CodEx – Design Manual

By the CodEx Team. Revision: 1474

Martin Mares
Tomas Gavenciak
Martin Krulis
Jiri Svoboda
Lukas Turek

Table of Contents

1 Introduction

1.1 An Introduction to Automatic Grading Systems

Since checking programs written by studens as their homework is very tedious and time-consuming, the idea of automating this process has been around for quite some time. In 1965 G. E. Forsythe and N. Wirth of Stanford University devised a system where the students wrote sub-routines in a dialect of Algol, submitted them on punch cards and these were evaluated by an automatic grading program.

Kassandra was an automatic grading system developed at ETH Zurich in the 1990s. Similar to Forsythe and Wirth's design, it had improved security, as the tested program ran seperately from the grader and communicated with it through network sockets. It also sported a window-based GUI for ease of use. It was able to evaluate solutions written in Matlab, Maple and Oberon (a descendant of Modula-2). A slight drawback of the system was that the students needed to use a special module provided by Kassandra to communicate with the grader.

Generally an automatic grading system can evaluate the source code statically (i.e. by examining the code without running it, for example assessing the coding style) or dynamically, running the code on some testing data. The testing data can be provided by the author of the problem, or sometimes even generated on the fly.

1.2 Overview of the CodEx System

CodEx is a system for automatic evaluation of source code based on dynamic analysis. Via its user interface, algorithmic problems can be entered for solving by other users (e.g. students), who submit their solutions to the problem in the form of a source-code file. The solutions are compiled and run in a sandboxed environment (MO-Eval) on a series of test inputs (provided by the author of the problem). The solutions are graded depending on whether they produce the correct output within the prescribed time and memory limits.

For evaluation we make use of the *MO Contest Environment* a.k.a. MO-Eval, used at the Czech Olympiad in programming (`http://mo.mff.cuni.cz/p/index.html.en`), which isolates the tested program from the system by running it in a sandbox, measures the time and memory consumed and makes sure the program does not exceed the imposed time and memory limits.

The system can currently evaluate solutions written in the C, C++, C#, GNU Pascal and Free Pascal languages, but as adding more languages is fairly easy, support for more languages is expected in the future.

In CodEx, some users create *exercises* which represent the problems to be solved and create *groups* which tie together a set of exercises (called *tasks*) and users (the *members* of the group). The members solve tasks for which they are awarded points. If they obtain the prescribed number of points and complete all obligatory tasks, then they have met the *group requirements*.

Among the main strengths of CodEx is its comfortable web-based interface, allowing the users to enter and solve problems from practically anywhere. CodEx is very modular and it can be easily run in a distributed fashion, making it highly scalable. Finally, having its security based on a sandboxing/virtualization principle, CodEx allows the users to solve the problems in a standard programming environment, bypassing the need for them to get aquainted with any special libraries or protocols.

1.3 Development Timeline

October 2007
: Agreed upon using the Zend Framework. MVC development started.

November 2007 – February 2008
: Development of C# evaluation support.

February – April 2008
: Architecture design.

April - July 2008
: Preliminary work. Implementation of libraries and *components*.

July 9th, 2008
: Principal development started.

July 2008 Implementation of individual modules and pages.

August 2008
: Debugging, testing, documentation, translation.

1.4 CodEx Components and Dependencies

From a birds-eye perspective, CodEx consists of a WWW front-end (running on the LAMP platform) and an evaluation back-end that runs asynchronously from the front-end. A more detailed view of its components and other sofware packages employed by the system follows.

1.4.1 Components of the CodEx Project

All components of CodEx project are licensed under GNU General Public License version 2 or later (`http://www.gnu.org/copyleft/gpl.html`).

WWW front-end
: User interface, written in PHP.

Queue Manager (Qman)
: Schedules jobs for evaluation, written in C.

Judges and Filters
: Simple programs in C which verify the output of evaluated programs.

pthread-fake
: Library that replaces pthread in GNU libc. Required for C# evaluation.

1.4.2 Third-party Components Distributed with CodEx

Zend Framework 1.5.3
: PHP development framework. Contains essential libraries used by the WWW front-end. Available at `http://framework.zend.com/`, licensed under the 3-clause BSD license.

libucw Library from Sherlock Holmes Search Engine (`http://www.ucw.cz/holmes/`), licensed under the GNU Lesser General Public Licence (LGPL). Required by *Qman*.

The MO Contest Environment (a.k.a. MO-Eval)
: Available at `http://mj.ucw.cz/mo-eval/`, licensed under the GNU General Public License version 2. CodEx uses a slightly modified version of MO-Eval as a worker that evaluates submitted source code.

Mono 1.2.5.1
: C# compiler and MSIL interpreter, available at http://www.mono-project.com/. Required for C# evaluation. CodEx uses a slightly patched version, the original version is not compatible with sandbox.

Reader and Wrapper
: C# classes that are linked to every evaluated C# program. Released as public domain.

1.4.3 External Dependencies

- Linux kernel 2.6.13 or later
- Apache WWW server 2.0 or later
- PHP 5.1.6 or later (both CLI and the Apache module)
- MySQL 5.0.32 or later
- GCC 4.1 or later
- Free Pascal compiler 2.0.0 or later
- GNU Pascal Compiler 3.4.6 or later

1.5 Assessment

The major goal of this project was to create a stable application that will be used for the first-year programming course at the Faculty of Mathematics and Physics, Charles University. These ambitions were met quite completely, since CodEx has already been used and tested on Summer School for Teachers of Informatics and will be deployed at the faculty during the next scholar year.

The main goal has been accomplished, however many issues were left unfinished and will be attended in the future. GUI ergonomics are insufficient and graphic design is poor. Some unimportant issues were left undocumented for the time being and there are minor redundancies in the written code. All these issues were caused by lack of time and will be solved in the nearest future.

Even though there are some unfinished bits and pieces the application is fully operational and ready to be used in production. Thanks to huge experience from the previous version of the CodEx application that was in use on MFF last scholar year, the developement was smooth and we did not walk into any serious obstacles.

2 Web User Interface

2.1 Page Hierarchy

2.1.1 Page Overview

2.1.2 Default Module

`index` This page only contains a login form. It is displayed when the user is not logged in or when his session expires. The login form triggers a `login` action handled by this form.

This form also handles a `logout` action triggered by a *logout* button that is displayed in the page header when the user is logged in.

This page also acts as a *sink*. If user wants to access another page but does not have sufficient rights, he/she is transfered to *fallback* page. Index is transitive fallback for all pages. Therefore it respects given URI and tries to keep it in case user is just re-logging.

If this page is accessed and user is logged in, the browser is redirected to `welcome` page so the user will not try to log in again.

`welcome` The first page displayed to user when he/she logs in. The page contains a welcome message, recent news, some other useful information and links (to help pages, tutorials, FAQ, etc.).

The welcome page does not have any actions since it is only meant for reading.

`settings` A page with the user's personal settings. It contains a form where the user may change his/her account properties (including password).

Besides the properties user may yield his/her delegations. Delegations are displayed in two separated tables (one for groups and one for exercises). Every delegation is equipped with check box. A *yield* button that removes all selected delegations is placed beneath the table. It triggers `yieldQuery` action which displays confirmation dialog and (if yielding is confirmed) calls `yield` action. Both actions are handled by this page.

`mail` A page enabling a user to send a bug report or a message to another CodEx user (or multiple users). It displays a form allowing the user to enter the subject and text of the message. It also displays a fixed list of recipients that cannot be modified here (it is determined by the arguments of the page) and the automatically generated subject prefix and body preamble.

When used to report a bug, the page takes one GET parameter, `bugUrl`. All 'Report a Bug' links are generated with this argument set to the (server-relative) URL of the current page. The URL is attached to the message and allows the administrator to see from which page the bug was issued.

