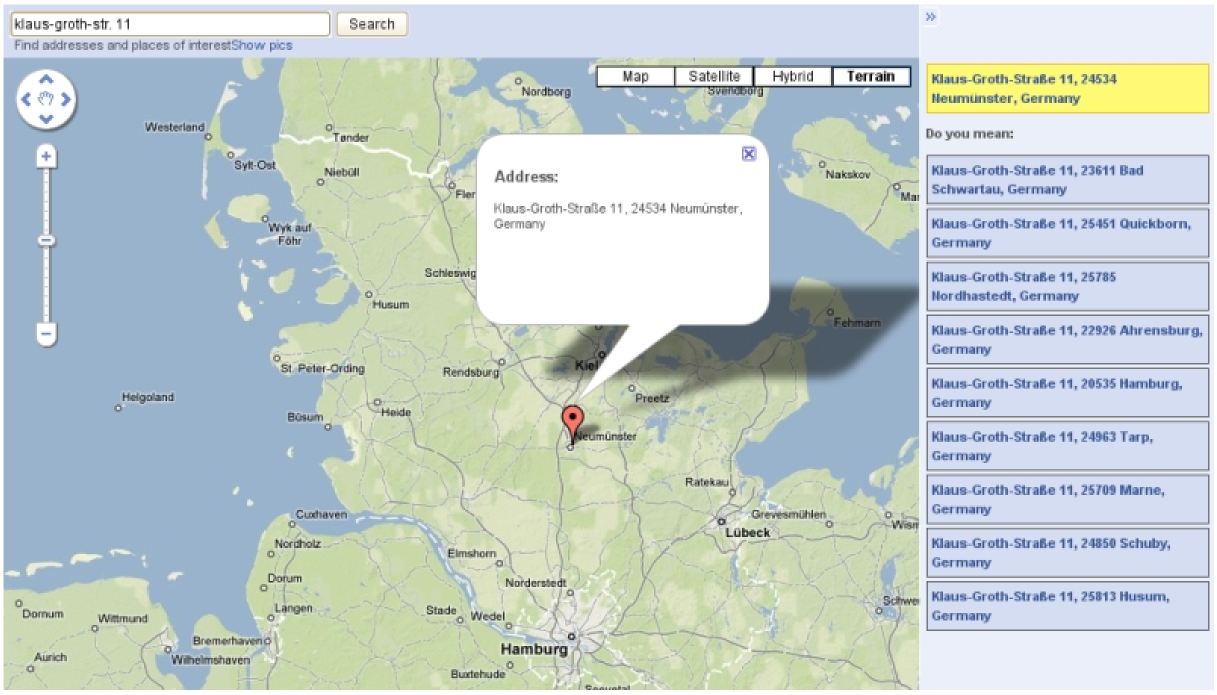
The search form uses the *extended_search(params)* method in User model to search for other community users in Users database table. The following code snippet shows the the extended search method.

```
def self.extended_search(params)
    query= [] #here go the SQL strings with "?" placeholders
    values= []#here go the values to be inserted in the strings of the query
    if params[:login] && params[:login] != ""
      query << " users.login = ?"
      values << params[:login]
    end
    if params[:gender] && params[:gender] != "all"
      query << " profiles.gender = ?"
      values << params[:gender]
    end
    if params[:status] && params[:status] != "all"
      query << " profiles.status = ?"
      values << params[:status]
    end
    if params[:city] && params[:city] != ""
      query << " profiles.city = ?"
      values << params[:city]
    end
    if params[:country] && params[:country].to_s != "all"
      query << " profiles.country = ?"
      values << params[:country]
    end
    conditions = []
    conditions << query.join(" AND ")
    conditions += values
    find(:all, :conditions => conditions, :include => [:profile])
  end
```

The *extended_search(params)* method checks the selected by the user search form fields. Each field value has a query string and parameter value, which are pushed into a *query*, respectively *values* arrays. After all the parameters are checked, *query* array entries are joined by an AND phrase and pushed into *conditions* array. At the end *conditions* and *values* are joined, Users table is searched and all the matching users are returned.

5.8 Integrating Google Maps

Community user has the ability to interact with the World's map. On current point of development the user can search places and addresses on the map, he can also link photos from his albums to map locations. Figure 5.16 shows how the search places and addresses work.



Address:

Klaus-Groth-Straße 11, 24534 Neumünster, Germany

Klaus-Groth-Straße 11, 24534 Neumünster, Germany

Do you mean:

- Klaus-Groth-Straße 11, 23611 Bad Schwartau, Germany
- Klaus-Groth-Straße 11, 25451 Quickborn, Germany
- Klaus-Groth-Straße 11, 25785 Nordhastedt, Germany
- Klaus-Groth-Straße 11, 22926 Ahrensburg, Germany
- Klaus-Groth-Straße 11, 20535 Hamburg, Germany
- Klaus-Groth-Straße 11, 24963 Tarp, Germany
- Klaus-Groth-Straße 11, 25709 Marne, Germany
- Klaus-Groth-Straße 11, 24850 Schuby, Germany
- Klaus-Groth-Straße 11, 25813 Husum, Germany

Fig. 5.16 Map page of the Web Community application.

After entering an address in the search field and pressing “Search” button, the search action in MapsController is called. The found locations are displayed in the sidebar, the first found location is focused and pointed on the map.

For linking user photos to the map, the Locations database table is used to store photo location. Each photo can be linked to only one location on the map. This is done from the show photo page in Web Community platform. The following figure shows the add location form.

