

UNIVERSITA' DEGLI STUDI DI MESSINA  
DIPARTIMENTO DI MATEMATICA E INFORMATICA

DATABASE COURSE PROJECT

---

**veeForum**

29 May 2015

---

**Author:**

Vittorio ROMEO

**Professor:**

Massimo VILLARI



<http://vittorioromeo.info>



<http://unime.it>

# Contents

<b>I</b>	<b>Project specifications</b>	<b>1</b>
<b>1</b>	<b>Client request</b>	<b>3</b>
<b>2</b>	<b>Software Requirements Specification</b>	<b>5</b>
1.	Introduction . . . . .	5
1.1	Software engineering . . . . .	5
1.2	SRS . . . . .	6
1.3	Purpose . . . . .	7
1.4	Scope . . . . .	7
2.	General description . . . . .	8
2.1	Product perspective and functions . . . . .	8
2.2	User characteristics . . . . .	8
3.	Specific requirements . . . . .	8
3.1	External interface requirements . . . . .	8
3.2	Functional requirements . . . . .	9
3.3	Example use cases . . . . .	11
3.4	Non-functional requirements . . . . .	15
4.	Analysis models . . . . .	16
4.1	Activity Diagrams . . . . .	16
4.2	Sequence Diagrams . . . . .	19
<b>II</b>	<b>Technical analysis</b>	<b>20</b>
<b>3</b>	<b>Development process</b>	<b>22</b>
1.	Environment and tools . . . . .	22
2.	Docker . . . . .	22
3.	Version control system . . . . .	23
4.	LAMP stack . . . . .	23
5.	Thesis . . . . .	24

5.1	LatexPP . . . . .	24
<b>4</b>	<b>Project structure</b>	<b>25</b>
<b>5</b>	<b>Database design</b>	<b>27</b>
1.	Diagrams . . . . .	27
<b>6</b>	<b>SQL</b>	<b>32</b>
1.	Database setup . . . . .	32
1.1	db . . . . .	32
2.	Tables . . . . .	34
2.1	log . . . . .	34
2.2	tag . . . . .	35
2.3	group . . . . .	36
2.4	user . . . . .	37
2.5	section . . . . .	39
2.6	fileData . . . . .	40
2.7	contentBase . . . . .	41
2.8	contentThread . . . . .	42
2.9	contentPost . . . . .	44
2.10	contentAttachment . . . . .	45
2.11	subscriptionBase . . . . .	47
2.12	subscriptionThread . . . . .	48
2.13	subscriptionUser . . . . .	50
2.14	subscriptionTag . . . . .	51
2.15	notificationBase . . . . .	52
2.16	notificationUser . . . . .	53
2.17	notificationThread . . . . .	55
2.18	notificationTag . . . . .	57
2.19	tagContent . . . . .	58
2.20	groupSectionPermission . . . . .	59
3.	Stored procedures . . . . .	60
3.1	mkContent . . . . .	60
3.2	mkSubscription . . . . .	63
3.3	mkNotification . . . . .	66
3.4	utils . . . . .	69
3.5	gNUser . . . . .	72
3.6	gNThread . . . . .	74
3.7	gNTag . . . . .	76

3..8	calcPrivs . . . . .	78
3..9	calcPerms . . . . .	80
4.	Triggers . . . . .	82
4..1	notifications . . . . .	82
4..2	contentBase . . . . .	84
4..3	subscriptionBase . . . . .	86
4..4	notificationBase . . . . .	88
4..5	subscriptionNtf . . . . .	90
4..6	delSubCnt . . . . .	92
5.	Database test data initialization . . . . .	94
5..1	initialize . . . . .	94
<b>7</b>	<b>PHP</b>	<b>97</b>
1.	Library module . . . . .	97
1..1	settings . . . . .	98
1..2	session . . . . .	99
1..3	debug . . . . .	100
1..4	db . . . . .	100
1..5	privs . . . . .	101
1..6	pages . . . . .	102
1..7	utils . . . . .	104
1..8	gen . . . . .	104
1..9	tbl . . . . .	107
1..10	sprocs . . . . .	109
2.	Core module . . . . .	110
<b>8</b>	<b>Web interface</b>	<b>112</b>
<b>III</b>	<b>Conclusion</b>	<b>118</b>
<b>9</b>	<b>Final product</b>	<b>120</b>
<b>10</b>	<b>What I learned</b>	<b>121</b>
<b>11</b>	<b>Future</b>	<b>122</b>
<b>12</b>	<b>References</b>	<b>123</b>

# Part I

## Project specifications

The following part of the document describes the project and its design/development process without exploring its implementation details.

The part begins with a synthesis of the **client request**. After a careful analysis of the request, a **Software Requirements Specification** (SRS) was written.

Writing a correct and informative SRS is of utmost importance to achieve an high-quality final product and ensuring the development process goes smoothly.

The SRS will cover the following points in depth:

- **Scope and purpose.**
- **Feature and functions.**
- **External interface requirements.**
- **Functional requirements.**
- **Example use cases.**
- **Non-functional requirements.**
- **Analysis models.**

# Chapter 1

## Client request

The client requests the design and implementation of a **forum creation/management framework** and a **modern responsive web forum browsing/management application**.

The client intends using the requested forum framework **to build communication platforms** for various projects, both for internal employee usage and interaction with the public.

It is imperative for the system to allow administrators to easily well-organized create **content-section hierarchies** and **user-group hierarchies**.

Administrators also need to be able to **give groups specific permissions for every section**.

Some sections will only be visible and editable to employee groups (e.g. internal discussion), some sections will be visible but not editable by the public (e.g. announcements), and others will need to be completely open to the public (e.g. technical support).

Being able to **keep track of user-created content** is also very important for the client.

Initially, tracking the date and the author of the content will be enough, but the system has to be designed in such a way that inserting additional creation information (e.g. browser/operating system used to post) will be trivial.

In the future, additional content types (e.g. videos, attachments) may be added to the system and their creation will have to be tracked as well.

Users and moderators will also need to be able to track user content through a **real-time notification system** directly from the web application interface.

This data needs to be independent from the contents, in order to easily allow administrators and project managers to gather statistical data on forum usage.

The web application has to be extremely simple but flexible as well. Administrators need be able to perform all functions described above through a responsive admin panel.

Content consumers and creators should be able to view and create content from the same responsive interface.

Moderators and administrators should be able to edit and delete posts through the same interface as well. User interface controls will be shown/hidden depending on the users permissions.



# Chapter 2

## Software Requirements Specification

### 1. Introduction

#### 1.1 Software engineering

**Software engineering** is the study and an application of engineering to the design, development, and maintenance of software.

The Bureau of Labor Statistics' definition is Research, design, develop, and test operating systems-level software, compilers, and network distribution software for medical, industrial, military, communications, aerospace, business, scientific, and general computing applications.

Typical formal definitions of software engineering are:

- The systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software.
- The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.
- An engineering discipline that is concerned with all aspects of software production.
- The establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines.

##### 1.1.1 Background

The term **software engineering** goes back to the '60s, when more complex programs started to be developed by teams composed by experts.

There was a radical transformation of software: from **artisan product** to **industrial product**.

A software engineer needs to be a good programmer, an algorithm and data structures expert with good knowledge of one or more programming languages.

He needs to know various design processes, must have the ability to convert generic requirements in well-detailed and accurate specifications, and needs to be able to communicate with the end-user in a language comprehensible to him.

Software engineering, is, however, a discipline that's still evolving. There still are no definitive standards for the software development process.

Compared to traditional engineering, which is based upon mathematics and solid methods and where well-defined standards need to be followed, software engineering is greatly dependent on personal experience rather than mathematical tools.

Here's a brief history of software engineering:

- **1950s:** Computers start to be used extensively in business applications.

- **1960s:** The first software product is marketed.

IBM announces its unbundling in June 1969.

- **1970s:** Software products are now regularly bought by normal users.

The software development industry grows rapidly despite the lack of financing.

The first software houses begin to emerge.

### 1..1.2 Differences with programming

- A programmer writes a complete program.
- A software engineer writes a software component that will be combined with components written by other software engineers to build a system.
- Programming is primarily a personal activity.
- Software engineering is essentially a team activity.
- Programming is just one aspect of software development.
- Large software systems must be developed similar to other engineering practices.

## 1..2 SRS

This **Software Requirements Specification** (SRS) chapter contains all the information needed by software engineers and project managers to design and implement the requested forum creation/management framework.

The SRS was written following the **Institute of Electrical and Electronics Engineers** (IEEE) guidelines on SRS creation.

## 1..3 Purpose

The SRS chapter is contained in the **non-technical** part of the thesis.

Its purpose is providing a **comprehensive description** of the objective and environment for the software under development.

The SRS fully describes **what the software will do** and **how it will be expected to perform**.

## 1..4 Scope

### 1..4.1 Identity

The software that will be designed and produced will be called **veeForum**.

### 1..4.2 Feature extents

The complete product will:

- Provide a framework for the **creation and the management of a forum system**.
- Allow its users to **deploy and administrate** multi-purpose forums.
- Give access to a **modern responsive web application** to setup, browse and manage the forum.

veeForum, however, will not:

- Provide infrastructure or implementation for a complete blog/website. The scope of the software is forum building.
- Implement instant private messaging - user-to-user chat is beyond the scope of the project.

### 1..4.3 Benefits and objectives

Deploying veeForum will give its users a number of important benefits and will fulfill specific objectives.

- Companies and individuals making use of veeForum will have access to an **easy-to-install** and **easy-to-use** forum creation and management platform.
- Users and moderators of the deployed forums will be able to **easily create, track and manage** content and other forum users.
- Forum administrators will be given **total control** of the forum structure, users and permissions through an **easy-to-use** responsive administration panel.

## 2. General description

### 2.1 Product perspective and functions

The product shares many basic aspects and features with existing forum frameworks such as **phpBB** or **vBulletin**: flat/threaded discussion support, nested sections, user attachments, etc.

veeForum improves on existing forum frameworks in the following ways:

- Provides a responsive web interface without postbacks.
- Allows users and moderators to subscribe and unsubscribe not only to posts, but to users and sections as well.
- Has a powerful **real-time** Facebook-like notification system that notifies users when tracked content has been added or edited.
- Gives administrator the possibility to **design and manage complex permission hierarchies** for user groups and single users.

### 2.2 User characteristics

veeForum needs to target both users that **only consume the content offered by deployed forums**, users that **actively create and manage content in deployed forums**, and users that **build and deploy forum instances**.

User-friendliness is essential for every target, but all the required functionality is effectively exposed to different user groups.

It is therefore required to have clear interfaces that do not negatively affect the user experience by being either too complex or too simple (all features need to be exposed).

## 3. Specific requirements

### 3.1 External interface requirements

**External interface requirements** identify and document the interfaces to other systems and external entities within the project scope.

#### 3.1.1 User interfaces

The product will provide both a desktop and a mobile user web interface.

- **Web interface:** it is required to provide a modern responsive web interface, compatible and tested with the most popular browsers (Internet Explorer 10+, Google Chrome, Mozilla Firefox). The web interface will give forum access to users and moderators, and administrator access to forum management staff.
- **Mobile interface:** is is required to provide a modern mobile application for the major platforms (Android, iOS, Windows Phone). The mobile application will allow browsing and content management of forums created with the product.

### 3..1.2 Software interfaces

The **open-source policy** of veeForum will allow framework users to expand or improve existing functionality and to interact with other existing technologies.

Accessing and modifying forum data (assuming permission requirements are satisfied by the user) will be possible through **RESTful** requests, returning and accepting **JSON** (Javascript Object Notation).

## 3.2 Functional requirements

In software engineering, a **functional requirement** defines a function of a system and its components.

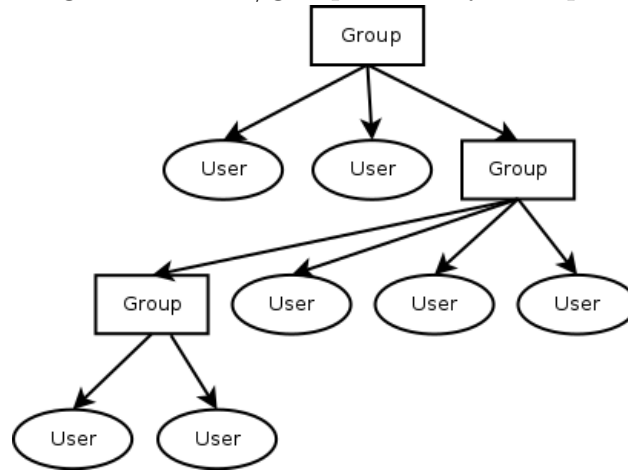
Functional requirements may be **calculations, technical details, data manipulation and processing** and other specific functionality that define what a system is supposed to accomplish.

Behavioral requirements describing all the cases where the system uses the functional requirements are captured in **use cases**.

### 3..2.1 User/group management

- **Users:** users will be managed by the system. Users can register (or be manually added by an administrator). Registration can be configured to require a confirmation email or not.
- **Groups:** every user will be part of at least one group at all times. Groups are part of an hierarchy: they can inherit from each other. Groups can have permissions specific to sections and system-wide permissions.

Figure 2.1: User/group hierarchy example.



### 3..2.2 Content hierarchy

- **Posts:** posts will be the base of the content hierarchy. They will contain HTML-enabled text and any number of attachments. Posts can be edited and deleted by the original owner.
- **Threads:** threads are groups of posts. Users with the correct permissions can create a thread in a specific section and have other users add posts or subscribe to it. Threads can be edited and deleted by the original owner.
- **Sections:** sections are content containers intended to group threads related to the same subject. Forum administrators and moderators can create sections and give users permissions to view or edit them.
- **Attachments:** users with the correct permissions can upload files and attach any number of them to one of their posts.

### 3..2.3 Content tracking system

- **Creation data:** user-created content (posts, threads, attachments, etc) will have some data specific to its creation can be extended by forum administrators. Basic predefined data will consist of creation date and time. It will be possible to run statistical queries on content creation data.
- **Subscriptions:** users and moderators will be able to subscribe to specific sections, threads or user to track their contents. They will receive real-time notifications upon addition/editing of tracked content.

Figure 2.2: Content hierarchy example.

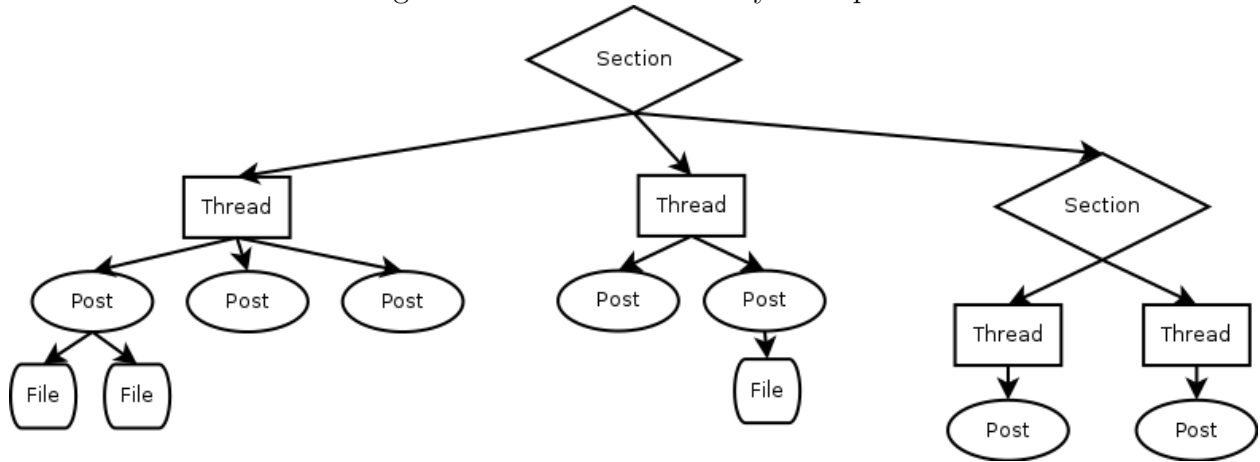
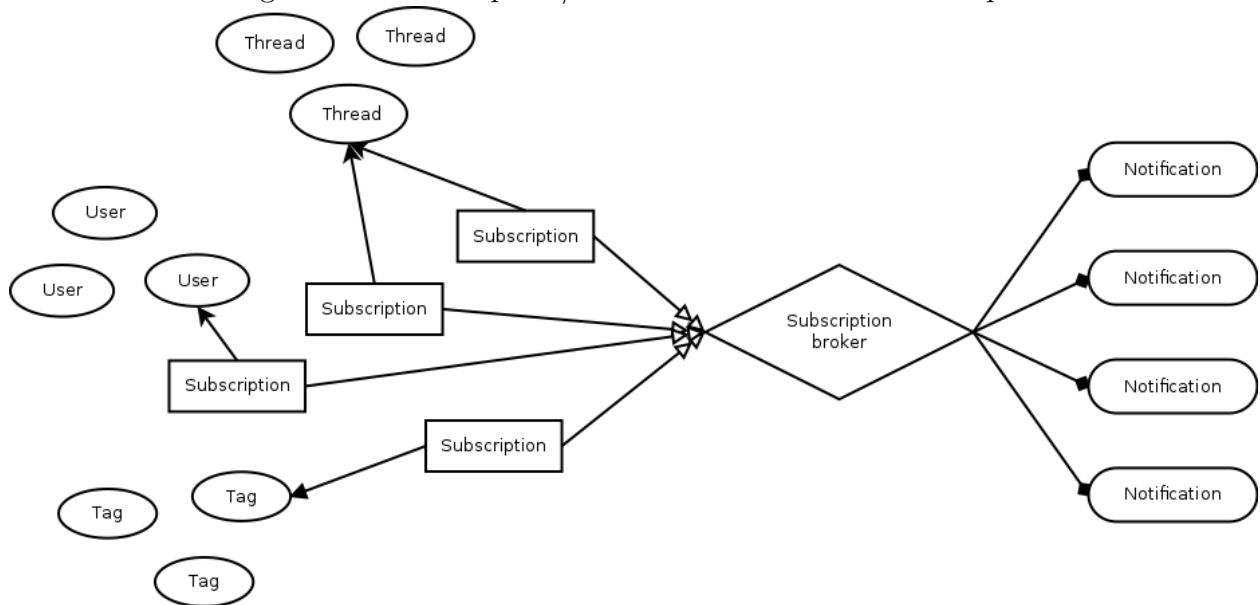


Figure 2.3: Subscription/notification architecture example.



### 3.3 Example use cases

In software and systems engineering, a **use case** is a list of steps, typically defining interactions between one or more actors and a system, to achieve a goal.

#### 3.3.1 Mobile game forum

A company developed a popular mobile game, with a wide audience. The company uses the **veeForum framework** to give users a place to discuss game strategy, give feedback on the quality of their product and receive technical support.

#### **3..3.1.1 Actors**

- Game developers.
- Game players.
- Forum management team.
- Technical support team.
- Feedback (PR) team.

#### **3..3.1.2 Pre-conditions**

- Release of a popular product with a wide audience.
- Game users need to register on the forum.

#### **3..3.1.3 Flow of events**

- Installation and configuration of a veeForum-enabled forum system by the forum management team.
- Creation of the sections and permission hierarchies by the forum management team and the developers.
- Registration and content creation by the game developers and game players.

#### **3..3.1.4 Post-conditions**

- Game players will be able to share their strategies and thoughts on the product.
- The technical support team will find all technical issues grouped in a convenient way and will be able to track individual issues. Technical support members will be able to communicate with each other in a private section.
- The feedback team will be able to track user suggestions and forward potential product improvements to the developer team.



Figure 2.4: Basic forum usage use case diagram.