When used for sending messages to other users, the page takes the parameter `recipients` instead. It is an array of IDs of users who are to receive the message.

When the user is logged in, his name and e-mail address are set automatically and cannot be changed. When he is not logged in, he must fill these in.

A user who is not logged in can only send a bug report (which will be delivered to *admin*). A user who is logged in but has no special rights can send a message to a single recipient. A user possessing RIGHT_READ for users may send a message to an arbitrary number of recipients.

2.1.3 Module `groups`

All pages in this module are intended for displaying and managing groups and everything related such as tasks, submits, points etc.

Only group members or users with at least `RIGHT_READ` for a given group may see the following pages (except for the `index` page). Some of these pages also require additional rights.

`index` List of available groups. Groups are displayed in three separated lists:

- member groups (i.e. groups where the user has membership)
- owned groups
- other groups (Naturally, the user can only see public groups or groups for which he/she has at least `RIGHT_READ`.)

A *join* button is displayed next to public groups which the user is not a member of. This button generates a `join` action that is handled by this page.

If the user has sufficient rights for a group (at least `RIGHT_DELETE`) he/she may also see a *delete* button next to the group. It generates a `delete` action that is also handled by this page.

`create` A form for creating new groups. The creation itself is handled by this page and, if completed successfully, the browser is redirected to the `index` page.

This page is only visible for users with at least `RIGHT_CREATE_PRIVATE` general right for groups.

`results` A table displaying an overview of results. One row is displayed for each group member, showing the points for every task in the group, bonus points and final point totals.

This page is only visible if the group is not discreet (see the `groups.discreet` database field) or by users who have at least `RIGHT_READ` for this group.

If the user has `RIGHT_EDIT` for the group, a *Remove* button is displayed in each row of the table, allowing to renive the user from the group. This button generates a `remove` action that is also handled by this page.

The page requires the GET parameter `groupId` to hold the ID of the group being displayed. If this ID is omitted, the results from all groups (more precisely, all groups visible to the logged user) are displayed.

`properties`

A form where group properties (point limit, flags, comment ...) can be edited. It also displays a table of users holding some delegations for this group. Each delegation line is equipped with a *revoke* button that triggers a `revoke` action (handled by this page). Delegations can only be revoked here. The only place where they can be granted is at the `users/rights` page.

At the bottom of the page, there is an additional form that allows changing the owner of the group. This form is visible only to users with `RIGHT_ADMIN` for the group. When the owner is changed and *Keep Admin Rights* box is checked, admin rights are delegated to former owner. Granter of these rights is implicitly an actor. If the actor is also the former owner, rights are delegated by admin (since nobody may grant rights to himself/herself).

This page is only visible for users with at least `RIGHT_EDIT` and the delegation part requires even `RIGHT_ADMIN` since only the *admin* or owner can grant and revoke delegations. The page also requires the GET parameter `groupId` to hold the ID of the group being displayed. If this parameter is omitted or invalid, the browser is redirected to the `index` page.

`bonusPoints`

A list of bonus points for all users and a simple form that allows granting bonus points to multiple users at once. Bonus points are arranged in a table where each row represents one user and columns are bonus points aggregated by their comments (bonus points with the same comment are displayed in one column).

If the user has no special rights and the group is *discreet* (see `groups.discreet` database field), he/she can only see his/her bonus points. Users with `RIGHT_READ` can see all bonus points and users with `RIGHT_EDIT` may even grant and revoke them.

Users with `RIGHT_EDIT` can see the points in text edit boxes so they can edit and revoke them. There's also a simple form underneath that allows to create a new column (comment group) in the table. The new column is created empty and it will not be saved into database unless some points are entered into it.

2.1.3.1 Module `groups/tasks`

All the following pages require the GET parameter `groupId` to hold the ID of the group being operated on. If this parameter is omitted or invalid, the browser is redirected to the `groups/index` page.

These pages are only visible to group members or users with `RIGHT_READ` for the selected group.

`index` A list of tasks assigned to the selected group. The user can select a task by clicking on, so that he/she can see its specification, submit a solution etc.

If the user has at least `RIGHT_EDIT` for the selected group, a *delete* button is displayed at each task. This button generates a `delete` action which is handled by this page.

`specification`

Page displaying the specification of the task (exercise). The specification an XHTML text with all instructions necessary for a user who wants to solve this task (problem description, input/output format, etc).

This page requires the GET parameter `taskId` that specifies the ID of the task whose specification is being displayed. If omitted, the browser is redirected to the `index` page.

`newSubmit`

A form that allows the user to submit a new solution for this task. The page is responsible for receiving the submitted source code, creating a new entry in the database and inserting int into the eval queue.

The user can only submit one source file with the proper extension. The file can either be uploaded or its content can be entered directly into a text field. If the submit is received successfully, the browser is redirected to the `submits` page.

This page is only visible for group members. When an operator wants to test the exercise before it is being assigned he/she may use exercises/solutions page to do so.

`properties`

Overview of the task's properties (points, deadline, comment, etc). Properties are displayed in a form, thus they can be directly edited and saved. The page is responsible for saving the properties into the database.

This page is only visible for users with `RIGHT_EDIT`. It also requires the GET parameter `taskId` to hold the ID of the task whose parameters are displayed and edited. If omitted, the browser is redirected to the `index` page.

`submits` A list of all submits that the user is allowed to see. Group members with no special rights can only see their own submits. Users with `RIGHT_READ` (for the selected task) can see all submits from all group members, and finally, users with `RIGHT_EDIT` may even delete and re-evaluate submits.

If the GET parameter `taskId` is specified, only submits related to that task are displayed. Otherwise, all submits for all tasks in the selected group are displayed. On this page, the user can also change which task is selected by selecting another task from a pull-down list.

The table of submits is equipped with a filter form, where the selected task may be changed and other filter restrictions applied. It is possible to restrict the listing just to submits from a given user, time period or points range.

A checkbox is displayed next to each submit for users with `RIGHT_EDIT`. These checkboxes mark submits that are affected by *delete* and *reevaluate* buttons beneath the form. Both buttons are part of the same form that encapsulates also the checkboxes. This form is also handled by this page.

`submitInfo`

More detailed information about one submit. It is visible for all users with `RIGHT_READ` for the submit. Users with `RIGHT_EDIT` may even change attached bonus points, delete the submit or send it for re-evaluation. Changing the amount of bonus points is handled by a form component. Re-evaluation and deletion is handled by the actions `reevaluate` and `delete` that are handled by this page.

The page displays summary information (time, points...), the evaluation log and the submitted source code. A user with `RIGHT_EDIT` can also see additional information from *eval* (such as consumed time and memory for each test, results from judges etc.) Some data (e.g. the evaluation log) will not be available if the submit has not been evaluated yet.

This page requires the GET parameter `submitId` to point to the submit that is being displayed. If omitted, the browser is redirected to the `submits` page.

2.1.4 Module `exercises`

Pages in this module display and manage exercises, their specifications and exemplary solutions. The user must have at least `RIGHT_READ` for exercises in order to see these pages (even the `index` page).

`index` A list of all exercises that are visible to the user. The user may open an exercise by clicking on it, which brings him to the *specification* page and sets the `exerciseId` argument, allowing him to access other exercise-specific pages.

The user can see all public exercises and those private exercises for which he/she has `RIGHT_ADMIN`. Each exercise for which the user has `RIGHT_DELETE` is equipped with a *delete* button that triggers a `delete` action. This action is also handled by this page.

`create` A form for creating new exercises. Creation itself is handled by this page and, if completed successfully, the browser is redirected to the `editText` page with `exerciseId` set to the ID of the newly created exercise.

This page is visible only for users with a general `RIGHT_CREATE` for exercises.