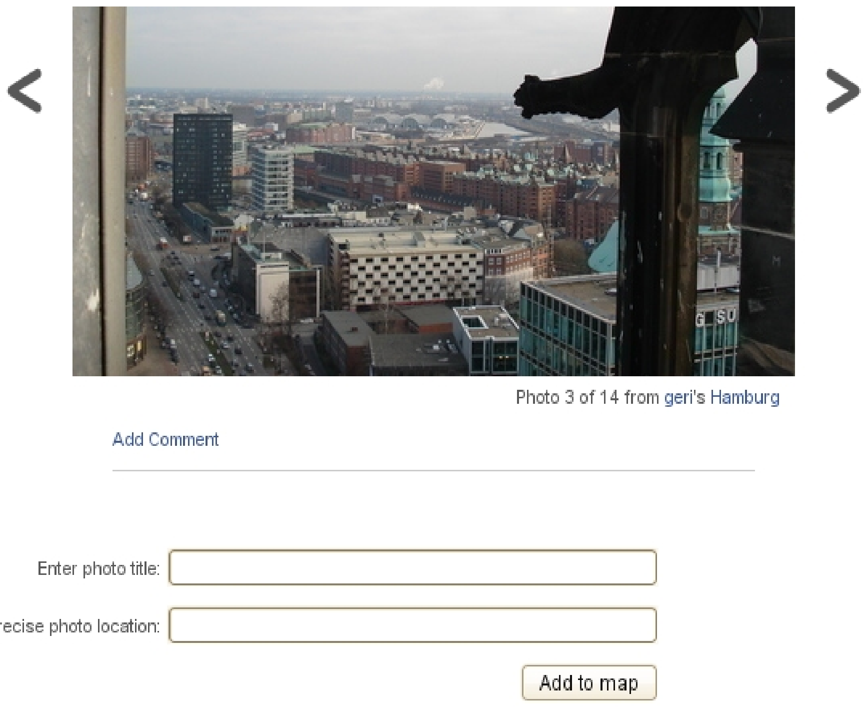


Photo 3 of 14 from geri's Hamburg
Add Comment
Enter photo title:
Enter precise photo location:
Add to map

Fig. 5.17 Add photo location form.

The add photo location form consist of two text fields – “Enter photo title” and “Enter photo location”. User should enter both photo title and precise location, when pressing the “Add to Map” button, the *create* action in LocationsController is called.

```
def create
   @photo = Photo.find(params[:id])
   @loc = GoogleGeocoder.geocode(params[:address])
   if @loc.success
     @location = Location.new(:photo_id => @photo.id, :name => params[:name],
       :address => @loc.full_address, :city => @loc.city, :state => @loc.state,
       :zip => @loc.zip, :country => @loc.country_code, :lat => @loc.lat, :lng => @loc.lng)
       if  @location.save
           ...
end
```

The *create* action uses Geokit library for Ruby to geocode the address entered. If successfully geocoded, a new Location object is created with the geocoded address parameters. In the page sidebar a small map with a marker, pointing exact photo location, will appear. All the entries, stored in the Locations database table, can be viewed on the map as small thumbnails. By clicking them an info window with the photo is showed on the map. The photo is a link to show photo page of user, uploaded that photo. Figure 5.18 shows a map example with linked photos on it. By clicking the info window photo, user will be redirected to the uploader's profile.

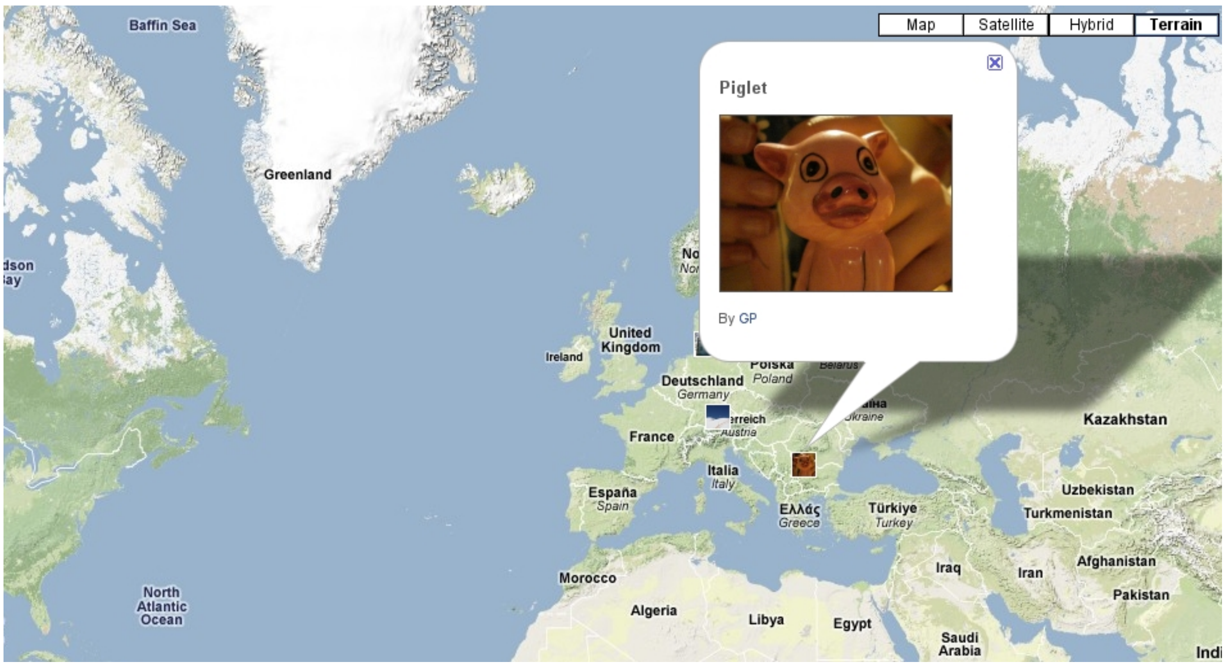


Fig. 5.18 Show map page with photo markers.