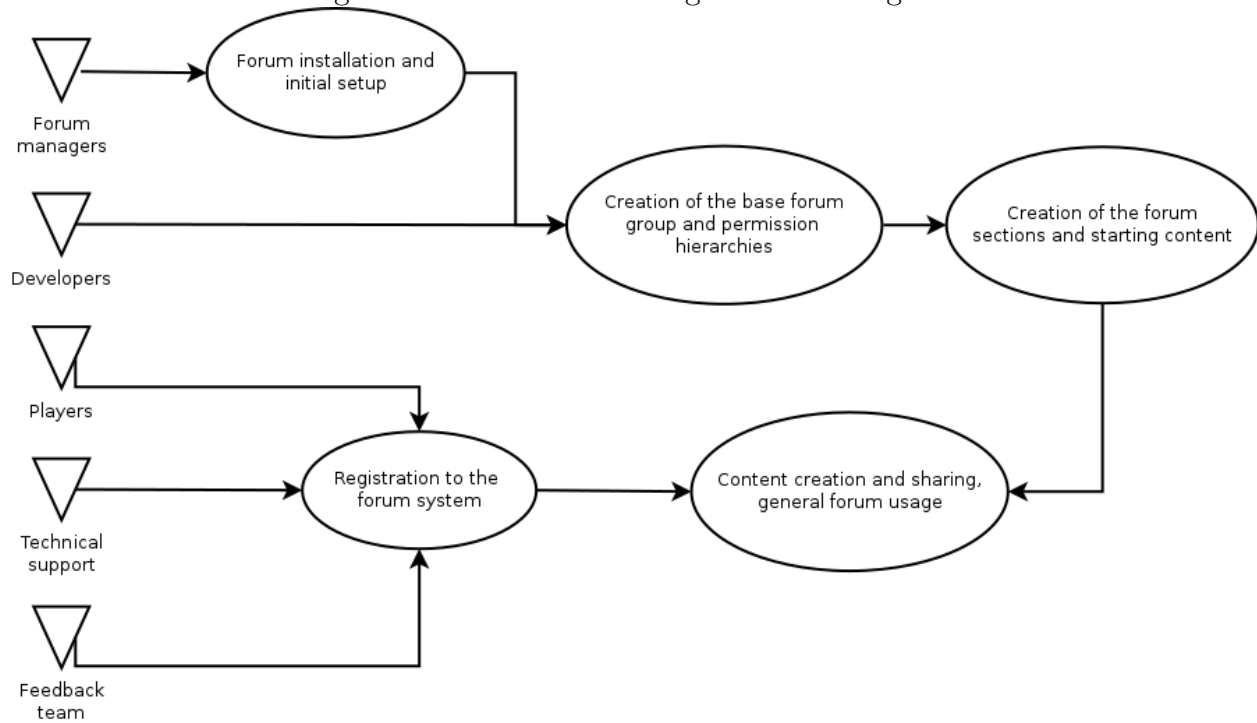
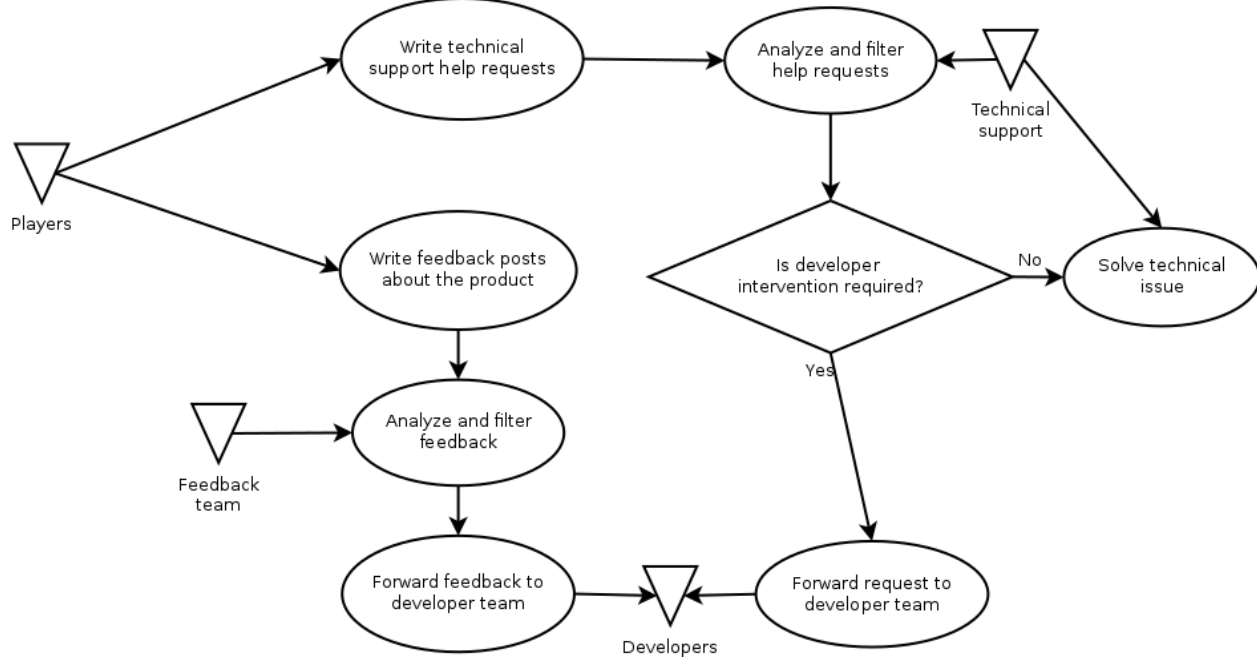


Figure 2.5: Technical support and feedback use case diagram.



### **3..3.2 Local city GNU/Linux usergroup forum**

Some GNU/Linux users from the same city decide to start a local usergroup to discuss the GNU/Linux ecosystem and make new friends. In spirit with the open-source nature of the system, collaboration is extremely important. They require to easily assign specific permissions to users and groups to allow the forum to grow and be well-organized.

#### **3..3.2.1 Actors**

- Usergroup creators.
- Usergroup members.
- External visitors.

#### **3..3.2.2 Pre-conditions**

- Interest in a local GNU/Linux usergroup.
- Availability of people willing to collaborate.

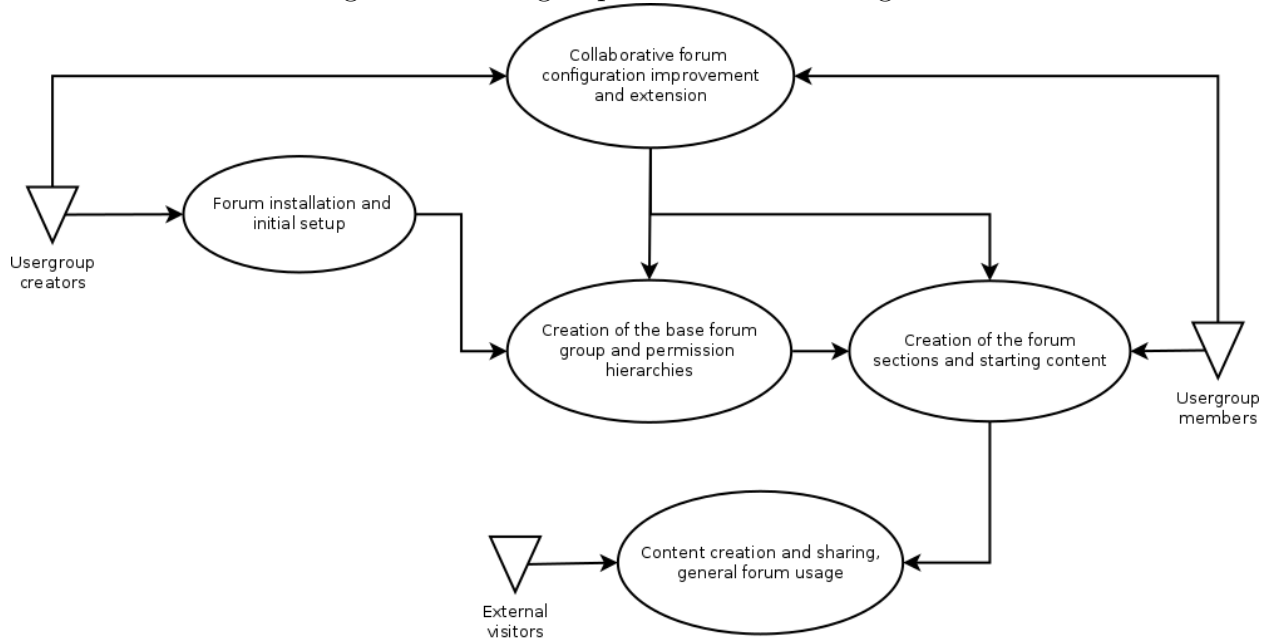
#### **3..3.2.3 Flow of events**

- Installation and configuration of a veeForum-enabled forum system by the usergroup creators.
- Creation of the initial sections and permission hierarchies by the usergroup creators.
- Registration of usergroup members and external visitors.
- The usergroup creators give other usergroup members permissions to create and manage sections and users, starting a chain of collaborative forum content development.
- Usergroup members and external visitors contribute and make use of the content.

#### **3..3.2.4 Post-conditions**

- Local city usergroup members will be able to get to know and speak to each other.
- Usergroups members willing to contribute will be able to easily manage sections and write posts/articles.
- External visitors will be able to make use of the public content.

Figure 2.6: Usergroup forum use case diagram.



### 3.4 Non-functional requirements

Functional requirements are supported by **non-functional requirements** (also known as quality requirements), which impose constraints on the design or implementation (such as performance requirements, security, or reliability).

#### 3.4.1 Performance

The system will be designed from the ground-up with emphasis on performance. As the forum may have huge amounts of contents and concurrent usage after its deployment, optimizing is a must.

When possible, functions will be implemented **directly in the database**, for maximum performance.

Web backend functions will also be carefully **optimized both for memory and speed**.

#### 3.4.2 Reliability

The system will have to be reliable and keep working in case of errors.

Database queries and functions will be executed in **safe wrappers** that catch and handle errors carefully.

### 3..4.3 Security

veeForum needs to guarantee privacy and security for users and administrator of the system.

Well-tested and well-received **security idioms** and **encryption algorithms** will have to be used throughout the implementation of the whole system.

### 3..4.4 Maintainability and portability

Being an open-source project, **maintainability**, **extensibility** and **portability** are key.

The code layer will be carefully designed and organized to allow easy maintenance, bug-fixing and feature addition.

To ensure maximum portability, the product will be designed to work on the most popular **GNU/Linux** distributions and will be thoroughly tested on different platforms.

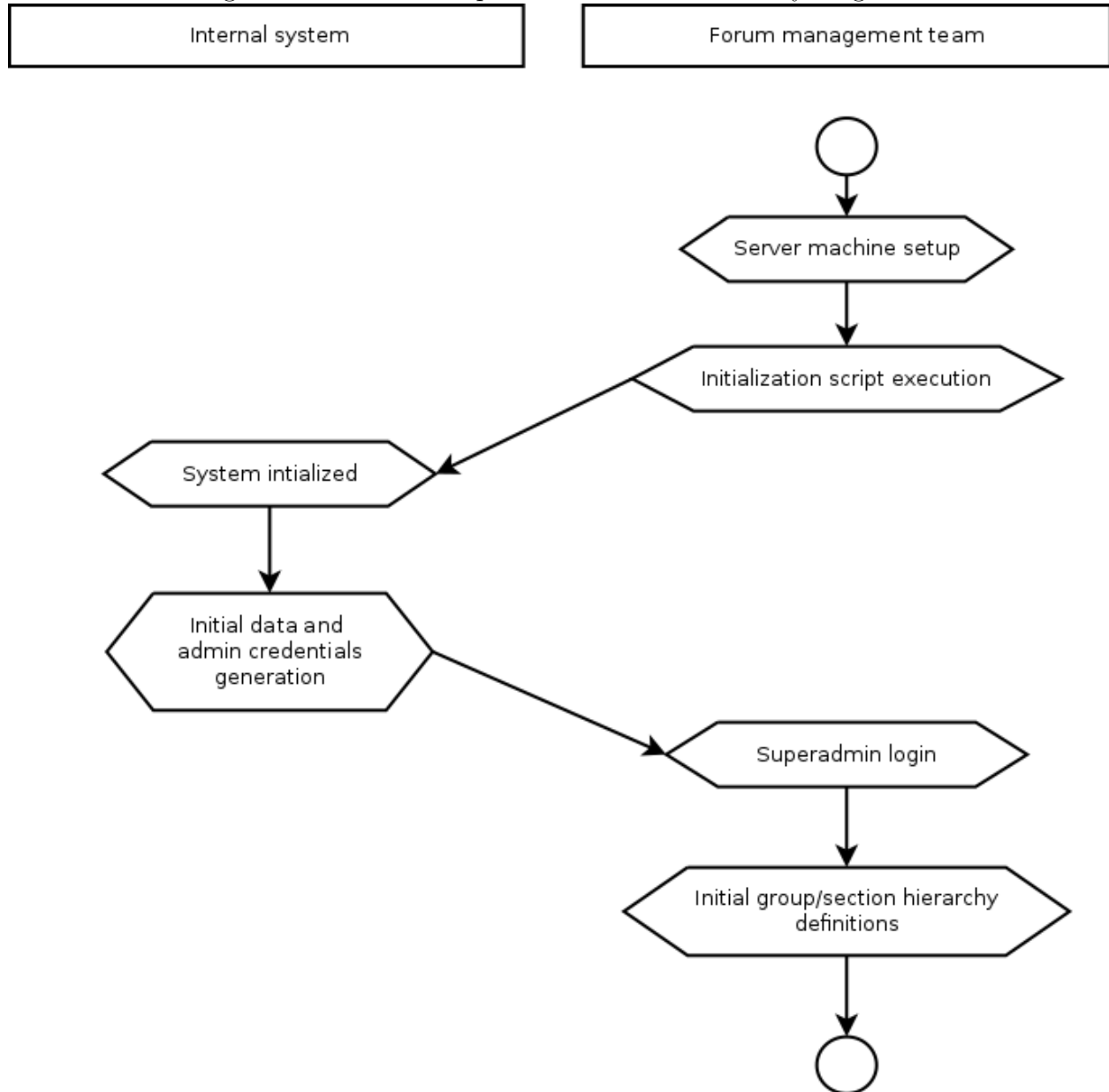
## 4. Analysis models

### 4.1 Activity Diagrams

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. In the Unified Modeling Language, activity diagrams are intended to model both computational and organisational processes (i.e. workflows). Activity diagrams show the overall flow of control.

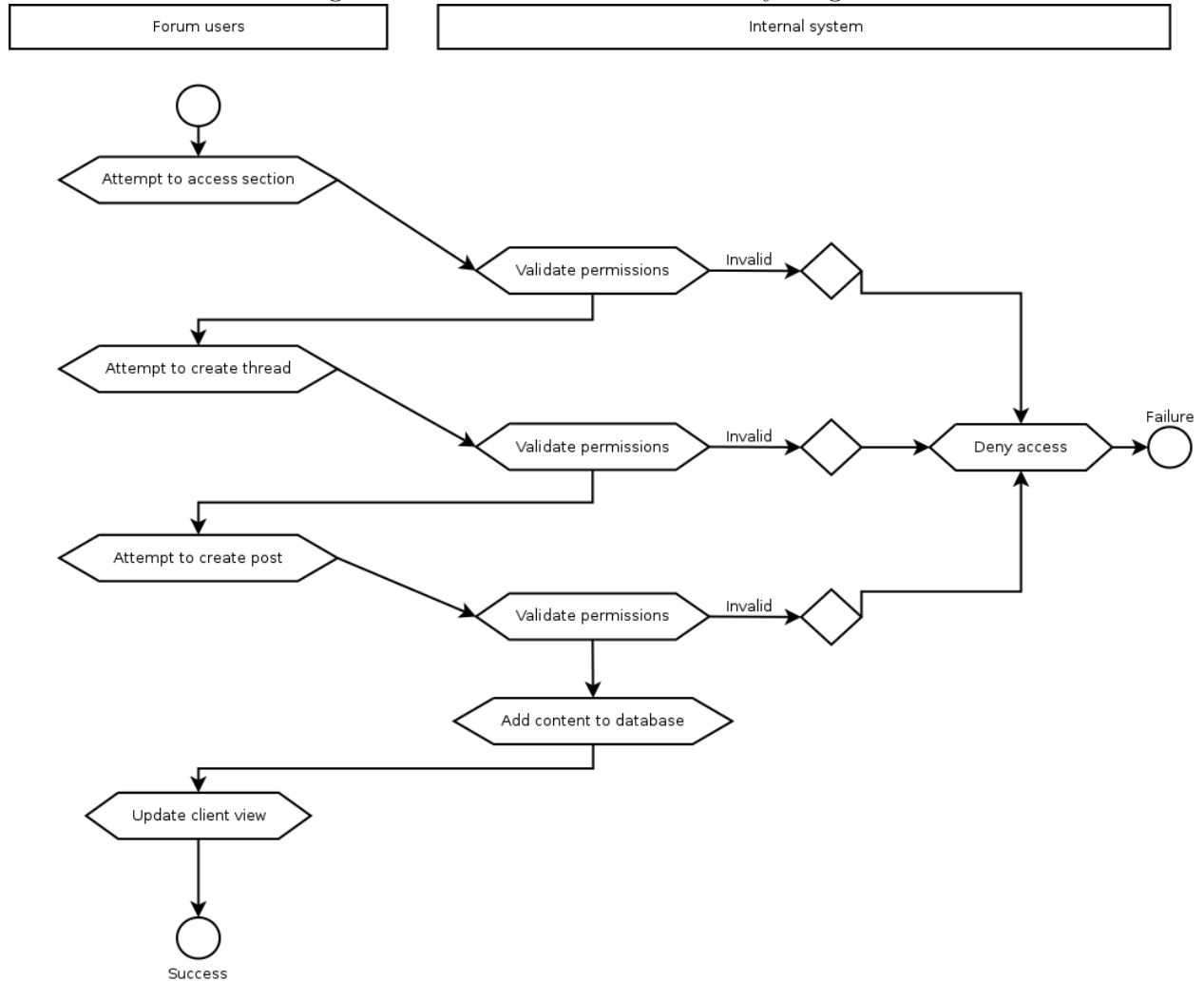
The following diagram shows the steps that the **forum management team** must take in order to setup and initialize a veeForum-enabled forum.

Figure 2.7: Forum setup and initialization activity diagram.



The following diagram shows the steps that the **forum users** must take in order to add content to the forum system.

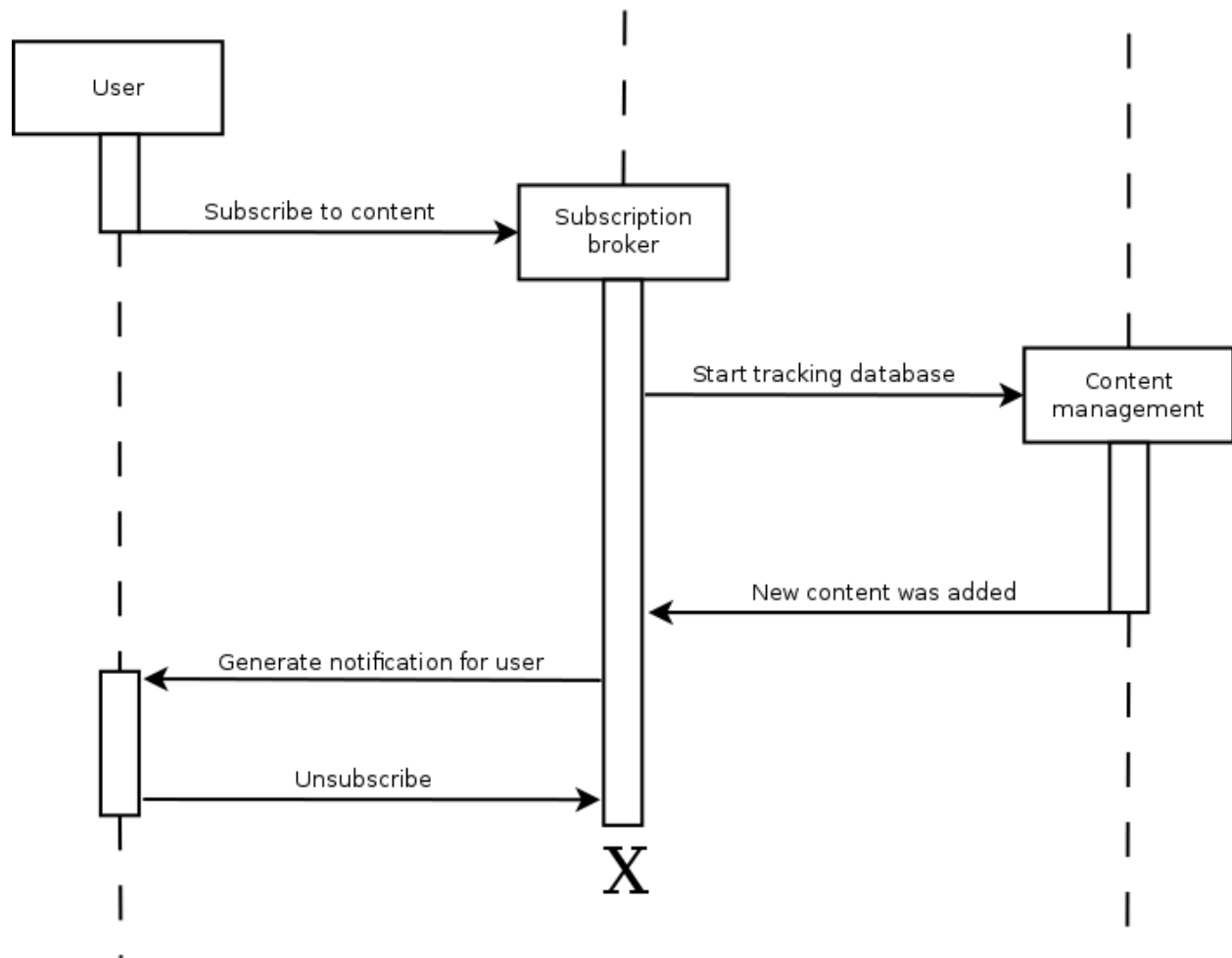
Figure 2.8: Content creation activity diagram.