`properties`

A form where properties of the exercise are displayed and edited and a list of delegated rights for the selected exercise. Submission of the properties form is handled by this page.

Delegations are displayed in a table and each delegation is equipped with a *revoke* button that triggers a `revoke` action (which is also handled by this page). This is the only place where delegations can be revoked. The only place where they can be granted is the `users/rights` page.

At the bottom of the page, there is an additional form that allows changing the author of the exercise. This form is visible only to users with `RIGHT_ADMIN` for the exercise. When the author is changed and *Keep Admin Rights* box is checked, admin rights are delegated to former author. Granter of these rights is implicitly an actor. If the actor is also the former author, rights are delegated by admin (since nobody may grant rights to himself/herself).

The properties page is only visible for users with at least `RIGHT_READ` for the selected exercise. To edit the properties, however, one needs at least `RIGHT_EDIT`. Users with `RIGHT_ADMIN` can also see the delegations. The Page also requires the GET parameter `exerciseId` to hold the ID of the exercise for which the properties are displayed. If omitted, the browser is redirected to the `index` page.

`specification`
: The text of the current exercise specification and a list of all specification versions, identified by their date of creation. The user can also examine the texts of other versions by clicking on their captions in the list. With `RIGHT_EDIT` for the selected exercise the user may also delete versions and change which version is the current one. Corresponding buttons in the list generate `delete` and `setActual` actions that are handled by this page.

  Specification versions can be also be edited. An edit button directs the browser on the `editText` page with the proper GET parameters set.

  The page requires the GET parameter `exerciseId` to hold the ID of the exercise whose specification is being displayed. Also the page may receive the GET parameter `textId` that holding the ID of the specification version that should be displayed. If this parameter is omitted, the current version is displayed.

  If there are no specifications of the exercise yet, the browser is redirected to the `editText` page.

`editText`
: A form for editing the selected version of the specification. Modifications submitted by this form are handled by this page. The form may be submitted either in two ways: simply a *save* (modifications are saved into the existing specification version) or *saved as new version* (which creates new version of the specification and makes it the current one). If the exercise has no specification versions yet, the form can only be saved as a new version.

  This page is only visible for users with at least `RIGHT_EDIT` for the selected exercise. The page also requires the GET parameters `exerciseId` and `textId` that refer to the exercise and its specification that is being edited. If `textId` is omitted, the form is blank and can be only used for creating new text.

`attachedFiles`
: List of files attached to a given specification. Each specification can have its own directory tree with pictures, files for download etc. These are managed with a file manager component on this page.

  Only users with at least `RIGHT_EDIT` may modify files in the directory. The page also requires the GET parameter `exerciseId` pointing to the exercise and optionally `textId` pointing to the specification version to which the displayed directory belongs to. If `textId` is not set, the files attached to the current version are displayed. If the exercise does not have any specification, the page is inaccessible.

`configuration`

A form for editing basic test configurations. All general test properties may be modified here except points and resource limits. Configuration parameters are parsed directly from eval's config test file and when editing form is submitted the file is atomically replaced.

This page is visible for all users (with `RIGHT_READ` of course) but only users with at least `RIGHT_EDIT` may edit parameters and save them. The page also requires GET the parameter `exerciseId` which holds the ID of the exercise whose test's configuration is being edited.

`limits` Special form for modifying points per test, time limit and memory limit values. This page can either edit general limits for all extensions or limits that are designed for one extension only.

At the top of the page there are text boxes for editing general values for all tests (in given extension). They are followed by table where individual values for each test may be defined. These values are combined in following order (ordered from most to least important:

- Values defined for individual test and individual extension.
- Values defined for individual test and all extensions.
- General values for all tests in individual extension.
- General values for all test and all extensions.

The page also contains special interface that allows experts add and modify extra (non-standard) parameters to config files.

All values displayed on this page are parsed directly from exercise's config file and if the form is submitted, the file is atomically replaced.

Every user with `RIGHT_READ` has access to this page but only user with `RIGHT_EDIT` for given exercise may change its values. Page also requires `exerciseId` GET parameter that holds id of an exercise whose config file is being edited and optional `extension` parameter that holds the extension (defining programming language) for which the limits applies. If omitted, general limits are edited.

`testFiles`

Contents of the directory attached to the exercise. This directory contains files used for testing submitted solutions of the exercise (input data and corresponding output). How these files are used is defined by test configuration (see `configuration` page).

This page is visible for all users (with `RIGHT_READ`, of course) but only users with at least `RIGHT_EDIT` may modify the files. The page requires the GET parameter `exerciseId` that specifies the exercise whose test-files are being displayed (or edited). If omitted, the browser is redirected to the `index` page.

`assignTasks`

Assigning exercises (creating tasks) to groups. This page displays some information about the exercise that might be useful for a group owner who wants to use this exercise, and a list of all groups to which the user has at least `RIGHT_EDIT`. The user may check groups to which he wishes to assign the task (thus, multiple tasks may be created at a time).

The page also contains a form with basic task parameters (the deadline, points etc.) that are used as defaults for newly created tasks.

The page requires the GET parameter `exerciseId` to hold the ID of the exercise that is being assigned.

2.1.4.1 Module `exercises/solutions`

This module contains pages managing solutions of one exercise. All the following pages require the GET parameter `exerciseId`, which holds a reference to the exercise whose solutions are being displayed and managed.

`index` List of all solutions belonging to the selected exercise. The user can only see public solutions and all of his own solutions. Users with `RIGHT_ADMIN` for the current exercise can also see all private solutions.

Every solution for which the user has at least `RIGHT_EDIT` is equipped with a checkbox. These checkboxes can be used to select solutions affected by the *reevaluate* and *delete* buttons. The checkboxes and the buttons are part of a form that is also handled by this page.

`newSolution`
A form that allows the user to create (submit) a new solution. The page is responsible for processing submitted source code (saving it into a file, creating a database record and sending the solution to the evaluation system). This page is similar to the `groups/tasks/newSubmit` page.

The user can only submit one source file with the proper extension. The file can either be uploaded or its content can be entered directly into a text field. If the solution is received successfully, the browser is redirected to the `index` page.

This page is visible only users with at least `RIGHT_EDIT_BASIC` for the selected exercise. However, only users with `RIGHT_EDIT` can make their solutions public.

`solutionInfo`
More detailed information about one particular solution. Users with `RIGHT_EDIT` for the solution may send it for re-evaluation or delete it. Actions `reevaluate` and `delete` are triggered by corresponding buttons. Both are handled by this page.

The page contains summary information (time, points...), the evaluation log (with additional information such as time consumed) and the submitted source code. Some data (e.g. the evaluation log) will not be available if the solution is not evaluated yet.

If the user has RIGHT_EDIT for the solution, he can change the comment and the `public` flag.

This page requires the GET parameter `solutionId`, which points to the solution that is being displayed. If omitted, the browser is redirected to the `index` page.

2.1.5 Module `exercises/comments`

This module encapsulates all functions related to exercise comments. Exercise comments are text written by operators which contains information for other operators, experience and best practices to share.

All pages in this module requires GET parameter `exerciseId` that points to an exercise to which all comments belongs to. Furthermore all the pages are visible only to users with at least `RIGHT_READ` for selected exercise.

`index` Displays table with all exercise comments. The whole comment text content is displayed in the table. It also contains link to edit page and *delete* button that triggers the `delete` action which is embedded in the page.

The `delete` action requires `commentId` GET parameter and is protecte by `deleteQuery` query action. The query action is bypassed by JavaScript query as all the action queries are. The delete action may be performed only by users with at least `RIGHT_EDIT` for selected exercise or by comment author.

`create` Page with general form that creates new comments. Data from the form are processed by the page and after successfull submit the browser is redirected to the `index` page of current module.

Only users with at least `RIGHT_EDIT_BASIC` for selected exercise may see this page.