For integrating Google Maps in the web community project, an API key is needed. The API key can be generated on the Google Maps API website [14] and it can be used only for one domain name. The Google Maps API is implemented in Javascript and to integrate it in the application, a script tag in the page head should be included.

```
<script type="text/javascript" src="http://maps.google.com/maps?
file=api&v=2&key=ABQIAAAAdAtcnlwG_jTGwL1CAqxcVxTJQa0g3lQ9GZqIMmInSLzwtGDKaBT1Jt-0
jpamfPG4jJsvBmLBDf0XIg&sensor=true">
</script>
```

The following code includes the Google Javascript files if the key provided matches the domain name of the application. Since the web community application is still in development process, the key used is for http://localhost:3000/. All the functions used for manipulating the map - map initialization, adding and removing markers are written in Javascript and can be found in the *public/javascripts/application.js* file. The Web Community application CSS files can be found in *public/stylesheets* directory.

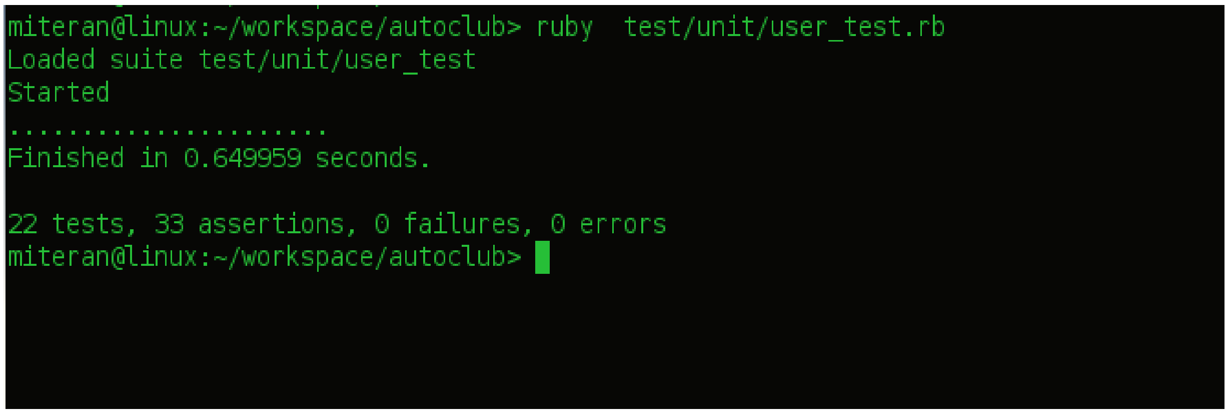
Chapter 6 – Testing and Further Development

The purpose of tests is to check the correctness and stability of the front-end and also of the back-end of the application. The web community project has been tested with and without using a web browser.

6.1 Automated Tests

Developing with Ruby on Rails allows to write automated tests and fixtures for the Web Community application. By convention tests are divided into unit tests – for testing the models, functional tests – for testing a single action in a controller, and integration tests – for testing the flow through one or more controllers. Fixtures are files in YAML or CSV format, which contain the data for a single model. The name of the fixture file is significant, because it should match the name of a database table. All the automated test classes and fixtures files are located in *test* directory of the application's root directory.

Unit tests are used to test the Web Community application models. The unit test classes can be found in *test/unit* directory of the application. Unit test classes are subclasses of Test::Unit::TestCase class, part of the Test::Unit framework that comes pre-installed with Ruby. Executing can be done easily from the console terminal. Figure 6.1 shows how to run unit tests. Here we ran all unit tests, written for the User model in the Web Community application. The tests are stored in *user_test.rb* file in *unit/test* directory.



```
miteran@linux:~/workspace/autoclub> ruby  test/unit/user_test.rb
Loaded suite test/unit/user_test
Started
......................
Finished in 0.649959 seconds.

22 tests, 33 assertions, 0 failures, 0 errors
miteran@linux:~/workspace/autoclub>
```

Fig. 6.1 Run all unit tests for User model.

When all unit tests have finished, a message with the elapsed time in seconds, number of tests executed, number of assertions, failures and errors is displayed. If the test is passed, 0 failures and 0 errors should appear. Lets have a look now at some unit test examples.

```
class UserTest < ActiveSupport::TestCase
  fixtures :users #load fixtures which will be used
  ...
```

```
end
```

The fixtures directive loads the fixture data, corresponding to the given model name, into the corresponding database table before each test method in the test case is run. The name of the fixture file determines the table that is loaded, in case of using *:users* will cause the *users.yml* fixture file to be used. Here is how a fixture from *users.yml* file look like.

```
quentin:
  id: 3
  login: quentin
  name: Quentin Tarantino
  email: quentin@hotmail.com
  salt: aca9e1a5cd7629909bac4d647d9484cdc54008c3
  crypted_password: 26c9e3a8ed1204e22ab09347aa87895806d38f0c
  state: active
  activated_at: 2009-05-08 21:05:36
  activation_code:
  deleted_at:
  reset_code:
  remember_token:
  remember_token_expires_at:
```