## 4.2 Sequence Diagrams

The following diagram shows the interaction between **forum users**, the **subscription broker** and the **content management** system in order to manage subscriptions and generate notifications.

Figure 2.9: Subscription/notification system sequence diagram.



# Part II

## Technical analysis



The following part of the thesis will cover all implementation choices and details for veeForum in depth.

Firstly, the **development environment and tools** and **chosen technologies** will be described and motivated.

Afterwards, the technical details, including code examples and APIs, will be described for the two modules of the application: the **database** and the **web application**.

Every **table** of the database will be analyzed in detail, directly showing commented **DDL** code. The database also contains important **stored procedures** and **triggers** that are core part of the system's logic and that need to be explained in depth - the related **DML** code will be shown and commented.

The web application itself is divided in multiple modules:

- A **database interface backend module**, that interfaces with the database and wraps its tables and stored procedures.
- A **HTML5 generation module**, that greatly simplifies the creation of dynamic forum web pages by wrapping HTML5 controls in **object-oriented wrappers** that can be easily bound to callbacks and database events.
- A **modern responsive AJAX frontend** that allows users and interact with the backend module from multiple device, limiting postbacks and page refreshes.

# Chapter 3

## Development process

### 1. Environment and tools

All modules of veeForum have been developed on **Arch Linux x64**, a lightweight GNU/Linux distribution.

Arch is installed as a minimal base system, configured by the user upon which their own ideal environment is assembled by installing only what is required or desired for their unique purposes. GUI configuration utilities are not officially provided, and most system configuration is performed from the shell and a text editor. Based on a rolling-release model, Arch strives to stay bleeding edge, and typically offers the latest stable versions of most software.

No particular integrated development environments (IDEs) were used during the development - a modern graphical text editor, **Sublime Text 3**, was used instead.

### 2. Docker

Docker is an open-source project that **automates the deployment of applications** inside software containers, by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux.

Docker uses resource isolation features of the Linux kernel such as **cgroups** and **kernel namespaces** to allow independent containers to run within a single Linux instance.

This technology has been used since the beginning of the development process to **separate veeForum data and packages** from the host system and to dramatically increase **portability** and **ease of testing**.

Docker is also used for the installation of the product on target systems - with a single command it is possible to **retrieve all required dependencies**, correctly **configure the system** and **automatically install veeForum**.

### 3. Version control system

Version control systems (VCSs) allow the **management of changes** to documents, computer programs, large web sites, and other collections of information.

Nowadays, a version control system is **essential** for the development of any project. Being able to track changes, develop features in separate **branches**, have multiple programmers work on the same code base without conflicts and much more is extremely important for projects of any scope and size.

The chosen VCS is **Git**, a distributed revision control system with an emphasis on **speed**, **data integrity**, and support for **distributed, non-linear workflows**.

Git is widely appreciated in the private and open-source programming communities - it was initially designed and developed by **Linus Torvalds** for Linux kernel development in 2005, and has since become the most widely adopted version control system for software development.

The veeForum project is **open-source** and **appreciates feedback and contributions**. It is hosted on **GitHub**, a web-based Git repository hosting service, which offers all of the distributed revision control and source code management (SCM) functionality of Git, while adding **additional features** that make collaboration and public contributions easy and accessible.

### 4. LAMP stack

The server and web application run on a **LAMP stack**, on a GNU/Linux machine.

A LAMP stack is composed by the following technologies:

- **L**: GNU/Linux machine.
- **A**: Apache HTTP server.

The Apache HTTP server is the world's most widely used web server software.

Apache has been under open-source development for about 20 years - it supports all modern server-side technologies and programming languages, and also is **extremely reliable** and **secure**.

- **M**: Stands for MySQL server, but **MariaDB**, a modern drop-in replacement for MySQL is used as the DBMS.

MariaDB is fully compliant with the MySQL standard and language, but it is more performant and has additional features. It is the default DBMS in the Arch Linux distribution.

- **P**: PHP5, the server backend language.

HTML5, PHP5 and JavaScript conformant to the 5.1 ECMAScript specification (along with the JQuery library) are used for the development of the web application.

The **AJAX** (Asynchronous JavaScript and XML) paradigm will be used to ensure that the application feels responsive and that user interaction is immediately reflected on the web application.

## 5. Thesis

The current document was written using  $\text{\LaTeX}$ , an high-quality typesetting system; it includes features designed for the production of **technical and scientific documentation**.

$\text{\LaTeX}$  was chosen for the current document because of the visually pleasant typography, its extensibility features and its abilities to include and highlight source code.

### 5.1 LatexPP

A small **C++14**  $\text{\LaTeX}$  preprocessor named **LatexPP** was developed for the composition of this thesis.

LatexPP allows to use an intuitive syntax that avoids markup repetition for code highlighting and macros.

Preprocessing and compiling a  $\text{\LaTeX}$  document using LatexPP is simple and can be automated using a simple **bash** script.

---

```
1  #!/bin/bash
2
3  latexpp ./thesis.lpp > ./thesis.tex
4  pdflatex -shell-escape ./thesis.tex && chromium ./thesis.pdf
```

---

LatexPP is available as an open-source project on GitHub:

<https://github.com/SuperV1234/Experiments/Random>

# Chapter 4

## Project structure

The project folder and file structure is organized as such:

- **./doc/**

Folder containing the documentation of the project.

- **./latex/**

LatexPP and L<sup>A</sup>T<sub>E</sub>X source and output files.

- **./sql/**

Folder containing the SQL DDL scripts.

- **./scripts/**

Contains all the parts that make up the complete SQL initialization script.

- **./mkScript.sh**

Builds the complete SQL initialization scripts from the files in ./scripts/.

- **./script.sql**

Complete SQL initialization scripts that sets up a database suitable veeForum.

- **./exe/**

Folder containing executable scripts to setup the system.

- **./docker/**

Docker-related scripts.

- \* **./start.sh**

Starts a Docker instance containing veeForum.

- \* **./cleanup.sh**

Cleans any running veeForum Docker instance.

- \* **./shell.sh**  
Starts a Docker instance containing veeForum, controlling an instance of bash inside it.
- \* **./httpdLog.sh**  
Prints the Apache error log of the current running veeForum Docker instance.
- **./www/**  
Folder containing web application data.
  - **./css/**  
CSS3 stylesheets.
  - **./js/**  
ECMAScript 5 script files.
  - **./json/**  
Non-relational data storage files, in JSON format.
  - **./php/**  
PHP backend code.
    - \* **./lib/**  
Backend to database interface library and HTML5 generation library.
    - \* **./core/**  
PHP frontend files that generate the responsive HTML5 web application user interface.

# Database design

## 1. Diagrams

Figure 5.1: Complete DB Entity-Relationship diagram.