`edit` Page with the very same module as the `create` page. Except the form is filled by data from selected comment and when submitted the comment is updated. Submitted data are also handled by this page. After submit or cancel operation, the browser is redirected back to `index` page.

This page requires GET parameter `commentId` which points to exercise comment that is being edited.

Access to this page is restricted only for users with `RIGHT_EDIT` for selected exercise or for author of the comment.

2.1.6 Module `users`

This module contains pages for viewing and managing user accounts, their rights and users' memberships in groups. All the following pages require at least the `RIGHT_READ` general right for users.

Since the word *user* becomes little overused in this module we will call the user who is logged in and performs all the actions the *actor*. Furthermore, only general rights are considered every time when speaking about rights (since there are no other rights for users anyway).

`index` A listing of all users. The list can be sorted and filtered by various attributes (user role, study program, study group, study year). Any group for which the actor has RIGHT_READ can be selected and only members of that group will be displayed. This condition can also be negated, so that only users not belonging to a group can be displayed. A text-search based filter is also available, displaying only those users whose login name, e-mail address or full name (i.e. *name* `<space>` *middleName* `<space>` *surname*) contain the specified text as a substring.

Every user entry is equipped with a checkbox for selecting it. If the actor has `RIGHT_DELETE` right for users, a *delete* button is displayed that deletes all the selected users. Every entry is also equipped with two links, '`Edit`' and '`Rights`', provided that the actor has sufficient rights to edit the users properties and rights, respectively.

This page is also used for assigning users to groups. If the actor has `RIGHT_EDIT` for at least one group, he/she may add selected users to this group from here.

`create` A form for creating new users. Submitted data are handled by this page, which is responsible for creating the new user. A new user is always created with zero rights.

This page is visible if the actor has at least `RIGHT_CREATE` for users. After successful creation, the browser is redirected to the `index` page. If the actor has at least `RIGHT_EDIT`, the browser is redirected to the `rights` page instead, so rights for newly created user can be edited.

`properties`

A form where basic properties of the selected user (such as login, password, e-mail, etc.) are displayed and edited. Modifications are handled by this page and saved into the database.

An actor with `RIGHT_EDIT_BASIC` may edit properties of any other user with no general rights. With `RIGHT_EDIT` he/she may edit any user who has strictly less rights (using vector comparison, i.e. all components are less than or equal and at least one component is strictly less than). A user with `RIGHT_ADMIN` the actor may edit any user except the administrator (i.e. the user with `id == 1`).

Also, any user having access to this page (i.e. general `RIGHT_READ` for users) including *admin* may edit his own properties.

This page requires a GET parameter `userId` that holds the ID of the user who is being edited.

`rights` Overview of general rights and delegations. Rights are displayed in a form where the actor can select from prepared roles or set general rights manually.

Delegations are displayed in two tables – one for groups and one for exercises. Actor may see only those delegations he/she may also revoke. Every delegation is equipped with a check box. A *revoke* button that removes all selected delegations is placed beneath the table. It triggers a `revokeQuery` action that shows query dialog and if confirmed triggers `revoke` action. Both actions are handled by this page.

This page is also the only place in the system where delegations can be granted. There are also two forms (for groups and exercises) where the actor can select a group/exercise for which he/she has `RIGHT_ADMIN` and delegate rights for it. Data from these forms are also handled by this page.

This page is accessible for an actor having `RIGHT_EDIT_BASIC`. The actor must have at least `RIGHT_EDIT` to edit general rights, although he/she may never grant nor revoke right that he/she does not have. An actor with `RIGHT_ADMIN` for users may edit general rights without restrictions.

Nobody can modify his/her own rights.

This page requires a GET parameter `userId` that holds the ID of the user whose rights are edited.

2.1.7 Module news

All pages in this module are related to news items and their management. News items are short messages from administrator or group owners to other users.

`index` Overview of the news. News items are typically sorted by their time of creation and may be filtered by groups.

The user can only see those news items that are visible for everyone or that are visible for members of some group and the user is a member of this group, its owner or a trustee. The user must also have sufficient general rights. That means `groupRight`, `exerciseRights` and `usersRights` defined for each news must be lesser or equal to the user's general rights. The administrator (the user with `id == 1`) may see all news items. See Section 2.3 [Security], page 21

If the user has sufficient rights, he/she may also see buttons for news editing (`RIGHT_EDIT`) or deletion (`RIGHT_DELETE`). The *edit* button leads to the `edit` page and the *delete* button performs action a `delete` action that is handled by this page.

`create` A form for creating a new news item. Only a user with general right `RIGHT_CREATE_PRIVATE` for news (or higher) can see this page. User with `RIGHT_CREATE_PRIVATE` may only create news for his own groups and a user with `RIGHT_CREATE` may also create global news.

Data from this form are handled by this page. If the news item is successfully created, the browser is redirected back to the `index` page.

`edit` A form for editing the news. This form is similar to the form on the `new` page. This page is also responsible for saving the modified data into the database. If the data are correctly saved, the browser is redirected back to the `index` page.

The page requires a GET parameter `newsId` that specifies the ID of the news item that is being edited. If this ID is omitted or invalid, the browser is redirected to the `index` page.

2.1.8 Query Dialog

Time to time user needs to perform serious operation such as delete an object (group, exercise etc.). Since this operation may be performed by one-click action, the user should be asked to confirm his/her decision. Therefore a query dialog was introduced.

Query dialog is technically a page. It has it's controller and template and uses MVC routing and dispathing system as any other page. However there are some differences since multiple dialogs with different messages may be shown at the same time.

Protocol for showind dialog is quite simple. Page that requires user's confirmation follows these steps.

1. Aquire the dialog controller object via `getDialogObject` method of the page manager.
2. Initializes the dialog (method `initializeDialog`) and specify fallback page. When the validity of session data for created dialog instance has expired, the dialog redirects browser to the fallback page. Therefore this page SHOULD be the page that creates the dialog. Initialization also generates random ID for the instance. Each instance has it's own ID that identifies data in the session.
3. Set dialog parameters using controllers public methods `setDialogName` and `setMessage`. These methods prepares given values into special session storage room used for given dialog instance.
4. Add dialog buttons using `addButton` method. This method can add both *action* and *link* buttons. That means buttons that are performed as POST actions (thus not indempotent) and simple links. In fact all the buttons are rendered as *action* buttons but link-like buttons are dirrected to the '`redirect`' action of the dialog. This si explained more thoroughly in Session Data Management subsection.
5. Perform a redirect to the dialog page.
6. Dialog page is displayed. User may leave dialog page by clicking on one of the given buttons.
7. After clicking on some of the buttons taget action is executed or the browser is redirected to given URL.

Here is a trivial sample of PHP code that uses query dialog:

```
public function testQueryAction()

$dialog = PageManager::getInstance()->getDialogObject('query');
$dialog->initializeDialog($this->createUrl('', array(), true, false));
$dialog->setDialogName(tr('Important Decision'));
$dialog->setMessage(tr('Do you wish to perform %s action?'), 'Test');
$dialog->addButton(tr('Yes'), $this->createUrl('test'));
$dialog->addButton(tr('No'), $this->createUrl('', array(), false), false);
$dialog->redirect();



public function testAction()

// Perform the real action...
```

2.1.8.1 Session Data Management

Since multiple instances of the dialog may be shown simultaneously, dialog may not use common page namespace and persistent namespace and more sophisticated approach is required. When the dialog is initialized, unique ID is assigned. This ID is used to identify session storage.

ID is passed to the dialog along with fallback page URL via GET parameters `dialogQueryId` and `fallbackUrl`. Both parameters are mandatory since dialog can not work properly without them.