Each test method always starts with the prefix “test_”. The following code snippet shows two tests – *test_should_require_email* and *test_should_reset_password* from *user_test.rb* file

```
class UserTest < ActiveSupport::TestCase
 ...
 def test_should_require_email
    u = create_user(:email => nil)
    assert !u.save, "User with no email saved!"
 end

 def test_should_reset_password
   users(:quentin).update_attributes(:password => 'new password',
:password_confirmation => 'new password')
   assert_equal users(:quentin), User.authenticate('quentin@hotmail.com', 'new password')
 end
 ...
end
```

The first method – *test_should_require_email,* creates a User object without an email. The assert statement checks if user can be saved. In case of user saved, the assertion is not true and generates a failure, “User with no email saved!” is displayed and test fails. The second method resets the password of user *quentin* if the two user objects are the same. If password is not reseted, assertion will generate a failure and test will fail.

Functional tests are used to test the Web Community application controllers. The functional test classes can be found in *test/functional* directory of the application. They are subclasses of ActionController::TestCase class. The purpose of writing

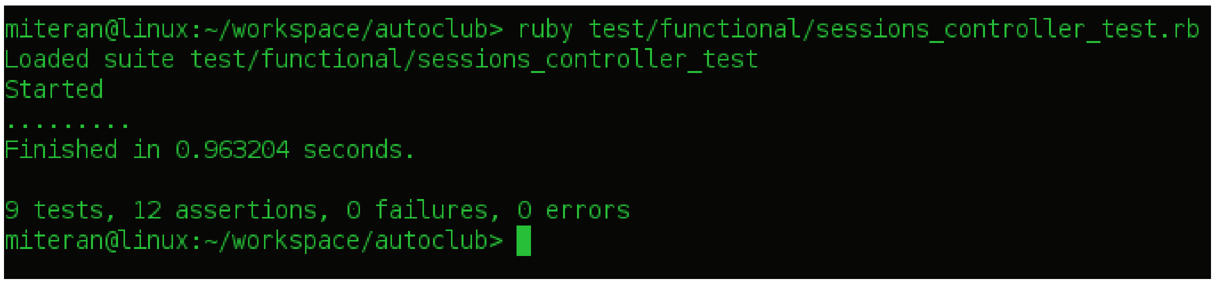
functional tests is to test various actions of a single controller. Controllers handle the incoming web request to the Web Community application and eventually respond with a rendered view.
Application functional tests inspect if the web request was successful, if the community user was redirected to the right page, if the community user was successfully authenticated, if the correct object was stored in the response template, if the appropriate message was displayed, and etc. The following code snippet shows functional tests from the *sessions_controller_test.rb* file, found in *test/functional* directory.

```
class SessionsControllerTest < ActionController::TestCase

  fixtures :users, :folders
  def test_should_login_and_redirect
    post :create, :email => 'quentin@hotmail.com', :password => 'monkey'
    assert session[:user_id]
    assert_response :redirect
  end
   ...
  def test_should_logout
    login_as :quentin
    get :destroy
    assert_nil session[:user_id]
    assert_response :redirect
  end

  def test_should_remember_me
    @request.cookies["auth_token"] = nil
    post :create, :email => 'quentin@hotmail.com', :password => 'monkey', :remember_me
=> "1"
    assert_not_nil @response.cookies["auth_token"]
  end
   ...
end
```

User opens the start page, enters his or her email account and password, and clicks the “Login” button. A POST request to the create action in SessionsController is sent, if user is valid, he or she is redirected to user home page. Now we check this scenario by writing an automated test. First, all the fixtures needed for the tests are loaded. The *test_should_login_and_redirect* method generates a POST request to SessionsController, calling the create method with *:email* and *:password* parameters set. On the next code line assertion checks if new user session has been created. Finally, assertion checks if the response was redirect.

User is logged in and he or she decides to logout. The user clicks the “Logout” link, a GET request to the destroy action in SessionsController is sent, user session is set to *nil*, user is redirected to start page. We check this scenario by writing the second method – *test_should_logout.* It generates a GET request to the destroy action in SessionsController. After the destroy action is executed, the test checks with an assertion if user session is set to nil and if the response is redirect. The test will be successful when all assertions have passed, and no failure or error has been generated.

The third test - *test_should_remember_me*, generates a POST request to create action in SessionsController with *remember_me* parameter set to '1', and checks if after the log-in, the *auth_token* cookie is set. The two instance variables - @*request* and @*response*, are available by default to all functional tests. Figure 6.2 shows the results after running all sessions controller functional tests.



```
miteran@linux:~/workspace/autoclub> ruby test/functional/sessions_controller_test.rb
Loaded suite test/functional/sessions_controller_test
Started
.........
Finished in 0.963204 seconds.

9 tests, 12 assertions, 0 failures, 0 errors
miteran@linux:~/workspace/autoclub>
```

Fig. 6.2 Running sessions controller functional tests.

The Web Community application is working in development environment. Not all automated tests are written. In order to bring the project to production environment, unit and functional tests should be included for each application model, respectively application controller. Unit tests should include at least one test method for each model function. Functional tests should include at least one test method for each controller action.