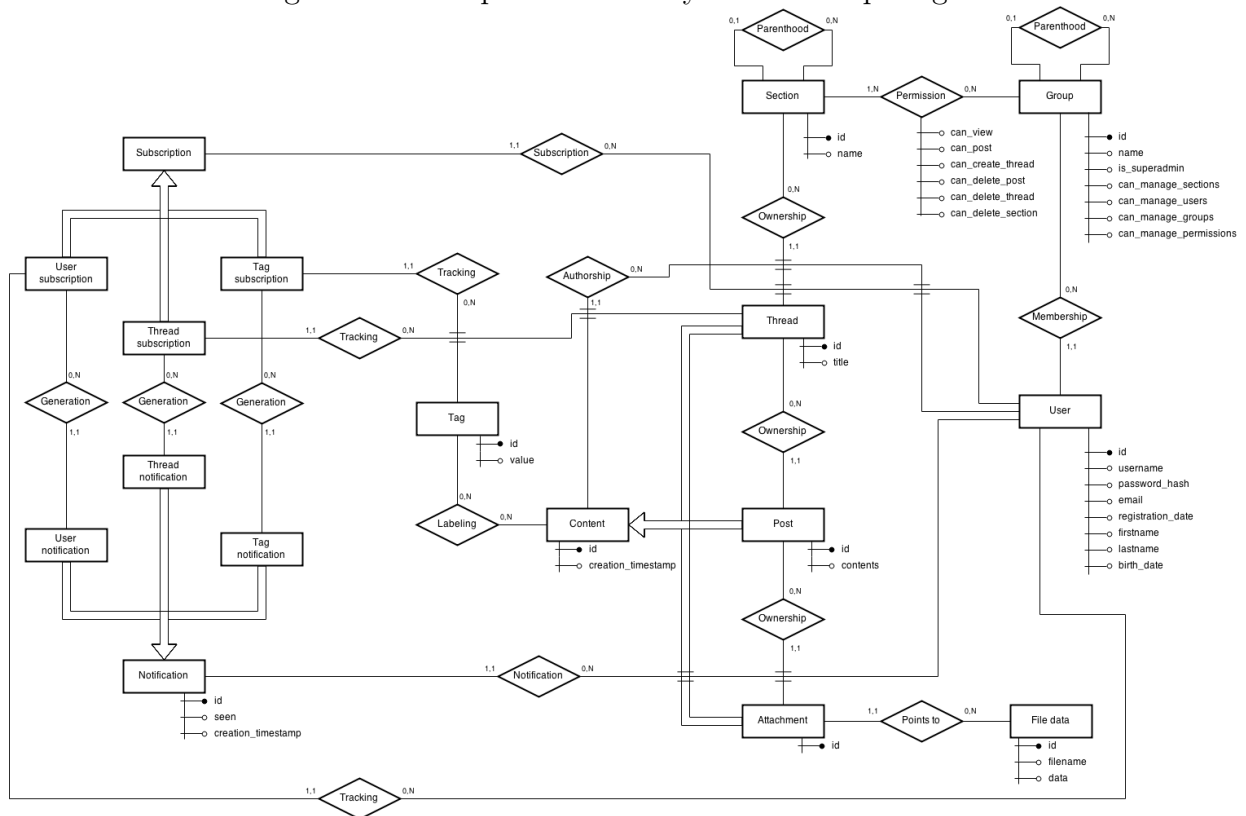


Figure 5.2: ER diagram zoom: content hierarchy

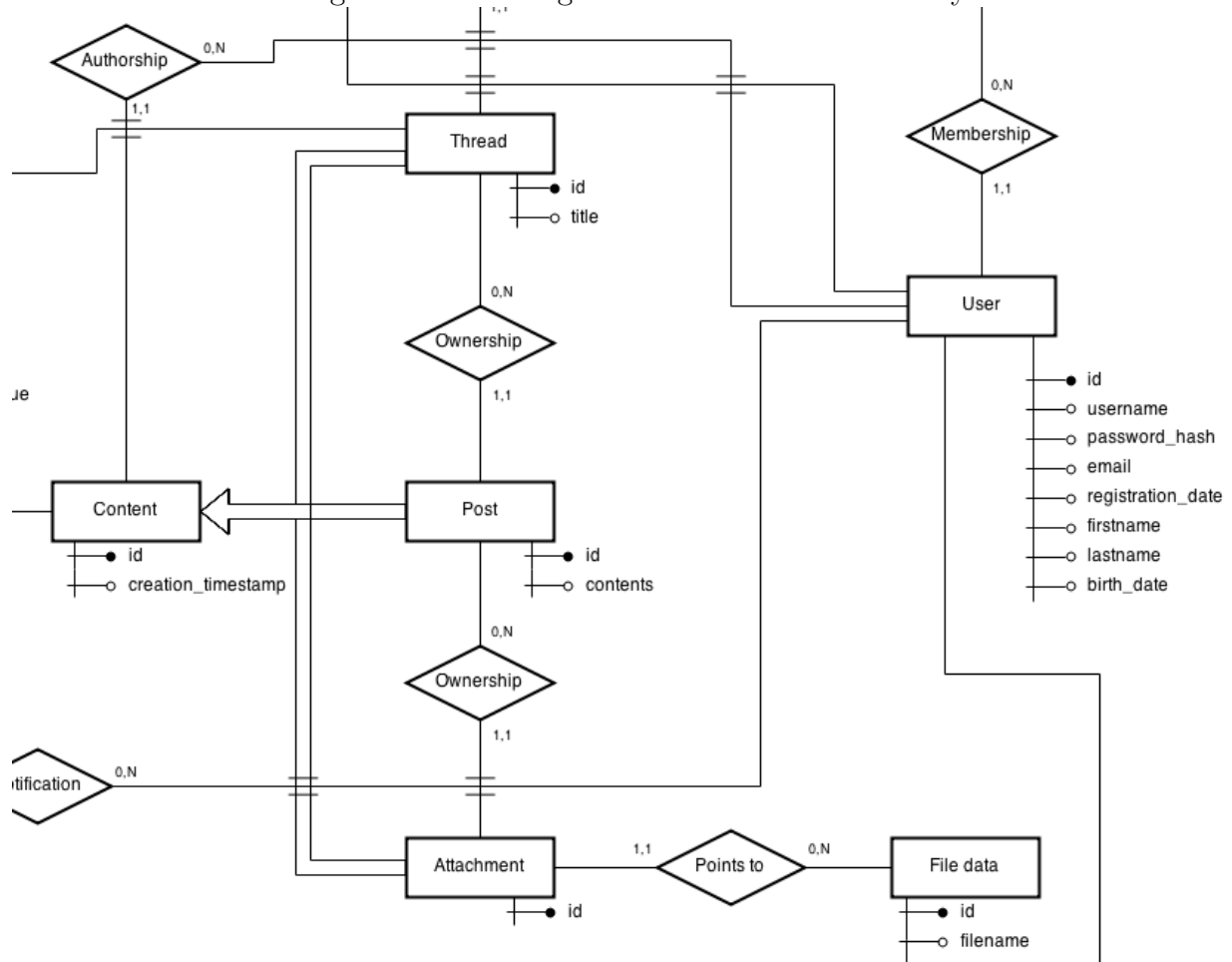




Figure 5.3: ER diagram zoom: section hierarchy

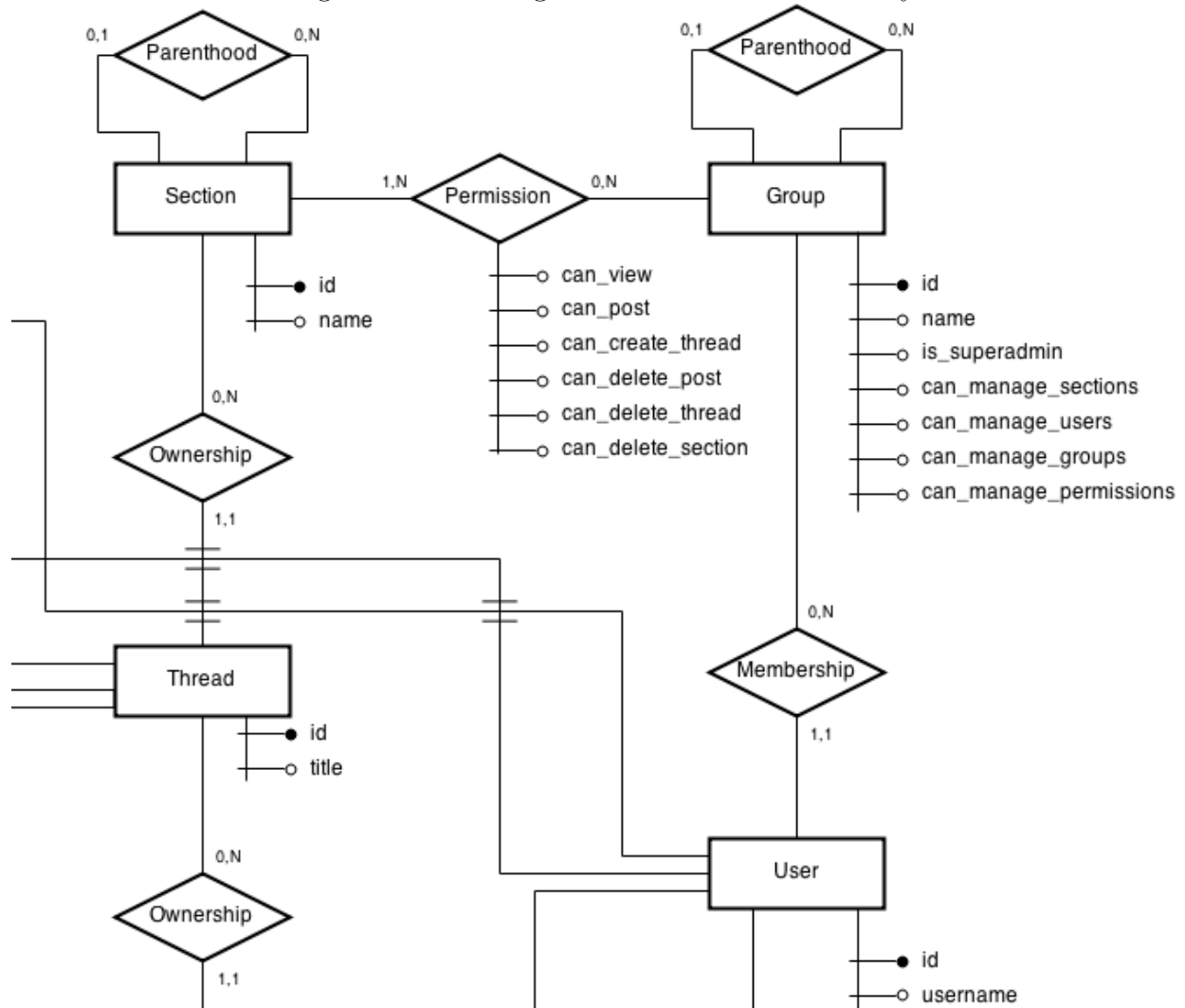


Figure 5.4: ER diagram zoom: content subscription/notification system

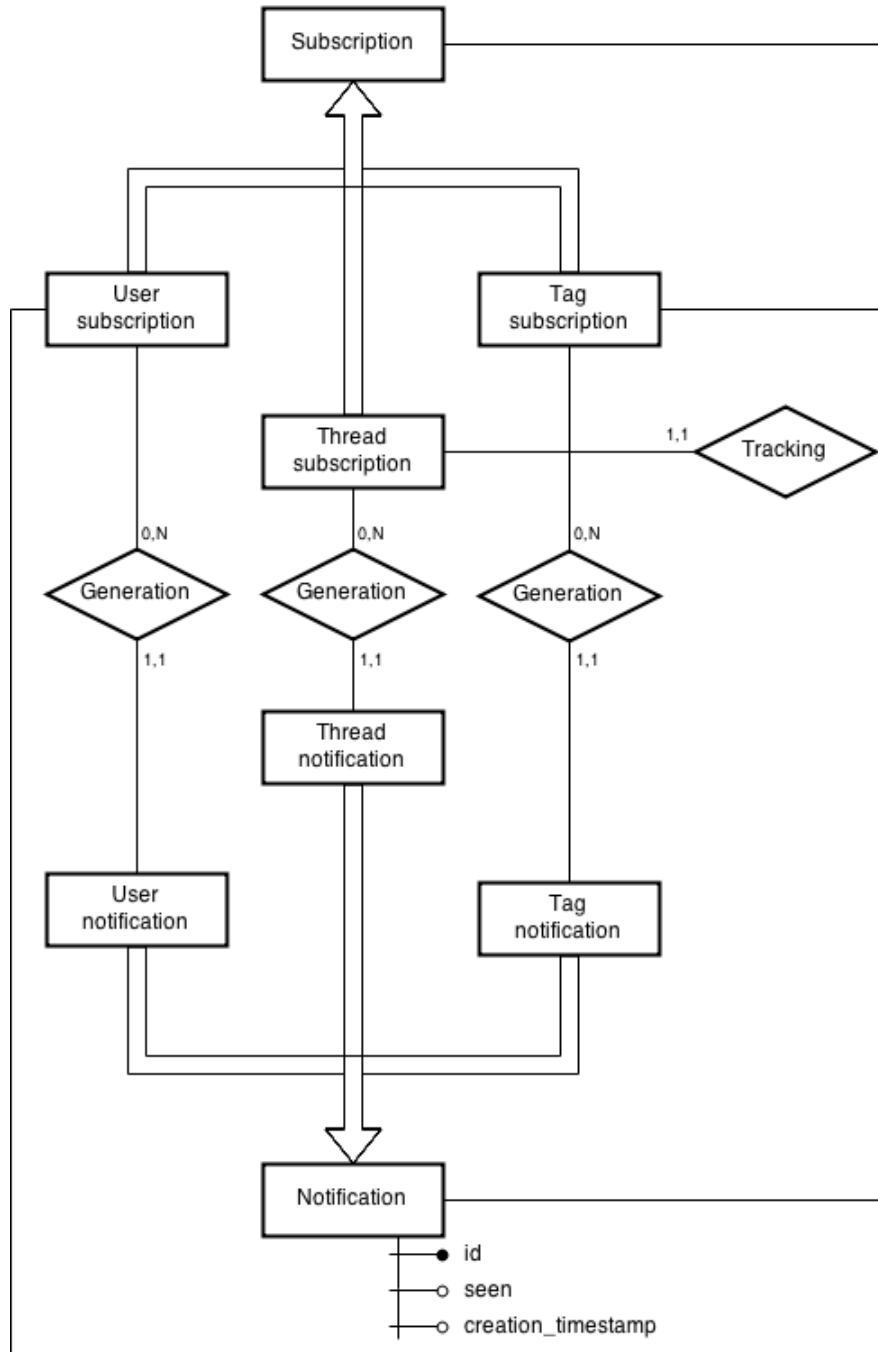
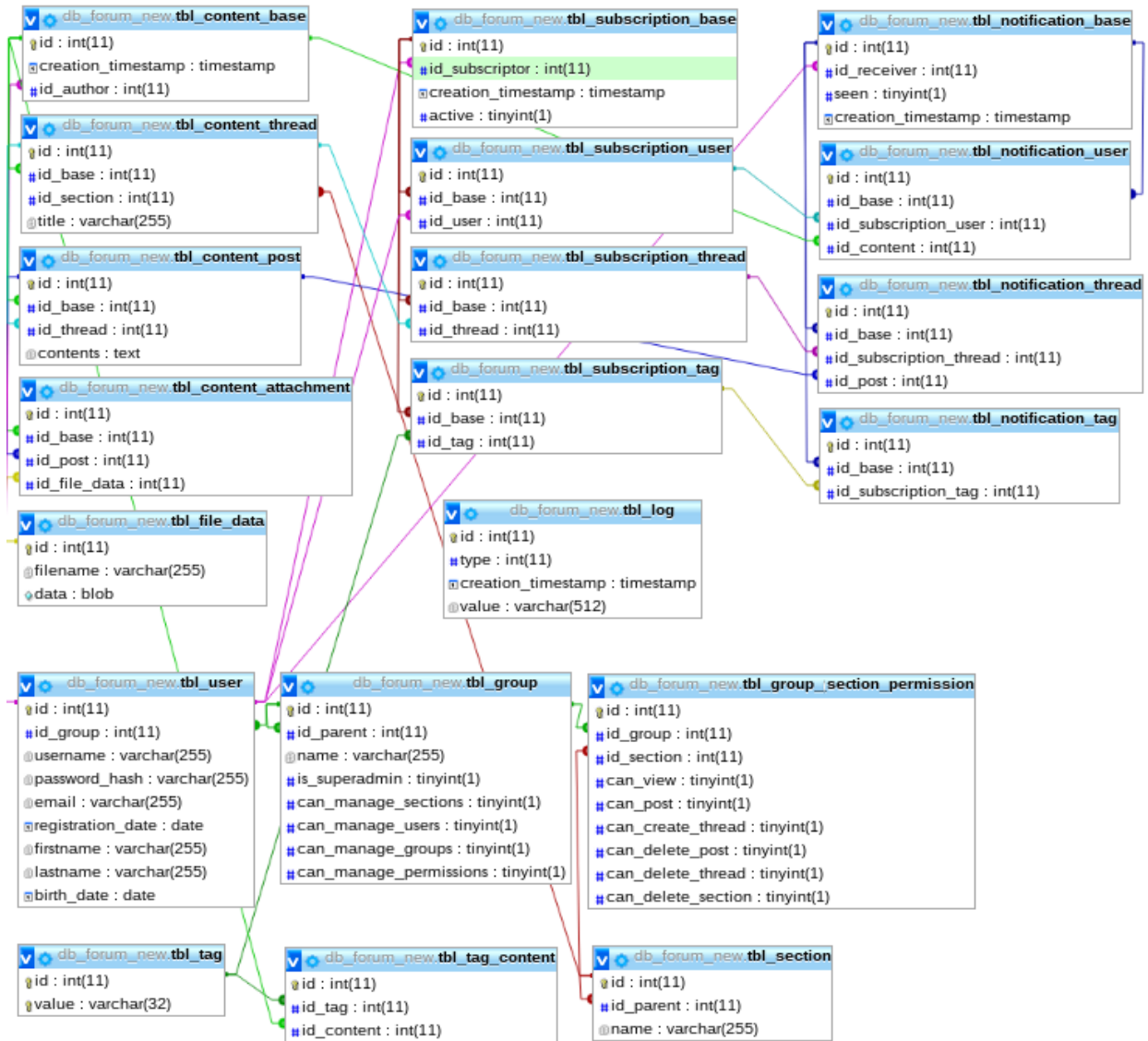


Figure 5.5: Complete DB logic diagram.



# Chapter 6

## SQL

### 1. Database setup

veeForum is supposed to be installed on a clean instance of MySQL server. The following script correctly initializes the required database and cleans any previous version of veeForum.

#### 1.1 db

##### 1.1.1 Code

```
1 #####
2 # Copyright (c) 2013-2015 Vittorio Romeo
3 # License: Academic Free License ("AFL") v. 3.0
4 # AFL License page: http://opensource.org/licenses/AFL-3.0
5 #####
6 # http://vittorioromeo.info
7 # vittorio.romeo@outlook.com
8 #####
9
10 #####
11 # veeForum forum framework initialization and creation script
12 #####
13
14 #####
15 # This script is meant to be run once to create and initialize
16 # from scratch the whole MySQL veeForum backend.
17 # Therefore, we drop the database if exists and re-create it.
18 drop database if exists db_forum_new$
19 create database db_forum_new$
20 use db_forum_new$
21 #####
```

### **1..1.2 Explanation**

This script is meant to be run once to create and initialize from scratch the whole MySQL veeForum backend. Therefore, we drop the database if exists and re-create it.

## 2. Tables

A big amount of tables is required to make veeForum satisfy all requirements. Every table in the project is documented in the following section - the full **DDL** commented code and an explanation is provided for every table.

### 2..1 log

#### 2..1.1 Code

```
1 #####
2 # TABLE
3 # * This table deals with log messages.
4 #####
5 create table tbl_log
6 (
7     # Primary key
8     id int auto_increment primary key,
9
10    # Log type
11    type int not null default 0,
12
13    # Entry timestamp
14    creation_timestamp timestamp not null default 0,
15
16    # Name
17    value varchar(512) not null
18 )$
19 #####
```

#### 2..1.2 Explanation

The **log** table is a simple non-relational list of log messages that can be used for debugging and security purposes.

## 2..2 tag

### 2..2.1 Code

```
1 #####
2 # TABLE
3 # * This table deals with tag archetypes.
4 #####
5 create table tbl_tag
6 (
7     # Primary key
8     id int auto_increment primary key,
9
10    # Name
11    value varchar(32) not null unique
12 )$
13 #####
```

### 2..2.2 Explanation

The **tag** table is a simple non-relational list of unique tags that can be attached to user-created content.

## 2..3 group

### 2..3.1 Code

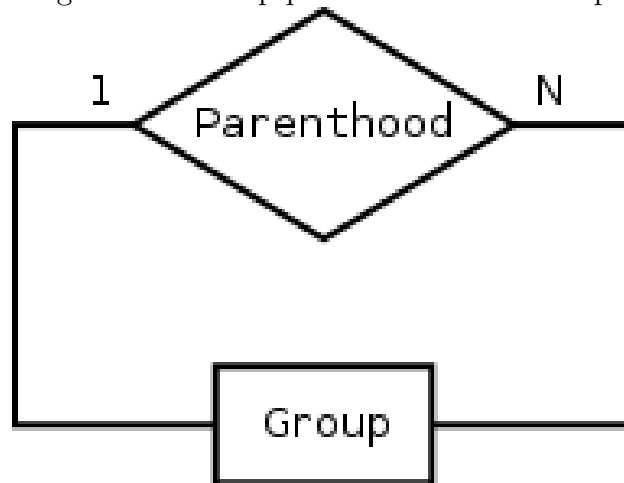
```
1 #####
2 # TABLE
3 # * This table deals with groups.
4 # * Every group row also contains its forum-wide privileges.
5 #####
6 create table tbl_group
7 (
8     # Primary key
9     id int auto_increment primary key,
10
11     # Parent group (null is allowed)
12     id_parent int,
13
14     # Name,
15     name varchar(255) not null,
16
17     # Privs
18     is_superadmin boolean not null default false,
19     can_manage_sections boolean not null default false,
20     can_manage_users boolean not null default false,
21     can_manage_groups boolean not null default false,
22     can_manage_permissions boolean not null default false,
23
24     foreign key (id_parent)
25         references tbl_group(id)
26         on update cascade
27         on delete cascade
28 )$
29 #####
```

### 2..3.2 Explanation

The **group** table defines the groups users can belong to. Every row defines a different group and assigns forum-wide permissions to them. Groups can inherit from each other thanks to the `id_parent` field, which is the id of the parent group and can be `NULL`.



Figure 6.1: Group parenthood relationship.



## 2..4 user

### 2.4.1 Code

```

1  #####
2  # TABLE
3  # * This table deals with users.
4  #####
5  create table tbl_user
6  (
7      # Primary key
8      id int auto_increment primary key,
9
10     # Group of the user
11     id_group int not null,
12
13     # Credentials
14     username varchar(255) not null,
15     password_hash varchar(255) not null,
16     email varchar(255) not null,
17     registration_date date not null,
18
19     # Personal info
20     firstname varchar(255),
21     lastname varchar(255),
22     birth_date date,
23
24     foreign key (id_group)
25         references tbl_group(id)
26         on update cascade

```

```

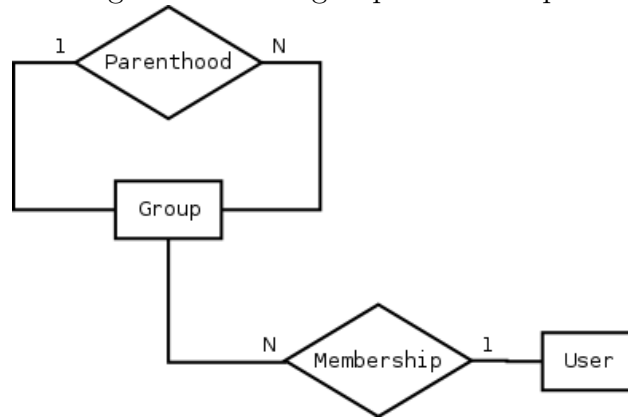
27         on delete cascade
28     )$
29     #####

```

### 2..4.2 Explanation

The **group** table contains the users registered to the forum system. Every user **needs** to belong to a group, whose id is stored in **id\_group**. Every row stores user credentials data and personal info.

Figure 6.2: User-group relationship.



## 2..5 section

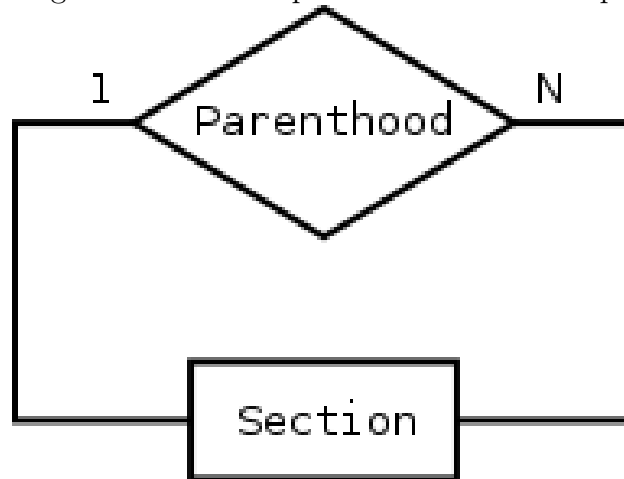
### 2..5.1 Code

```
1 #####
2 # TABLE
3 # * This table deals with sections.
4 #####
5 create table tbl_section
6 (
7     # Primary key
8     id int auto_increment primary key,
9
10    # Parent section (null is allowed)
11    id_parent int,
12
13    # Data
14    name varchar(255) not null,
15
16    foreign key (id_parent)
17        references tbl_section(id)
18        on update no action
19        on delete no action
20 )$
21 #####
```

### 2..5.2 Explanation

The **section** table contains all forum sections, defining the base hierarchy for content. Sections have a name and can inherit from each other thanks to the `id_parent` field, which is the id of the parent section and can be **NULL**.

Figure 6.3: Section parenthood relationship.



## 2..6 fileData

### 2..6.1 Code

```

1  #####
2  # TABLE
3  # * This table deals with binary file data.
4  # * Used for attachments.
5  #####
6  create table tbl_file_data
7  (
8      # Primary key
9      id int auto_increment primary key,
10
11     # Data
12     filename varchar(255) not null,
13     data blob not null
14 )$
15 #####
    
```

### 2..6.2 Explanation

The **fileData** table stores binary data and a filename for attachments. It makes use of the **blob** MySQL data type to directly store binary data in the database backend.

## 2..7 contentBase

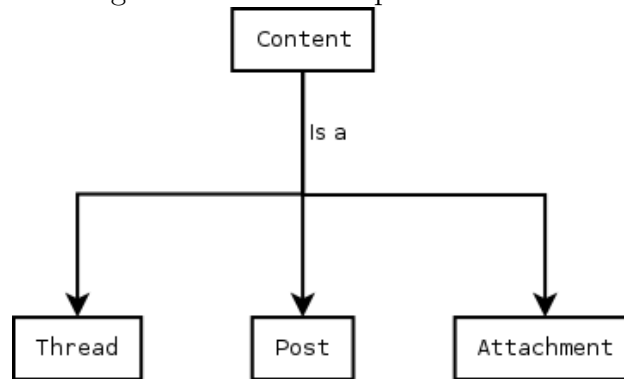
### 2..7.1 Code

```
1 #####
2 # TABLE
3 # * This table deals with content shared data.
4 #####
5 # HIERARCHY
6 # * Is base of: tbl_content_thread, tbl_content_post,
7 #               tbl_content_attachment
8 #####
9 create table tbl_content_base
10 (
11     # Primary key
12     id int auto_increment primary key,
13
14     # Data
15     creation_timestamp timestamp not null default 0,
16     id_author int not null,
17
18     foreign key (id_author)
19         references tbl_user(id)
20         on update no action
21         on delete no action
22 )$
23 #####
```

### 2..7.2 Explanation

The **contentBase** table defines the base entity of the content inheritance tree. Derived content types are: **threads**, **posts** and **attachments**. All content types share a **creation\_timestamp** and an author, identified by **id\_author**.

Figure 6.4: Content specializations.



## 2..8 contentThread

### 2..8.1 Code

```

1 #####
2 # TABLE
3 # * This table deals with threads, a type of content.
4 #####
5 # HIERARCHY
6 # * Derives from: tbl_content_base
7 #####
8 create table tbl_content_thread
9 (
10     # Primary key
11     id int auto_increment primary key,
12
13     # Content base
14     id_base int not null,
15
16     # Parent section
17     id_section int not null,
18
19     # Data
20     title varchar(255) not null,
21
22     foreign key (id_base)
23         references tbl_content_base(id)
24         on update cascade
25         on delete no action,
26
27     foreign key (id_section)
28         references tbl_section(id)
29         on update no action

```

```

30         on delete no action
31     )$
32 #####

```

### 2..8.2 Explanation

Content specialization for **threads**. A thread belongs to a section (identified by `id_section`) and has a `title`. The base content instance is identified by `id_base`.

## 2..9 contentPost

### 2..9.1 Code

```
1 #####
2 # TABLE
3 # * This table deals with posts, a type of content.
4 #####
5 # HIERARCHY
6 # * Derives from: tbl_content_base
7 #####
8 create table tbl_content_post
9 (
10     # Primary key
11     id int auto_increment primary key,
12
13     # Creation data
14     id_base int not null,
15
16     # Parent thread
17     id_thread int not null,
18
19     # Data
20     contents text not null,
21
22     foreign key (id_base)
23         references tbl_content_base(id)
24         on update cascade
25         on delete no action,
26
27     foreign key (id_thread)
28         references tbl_content_thread(id)
29         on update no action
30         on delete no action
31 )$
32 #####
```

### 2..9.2 Explanation

Content specialization for **posts**. A post belongs to a thread (identified by `id_thread`) and has text `contents`. The base content instance is identified by `id_base`.



## 2..10 contentAttachment

### 2..10.1 Code

```
1 #####
2 # TABLE
3 # * This table deals with attachments, a type of content.
4 #####
5 # HIERARCHY
6 # * Derives from: tbl_content_base
7 #####
8 create table tbl_content_attachment
9 (
10     # Primary key
11     id int auto_increment primary key,
12
13     # Creation data
14     id_base int not null,
15
16     # Parent post
17     id_post int not null,
18
19     # File data
20     id_file_data int not null,
21
22     foreign key (id_base)
23         references tbl_content_base(id)
24         on update cascade
25         on delete cascade,
26
27     foreign key (id_post)
28         references tbl_content_post(id)
29         on update no action
30         on delete no action,
31
32     foreign key (id_file_data)
33         references tbl_file_data(id)
34         on update no action
35         on delete no action
36 )$
37 #####
```

### 2..10.2 Explanation

Content specialization for **attachments**. An attachment belongs to a post (identified by `id_post`) and points to a specific file data instance `id_file_data`. The base content instance

is identified by `id_base`.

## 2..11 subscriptionBase

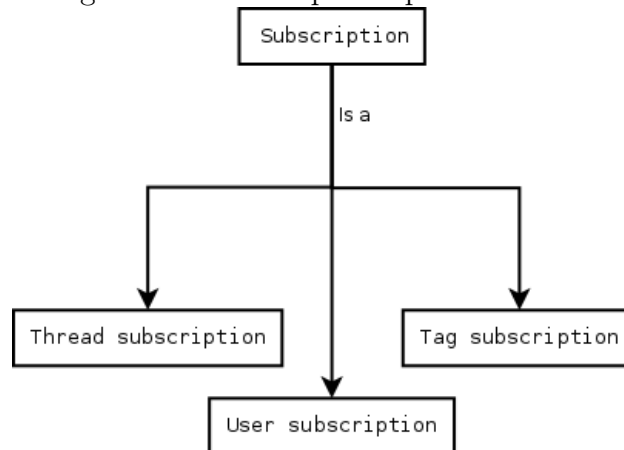
### 2..11.1 Code

```
1 #####
2 # TABLE
3 # * This table deals with subscription shared data.
4 # * Subscriptions allow users to track content or other users.
5 #####
6 # HIERARCHY
7 # * Is base of: tbl_subscription_thread, tbl_subscription_tag,
8 #               tbl_subscription_user
9 #####
10 create table tbl_subscription_base
11 (
12     # Primary key
13     id int auto_increment primary key,
14
15     # Subscriptor user
16     id_subscriptor int not null,
17
18     # Timestamp of beginning
19     creation_timestamp timestamp not null default 0,
20
21     # Active/inactive
22     active boolean not null default true,
23
24     foreign key (id_subscriptor)
25         references tbl_user(id)
26         on update cascade
27         on delete cascade
28 )$
29 #####
```

### 2..11.2 Explanation

The **subscriptionBase** table defines the base entity of the subscription inheritance tree. Derived subscription types are: **thread subscriptions**, **user subscriptions** and **tag subscriptions**. All subscription types share a **creation\_timestamp** (beginning of the subscription), a subscriptor (identified by **id\_subscriptor**) and an **active** flag that can be turned on and off from the web interface by the subscriptor.

Figure 6.5: Subscription specializations.