Described approach brings one severe problem – session growth. Every time new dialog is created, new session storage is allocated and filled with data. Therefore a mechanism must exist to unset storages that are no longer needed. This storage is released when user confirmes of rejectes the dialog.

All buttons are *action* buttons therefore any user's decision will invoke a POST request. Link-like buttons are directed to the `redirect` action of dialog controller which releases the session storage and preforms the redirect. Other buttons are slightly modified and URL of each one is provided with extra GET parameter (`_dialogQueryId`) that carries ID of the dialog instance. When action is performed this parameter is intercepted by MVC dispatcher who is responsible for removing the data storage.

If the user leaves dialog page by other means (e.g. by *Back* button in his/her browser), the data will endure until user logs out. This problem is known yet unattended for the time being.

Whole dialog matter has been affected by spaghetti design and code injection pattern. However re-factorization is not planed since the dialog is functional and quite separated.

2.1.8.2 JavaScript Bypass

The dialog query adds at least one client-server round trip to every confirmed operation. This might be little frustrating for users with slow Internet connection therefore a JavaScript bypass was implemented for every confirmation.

Principle of the bypass is to intercept mouse clicking (or form submitting) event by JS, displaying simple confirmation dialog directly in user's browser and then updating URL of action button, so that the *slow* dialog query is skipped. In fact a `jsConfirmed=1` GET parameter is added into URL and query action handler is programmed to skip dialog if this parameter is found.

Therefore users with operational JavaScript may speed up their work with CodEx and users witout it will not be deprived of the original functionality.

2.1.9 Main menu

- Welcome Page
- News
  - Create News
- Groups
  - Create Group
  - Tasks
    - Specification
    - New Submit
    - Properties
    - Submits
  - Properties
  - Results
  - Bonus Points
- Exercises
  - Create Exercise
  - Properties

    - Specification
        - Attached Files
    - Tests Configuration
    - Tests Files
    - Assign Tasks
    - Solution Database
        - Submit Solution
- Users
    - Create User
    - Properties
    - Rights
- Personal Settings

2.2 Components

2.2.1 Component Overview

Since we try to implement every piece of functionality only once, the component system was introduced. This system is tightly integrated with MVC. Therefore, we recommend reading the MVC section first. See ⟨undefined⟩ [MVC], page ⟨undefined⟩

Components are very similar to pages. They have a controller class that must be derived from `PageComponentAbstract` and a template. Both the controller and the template are stored in the '`components/`*`component name`*' directory in the files '`controller.php`' and '`template.php`'. Access to these files is managed by the `PageManager` class.

Each component must be registered in the the controller class of the page where it is supposed to be displayed. Upon registration a unique ID is assigned to the component. Multiple components of the same type can be placed simultaneously on the page.

2.2.1.1 `PageComponentAbstract`

The controller of a component must be derived from the `PageComponentAbstract` class. Much like the page controller, the component controller is responsible for rendering and handling actions.

The component can define a list of internal GET parameters and their filtering rules. General rules are stored in the `$getParams` member variable and action GET parameter rules are in `$getActionParams`. These two fields are formatted the same way as the corresponding fields in the page controller object. When a component is registered, the GET parameter rules of the component are merged with GET parameter rules of the page.

The GET parameters live in separate namespace. A simple name-mangling scheme is applied, composing the final name of the parameter as `Ccomponent Id_`. I.e.m if the component 42 has a parameter named '`foo`', this parameter will have the real name `C42_foo`). The mangling is performed automatically by the `registerComponent` method.

After registration, the `initialize` method is called. This method does nothing, however it can be redefined in derived classes. The initialization routine can also hold code that cannot go into the constructor for some reason (for in the constructor the component is still not registered). It is strongly recommended to put as much code as possible here instead of the constructor, since initialization is called only when its clear that the component will really be needed.

The component can also handle actions. Component actions are handled the very same way as page actions. The only difference is that the `action` GET parameter should be formatted

as an array. The key of this array is the component ID and the value is the action name. For example, a request with the parameter '`action[42]=delete`' parameter will be dispatched to the component with ID 42 on which the `deleteAction` method will be called.

Rendering is performed by the `renderView` method. This method prepares the view object and renders it using the component template. Relative path to the template is defined in the `$template` member variable. If no template is defined, the default '`template.php`' is used from the directory of the component. After the view object is created, the `prepareView` method is called. This method does nothing, but descendant classes can use it for additional initialization of view data.

Each component has its own list of CSS and JavaScript files. They are stored in the `css` and `js` member variables as arrays. When the component is registered, these lists are merged with lists on the page. Scripts and styles should be written with respect to the independent nature of components. For example, the CSS selectors should only use general classes (not selectors for individual IDs).

2.2.1.2 Component Session Storage

There can be a special field inside the session namespace of a page (both normal and persistent) called '`components`'. This field is an array indexed with component IDs and its values are also arrays. Each array acts as the private storage of a component and it is passed to the component reference. This way the components have their own storage that will not collide with storage of other components even if there is more than one component of the same type on the page.

For example, if the component with ID 42 attempts to save parameter '`offset`' to the session, it will be stored in:

pageNamespace`->components[42]['offset']`

Access to component session storage is provided by the `getSessionStorage` and `getPersistentSessionStorage` methods that return a reference to the storage array. These methods must not be called before the component is registered, since unregistered components do not have access to any page session namespace.

2.2.2 Menu Component

2.2.2.1 Overview

The Menu Component provides a generic hierarchical menu and, together with the CodEx menu model, provides the main menu for the CodEx front-end.

2.2.2.2 Generic Menu

The Menu Component displays a list of menu entries. Each entry has the following properties: a caption, the URI it links to, a hint (usually displayed when the mouse hovers over it) and an *active* flag. There should be always exactly one entry with the active flag enabled, corresponding to the current page. Also, each entry can have an arbitrary number of sub-entries and entries can be nested to an arbitrary level.

The actual list of entries is provided by the model. Any model must implement the interface `IMenuModel` which contains a single function `getMenuItems()` that returns an array of `MenuItemAbstract` objects. These have the following public properties: `caption`, `url`, `hint`, `active` and a public method `hasSubItems()`. If this method returns `true`, then the entry has sub-entries and these can be obtained by iterating the entry object (as it implements the `Iterator` interface).

2.2.2.3 CodEx Menu Model

When the user logs into CodEx, only a few menu entries are displayed at first. As he navigates the menu, entries appear and disappear depending on the user's actions. Typically, only entries related to the user's current module and a few others are displayed.

The idea behind the magic is relatively simple. Most pages require some `GET` parameters specifying the object they are working with. For example, most pages in the *groups* module require the parameter `groupId`, i.e. the ID of the selected group. An entry is only displayed if the parameters required by the corresponding page are present and valid.

The required parameters are specified directly in the definition of the menu and they are two-fold. When switching from one page to another using the menu, these parameters also get *preserved*, while others are *swept*. Thus while browsing pages working with a group, the group remains selected. When the user opens a page not working with a group, the selected group is forgotten and the entries of the pages working with a specific group are hidden).

The model, implemented in the `MenuModel` class, constructs the list of entries on the fly from an internal representation. It is a list of numbered-array entries, each containing the following fields: caption, module name, page name, list of `GET` parameters used, the hint, a list of sub-entries and a *hidden* flag.

An entry is only visible if all required `GET` parameters are present and the user has sufficient rights to display the page. (verified with `checkRights()`). Also, if the entry is marked as *hidden*, it will not be displayed unless it is active (i.e. unless it corresponds to the current page).

2.2.3 Table Component

Component table is ment for displaying plain ordinary data tables (like list of all users). Similar issues are attended every time a table is rendered in the page, such as HTML rendering, oddering by selected columns or displaying table with huge amoun of data per parts. Therefore it is usefull to have a component that will attend this issues.