6.2 Usability Tests

The Web Community application has been tested with the web browser. In order to test the project functionality, test users are used. Next step will be to run through a typical web page scenario. First, the community user opens the start page in his or her browser client. User enters his or her email account and password and tries to log-in (Figure 6.1). Log-in fails, because user has entered a wrong email address or wrong password. User is redirected to log-in page (Figure 6.2), an error message is displayed.

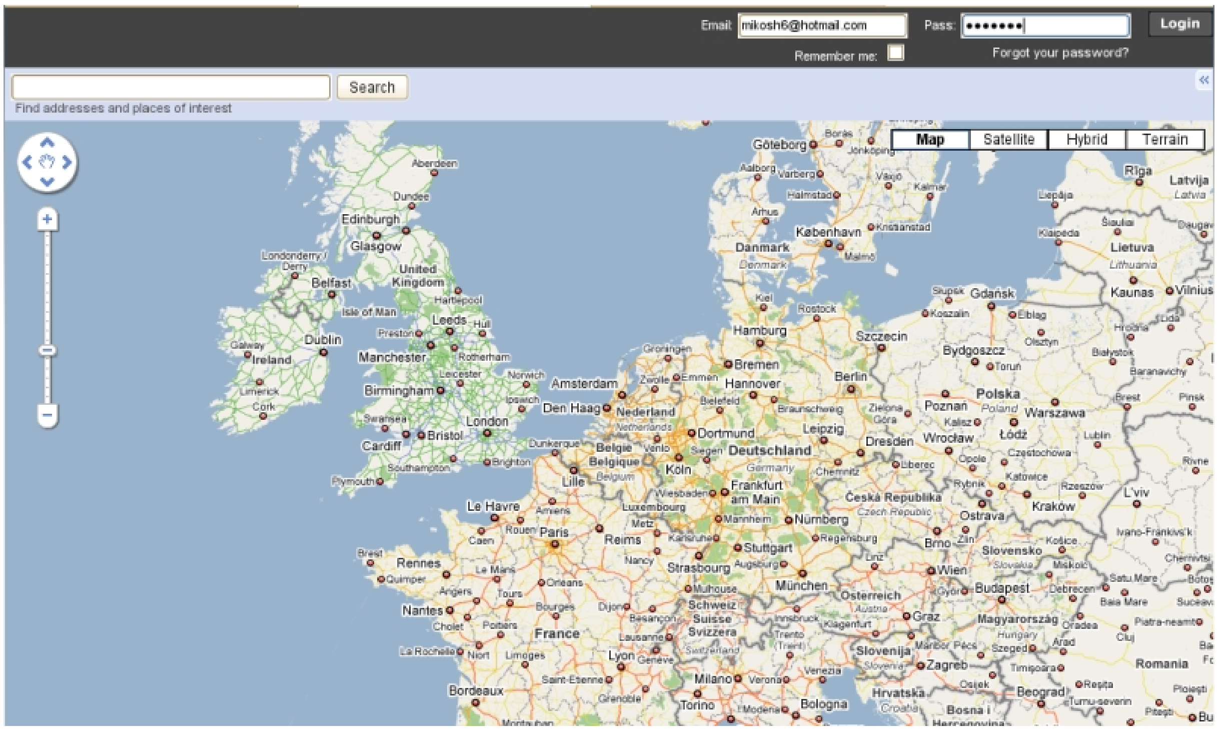
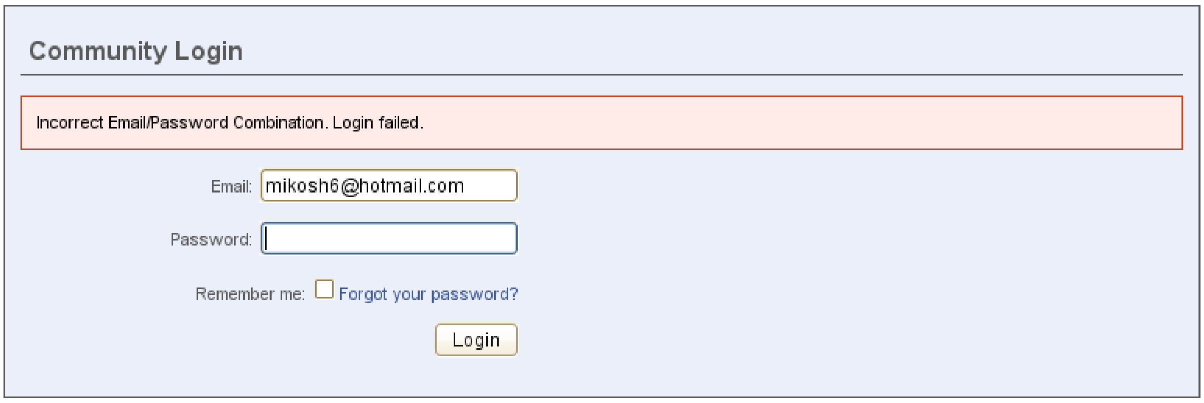


Fig. 6.3 User opens the start page and tries to log-in.


Community Login

Incorrect Email/Password Combination. Login failed.