## 2.12 subscriptionThread

### 2.12.1 Code

```

1 #####
2 # TABLE
3 # * This table deals with thread subscriptions.
4 #####
5 # HIERARCHY
6 # * Derives from: tbl_subscription_base
7 #####
8 create table tbl_subscription_thread
9 (
10     # Primary key
11     id int auto_increment primary key,
12
13     # Base implementation id
14     id_base int not null,
15
16     # Target thread
17     id_thread int not null,
18
19     foreign key (id_base)
20         references tbl_subscription_base(id)
21         on update cascade
22         on delete cascade,
23
24     foreign key (id_thread)
25         references tbl_content_thread(id)
26         on update cascade
27         on delete no action # Triggers do not get fired with 'cascade'

```

```
28 )$
29 #####
```

### 2..12.2 Explanation

Subscription specialization for **thread subscriptions**. Allows to track a thread (identified by `id_thread`) for new content additions. The base subscription instance is identified by `id_base`.

## 2..13 subscriptionUser

### 2..13.1 Code

```
1 #####
2 # TABLE
3 # * This table deals with user subscriptions.
4 #####
5 # HIERARCHY
6 # * Derives from: tbl_subscription_base
7 #####
8 create table tbl_subscription_user
9 (
10     # Primary key
11     id int auto_increment primary key,
12
13     # Base implementation id
14     id_base int not null,
15
16     # Target user
17     id_user int not null,
18
19     foreign key (id_base)
20         references tbl_subscription_base(id)
21         on update cascade
22         on delete cascade,
23
24     foreign key (id_user)
25         references tbl_user(id)
26         on update cascade
27         on delete no action # Triggers do not get fired with 'cascade'
28 )$
29 #####
```

### 2..13.2 Explanation

Subscription specialization for **user subscriptions**. Allows to track an user (identified by `id_user`) for new content additions. The base subscription instance is identified by `id_base`.

## 2..14 subscriptionTag

### 2..14.1 Code

```
1 #####
2 # TABLE
3 # * This table deals with tag subscriptions.
4 #####
5 # HIERARCHY
6 # * Derives from: tbl_subscription_base
7 #####
8 create table tbl_subscription_tag
9 (
10     # Primary key
11     id int auto_increment primary key,
12
13     # Base implementation id
14     id_base int not null,
15
16     # Target tag
17     id_tag int not null,
18
19     foreign key (id_base)
20         references tbl_subscription_base(id)
21         on update cascade
22         on delete cascade,
23
24     foreign key (id_tag)
25         references tbl_tag(id)
26         on update cascade
27         on delete no action # Triggers do not get fired with 'cascade'
28 )$
29 #####
```

### 2..14.2 Explanation

Subscription specialization for **tag subscriptions**. Allows to track a tag (identified by `id_tag`) for new content additions. The base subscription instance is identified by `id_base`.

## 2..15 notificationBase

### 2..15.1 Code

```
1 #####
2 # TABLE
3 # * This table deals with notification shared data.
4 # * Notifications are created when users need to be notified
5 #   about content they are subscribed to.
6 #####
7 # HIERARCHY
8 # * Is base of: tbl_notification_user, tbl_notification_thread,
9 #               tbl_notification_tag
10 #####
11 create table tbl_notification_base
12 (
13     # Primary key
14     id int auto_increment primary key,
15
16     # Receiver of the notification
17     id_receiver int not null,
18
19     # Notification seen?
20     seen boolean not null default false,
21
22     # Notification data creation timestamp
23     creation_timestamp timestamp not null default 0,
24
25     foreign key (id_receiver)
26         references tbl_user(id)
27         on update cascade
28         on delete cascade
29 )$
30 #####
```

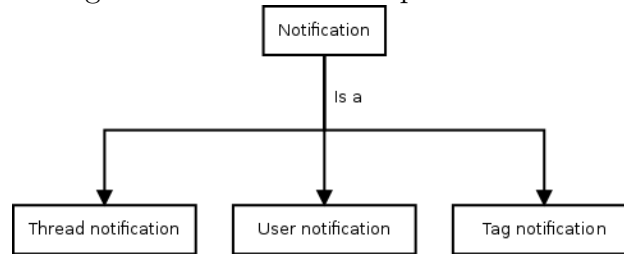
### 2..15.2 Explanation

The **notificationBase** table defines the base entity of the notification inheritance tree. Derived notification types are: **thread notifications**, **user notifications** and **tag notifications**. All notifications types share a **seen** flag (which is set to **true** if the receiver seen a particular notification), a receiver (identified by **id\_receiver**) and a **creation\_timestamp**.

Notifications are created from subscriptions, using triggers.



Figure 6.6: Notification specializations.



## 2..16 notificationUser

### 2..16.1 Code

```

1 #####
2 # TABLE
3 # * This table deals with user notifications.
4 #####
5 # HIERARCHY
6 # * Derives from: tbl_notification_base
7 #####
8 create table tbl_notification_user
9 (
10     # Primary key
11     id int auto_increment primary key,
12
13     # Base
14     id_base int not null,
15
16     # Subscription
17     id_subscription_user int not null,
18
19     # Content posted by the user
20     id_content int not null,
21
22     foreign key (id_base)
23         references tbl_notification_base(id)
24         on update cascade
25         on delete cascade,
26
27     foreign key (id_subscription_user)
28         references tbl_subscription_user(id)
29         on update cascade
30         on delete no action, # Triggers do not get fired with 'cascade'
31
32     foreign key (id_content)
33         references tbl_content_base(id)

```

```

34         on update cascade
35         on delete no action # Triggers do not get fired with 'cascade'
36     )$
37     #####

```

## 2..16.2 Explanation

Notification specialization for **user notifications**. Generated when a tracked user creates new content. Points to the subscription that generated the notification (identified by `id_subscription_user`) and to the created content (identified by `id_content`). The base notification instance is identified by `id_base`.

## 2..17 notificationThread

### 2..17.1 Code

```
1 #####
2 # TABLE
3 # * This table deals with thread notifications.
4 #####
5 # HIERARCHY
6 # * Derives from: tbl_notification_base
7 #####
8 create table tbl_notification_thread
9 (
10     # Primary key
11     id int auto_increment primary key,
12
13     # Base
14     id_base int not null,
15
16     # Subscription
17     id_subscription_thread int not null,
18
19     # Newly created post
20     id_post int not null,
21
22     foreign key (id_base)
23         references tbl_notification_base(id)
24         on update cascade
25         on delete cascade,
26
27     foreign key (id_subscription_thread)
28         references tbl_subscription_thread(id)
29         on update cascade
30         on delete no action, # Triggers do not get fired with 'cascade'
31
32     foreign key (id_post)
33         references tbl_content_post(id)
34         on update cascade
35         on delete no action # Triggers do not get fired with 'cascade'
36 )$
37 #####
```

### 2..17.2 Explanation

Notification specialization for **thread notifications**. Generated when new content is added to a tracked thread. Points to the subscription that generated the notification (identified

by `id_subscription_thread`) and to the created content (identified by `id_content`). The base notification instance is identified by `id_base`.

## 2..18 notificationTag

### 2..18.1 Code

```
1 #####
2 # TABLE
3 # * This table deals with tag notifications.
4 #####
5 # HIERARCHY
6 # * Derives from: tbl_notification_base
7 #####
8 create table tbl_notification_tag
9 (
10     # Primary key
11     id int auto_increment primary key,
12
13     # Base
14     id_base int not null,
15
16     # Subscription
17     id_subscription_tag int not null,
18
19     foreign key (id_base)
20         references tbl_notification_base(id)
21         on update cascade
22         on delete cascade,
23
24     foreign key (id_subscription_tag)
25         references tbl_subscription_tag(id)
26         on update cascade
27         on delete no action # Triggers do not get fired with 'cascade'
28 )$
29 #####
```

### 2..18.2 Explanation

Notification specialization for **tag notifications**. Generated when new content is labeled with the tracked tag. Points to the subscription that generated the notification (identified by `id_subscription_tag`) and to the created content (identified by `id_content`). The base notification instance is identified by `id_base`.

## 2..19 tagContent

### 2..19.1 Code

```
1 #####
2 # TABLE
3 # * This table deals with the many-to-many tag-content relationship.
4 #####
5 create table tbl_tag_content
6 (
7     # Primary key
8     id int auto_increment primary key,
9
10    # Tag
11    id_tag int not null,
12
13    # Content base
14    id_content int not null,
15
16    foreign key (id_tag)
17        references tbl_tag(id)
18        on update cascade
19        on delete cascade,
20
21    foreign key (id_content)
22        references tbl_content_base(id)
23        on update cascade
24        on delete cascade
25 )$
26 #####
```

### 2..19.2 Explanation

The **tagContent** table labels content to tags. It is a **N to N** relationship table.



## 2..20 groupSectionPermission

### 2..20.1 Code

```
1 #####
2 # TABLE
3 # * This table deals with the many-to-many group-section permissions
4 # relationship.
5 #####
6 create table tbl_group_section_permission
7 (
8     # Primary key
9     id int auto_increment primary key,
10
11     # Relationship (group <-> section)
12     id_group int not null,
13     id_section int not null,
14
15     # Data
16     can_view boolean not null,
17     can_post boolean not null,
18     can_create_thread boolean not null,
19     can_delete_post boolean not null,
20     can_delete_thread boolean not null,
21     can_delete_section boolean not null,
22
23     foreign key (id_group)
24         references tbl_group(id)
25         on update cascade
26         on delete cascade,
27
28     foreign key (id_section)
29         references tbl_section(id)
30         on update cascade
31         on delete cascade
32 )$
33 #####
```

### 2..20.2 Explanation

The **groupSectionPermission** table links groups to sections, giving users belonging to the selected group a set of permissions for the selected section. It is a **N to N** relationship table.

## 3. Stored procedures

To ensure **maximum performance** and to **minimize coupling** with the PHP backend, the logic of the forum system is, where possible, implemented with SQL **stored procedures**. A stored procedure is a subroutine available to applications that access a relational database system, and it is actually stored in the database data dictionary.

### 3.1 mkContent

#### 3.1.1 Code

```
1  #####
2  # PROCEDURE
3  # * Create a content base and return its ID.
4  #####
5  create procedure mk_content_base
6  (
7      in v_id_author int,
8      out v_created_id int
9  )
10 begin
11     insert into tbl_content_base
12         (id_author, creation_timestamp)
13         values(v_id_author, now());
14
15     set v_created_id := LAST_INSERT_ID();
16 end$
17 #####
18
19
20
21 #####
22 # PROCEDURE
23 # * Create a content base + content thread.
24 #####
25 create procedure mk_content_thread
26 (
27     in v_id_author int,
28     in v_id_section int,
29     in v_title varchar(255)
30 )
31 begin
32     call mk_content_base(v_id_author, @out_id_base);
33
34     insert into tbl_content_thread
```



```

35         (id_base, id_section, title)
36         values(@out_id_base, v_id_section, v_title);
37     end$
38     #####
39
40
41
42     #####
43     # PROCEDURE
44     # * Create a content base + content post.
45     #####
46     create procedure mk_content_post
47     (
48         in v_id_author int,
49         in v_id_thread int,
50         in v_contents text
51     )
52     begin
53         call mk_content_base(v_id_author, @out_id_base);
54
55         insert into tbl_content_post
56             (id_base, id_thread, contents)
57             values(@out_id_base, v_id_thread, v_contents);
58     end$
59     #####
60
61
62
63     #####
64     # PROCEDURE
65     # * Create a content base + content attachment.
66     #####
67     create procedure mk_content_attachment
68     (
69         in v_id_author int,
70         in v_id_post int,
71         in v_id_file_data int
72     )
73     begin
74         call mk_content_base(v_id_author, @out_id_base);
75
76         insert into tbl_content_attachment
77             (id_base, id_post, id_file_data)
78             values(@out_id_base, v_id_post, v_id_file_data);
79     end$
80     #####

```

### 3..1.2 Explanation

The procedures in the code listed above deal with the creation of content. To create content, it is necessary to instantiate both a `content_base` row and a specialization data row. These procedures automatically create both the required rows and make sure they relate to each other correctly, thanks to the `LAST_INSERT_ID()` MySQL function.

- `mk_content_base`: creates a content base record and returns its id.
- `mk_content_thread`: calls `mk_content_base`, then creates a thread specialization row linked to it. Takes the author id and title of the thread as input parameters.
- `mk_content_post`: calls `mk_content_base`, then creates a thread specialization row linked to it. Takes the author id and id of the parent thread as input parameters.
- `mk_content_attachment`: calls `mk_content_base`, then creates a thread specialization row linked to it. Takes the author id and id of the parent post as input parameters.

## 3.2 mkSubscription

### 3.2.1 Code

```
1  #####
2  # PROCEDURE
3  # * Create a subscription base and return its ID.
4  #####
5  create procedure mk_subscription_base
6  (
7      in v_id_subscriptor int,
8      out v_created_id int
9  )
10 begin
11     insert into tbl_subscription_base
12         (id_subscriptor, creation_timestamp, active)
13         values(v_id_subscriptor, now(), true);
14
15     set v_created_id := LAST_INSERT_ID();
16 end$
17 #####
18
19
20
21 #####
22 # PROCEDURE
23 # * Create a subscription base + subscription user.
24 #####
25 create procedure mk_subscription_user
26 (
27     in v_id_subscriptor int,
28     in v_id_user int
29 )
30 begin
31     call mk_subscription_base(v_id_subscriptor, @out_id_base);
32
33     insert into tbl_subscription_user
34         (id_base, id_user)
35         values(@out_id_base, v_id_user);
36 end$
37 #####
38
39
40
41 #####
42 # PROCEDURE
43 # * Create a subscription base + subscription thread.
```

```

44 #####
45 create procedure mk_subscription_thread
46 (
47     in v_id_subscriptor int,
48     in v_id_thread int
49 )
50 begin
51     call mk_subscription_base(v_id_subscriptor, @out_id_base);
52
53     insert into tbl_subscription_thread
54         (id_base, id_thread)
55         values(@out_id_base, v_id_thread);
56 end$
57 #####
58
59
60
61 #####
62 # PROCEDURE
63 # * Create a subscription base + subscription tag.
64 #####
65 create procedure mk_subscription_tag
66 (
67     in v_id_subscriptor int,
68     in v_id_tag int
69 )
70 begin
71     call mk_subscription_base(v_id_subscriptor, @out_id_base);
72
73     insert into tbl_subscription_tag
74         (id_base, id_tag)
75         values(@out_id_base, v_id_tag);
76 end$
77 #####

```

### 3..2.2 Explanation

The procedures in the code listed above deal with the creation of subscriptions. To create subscriptions, it is necessary to instantiate both a `subscription_base` row and a specialization data row. These procedures automatically create both the required rows and make sure they relate to each other correctly, thanks to the `LAST_INSERT_ID()` MySQL function.

- `mk_subscription_base`: creates a subscription base record and returns its id.
- `mk_subscription_user`: calls `mk_subscription_base`, then creates a user specialization row linked to it. Takes the subscriptor id and id of the user as input parameters.

- `mk_subscription_thread`: calls `mk_subscription_base`, then creates a thread specialization row linked to it. Takes the subscriber id and id of the thread as input parameters.
- `mk_subscription_tag`: calls `mk_subscription_base`, then creates a tag specialization row linked to it. Takes the subscriber id and id of the tag as input parameters.

## 3.3 mkNotification

### 3.3.1 Code

```
1  #####
2  # PROCEDURE
3  # * Create a notification base and return its ID.
4  #####
5  create procedure mk_notification_base
6  (
7      in v_id_receiver int,
8      out v_created_id int
9  )
10 begin
11     insert into tbl_notification_base
12         (id_receiver, seen, creation_timestamp)
13         values(v_id_receiver, false, now());
14
15     set v_created_id := LAST_INSERT_ID();
16 end$
17 #####
18
19
20
21 #####
22 # PROCEDURE
23 # * Create a notification base + notification user.
24 #####
25 create procedure mk_notification_user
26 (
27     in v_id_receiver int,
28     in v_id_subscription_user int,
29     in v_id_content int
30 )
31 begin
32     call mk_notification_base(v_id_receiver, @out_id_base);
33
34     insert into tbl_notification_user
35         (id_base, id_subscription_user, id_content)
36         values(@out_id_base, v_id_subscription_user, v_id_content);
37 end$
38 #####
39
40
41
42 #####
43 # PROCEDURE
```

```

44  # * Create a notification base + notification thread.
45  #####
46  create procedure mk_notification_thread
47  (
48      in v_id_receiver int,
49      in v_id_subscription_thread int,
50      in v_id_post int
51  )
52  begin
53      call mk_notification_base(v_id_receiver, @out_id_base);
54
55      insert into tbl_notification_thread
56          (id_base, id_subscription_thread, id_post)
57          values(@out_id_base, v_id_subscription_thread, v_id_post);
58  end$
59  #####
60
61
62
63  #####
64  # PROCEDURE
65  # * Create a notification base + notification tag.
66  #####
67  create procedure mk_notification_tag
68  (
69      in v_id_receiver int,
70      in v_id_subscription_tag int
71  )
72  begin
73      call mk_notification_base(v_id_receiver, @out_id_base);
74
75      insert into tbl_notification_tag
76          (id_base, id_subscription_tag)
77          values(@out_id_base, v_id_subscription_tag);
78  end$
79  #####

```

### 3.3.2 Explanation

The procedures in the code listed above deal with the creation of notifications. To create notifications, it is necessary to instantiate both a `notification_base` row and a specialization data row. These procedures automatically create both the required rows and make sure they relate to each other correctly, thanks to the `LAST_INSERT_ID()` MySQL function.

- `mk_notification_base`: creates a notification base record and returns its id.

- `mk_notification_user`: calls `mk_notification_base`, then creates a user specialization row linked to it. Takes the notification receiver id, the user subscription id and id of the new content as input parameters.
- `mk_notification_thread`: calls `mk_notification_base`, then creates a thread specialization row linked to it. Takes the notification receiver id, the thread subscription id and id of the new content as input parameters.
- `mk_notification_tag`: calls `mk_notification_base`, then creates a tag specialization row linked to it. Takes the notification receiver id, the tag subscription id and id of the new content as input parameters.



## 3..4 utils

### 3..4.1 Code

```
1  #####
2  # PROCEDURE
3  # * Return the subscriptor ID from a subscription base ID.
4  #####
5  create procedure get_subscriptor
6  (
7      in v_id_base int,
8      out v_id_subscriptor int
9  )
10 begin
11     select id_subscriptor
12     into v_id_subscriptor
13     from tbl_subscription_base
14     where id = v_id_base;
15 end$
16 #####
17
18
19
20 #####
21 # PROCEDURE
22 # * Returns true if an unseen notification user with a specific
23 #   subscriptor ID and a specific user ID exists.
24 #####
25 create procedure check_notification_unseen_existance_user
26 (
27     in v_id_subscriptor int,
28     in v_id_user int,
29     out v_result boolean
30 )
31 begin
32     set v_result := exists
33     (
34         select tb.id_receiver, tb.seen, ts.id_user
35         from tbl_notification_base as tb
36             inner join tbl_notification_user as td on tb.id = td.id_base
37             inner join tbl_subscription_user as ts on td.id_subscription_user = ts.id
38         where
39             tb.seen = false
40             and tb.id_receiver = v_id_subscriptor
41             and ts.id_user = v_id_user
42     );
43 end$
```

```

44 #####
45
46
47
48 #####
49 # PROCEDURE
50 # * Returns true if an unseen notification thread with a specific
51 #   subscriber ID and a specific thread ID exists.
52 #####
53 create procedure check_notification_unseen_existance_thread
54 (
55     in v_id_subscriber int,
56     in v_id_thread int,
57     out v_result boolean
58 )
59 begin
60     set v_result := exists
61     (
62         select tb.id_receiver, tb.seen, ts.id_thread
63         from tbl_notification_base as tb
64             inner join tbl_notification_thread as td on tb.id = td.id_base
65             inner join tbl_subscription_thread as ts on td.id_subscription_thread = ts.id
66         where
67             tb.seen = false
68             and tb.id_receiver = v_id_subscriber
69             and ts.id_thread = v_id_thread
70     );
71 end$
72 #####
73
74
75
76 #####
77 # PROCEDURE
78 # * Returns true if an unseen notification user with a specific
79 #   subscriber ID and a specific tag ID exists.
80 #####
81 create procedure check_notification_unseen_existance_tag
82 (
83     in v_id_subscriber int,
84     in v_id_tag int,
85     out v_result boolean
86 )
87 begin
88     set v_result := exists
89     (
90         select tb.id_receiver, tb.seen, ts.id_tag

```

```

91         from tbl_notification_base as tb
92             inner join tbl_notification_tag as td on tb.id = td.id_base
93             inner join tbl_subscription_tag as ts on td.id_subscription_tag = ts.id
94         where
95             tb.seen = false
96             and tb.id_receiver = v_id_subscriptor
97             and ts.id_user = v_id_tag
98     );
99 end$
100 #####

```

### 3.4.2 Explanation

The code listed above is composed of several utility stored procedures.