As all other components the table implements Model-View-Controller pattern. Controller is represented by class that handles common issues (i.e. data paging) and prepares the data for visualisation. Data are stored inro view object and rendered into HTML by common template. Model is impelemented by separated class and each table has it's own model.

2.2.3.1 `ITableModel` Interface

Every table model implements this interface. This interface is defined that the table controller will not know anything about data storage, however it will have means how to get required data.

Model also provides way to other information about the table such as list of columns, their captions, alignment etc.

2.2.3.2 `DBTableModel` Abstract Class

Komponenta Table ========== Komponenta table slou k zobrazovn tabulek dat a k jejich ppadnmu nastrnkovn. Ke sv innosti potebuje datov model (objekt implementujc interface ITableModel), kter slou jako zdroj dat (poskytuje informace o sloupcch a na podn vrt dan poet dk ve zvolenm uspodn).

Konstruktor oekv referenci na model a ppadn i dal (nepovinn) parametry. Parametry se uvdj v tomto poad: - $showHeader - bool flag, zda se m zobrazovat header (1. dek tabulky) - $paging - zda se m pout strnkovn dat - $cssClass - tda CSS, kter se nastav tabulce. - $noRowsMessage - textov zprva, kter se zobraz, kdy v tabulce nejsou dn data.

Tabulka si ukld potebn informace do persistentnho loit v session: amount - poet zznam zobrazovanch na strnce offset - offset prvnho zznamu na strnce sorting_column - alias sloupce, podle kterho se td sorting_desc - bool. flag, zda se td sestupn, i nikoli.

ITableModel ========== Interface, kter mus implementovat model tabulky. Jednotliv metody jsou popsny ve zdrojovm kdu.

Soust definice interface je i tda TableModelColumn, jej objekty uchovvaj informace vdy o jednom sloupci tabulky. Objekt m pouze konstruktor a adu poloek pro ten: $caption - popisek sloupce (me obsahovat i jednoduch XHTML (nap) $align - XHTML hodnota stejnojmenho atributu (left, right, center), kterm smrem budou formtovny vechny buky sloupce (vchoz je left) $valign = XHTML hodnota stejnojmenho atributu (top, bottom, middle); nastavuje vertikln zarovnn vech bunk (vchoz je top) $nowrap - boolean flag, kter kdy je nastaven na true, zake zalamovn obsahu bunk (vchoz je false) $expand - boolean flag, kter nastavuje, zda se m sloupec automaticky rozthnout, pokud je msto (vchoz je false) $sortable - boolean flag, kter uruje, zda lze podle poloek ve sloupci tdit dky (vchoz je false) $sorting - uruje, jak je tento sloupec pouit ke tdn (0 = vbec, 1 = vzestupn, -1 = sestupn)

DbTableModel ========== Model, kter erp data pro tabulku pmo z databze. V ppad jednoduchch dotaz (kter neobsahuj agregan funkce, ani klauzuli where), lze tabulku navrhnout velice rychle (i kdy je dotaz sloen z vce tabulek pes standardn operac join). Rozhran je ale dostaten flexibiln, aby sneslo jakkoli SQL SELECT dotaz.

Pi vytven modelu je teba pedat konstruktoru adu parametr, kter zinicializjuj model:

$caption - Popisek tabulky, kter se zobraz v HTML.

$columnParams - definice sloupc a jejich mapovn na SQL sloupce. Vce viz "Definice sloupc".

$defaultTable - Nzev "vchoz" DB tabulky. V ppad, e se nepouv vce tabulek (spojench operac JOIN), tak je to tak jedin DB tabulka. Zde me bt uloen bu etzec (nzev tabulky), nebo pole s jednm prvkem (alias => nzev), pokud potebujeme tabulku pejmenovat. Na tabulku se pak vdy odkazujeme jejm aliasem (pokud nen definovn, je aliasem jmno tabulky samotn).

$joinTables - Pole "ostatnch" tabulek, kter se s vchoz spojuj operac JOIN. Kad poloka je dvouprvkov pole: array(selektorTabulky, podmnka). Podmnka je SQL etzec, kter se pouije jako spojovac podmnka ("ON") u klauzule JOIN v SQL dotazu. Selektor tabulky m stejn formt, jako u vchoz tabulky (me to bt jmno tabulky, nebo pole alias => jmno).

$dbColumnFilters - Pole filtr, kter se aplikuj po naten dat z DB. Vce viz "Filtry".

$orderBy - Pole sloupc, podle kterch se bude tabulka tdit v ppad, e nen vybrn dn sloupec ke tdn. Toto pole je jako argument rovnou pedno metod order() objektu Zend_Db_Select, take mus dodrovat jeho syntax. Vchoz hodnota je array('id').

SQL dotaz pro ten z DB se automaticky poskld z uvedench parametr a specifikac jednotlivch sloupc. V ppad, e je poteba pout sloitj dotaz, je mon (funkc setSQLQuery) pedefinovat vnitn objekt Zend_Db_Select a pout tak v podstat libovoln dotaz. Takto pidan objekt by neml mt nastaveny limity pro zobrazen (LIMIT) ani klauzuli ORDER BY. Rovn je teba zachovat korespondenci mezi nzvy sloupc uvedenmi v SQL dotazu a ve specifikaci sloupc modelu tabulky.

Pi zavoln funkce loadData() se nejprve piprav a zavol SQL dotaz. Na zskan data se pouij filtry definovan v dbColumnFilters a nsledn se kad dek proene renderovacmi funkcemi vech sloupc. dek se pak ulo do novho objektu typu stdClass, jeho poloky jsou pojmenovny po nzvech sloupc (objekty se narozdl od pole nekopruj, ale pedvaj referenc).

Definice sloupc: ———— Sloupce jsou v modelu uloeny v promnn $columns jako pole objekt DbTableModelColumn (co je tda pmo odvozen od TableModelColumn). Zachovv vechny dosavadn lensk promnn a navc pidv nkter dal:

$template - Triviln XHTML template, kter je pouit pro renderovn dat do buek pslunho sloupce. Obsahovat sm pouze prost text (krom znaku dolar - $) a promnn ve formtu $nazevSloupce. V nzvu sloupce se sm vyskytovat pouze psmena, sla a znak podtrtko. Jedn se o nzev DB sloupce (nikoli sloupce tabulky) - viz dle. Pklad: '$name $surname' - zobraz jmno a pjmen, kde pjmen je tun.

$dbCols - Seznam databzovch sloupc, kter jsou poteba k renderovn tohoto tabulkovho sloupce. Kad sloupec je definovn v poli jako alias => definiceSloupce. Pokud je alias vynechn, pouije se nzev sloupce z SQL. definiceSloupce me bt jednoduch (jen etzec pro SQL), nebo pole array(aliasTabulky, sloupec), kde alias tabulky je odkaz na nkterou z tabulek definovanou v $defaultTable a $joinTables. Pokud je alias tabulky vynechn, bere se automaticky $defaultTable. SQL alias sloupce (ppadn jeho DB jmno - pokud nem alias) pak lze pout uvnit $template.

$orderBy a $orderByDesc - Ukldaj nastaven poloky ORDER BY, pokud se pouije tdn podle tohoto sloupce. Nastaven je uloeno ve formtu array(nzevDBSloupce => smr), kde smr me bt 1 (vzestupn), nebo -1 (sestupn). V ppad, e nejsou poloky definovny pi inicializaci, dopln se do nich automaticky sloupce definovan v $dbCols (ve stejnm poad, v jakm jsou v dbCols definovny).

Vechny doplnn lensk promnn lze inicializovat stejn, jako u rodiovsk tdy. V parametrech se uvd ve stejnm poad, v jakm jsou popsny zde a a za parametry rodiovsk tdy.