Email: mikosh6@hotmail.com

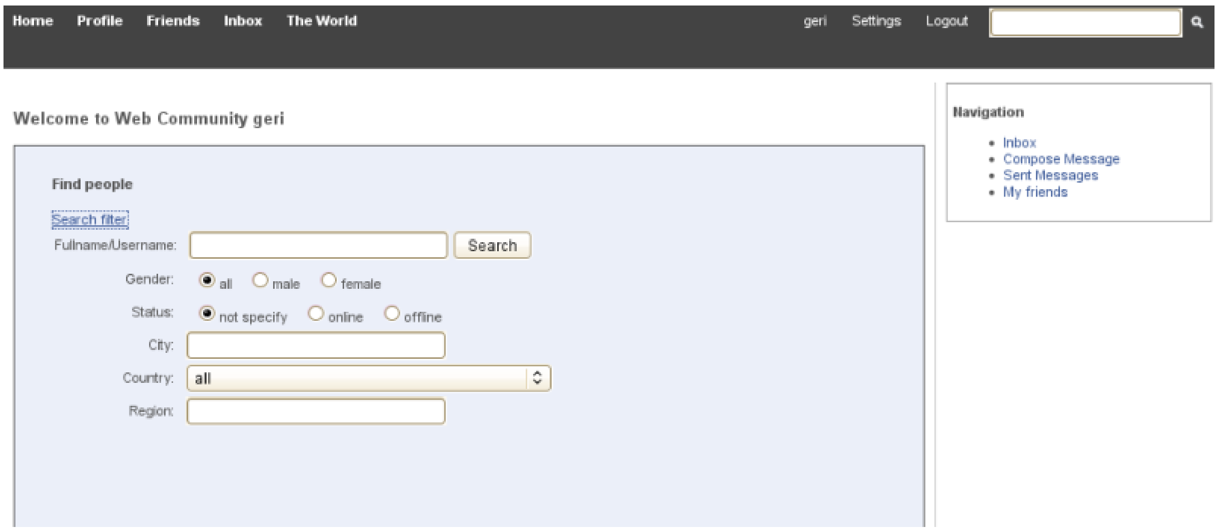
Password:

Remember me: ☐ Forgot your password?

Login

Fig. 6.4 User is asked to enter correct email/password combination.

After user enters the correct email and password combination, he or she is redirected to the home page (Figure 6.5). User decides to add new image to his or her album. He or she goes to the profile page (Figure 6.6) and clicks on the the desired album in Albums section. Edit album page is displayed. Community user selects the add image form and tries to update the album. He or she decides to test the form and selects a non image file or a image file bigger than 3 MB. When clicking the "Update" button, the page is redisplayed with an error message on top (Figure 6.5). Album can not be updated.



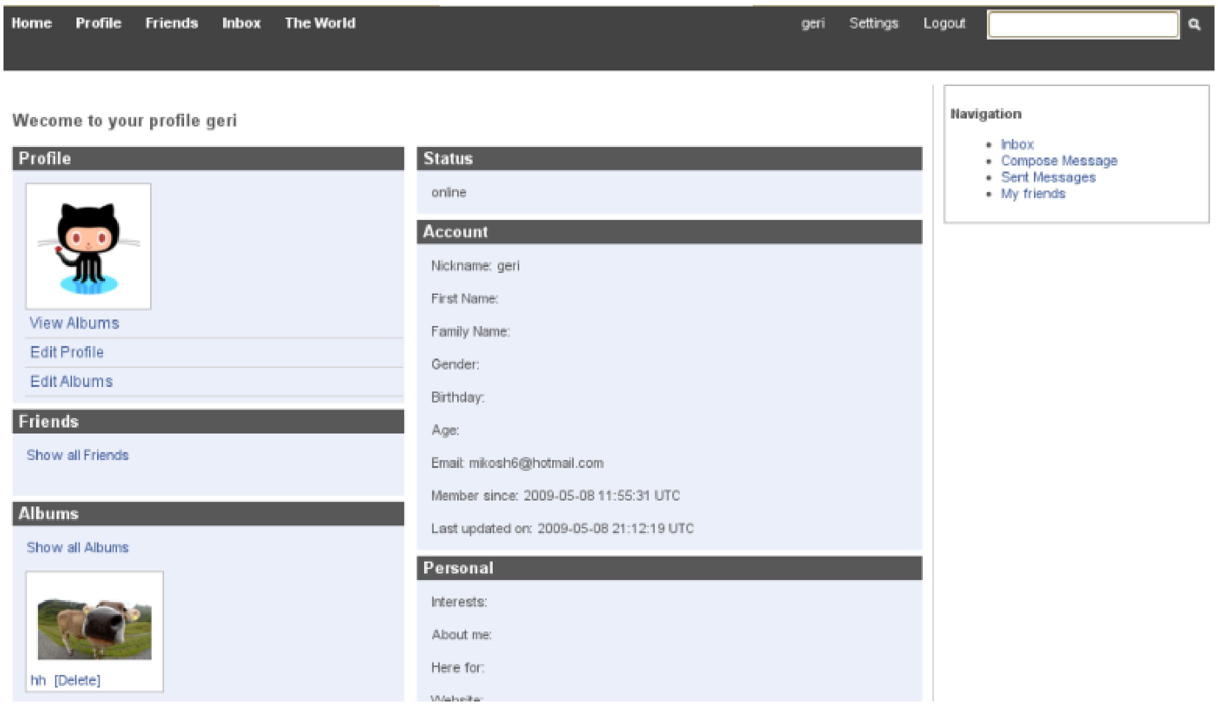
Welcome to Web Community geri

Find people

Search filter

Fullname/Username:	
Gender:	[✓ all] male female
Status:	[✓ not specify] online offline
City:	
Country:	all
Region:	

Fig. 6.5 User is redirected to home page.