- **get\_subscriptor**: takes a **subscription base id** as an input parameter and returns the id of the subscriptor.
- **check\_notification\_unseen\_existance\_user**: takes a **subscriptor id** and a **target subscribed user id** as input parameters and returns **true** if an unseen user notification with the passed parameters exists.
- **check\_notification\_unseen\_existance\_thread**: takes a **subscriptor id** and a **target subscribed thread id** as input parameters and returns **true** if an unseen thread notification with the passed parameters exists.
- **check\_notification\_unseen\_existance\_tag**: takes a **subscriptor id** and a **target subscribed tag id** as input parameters and returns **true** if an unseen tag notification with the passed parameters exists.

## 3..5 gNUser

### 3..5.1 Code

```
1  #####
2  # PROCEDURE
3  # * Generate notifications for every subscriber to the author of the
4  #   last created content.
5  #####
6  create procedure generate_notifications_user
7  (
8      in v_last_content_id int,
9      in v_last_content_author int
10 )
11 begin
12     declare loop_done int default false;
13     declare var_id_sub, var_id_sub_base, var_id_sub_tracked_user,
14             current_id_subscriptor int;
15     declare itr cursor for select id, id_base, id_user from tbl_subscription_user;
16     declare continue handler for not found set loop_done = true;
17
18     open itr;
19
20     label_loop:
21     loop
22         fetch itr into var_id_sub, var_id_sub_base, var_id_sub_tracked_user;
23
24         if loop_done then
25             leave label_loop;
26         end if;
27
28         if var_id_sub_tracked_user = v_last_content_author then
29             call get_subscriptor(var_id_sub_base, current_id_subscriptor);
30             call mk_notification_user(current_id_subscriptor, var_id_sub,
31                                     ↪ v_last_content_id);
32         end if;
33     end loop;
34     close itr;
35 end$
36 #####
```

### 3..5.2 Explanation

The stored procedure listed above deals with the **generation of user notifications**. It is automatically called by the `trg_notifications_user` trigger, which fires after the addition

of new content to the system.

The procedure takes the **last added content id** and its **author id** as input parameters, and generates (if matching subscriptions exists) notification records for every subscriber.

The code makes use of **complex MySQL features** like **cursors**, **variable declarations** and **loops**. These features are required to efficiently traverse the subscription hierarchies and retrieve the necessary identifiers.

To generate the notifications, the `get_subscriber` and `mk_notification_user` procedures are called inside the loop, for each matching subscriber.

## 3..6 gNThread

### 3..6.1 Code

```
1  #####
2  # PROCEDURE
3  # * Generate notifications for every subscriber to the thread of the
4  #   last created post.
5  #####
6  create procedure generate_notifications_thread
7  (
8      in v_last_post_id int,
9      in v_last_post_thread int
10 )
11 begin
12     declare loop_done int default false;
13     declare var_id_sub, var_id_sub_base, var_id_sub_tracked_thread,
14             current_id_subscriptor int;
15     declare itr cursor for select id, id_base, id_thread from tbl_subscription_thread;
16     declare continue handler for not found set loop_done = true;
17
18     open itr;
19
20     label_loop:
21     loop
22         fetch itr into var_id_sub, var_id_sub_base, var_id_sub_tracked_thread;
23
24         if loop_done then
25             leave label_loop;
26         end if;
27
28         if var_id_sub_tracked_thread = v_last_post_thread then
29             call get_subscriptor(var_id_sub_base, current_id_subscriptor);
30             call mk_notification_thread(current_id_subscriptor, var_id_sub,
31                                         ↪ v_last_post_id);
32         end if;
33     end loop;
34     close itr;
35 end$
36 #####
```

### 3..6.2 Explanation

The stored procedure listed above deals with the **generation of thread notifications**. It is automatically called by the `trg_notifications_thread` trigger, which fires after the

addition of new content to the system.

The procedure takes the **last added post id** and its **parent thread id** as input parameters, and generates (if matching subscriptions exists) notification records for every subscriber.

The code makes use of **complex MySQL features** like **cursors**, **variable declarations** and **loops**. These features are required to efficiently traverse the subscription hierarchies and retrieve the necessary identifiers.

To generate the notifications, the `get_subscriber` and `mk_notification_thread` procedures are called inside the loop, for each matching subscriber.

## 3..7 gNTag

### 3..7.1 Code

```
1  #####
2  # PROCEDURE
3  # * Generate notifications for every subscriber to the tag of the
4  #   last created content.
5  #####
6  create procedure generate_notifications_tag
7  (
8      in v_last_tc_tag int,
9      in v_last_tc_content int
10 )
11 begin
12     declare loop_done int default false;
13     declare var_id_sub, var_id_sub_base, var_id_sub_tracked_tag,
14             current_id_subscriptor int;
15     declare itr cursor for select id, id_base, id_tag from tbl_subscription_tag;
16     declare continue handler for not found set loop_done = true;
17
18     open itr;
19
20     label_loop:
21     loop
22         fetch itr into var_id_sub, var_id_sub_base, var_id_sub_tracked_tag;
23
24         if loop_done then
25             leave label_loop;
26         end if;
27
28         if var_id_sub_tracked_tag = v_last_tc_tag then
29             call get_subscriptor(var_id_sub_base, current_id_subscriptor);
30             call mk_notification_tag(current_id_subscriptor, var_id_sub);
31         end if;
32     end loop;
33
34     close itr;
35 end$
36 #####
```

### 3..7.2 Explanation

The stored procedure listed above deals with the **generation of tag notifications**. It is automatically called by the `trg_notifications_tag` trigger, which fires after the addition of new content to the system.



The procedure takes the **last added content's tag id** and its **content base id** as input parameters, and generates (if matching subscriptions exists) notification records for every subscriber.

The code makes use of **complex MySQL features** like **cursors**, **variable declarations** and **loops**. These features are required to efficiently traverse the subscription hierarchies and retrieve the necessary identifiers.

To generate the notifications, the `get_subscriber` and `mk_notification_tag` procedures are called inside the loop, for each matching subscriber.

## 3..8 calcPrivs

### 3..8.1 Code

```
1  #####
2  # PROCEDURE
3  # * Calculate the final privileges of a user by inheriting them from the group hierarchy
4  #   they belong to.
5  #####
6  create procedure calculate_final_privileges
7  (
8      in v_id_user int,
9      out v_is_superadmin boolean,
10     out v_can_manage_sections boolean,
11     out v_can_manage_users boolean,
12     out v_can_manage_groups boolean,
13     out v_can_manage_permissions boolean
14 )
15 begin
16     # Set initial out values
17     set v_is_superadmin := false;
18     set v_can_manage_sections := false;
19     set v_can_manage_users := false;
20     set v_can_manage_groups := false;
21     set v_can_manage_permissions := false;
22
23     # Get user group
24     select id_group
25     into @current_id_group
26     from tbl_user
27     where id = v_id_user;
28
29     # Traverse the hierarchy and set privileges
30     label_loop:
31     loop
32         set @last_id_group := @current_id_group;
33
34         select id_parent, is_superadmin, can_manage_sections,
35                can_manage_users, can_manage_groups, can_manage_permissions
36         into @current_id_group, @p0, @p1, @p2, @p3, @p4
37         from tbl_group
38         where id = @last_id_group;
39
40         set v_is_superadmin := v_is_superadmin or @p0;
41         set v_can_manage_sections := v_can_manage_sections or @p1;
42         set v_can_manage_users := v_can_manage_users or @p2;
43         set v_can_manage_groups := v_can_manage_groups or @p3;
```

```

44         set v_can_manage_permissions := v_can_manage_permissions or @p4;
45
46         if @current_id_group is null then
47             leave label_loop;
48         end if;
49     end loop;
50 end$
51 #####

```

### 3..8.2 Explanation

The `calculate_final_privileges` stored procedure takes an **user id** input parameter and traverses its **user/group hierarchy** recursively, returning the final system-wide privilege bit set of the user.

The bit set contains the following privileges:

- `v_is_superuser`: `true` if the user is a super administrator.
- `v_can_manage_sections`: `true` if the user can manage (add/edit/remove) sections.
- `v_can_manage_users`: `true` if the user can manage (add/edit/remove) users.
- `v_can_manage_groups`: `true` if the user can manage (add/edit/remove) group hierarchies.
- `v_can_manage_permissions`: `true` if the user can manage (add/edit/remove) permission hierarchies.

## 3..9 calcPerms

### 3..9.1 Code

```
1  #####
2  # PROCEDURE
3  # * Calculate the final permissions of a user by inheriting them from the group hierarchy
4  #   they belong to, towards a specific section.
5  #####
6  create procedure calculate_final_permissions
7  (
8      in v_id_user int,
9      in v_id_section int,
10     out v_can_view boolean,
11     out v_can_post boolean,
12     out v_can_create_thread boolean,
13     out v_can_delete_post boolean,
14     out v_can_delete_thread boolean,
15     out v_can_delete_section boolean
16 )
17 begin
18     # Set initial out values
19     set v_can_view := false;
20     set v_can_post := false;
21     set v_can_create_thread := false;
22     set v_can_delete_post := false;
23     set v_can_delete_thread := false;
24     set v_can_delete_section := false;
25
26     # Get user group
27     select id_group
28     into @current_id_group
29     from tbl_user
30     where id = v_id_user;
31
32     # Traverse the hierarchy and set permissions
33     label_loop:
34     loop
35         set @last_id_group := @current_id_group;
36
37         select id_parent
38         into @current_id_group
39         from tbl_group
40         where id = @last_id_group;
41
42         select can_view, can_post, can_create_thread,
43                can_delete_post, can_delete_thread, can_delete_section
```

```

44     into @p0, @p1, @p2, @p3, @p4, @p5
45     from tbl_group_section_permission
46     where id_group = @last_id_group and id_section = v_id_section;
47
48     set v_can_view := v_can_view or @p0;
49     set v_can_post := v_can_post or @p1;
50     set v_can_create_thread := v_can_create_thread or @p2;
51     set v_can_delete_post := v_can_delete_post or @p3;
52     set v_can_delete_thread := v_can_delete_thread or @p4;
53     set v_can_delete_section := v_can_delete_section or @p5;
54
55     if @current_id_group is null then
56         leave label_loop;
57     end if;
58 end loop;
59 end$
60 #####

```

### 3..9.2 Explanation

The `calculate_final_permissions` stored procedure takes an **user id** and a **section id** as input parameters and traverses the user's **user/group hierarchy** recursively, calculating and returning the final user permissions related to the passed section.

The calculated permission set (a bit set) contains the following boolean values:

- `v_can_view`: **true** if the user can view/access the section.
- `v_can_post`: **true** if the user can post in threads existing in the section.
- `v_can_create_thread`: **true** if the user can create threads in the section.
- `v_can_delete_post`: **true** if the user can delete posts inside the section threads.
- `v_can_delete_thread`: **true** if the user can delete threads inside the section.
- `v_can_delete_section`: **true** if the user can delete the section and its subsections.

## 4. Triggers

### 4.1 notifications

#### 4.1.1 Code

```
1 #####
2 # TRIGGER
3 # * Generate notifications for user subscriptions after content
4 #   creation.
5 #####
6 create trigger trg_notifications_user
7     after insert on tbl_content_base
8     for each row
9 begin
10     call generate_notifications_user(NEW.id, NEW.id_author);
11 end$
12 #####
13
14
15
16 #####
17 # TRIGGER
18 # * Generate notifications for thread subscriptions after post
19 #   creation.
20 #####
21 create trigger trg_notifications_thread
22     after insert on tbl_content_post
23     for each row
24 begin
25     call generate_notifications_thread(NEW.id, NEW.id_thread);
26 end$
27 #####
28
29
30
31 #####
32 # TRIGGER
33 # * Generate notifications for tag subscriptions after content
34 #   creation.
35 #####
36 create trigger trg_notifications_tag
37     after insert on tbl_tag_content
38     for each row
39 begin
40     call generate_notifications_tag(NEW.id_tag, NEW.id_content);
```

```

41 end$
42 #####

```

#### 4.1.2 Explanation

These triggers deal with **notification generation**.

- `trg_notifications_user`: generates a notification when a tracked user creates content.
- `trg_notifications_thread`: generates a notification when a post is added to a tracked thread.
- `trg_notifications_tag`: generates a notification when content with the tracked tag is created.

## 4.2 contentBase

### 4.2.1 Code

```
1  #####
2  # TRIGGER
3  # * Delete content base left behind by derived content types.
4  #####
5  create trigger trg_del_content_base_thread
6      after delete on tbl_content_thread
7      for each row
8  begin
9      delete from tbl_content_base
10     where id = OLD.id_base;
11 end$
12 #####
13
14
15
16 #####
17 # TRIGGER
18 # * Delete content base left behind by derived content types.
19 #####
20 create trigger trg_del_content_base_post
21     after delete on tbl_content_post
22     for each row
23 begin
24     delete from tbl_content_base
25     where id = OLD.id_base;
26 end$
27 #####
28
29
30
31 #####
32 # TRIGGER
33 # * Delete content base left behind by derived content types.
34 #####
35 create trigger trg_del_content_base_attachment
36     after delete on tbl_content_attachment
37     for each row
38 begin
39     delete from tbl_content_base
40     where id = OLD.id_base;
41 end$
42 #####
43
```



```

44
45
46
47
48
49
50
51
52
53 #####
54 # TRIGGER
55 # * Delete notifications pointing to content that's about to be deleted.
56 #####
57 create trigger trg_del_ntf_user_on_post_del
58     before delete on tbl_content_base
59     for each row
60 begin
61     delete from tbl_notification_user
62     where id_content = OLD.id;
63 end$
64 #####
65
66
67 #####
68 # TRIGGER
69 # * Delete notifications pointing to content that's about to be deleted.
70 #####
71 create trigger trg_del_ntf_thread_on_post_del
72     before delete on tbl_content_post
73     for each row
74 begin
75     delete from tbl_notification_thread
76     where id_post = OLD.id;
77 end$
78 #####

```

#### 4..2.2 Explanation

These triggers deal with content deletion and cleanup.

The `trg_del_content_base_thread`, `trg_del_content_base_post`, and `trg_del_content_base_att` triggers automatically delete the content base instance upon derived content instance deletion.

The `trg_del_ntf_user_on_post_del` and `trg_del_ntf_thread_on_post_del` triggers delete notifications pointing to content that is about to get deleted.

## 4..3 subscriptionBase

### 4..3.1 Code

```
1  #####
2  # TRIGGER
3  # * Delete subscription base left behind by derived subscription types.
4  #####
5  create trigger trg_del_subscription_base_thread
6      after delete on tbl_subscription_thread
7      for each row
8  begin
9      delete from tbl_subscription_base
10     where id = OLD.id_base;
11 end$
12 #####
13
14
15
16 #####
17 # TRIGGER
18 # * Delete subscription base left behind by derived subscription types.
19 #####
20 create trigger trg_del_subscription_base_user
21     after delete on tbl_subscription_user
22     for each row
23 begin
24     delete from tbl_subscription_base
25     where id = OLD.id_base;
26 end$
27 #####
28
29
30
31 #####
32 # TRIGGER
33 # * Delete subscription base left behind by derived subscription types.
34 #####
35 create trigger trg_del_subscription_base_tag
36     after delete on tbl_subscription_tag
37     for each row
38 begin
39     delete from tbl_subscription_base
40     where id = OLD.id_base;
41 end$
42 #####
```

### 4..3.2 Explanation

These triggers deal with subscription base instance automatic deletion upon derived instance deletion.

## 4.4 notificationBase

### 4.4.1 Code

```
1  #####
2  # TRIGGER
3  # * Delete notification base left behind by derived notification types.
4  #####
5  create trigger trg_del_notification_base_thread
6      after delete on tbl_notification_thread
7      for each row
8  begin
9      delete from tbl_notification_base
10     where id = OLD.id_base;
11 end$
12 #####
13
14
15
16 #####
17 # TRIGGER
18 # * Delete notification base left behind by derived notification types.
19 #####
20 create trigger trg_del_notification_base_user
21     after delete on tbl_notification_user
22     for each row
23 begin
24     delete from tbl_notification_base
25     where id = OLD.id_base;
26 end$
27 #####
28
29
30
31 #####
32 # TRIGGER
33 # * Delete notification base left behind by derived notification types.
34 #####
35 create trigger trg_del_notification_base_tag
36     after delete on tbl_notification_tag
37     for each row
38 begin
39     delete from tbl_notification_base
40     where id = OLD.id_base;
41 end$
42 #####
```

#### 4.4.2 Explanation

These triggers deal with notification base instance automatic deletion upon derived instance deletion.

## 4..5 subscriptionNtf

### 4..5.1 Code

```
1  #####
2  # TRIGGER
3  # * Delete notifications that point to the deleted subscription.
4  #####
5  create trigger trg_del_subscription_ntf_thread
6      before delete on tbl_subscription_thread
7      for each row
8  begin
9      delete from tbl_notification_thread
10     where id_subscription_thread = OLD.id;
11 end$
12 #####
13
14
15
16 #####
17 # TRIGGER
18 # * Delete notifications that point to the deleted subscription.
19 #####
20 create trigger trg_del_subscription_ntf_user
21     before delete on tbl_subscription_user
22     for each row
23 begin
24     delete from tbl_notification_user
25     where id_subscription_user = OLD.id;
26 end$
27 #####
28
29
30
31 #####
32 # TRIGGER
33 # * Delete notifications that point to the deleted subscription.
34 #####
35 create trigger trg_del_subscription_ntf_tag
36     before delete on tbl_subscription_tag
37     for each row
38 begin
39     delete from tbl_notification_tag
40     where id_subscription_tag = OLD.id;
41 end$
42 #####
```

#### 4..5.2 Explanation

These triggers delete all notification belonging to a subscription that's about to be deleted.

## 4.6 delSubCnt

### 4.6.1 Code

```
1  #####
2  # TRIGGER
3  # * Delete subscriptions pointing to threads about to be deleted.
4  #####
5  create trigger trg_del_subscription_cnt_thread
6      before delete on tbl_content_thread
7      for each row
8  begin
9      delete from tbl_subscription_thread
10     where id_thread = OLD.id;
11 end$
12 #####
13
14
15 #####
16 # TRIGGER
17 # * Delete subscriptions pointing to users about to be deleted.
18 #####
19 create trigger trg_del_subscription_cnt_user
20     before delete on tbl_user
21     for each row
22 begin
23     delete from tbl_subscription_user
24     where id_user = OLD.id;
25 end$
26 #####
27
28
29
30 #####
31 # TRIGGER
32 # * Delete subscriptions pointing to tags about to be deleted.
33 #####
34 create trigger trg_del_subscription_cnt_tag
35     before delete on tbl_tag
36     for each row
37 begin
38     delete from tbl_subscription_tag
39     where id_tag = OLD.id;
40 end$
41 #####
```



#### **4..6.2 Explanation**

These triggers delete all subscriptions pointing to content that's about to be deleted.