Pklad pole, kter se m pedat konstruktoru: array('id' => array('Id', // popisek 'center', // align 'middle', // valign false, // nezalamovat false, // expandovat false, // tdit '$id', // template pouze zobraz sloupec "id" array('id')), // potebuje sloupec id z vchoz tabulky 'name' => array(caption => 'Jmno', align => 'left', valign => 'middle', sorting => true, template => '$name $surname', // slo jmno a pjmen tun dbCols => array('surname' => array('users', 'surname'), // users.surname AS surname 'name' => array('users', 'name') // users.name AS name)),);

Filtry: ——- V nkterch ppadech si nevystame pouze s primitivnmi templaty (viz ve), ale potebujeme sloitj operace nad petenmi daty, kter navc meme udlat efektivn pouze v PHP (nap. pkn pevod unix-timestampu na datum a as nebo vytvoen odkazu). Od toho jsou tu filtry.

Filtry jsou konverze definovan pmo nad daty vytaenmi z databze, ne se pedaj k zrenderovn pomoc triviln ablony sloupce.

Definice filtr jsou uloeny v poli $dbColumnFilters. Indexem je nzev sloupce, kter se m filtrovat, a hodnotou je filtrovac funkce. Filtrovac funkce je pedna bu jako etzec (identifiktor lensk funkce), nebo jako pole, kde prvn prvek je opt etzec (identifiktor) a nsledujc prvky jsou dodaten argumenty.

Filtr mus bt lensk funkce modelu (nejlpe s protected pstupem). Prvn dva jej parametry jsou &$row, $col, kde $row je objekt s daty jednoho dku, kter se prv renderuje a $col je nzev sloupce, pro kter je funkce urena (tj. kter se m filtrovat). Pochopiteln funkce me teoreticky sahat na vechna data v dku a modifikovat je, nicmn je dobr dodrovat, e funkce modifikuje pouze sloupec $col.

Filtr me mt i vce parametr, prvn dva zstvj &$row a $col, ostatn parametry mus korespondovat s parametry, kter jsou funkci pedny (viz formt zpisu filtr popsan ve).

2.2.4 Form Component

2.2.5 File Manager Component

2.2.5.1 Overview

The File Manager component (class `PageComponentFileManager`) allows a CodEx user remote file-access to a CFI file system on the server (typically residing inside a directory). The file manager allows browsing the directory structure, uploading, downloading, moving, renaming and deleting files, creating or removing directories.

2.2.5.2 Browsing Files and Directories

At any moment, the contents of exactly one directory, the working directory are displayed. The listing is similar to *full-file list* in a typical file manager. Each line represents one file and different file properties and controls are in different columns. Name, date of last modification

and size (for files) is displayed. Also there's a graphical icon indicating the type of the entry. Files also have a small floppy-disk icon in the rightmost column. Clicking on it will activate a download link for the file. Clicking on the name of the file will activate a view link (displaying the contents of a text file in the browser). Clicking on the name of a directory will switch to that directory.

When inside a directory, a special '..' entry is display at the first line for switching to the parent directory. Also, links to all directories on the active branch (i.e. the branch leading to the working directory) are displayed for fast switching to any superior directory.

2.2.5.3 Creating Directories

The user can create directories to organize his files. There is no a priori limit to the number or nesting of directories, but such limits might be specified by the underlying file system for specific purposes. Trying to create a directory with an invalid name will fail. The validity of a name is determined by the underlying file system.

2.2.5.4 Uploading Files

Up to a predefined number of files can be uploaded simultaneously. Apart from the filename input field, each file upload slot contains two checkboxes, *Unpack* and *Binary*. Checking *Unpack* has the following effects: If the file has a '`.gz`' or '`.bz2`' extension, it will be decompressed first. If the file has a '`.tar`', '`.tar.gz`', '`.tar.bz2`' or '`.tar.bz2`' extension, it is treated as an archive. In this case, the contents of the archive are copied to the working directory, not the original archive.

Unless `Binary` is checked, the file (or all files inside an archive) will be passed through the text-file filter, converting CRLF line-endings to LF and removing a possible UTF-8 BOM. (Most uploaded files are expected to be textual). When uploading binary files (such as images) this checkbox must be checked or the contents of the files will be broken.

Any ownership, access modes or symlinks stored in the archives are ignored. If the names some of the uploaded files (or files extracted from uploaded archives) contain forbidden characters, the will not be added to the directory and warning messages will be printed.

2.2.5.5 Downloading Files

The user may select an arbitrary number of files or directories in the working directory using checkboxes next to the file icons. Pressing the *Download* button will download all selected files packed in an archive. The type of this archive can be selected with a radio button (supported formats are *zip*, *tar*, *.tar.gz* and *.tar.bz2*).

2.2.5.6 Renaming, Moving and Deleting

As with downloads, any number of files can be selected. All selected files can be simultaneously deleted, moved to another directory or even renamed. To delete selected files the user simply hits the *Delete* button. To move files to another directory, the user selects this directory from a pull-down menu and hits *Move*. When moving directories, existing directories will be merged. Moving a file over an existing file will fail. An attempt to move a directory inside one of its descendants will be detected and silently ignored.

When the user selects some files and hits the *Rename* button, the names of selected files will become edit-boxes, where the user can change their names. When the user hits *OK* the file manager tries to rename all the files (simultaneously). Sophisticated renaming patterns are supported (See Section 4.2.7 [CFI Utilities], page 48) or the source file `cfi_util.php` for details). Trying to rename an entry to an existing name will fail (unless that entry is renamed to another entry simultaneously).

2.2.5.7 File Manager Error Handling and Reporting

The file manager always tries to complete as much work as possible. It will always try to perform the operation on all selected files or directories, regardless of the operation failing on any of the preceding entries. For each entry the requested operation failed, it will print a detailed warning message, explaining what kind of operation failed, on which file and why.

2.2.5.8 File Manager Security

The file manager component itself needn't do any specific checks whether the requested operations on files are allowed as this is done by the underlying CFI file system (See Section 4.2 [CFI], page 44).

2.3 Security

Security is one of the most delicate issues in every information system. The security model employed in CodEx consists of three principal parts:

- The Trusted Base and Its Protection
- Authentication
- Authorization

Note that this section does not aim to cover all security-related issues in CodEx. Also, the considerations that need to be taken into account when operating the system are beyond the scope of this manual.

2.3.1 The Trusted Base

The *trusted base* is a *safe place* from the point of view of the application. Our application requires that the database, related data files and all CodEx scripts are stored and run in the trusted base.

Furthermore, the application is only accessible from the outside world via the HTTP protocol. The administrator/deployer of the application must ensure that CodEx will run in an environment that meets the above criteria.

Other security-related issues (such as backups, physical protection and so on) are solely the administrator's responsibility. Henceforth, we expect that the trusted base is secured, since all the following security mechanisms rely on this fact.

2.3.2 Authentication

Authentication is the process of verifying the user's identity. Unauthenticated users may not perform any action within the system nor see any relevant data.

Authentication is performed by verifying the user's login name and password. The user logs in with his/her `login`, that is assigned to him by the system upon registration or with his `loginAlias` which the user may pick for himself/herself. The password is stored in the database in hashed form (for details see `users.passwd` in database description). Even though the database considered a part of the trusted base, the passwords are stored hashed due to their high level of confidentiality. Other users' passwords should be revealed to no one, not even the administrator. Moreover, this significantly reduces the confidentiality of database backups.

When the user logs in, a session object is created. PHP sessions are protected by a validation key that is stored in the session and in a cookie at the client side. The key is changed after every operation, so it is nearly impossible to steal. The session manager also verifies the client's IP address and the login information expires after a certain time interval.

2.3.3 Authorization

The authorization is the process of verifying that the user possesses rights to carry out a particular action. CodEx applies the concept of minimum rights. Therefore anything that is not explicitly permitted is forbidden.