Wecome to your profile geri

Profile

Friends

Albums

Status

online

Account

Nickname: geri

First Name:

Family Name:

Gender:

Birthday:

Age:

Email: mikosh6@hotmail.com

Member since: 2009-05-08 11:55:31 UTC

Last updated on: 2009-05-08 21:12:19 UTC

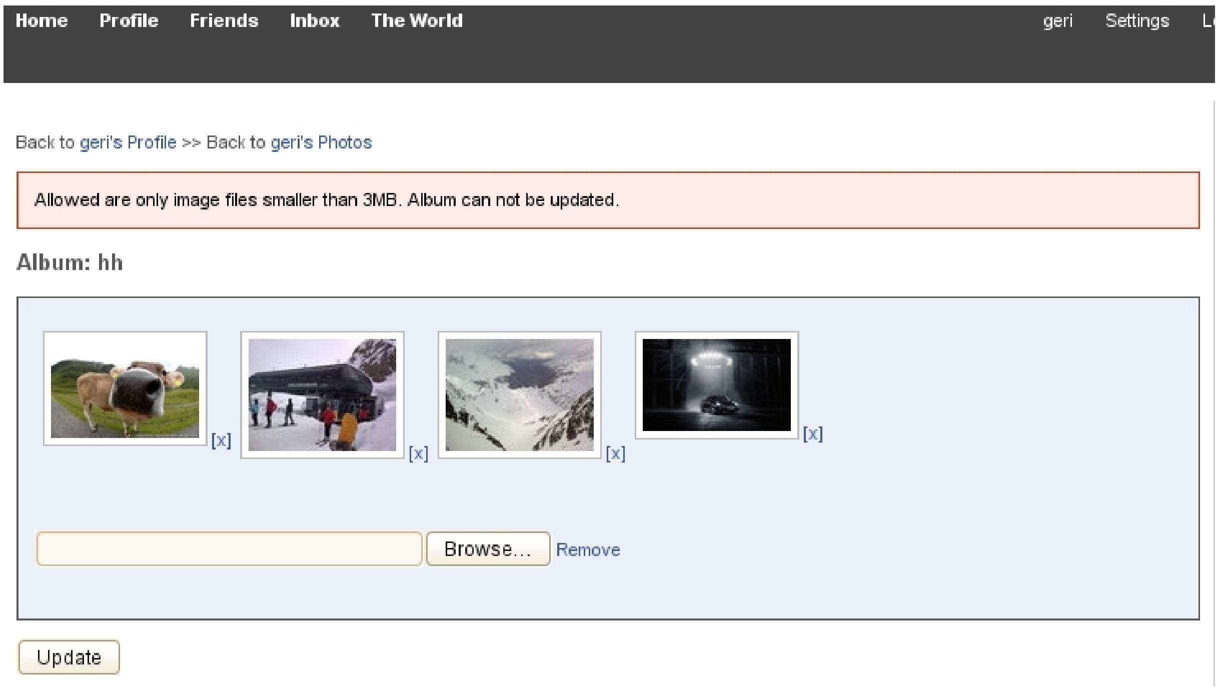
Personal

Interests:

About me:

Here for:

Fig. 6.6 User selects the desired album.



Allowed are only image files smaller than 3MB. Album can not be updated.

Fig. 6.7 User attempts to update album with invalid content.

Similar usability tests has been made for all the pages and user forms in the Web Community application. Usability tests help to find application errors and web browser bugs. On current level of development, application is tested only with Mozilla Firefox browser and full functionality with other web browser clients is not guaranteed. For future testing, the Web Community application will be loaded on real web server.

6.3 Further work

More Testing. The application is still running in development environment and includes some newly developed parts, which are not yet fully tested. Application errors and bugs could still occur. In order to bring the Web Community application to production environment additional tests are needed. Future development includes writing more automated unit and functional tests. Every model and controller method should be tested with and without a web browser support.

Extending web browsers support. Future development includes testing and modifying the application in order to work with other web browsers. Support for Internet Explorer and Google Chrome is planned to be developed. Different web browsers display in a different way the page and code optimization is needed. Application modification includes changes in some HTML view templates, as well as changes in the CSS files of the application.

Implementing unimplemented features.

Use SSL to transmit sensitive information. Future development includes implementing a HTTPS protocol for SessionsController and UsersController, in order to encrypt the traffic between the user web browser and the server. SSL will be used whenever forms that capture sensitive information appear, and whenever the server responds to user with a sensitive information. If a regular HTTP request comes along for a method that has been declared to require SSL, it will be intercepted and immediately issue a redirect back to the same URL, but with a protocol of HTTPS. That way the community user will automatically be switched to a secure connection without the need to perform any explicit protocol setting. Similarly, if an HTTPS request comes in for an action that shouldn't use SSL, it will be automatically redirected back to the same URL, but with a protocol of HTTP.

Implementing a Blog. Future development includes implementing a blog, where users can post new articles, other community users can comment and non community users can only read the articles and article's comments. In order to extend the Web Community application with a blog, new database table should be created. This database table will hold the blog articles. An Article model, as well as an ArticlesController will be needed. Each article will have many comments with *commentable_id* holding the id of the article and *commentable_type* will be of type Article. A normal article record will have an *id*, *article_title*, *article_text*. The blog page will show all the articles, by calling the *index* action in ArticlesController class. By selecting an article user will be redirected to show article page, the *show* action in ArticlesController will be called. The show article page will display the article with its full text, the comments for this article, and a HTML form for writing new comment. Only signed community users will be able to write comments to the article. Community blog will be used for additional communication between users. It will also provide community information to other, non-registered users.