## 5. Database test data initialization

### 5.1 initialize

#### 5.1.1 Code

```
1  #####
2  # PROCEDURE
3  # * Initialization procedure
4  # * Create necessary data for veeForum initialization
5  #####
6  create procedure initialize_veeForum()
7  begin
8      # Create Superadmin group (ID: 1)
9      insert into tbl_group
10         (id_parent, name, is_superadmin, can_manage_sections, can_manage_users,
11          can_manage_groups, can_manage_permissions)
12         values(null, 'Superadmin', true, true, true, true, true);
13
14      # Create Basic group (ID: 2) (default registration group)
15      insert into tbl_group
16         (id_parent, name, is_superadmin, can_manage_sections, can_manage_users,
17          can_manage_groups, can_manage_permissions)
18         values(null, 'Basic', false, false, false, false, false);
19
20      # Create SuperAdmin user (ID: 1) with (admin, admin) credentials
21      insert into tbl_user
22         (id_group, username, password_hash, email, registration_date, firstname,
23          lastname, birth_date)
24         values(1, 'admin', '21232f297a57a5a743894a0e4a801fc3',
25              'vittorio.romeo@outlook.com', curdate(), 'Vittorio', 'Romeo', curdate());
26
27      # Insert log message with the date of the forum framework installation
28      insert into tbl_log
29         (type, creation_timestamp, value)
30         values(0, now(), 'veeForum initialized');
31 end$
32 #####
33
34
35
36 #####
37 # PROCEDURE
38 # * Testing procedure
39 # * Create some test data to speed up development/testing
40 #####
```

```

41 create procedure create_test_data()
42 begin
43     insert into tbl_user
44         (id_group, username, password_hash, email, registration_date)
45         values(2, 'user1', 'pass1', 'email1', curdate());
46
47     insert into tbl_user
48         (id_group, username, password_hash, email, registration_date)
49         values(2, 'user2', 'pass2', 'email2', curdate());
50
51     insert into tbl_section
52         (id_parent, name)
53         values(null, 'section1');
54
55     insert into tbl_group_section_permission
56         (id_group, id_section, can_view, can_post, can_create_thread, can_delete_post,
57         can_delete_thread, can_delete_section)
58         values(1, 1, true, true, true, true, true, true);
59
60     call mk_subscription_user(2, 3);
61 end$
62 #####
63
64
65
66 #####
67 # COMMANDS
68 # * Initial commands required to set up veeForum
69 #####
70 call initialize_veeForum()$
71 call create_test_data()$
72 #####
73
74
75
76 #####
77 # Copyright (c) 2013-2015 Vittorio Romeo
78 # License: Academic Free License ("AFL") v. 3.0
79 # AFL License page: http://opensource.org/licenses/AFL-3.0
80 #####
81 # http://vittorioromeo.info
82 # vittorio.romeo@outlook.com
83 #####

```

### 5..1.2 Explanation

The `initialize` script generates initial data to allow administrator login and forum system testing.

The test data consists of two test users and an empty section.

One of the users subscribes to the other one, in order to quickly test the notification system.

# Chapter 7

## PHP

**Object-oriented design** will be used as much as possible in the PHP5 backend code. The web application will be divided in two major modules: **library** and **core**.

The library module will contain functions and classes used throughout the whole application.

The web module will contain the actual web pages, divided in individual self-contained modules.

### 1. Library module

The **library PHP module** interfaces with the database, provides HTML-generation function and has additional utilities used in the core web application implementation.

**Session-stored variables** will be managed through a static **Session** class, using statically-stored keys, creating a safe interface and making debugging easier.

**Debugging** will be handled through a static **Debug** class. Logging of errors and query information can be enabled and disabled from administrators, and will be automatically displayed using AJAX.

The **database connection** will be managed using the `mysqli` PHP5 module. Every global database operation such as queries and connection will be wrapped in a safe interface that allows easy debugging and prevents security breaches.

**Privileges and permissions** will be loaded/saved from/to the database using bitset-like class instances that support all basic bitset operations. Their underlying implementation is separated from their API this allows developers to optimize or modify the bit storage without affecting code in the web module.

AJAX and shortcut functions for HTML generation will be handled through the **Gen** static class and the **Actions** static class.

AJAX requests will directly call functions (if valid) from the **Actions** class, which return

HTML, JSON, or plain text. Gen functions will be used from the web module to make the page structure more modular and avoid markup duplication.

Signing in and out and current user data will be managed from the `Credentials` static class. It will contain easy-to-use functions to check privileges and permissions, and also to handle login/logout.

Last, but not least, **database table interaction** will be handled by a very developer-friendly object-oriented interface. Every table in the database will have a corresponding class, derived from a generic `Table` class.

The `Table` class provides an object-oriented interface for common queries and **CRUD operations**. It also provides some very convenient methods to perform an action on every row matching a specific predicate or every row that's part of a hierarchy.

Their usage, combined with **PHP5 lambda functions**, will make usually complex hierarchy-traversing operations easy to write and debug. These functions are available for every table in the database. The classes derived from `Table` will implement functionality that is unique for specific database entities. Insertion and edit fields will be specified in the constructor of these classes, allowing the developer to use a very convenient and clean syntax for the insertion/editing of table rows.

## 1.1.1 settings

The **settings** submodule uses an intermediate server JSON storage to load and save system-wide settings.

The JSON file contains two properties:

- **forumName**: name of the forum, displayed in the navigation bar and HTML `<head>` tag.
- **defaultGroup**: id of the group newly registered users are inserted into.

The properties mentioned above are accessible through the following PHP interface:

---

```
1  <?php
2
3  class Settings
4  {
5      // Initialize settings
6      public static function init();
7
8      // Load/save settings from/to JSON
9      public static function loadFromFile();
10     public static function saveToFile();
```

```

11
12     // Get/set settings
13     public static function setForumName($mX);
14     public static function setDefaultGroup($mX);
15     public static function getForumName();
16     public static function getDefaultGroup();
17 }
18
19 ?>

```

---

## 1..2 session

The **session** submodule provides useful functions to manage session-stored variables in a convenient and type-safe way.

An enumeration-like class is defined for session key-value pairs keys:

```

1  <?php
2
3  class SK
4  {
5      // Example keys
6      public static $userID = "A1";
7      public static $debugLog = "A2";
8      public static $debugEnabled = "A3";
9      public static $pageID = "pageid";
10     public static $threadID = "threadid";
11 }
12
13 ?>

```

---

Accessing session variables is then done through the following PHP static interface:

```

1  <?php
2
3  class Session
4  {
5      // Initialize session
6      public static function init();
7
8      // Get/set session variables
9      public static function get($mX);
10     public static function set($mX, $mVal);
11 }

```

12

13 `?>`

---

## 1..3 debug

The **debug** submodule gives the developer a convenient interface to toggle debugging features and access the debug log.

---

```
1  <?php
2
3  class Debug
4  {
5      // Toggle debug mode
6      public static function setEnabled($mX);
7      public static function isEnabled();
8
9      // Clear log
10     public static function clear();
11
12     // Logging functions
13     public static function lo($mX);
14     public static function loLn();
15     public static function echoLo();
16 }
17
18 ?>
```

---

## 1..4 db

The **db** submodule provides a friendly interface to the database backend, abstracting most common queries and correctly handling quoted or null arguments. Its public interface is shown below:

---

```
1  <?php
2
3  class DB
4  {
5      // Connects to the database
6      public static function connect();
7
8      // Executes a query and returns its result
9      public static function query($mX);
```



```

10
11     // Returns the last inserted ID in the database
12     public static function getInsertedID();
13
14     // Returns a correctly escaped version of a string
15     public static function esc($mX);
16
17     // Returns a correctly quoted version of a value
18     public static function v($mX);
19 }
20
21 ?>

```

---

## 1..5 privs

The **privs** submodule deals with system-wide privileges. Privileges are handled as bitsets.

The **Privs** static enumeration-like class assigns an unique integer to every privilege bit:

```

1  <?php
2
3  class Privs
4  {
5      const count = 5;
6
7      const isSuperAdmin = 0;
8      const canManageSections = 1;
9      const canManageUsers = 2;
10     const canManageGroups = 3;
11     const canManagePermissions = 4;
12 }
13
14 ?>

```

---

Privilege bitsets are **PrivSet** instances. They provide functions available in most bitset implementations.

The public interface is shown below:

```

1  <?php
2
3  class PrivSet
4  {
5      // Variadic constructor - constructs an
6      // instance of 'PrivSet' with the passed privileges

```

```

7     public function __construct(...$mPrivs);
8
9     // Instantiates a 'PrivSet' from a string
10    public static function fromStr($mX);
11
12    // Instantiates a 'PrivSet' from a group
13    public static function fromGroup($mX);
14
15    // Returns a string representing the current 'PrivSet'
16    public function toString();
17
18    // Add/delete a privilege
19    public function add($mX);
20    public function del($mX);
21
22    // Check availability of a privilege
23    public function has($mX);
24
25    // Returns true if this 'PrivSet' is equal to another one
26    public function isEqualTo($mX);
27
28    // Returns the logical or between two 'PrivSet' instances
29    public function getOrWith($mX);
30
31    // Returns the logical and between two 'PrivSet' instances
32    public function getAndWith($mX);
33 }
34
35 ?>

```

---

## 1..6 pages

The **pages** submodule provides a simple framework for web application paging.

The PK static enumeration-like class assigns an unique integer to every page:

---

```

1  <?php
2
3  class PK
4  {
5      public static $sections = 0;
6      public static $administration = 1;
7      public static $threadView = 2;
8  }
9
10 ?>

```

---

Every page has its own `PageData` instance, which stores its URL and required access privileges.

---

```
1  <?php
2
3  class PageData
4  {
5      // Variadic constructor - takes the URL of the page
6      // and any number of required privileges as parameters
7      public function __construct($mURL, ...$mPrivs);
8
9      // Returns the URL of the page
10     public function getURL();
11
12     // Returns the required access privileges of the page
13     public function getPrivs();
14
15     // Returns true if the passed privileges are enough
16     // to access the page
17     public function canViewWithPrivs($mX);
18 }
19
20 ?>
```

---

All `PageData` instances are stored in a static `Pages` class:

---

```
1  <?php
2
3  class Pages
4  {
5      // Adds a page to the storage
6      // Forwards the variadic arguments to the 'PageData' constructor
7      public static function add(...$mArgs);
8
9      // Gets a 'PageData' by unique page id
10     public static function get($mX);
11
12     // Gets or sets the current page in session
13     public static function setCurrent($mX);
14     public static function getCurrent();
15 }
16
17 ?>
```

---

Web application pages can then be added using the following syntax:

---

```
1  <?php
2
3  Pages::add("php/core/content/sections.php");
4  Pages::add("php/core/content/adminPanel.php", Privs::isSuperAdmin);
5  Pages::add("php/core/content/threadView.php");
6
7  ?>
```

---

## 1..7 utils

Self-documenting static class containing various utility functions.

---

```
1  <?php
2
3  class Utils
4  {
5      // Converts an array to a comma-separated-list
6      public static function getCSL($mArray);
7
8      // Calculates the hash for a password
9      public static function getPwdHash($mX);
10
11     // Returns false if the string is not valid, empty, or
12     // only whitespace
13     public static function checkEmptyStr($mX, &$mMsg);
14
15     // Returns the parent of the last inserted record in
16     // the database (can be null)
17     public static function getInsertParent(&$mTbl, $mIDParent);
18 }
19
20 ?>
```

---

## 1..8 gen

The **gen** submodule provides a complex HTML generation system that builds a hierarchy of polymorphic PHP class instances.

AJAX-enabled HTML can then be generated by traversing the hierarchy recursively.

**ControlBase** is a class that represents a control hierarchy. Its functions can be used to access/edit the control hierarchy, to move around in the tree or to generate HTML.

Shortcut functions starting in **in** go one level deeper in the hierarchy. Shortcut functions starting in **out** go one level above in the hierarchy. Shortcut functions starting in **for** execute a callable object while traversing the hierarchy.

---

```
1  <?php
2
3  class ControlBase
4  {
5      // Add a child to this control
6      public function &add(&$mChild);
7
8      // Go one level above
9      public function &out();
10
11     // Go to the root of the hierarchy
12     public function &root();
13
14     // Parses and includes a PHP file as a child control
15     public function &file($mX);
16
17     // Prints the entire hierarchy as HTML
18     public function &printRoot();
19
20     // Executes the function/lambda for every children
21     public function &forChildren($mFn);
22
23     // Executes the function/lambda for every children (recursively)
24     public function &forChildrenRecursive($mFn);
25
26     // Executes the function/lambda for every parent (recursively)
27     public function &forParentRecursive($mFn);
28
29     // Shortcuts for common HTML elements
30     public function &literal(...$mArgs);
31     public function &inDiv(...$mArgs);
32     public function &inSpan(...$mArgs);
33     public function &strong($mX);
34     public function &h($mHLevel, $mX);
35     public function &hr();
36     public function &br();
37     public function &inFooter(...$mArgs);
38     public function &inA(...$mArgs);
39     public function &label($mFor, $mCaption);
40
41     // Shortcuts for common Bootstrap elements
42     public function &bsIcon($mIcon);
```

```

43     public function &inBSLinkBtn($mID, $mClass = '');
44     public function &inBSLinkBtnActive($mID, $mOnClick, $mClass = '');
45     public function &inBSLinkBtnCloseModal();
46     public function &bsLinkBtnAddDismissModal();
47     public function &inBSModal($mID);
48     public function &inBSModalHeader($mTitle);
49     public function &inBSModalBody();
50     public function &inBSModalFooter();
51     public function &inBSBtnGroup($mClass);
52     public function &inBSPanelNoHeader($mClass = '');
53     public function &inBSPanelWithHeader($mHeader);
54     public function &inBSTable($mID);
55     public function &inBSNavbarTextbox($mID, $mCaption);
56     public function &bsNavbarTextbox($mID, $mCaption);
57     public function &inBSFormTextbox($mID, $mCaption);
58     public function &bsFormTextbox($mID, $mCaption);
59     public function &bsFormTextarea($mID, $mCaption, $mRows);
60 }
61
62  ?>

```

---

Complex elements that require additional stored data or special functions can be defined as classes that derive from `ControlBase`.

```

1  <?php
2
3  // Example derived controls
4  class Container extends ControlBase;
5  class Literal extends ControlBase;
6  class HTMLControl extends ControlBase;
7  class BSModal extends Container;
8
9  ?>

```

---

Here's an example usage of the HTML generation module:

```

1  <?php
2
3  (new Container())
4      ->h(1, 'Administration')
5      ->hr()
6      ->inDiv(['class' => 'row'])
7          ->file("$rootAP/panelDebug.php")
8          ->file("$rootAP/panelGroups.php")

```

```

9         ->out()
10     ->hr()
11     ->inDiv(['class' => 'row'])
12         ->file("$rootAP/panelGSPerms.php")
13         ->file("$rootAP/panelSections.php")
14         ->out()
15     ->hr()
16     ->inDiv(['class' => 'row'])
17         ->file("$rootAP/panelUsers.php")
18 ->printRoot();
19
20 ?>

```

---

## 1..9 tbl

The **tbl** submodule provides a complex and fully-featured database table wrapping and management system for the PHP backend.

Every database table is defined in the PHP backend as a class deriving from **Tbl**.

**Tbl** is an extremely powerful abstraction that offers developers an huge amount of convenient functions to manage the records of a database table:

---

```

1  <?php
2
3  class Tbl
4  {
5      // Constructs a 'Tbl' instance with a specific name
6      // and a set of fields for value insertion
7      public function __construct($mTblName, ...$mInsertFields);
8
9      // Set the fields required to insert a new value
10     public function setInsertFields(...$mFields);
11
12     // Inserts a new record with the passed variadic values in
13     // the database
14     public function insert(...$mValues);
15
16     // Inserts a new record with the passed variadic values in
17     // the database (correctly escapes the passed arguments)
18     public function insertValues(...$mValues);
19
20     // Finds a record by ID and updates it
21     public function updateByID($mID, $mArray);
22
23     // Returns all records in an array

```

```

24     public function getAll();
25
26     // Returns all records matching a specific predicate in an array
27     public function getWhere($mX);
28
29     // Returns the first record
30     public function getFirst($mX);
31
32     // Returns the first record matching a specific predicate
33     public function getFirstWhere($mX);
34
35     // Deletes all records matching a specific predicate
36     public function deleteWhere($mX);
37
38     // Finds a record by ID and deletes it
39     public function deleteByID($mID);
40
41     // Finds a record by ID and deletes its hierarchy, if existing
42     public function deleteRecursiveByID($mID);
43
44     // Finds and returns a record by ID
45     public function findByID($mID);
46
47     // Finds all children of a specific record by ID
48     public function findAllByIDParent($mIDParent);
49
50     // Returns true if any record matches the predicate
51     public function hasAnywhere($mX);
52
53     // Returns true if a record with the passed ID exists
54     public function hasID($mID);
55
56     // Executes the passed function/lambda on every record
57     public function forRows($mFn);
58
59     // Executes the passed function/lambda on every record matching
60     // a specific predicate
61     public function forWhere($mFn, $mWhere);
62
63     // Executes the passed function/lambda on every child of a
64     // record
65     public function forChildren($mFn, $mIDParent = null, $mDepth = 0);
66
67     // Executes the passed function/lambda on every parent of a
68     // record
69     public function forParent($mFn, $mID, $mDepth = 0);
70 }

```



71  
72 `?>`

---

Database tables can then be wrapped and instantiated in the following way:

---

```
1  <?php
2
3  // ...
4
5  TBS::$log = new TblLog('tbl_log',
6      'type', 'creation_timestamp', 'value');
7
8  TBS::$tag = new TblTag('tbl_tag',
9      'value');
10
11 TBS::$section = new TblSection('tbl_section',
12     'id_parent', 'name');
13
14 TBS::$cntBase = new Tbl('tbl_content_base');
15 TBS::$cntThread = new TblCntThread('tbl_content_thread');
16 TBS::$cntPost = new TblCntPost('tbl_content_post');
17 TBS::$cntAttachment = new TblCntAttachment('tbl_content_attachment');
18
19 // ...
20
21 ?>
```

---

## 1..10 sprocs

The **sprocs** submodule provides a powerful and convenient abstraction to manage and call MySQL stored procedures from the PHP backend.

Every database stored procedure can be wrapped in a **SProc** instance.

The user-friendly yet extremely useful **SProc** public interface is shown below:

---

```
1  <?php
2
3  class SProc
4  {
5      // Constructs a stored procedure with a specific name,
6      // and a number of in/out parameters
7      public function __construct($mProcedureName,
8          $mInParamCount, $mOutParamCount);
9  }
```

```

10      // Forwards the variadic arguments to the stored procedure
11      // and calls it
12      public function call(...$mArgs);
13
14      // Forwards the variadic arguments to the stored procedure
15      // and calls it, returning its out parameters in an array
16      public function callOut(&$mOutArray, ...$mArgs);
17  }
18
19  ?>

```

---

Stored procedures can then be instantiated and wrapped in the web application code like this:

```

1  <?php
2
3  // ...
4
5  SPRCS::$mkContentThread = new SProc('mk_content_thread', 3, 0);
6  SPRCS::$mkContentPost = new SProc('mk_content_post', 3, 0);
7  SPRCS::$mkContentAttachment = new SProc('mk_content_attachment', 3, 0);
8
9  SPRCS::$mkSubscriptionUser = new SProc('mk_subscription_user', 2, 0);
10 SPRCS::$mkSubscriptionThread = new SProc('mk_subscription_thread', 2, 0);
11 SPRCS::$mkSubscriptionTag = new SProc('mk_subscription_tag', 2, 0);
12
13 // ...
14
15 ?>

```

---

## 2. Core module

The **core PHP module** contains the implementation of the default **veeForum web application**, which makes use of all the library module features described above.

Its folder structure is as follows:

```

1  .
2  body
3  footer.php
4  loginControls.php
5  modalInfo.php
6  modalNotifications.php

```

```
7     navbarContents.php
8     navbar.php
9     profileControls.php
10    body.php
11    content
12     actions.php
13     adminPanel
14     modalGSPerms.php
15     modalUserActions.php
16     modalUserEdit.php
17     panelDebug.php
18     panelGroups.php
19     panelGSPerms.php
20     panelSections.php
21     panelTags.php
22     panelUsers.php
23     adminPanel.php
24     forbidden.php
25     register
26     modalRegister.php
27     sections
28     modalNewPost.php
29     modalNewThread.php
30     sections.php
31     threadView.php
32    head.php
```

---

The `body` folder contains the implementation of the main page's modules, such as information/error modals, the navigation bar, etc.

The `content` folder contains several subfolders:

- `adminPanel`: implementation of the administration section and all its modules.
- `register`: implementation of the registration interface.
- `sections`: implementation of the **new post** and **new thread** interfaces.

Screenshots of the web interface will be shown in the following chapter, showing the end-result of the combination of the PHP modules.