Extending map functionality. On current level of application development, Google Maps is integrated in the Web Community application. By loading the map, user has the option to change between different map types, zoom out and zoom in the map, search the map for places of interest and addresses. Logged-in users have the additional option to see linked images on the World's map. Extending the current functionality includes providing users to calculate routes and distances between two points of interest. Function to link images directly on the map is also planned to be implemented. These new functions will be programmed in Javascript. Community user will be able to save locations from the map in My Locations profile section, which is not yet implemented. My Locations will link other community members – friends to user's locations.

Chapter 7 – Conclusion

The title of the thesis is “Development of Web Community Application with Ruby on Rails framework”. This report described how the MVC architecture applies to the Web Community application under the Ruby on Rails framework environment. Using the RoR technology, the application has been divided into smaller components. Developing application's functionality into modules, has helped to separate stable code from frequently changed one, to reuse source code, and to maintain and extend the application more easily.

The main objectives at the start of the thesis were as follows

- To design and develop a data structure that can comprehensively define and hold each user properties.
- To design and develop a friendly user interface by which users of the system can interact with it in a browser environment.
- The users must be able to log-in the application by using a personal email address and a password.
- The users must be able to change their profile settings.
- The users must be able to link with other users and to share each other profiles.
- The users must be able to comment on other users profiles and images.
- The users must be able to send and receive messages from other users.

At the end of the thesis the following things are achieved

- A front-end is developed for the client browser which is a system by which users can interact with the application.
- A data structure that comprehensively defines and stores user properties, comments, friendships and messages is devised.
- The administrator user can create new users and add them or remove them from the community application.
- The user can log-in the application by using email address and a password.
- Once the user has logged-in, he is able to change his profile properties.
- Once the user has logged-in, he is able to search and add new friends.
- Once the user has logged-in, he is able to share his profile with other friends, comment on others profile, comment on others images.
- Once the user has logged-in, he is able to check his mailbox, write, send and receive messages from other community users.

Extra features that have been implemented

- User is able to create albums and add photos to his albums.
- User is able to interact with Google Maps by means of searching places, zooming and moving across the map, changing the map type.

- User is able to link his album photos to the map, making them visible to other users.

Further work involves continuing writing and executing tests, solving different errors and application bugs, implementing unimplemented features. After securing and testing is finished, the application will be brought to production environment and will be deployed on server.

Finally, it can be stated that all the important aspects of the project have been fulfilled by this thesis. Uncompleted work has been mentioned in the Further Work section of Chapter 6.

Appendix A - References

[1] Dave Thomas, David Heinemeier Hansson: Agile Web Development with Rails Second Edition, Pragmatic Bookshelf, ISBN 0-9776166-3-0

[2] Dave Thomas, Chad Fowler and Andy Hunt: Programming Ruby Second Edition, Pragmatic Bookshelf, ISBN 0-9745140-5-5

[3] Andre Lewis, Michael Purvis, Jeffrey Sambells and Cameron Turner: Beginning Google Maps Applications with Rails and AJAX, Apress, ISBN 1-59059-787-7

[4] Andy Budd, Cameron Moll and Simon Collison: CSS Mastery, Friendsof, ISBN 1-59059-614-5

[5] http://www.ruby-lang.org/en/ (27.03.2009)

[6] http://rubyonrails.org/ (27.03.2009)

[7] http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/app-arch/app-arch2.html (29.03.2009)

[8] http://thepaisano.files.wordpress.com/2008/04/rails2.png (29.03.2009)

[9] http://rewrite.rickbradley.com/pages/moving_to_rails/ (29.03.2009)

[10] http://en.wikipedia.org/wiki/List_of_social_networking_websites (04.04.2009)

[11] http://kb.linuxvirtualserver.org/images/6/60/Rails-apache-mongrel.png (05.04.2009)

[12] http://javatouch.googlepages.com/spring-overview.png/spring-overview-full.jpg (13.04.2009)

[13] http://rmagick.rubyforge.org/ (29.03.2009)

[14] http://code.google.com/apis/maps/signup.html (08.05.2009)

Appendix B - Glossary

AJAX	Asynchronous Javascript and XML
ATOM	Web syndication format used for web feeds
CGI	Common Gateway Interface
CMS	Content Management System
CSS	Cascaded Style Sheets
CSV	Comma Separated Values
DBMS	Database Management System
ERb	Embedded Ruby
Ext3	Third extended file-system commonly used by the Linux kernel
JSP	Java Server Pages
NTFS	Windows NT standard file-system
MVC	Model View Controller
OS	Operating System
RDBMS	Relational Database Management System
RSS	Web syndication format used to publish frequently updated works
RoR	Ruby on Rails
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol

HTTPS	Hyper Text Transfer Protocol Secure
XHTML	Extensible Hyper Text Markup Language
XML	Extensible Markup Language
YAML	Human-readable data serialization format
WCMS	Web Content Management System
WWW	World Wide Web

Appendix C - Source Code Listing

This Bachelor Thesis contains the source code and documentation of the project on a CD. The CD also includes API HTML Document of the application and the Bachelor Report in PDF format. This Appendix is deposited with Prof. Dr. Hotop.

Declaration

I declare within the meaning of section 25(4) of the Examination and Study Regulations of the International Degree Course Information Engineering that: this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

--------------------------------------	----------------------------------
City, Date	Signature