# Chapter 8

## Web interface

Figure 8.1: Main page - user not logged in.

testForum

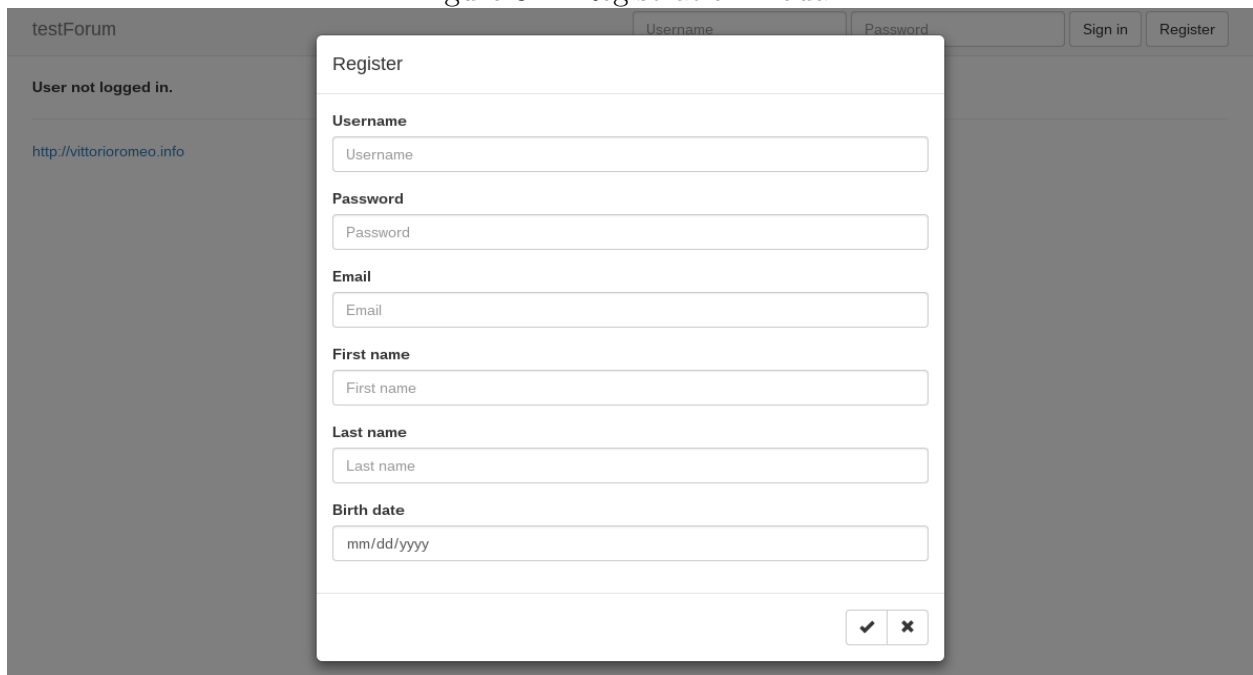
Username	Password	Sign in	Register
----------	----------	---------	----------

User not logged in.

---

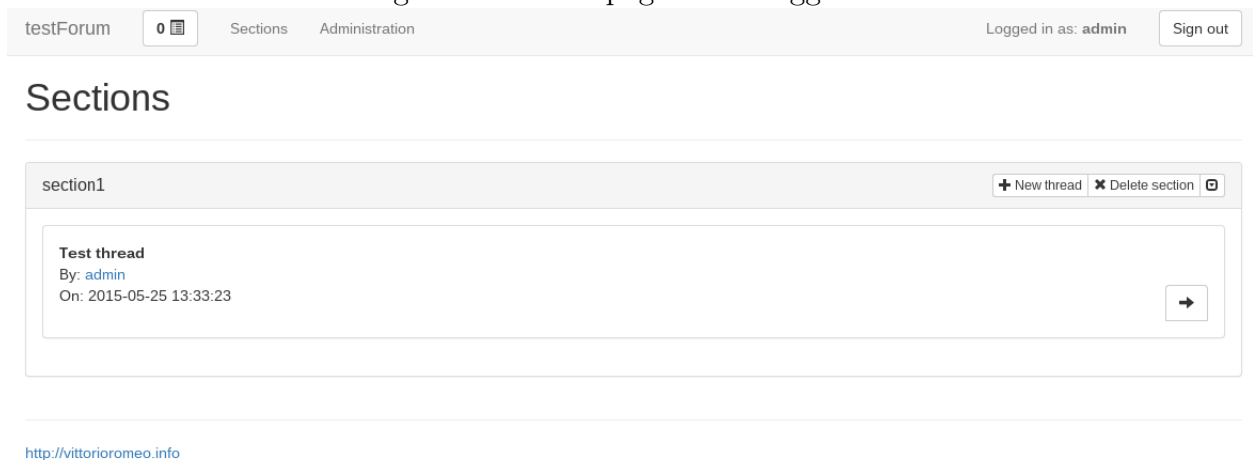
<http://vittorioromeo.info>

Figure 8.2: Registration modal.



The image shows a registration modal window titled "Register" overlaid on a forum page. The modal contains several input fields: "Username", "Password", "Email", "First name", "Last name", and "Birth date" (with a placeholder "mm/dd/yyyy"). At the bottom right of the modal are two buttons: a checkmark icon and an "X" icon. The background forum page shows the header "testForum" with "Sign in" and "Register" buttons, and a sidebar with the text "User not logged in." and a link "http://vittorioromeo.info".

Figure 8.3: Main page - user logged in.



The image displays the main page of the "testForum" when a user is logged in. The top navigation bar includes the forum name "testForum", a menu icon, and links for "Sections" and "Administration". On the right, it shows "Logged in as: admin" and a "Sign out" button. The main content area is titled "Sections" and features a section named "section1". Within this section, there is a "Test thread" by "admin" posted on "2015-05-25 13:33:23". Above the thread, there are buttons for "+ New thread", "X Delete section", and a trash icon. A right arrow button is located next to the thread information. At the bottom of the page, the URL "http://vittorioromeo.info" is displayed.

Figure 8.4: New thread modal.

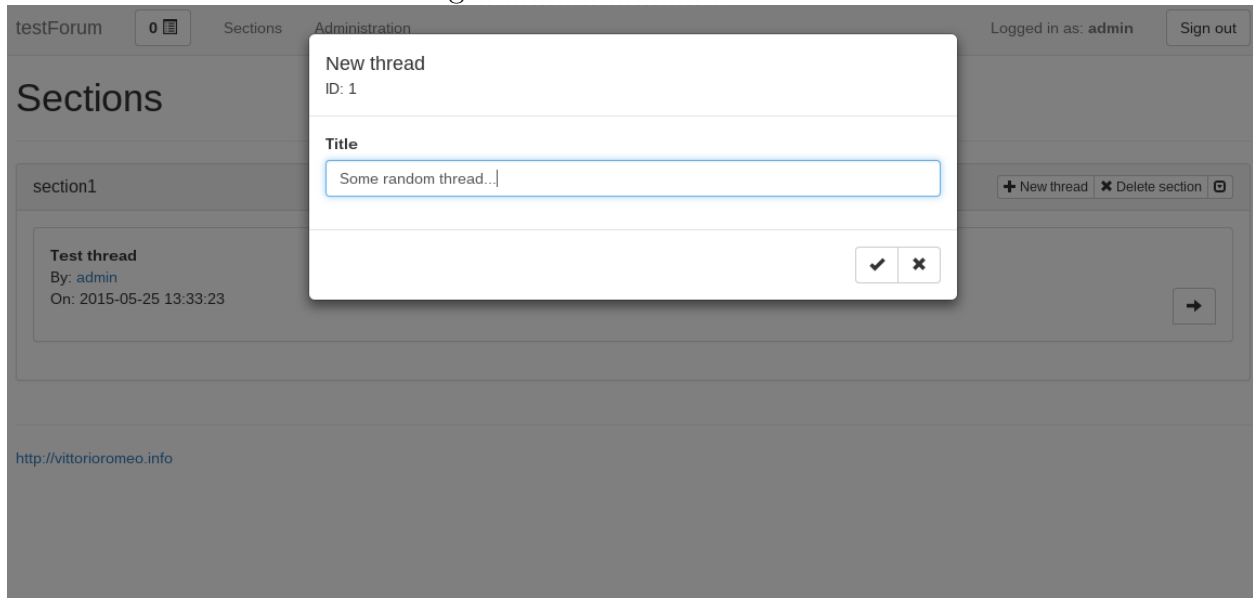


Figure 8.5: Thread view page.

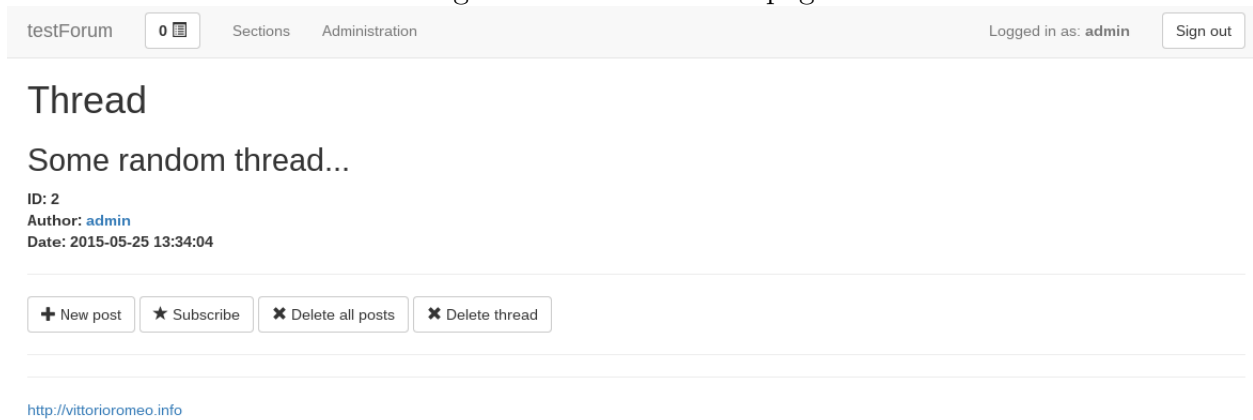


Figure 8.6: New post modal.

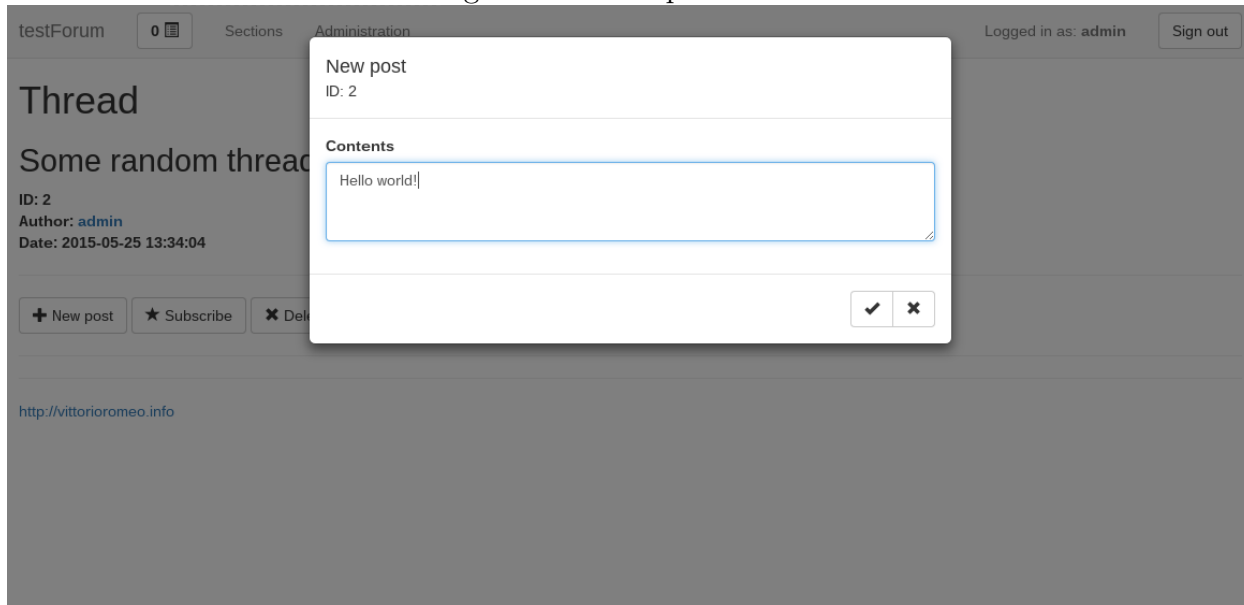


Figure 8.7: Thread view page - subscription.

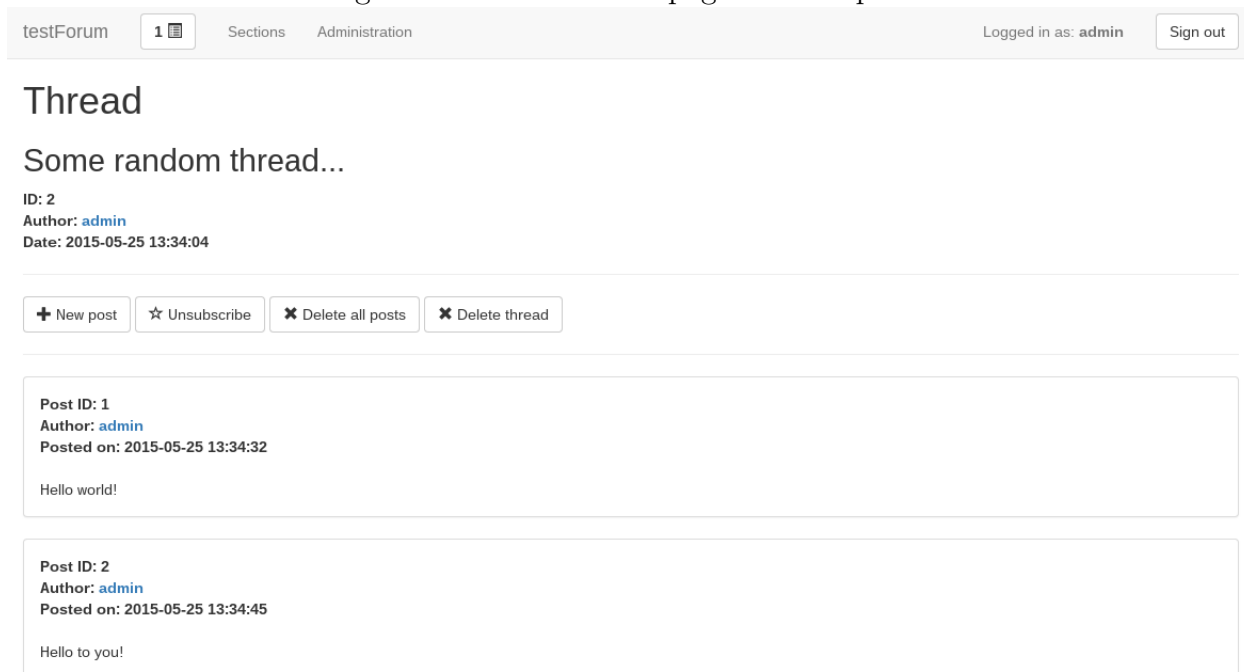


Figure 8.8: Notifications modal.

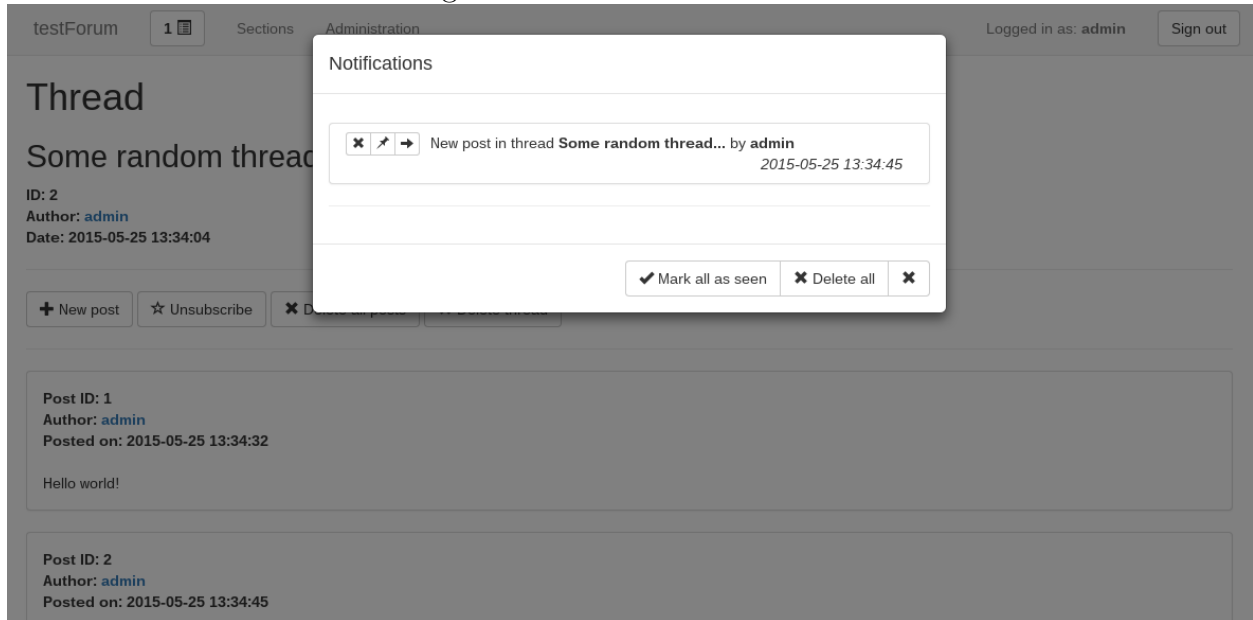


Figure 8.9: Administration panel - 1.

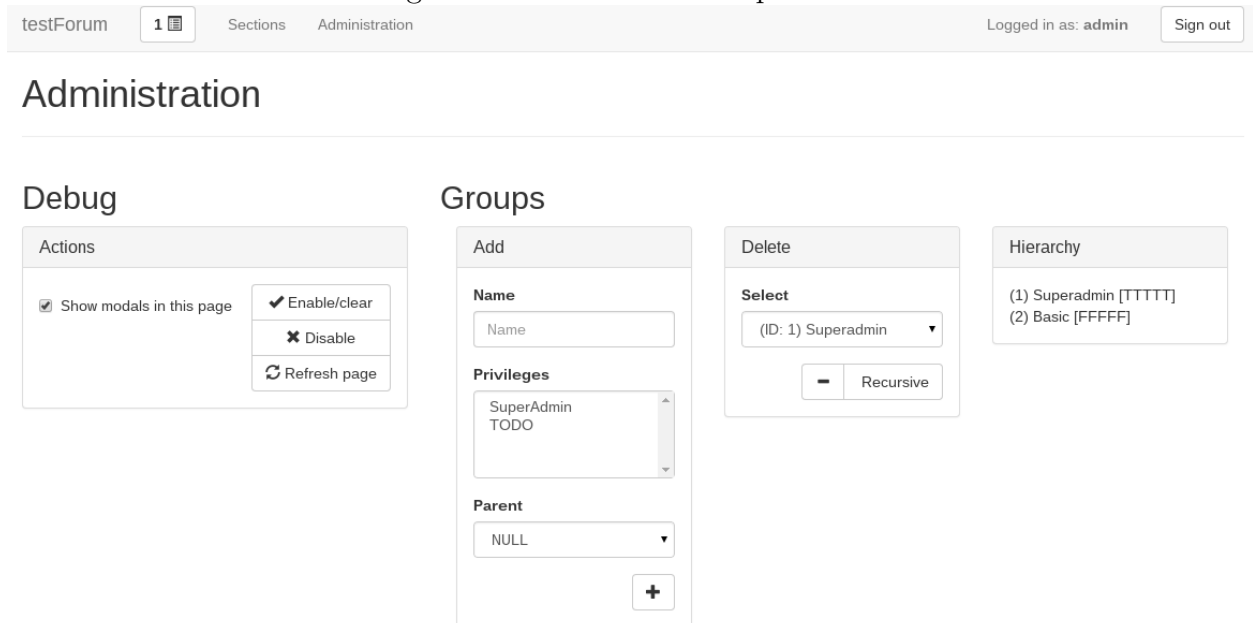




Figure 8.10: Administration panel - 2.

testForum 1 Sections Administration Logged in as: admin Sign out

### Group permissions

Manage

Group  
(ID: 1) Superadmin

Section  
(ID: 1) section1

→

### Sections

Add

Name  
Name

Parent  
NULL

+

Delete

Select  
(ID: 1) section1

- Recursive

Hierarchy

(1) section1

Figure 8.11: Administration panel - 3.

testForum 1 Sections Administration Logged in as: admin Sign out

(ID: 1) section1

→

NULL

+

### Users

Manage

+	ID	Username	Email	Group	First name	Last name	Registration date	Birth date
* ✎	1	admin	vittorio.romeo@outlook.com	Superadmin	Vittorio	Romeo	2015-05-25	2015-05-25
* ✎	2	user1	email1	Basic			2015-05-25	
* ✎	3	user2	email2	Basic			2015-05-25	
* ✎	4	test	test	Basic	test	test	2015-05-25	0000-00-00

<http://vittorioromeo.info>

# **Part III**

## **Conclusion**

aaa

# Chapter 9

## Final product

aaa

# Chapter 10

## What I learned

aaa

# Chapter 11

## Future

aaa

# Chapter 12

## References

aaa