The authorization process uses a simple security model based on Bell-LaPadula. Every time the **user** needs to perform a sensitive **action** with a protected **object**, the application asks the security model (the monitor) what maximum authorization level (called *rights*) the **user** has for that **object**. Also, every **action** has a minimum required authorization level (*rights*)r. If the user's rights $\geq$ required rights for the action, permission to execute the action is granted (otherwise it is denied).

Rights need not be bound to a single object. In some other cases the application may ask security model what rights the user has for all objects of the same kind. Rights to all objects are called *general rights*. These are described in detail in the Rights subsection.

Example: The user wants to delete a group. The security model says that the user has `RIGHT_ADMIN` for the group and, to delete a group, at least `RIGHT_DELETE` is required. Since `RIGHT_ADMIN` $\geq$ `RIGHT_DELETE`, the operation is permitted.

Another example: The user wants to create an exercise. There's no exercise object yet (because the application does not know whether the user may create it), so the security model is asked for general rights that user has for all exercises. Unfortunately the user has only `RIGHT_READ` for exercises and since `RIGHT_READ < RIGHT_CREATE`, the exercise cannot be created.

2.3.3.1 Admin

The system must have one special user account. This account is usually called '`admin`' (short for administrator) and it has pretty much the same role as the *root* account in UNIX systems.

The *admin* has `id == 1` and he/she is automatically granted maximum rights (`RIGHT_ADMIN`) for any object in the system. Furthermore, this account cannot be deleted nor modified by any other user.

The *admin* also plays a special role of a 'garbage collector'. When a user is deleted, some of his assets (groups and exercises) must be transferred to another user (since groups and exercises must have an owner). Typically, these assets are transferred to the user who performed the deletion. However, sometimes this need not be possible. In this case the assets are transferred to the *admin*. Therefore, the the *admin* account must always exist.

2.3.3.2 Protected Objects

The security manager cannot control access to every bit of data, for such fine granularity would produce intolerable overhead. Therefore, we have to decide what data need protecting and how small should the manager's granularity be made.

The protected objects mostly correspond with database records in tables with the same names as the object type. We recognize the following protected object types:

- users
- groups
- tasks
- submits
- exercises
- texts
- solutions
- exercise comments
- news

Each object is represented by one row in the corresponding table. Rights are always attached to one particular object and general rights are attached to object types (tables). There is no need for keeping rights to user objects, thus only general rights are applied for users (plus some special rules described later).

Some objects are not independent and instead belong to another object. Specifically, every submit belongs to a task and every task belongs to a group. Therefore, submits and tasks are considered to be a part of a group. When the user manipulates with them, rights of the corresponding group are tested instead. Likewise, the texts and solutions belong to an exercise.

These *substitutions* of rights may cause some complications, so one has to be careful when dealing with them. Fortunately, simple logic still works, so it is not so difficult to figure out which rights should be tested.

For example: The user wants to delete a task. Normally we would test whether the user has at least `RIGHT_DELETE` for given task. However, the task does not keep any rights at all and receives the rights of its group instead. But the rights for deleting a group (with all tasks and submits) are quite stronger than the rights to remove one tiny task from a group.

We must not be too dogmatic in testing the rights and use common sense in this case. Removing task from a group should be considered more like a modification of the group since all tasks are part of some group anyway. Therefore we should not test the task (or more precisely the group) for `RIGHT_DELETE`, but for `RIGHT_EDIT` instead.

Each case must be considered separately. Most common cases are described in the description of each page. See Section 2.1 [Page Hierarchy], page 4

2.3.3.3 Rights

Effective rights for the given object (that are returned by the security manager) are computed as the maximum value from:

- *General rights* are stored directly in the user's account and they represent rights for all objects of the specified kind. We recognize general rights for users, groups, exercises and news.
- *Owner's rights.* All objects except users, tasks and texts have an *owner*. The owner is the user who (typically) created that object and who administers it. The owner of any object except a submit has `RIGHT_ADMIN` for that particular object and the owner of a submit has `RIGHT_READ` for that submit.
- *Delegated rights* – will be described later.
- *Substitutional rights* are obtained from the object to which the tested object belongs to. See Section 2.3.3.2 [Protected Objects], page 22 For example, if the security manager examines the rights for a task, substitutional rights from the group where the task belongs to are consulted, too.

Rights (authorization levels) can have the following values:

`RIGHT_NONE`
: Means 'no particular rights'. It is default value if no explicit rights are defined (so everybody possesses this right to every object).

`RIGHT_CREATE_PRIVATE`
: This right only applies to groups and news. A user possessing this right may create obejects of the corresponding type, but with some restrictions. In case of groups, the group cannot be made public and in case of news, the news item must be designated for one group only (i.e. not public).

  This right is meaningful only when used as a general right.

`RIGHT_READ`
: Right to read a given object. What that exactly means depends on the object type. For example, the right to read an exercise also allows the user to read the test files and all public submits. However, there are some exceptions from this right. For example, the general right to read exercises does not permit the user to read private exercises.

`RIGHT_EDIT_BASIC`
: Right to perform minor modifications on an object. For example, a user with this right for exercises may submit his/her own solution, but he/she may not edit the exercise properties nor the specification.

`RIGHT_CREATE`
: The right to create a new instance of an object without any restrictions. This right is meaningful only when used as a general right.

`RIGHT_EDIT`
: Right to edit any parameter (which is editable) of the given object without any restrictions. This right, however, does not allow the user to delete the object.

`RIGHT_DELETE`
: Right to delete the object with all related parts (for example, deleting a group with all tasks and submits that belong in that group). Some objects may not be deleted despite the user having the right to do so (for example, an exercise may not be deleted if it has been assigned as a task).

`RIGHT_ADMIN`
: Ultimate rights to the object. Usually, only object owners and the administrator have such rights. This right permits any action on the object. A user with these rights may also hand over the object to another user or delegate rights to other users.

Levels are ordered from minimum to maximum rights. A higher level includes all rights from the level below. E.g., `RIGHT_READ` is a subset of `RIGHT_EDIT`.

2.3.3.4 Delegated Rights

Delegations are additional rights typically employed in less common situations. A delegation is kind of a permission to do certain things with an object. Rights are delegated by one user, called a *granter*, (who has sufficient rights to do so) to another user, a *trustee*. Delegations apply only for groups and exercises.

The granter must have at least (`RIGHT_ADMIN`) for the involved object. The trustee can receive more than one delegation, but he/she may only receive one delegation for a single object from a single granter. Also, a user cannot delegate rights to himself/herself. Delegation carries an access level with it, so it is possible to grant only `RIGHT_READ`, for example.

Delegations may be removed in three different ways:

- The trustee may yield the delegation.
- Granter may revoke any delegation he/she granted (no matter if he/she still has `RIGHT_ADMIN` for the objects involved).
- Any user with `RIGHT_ADMIN` for an object may revoke any delegations for that object (no matter who grants them).

Delegations are stored in a database table called `delegations`. See Section 5.13 [Table delegations], page 61

Illustrative example: Alice went on a vacation (without access to the Internet, of course) and she needed Bob to manage her group G. Fortunately Alice owned G (she had administrative

rights for it) so she decided to delegate `RIGHT_EDIT` to Bob (for group G). Bob could manage the group (create tasks, grant bonus points etc.), but he could not delete the group, change its owner nor delegate any rights for it to other users.

After a few days Bob had an accident (poor Bob!) and he ended in a hospital. So he asked Carl to take over for him. Bob could not delegate Carl any rights so he asked the almighty administrator to do so. *Admin* used his divine power and blessed Carl with `RIGHT_EDIT` delegation for group G, so Carl could perform the same actions as Bob.

When Bob finally got out of the hospital, he saw that Carl is doing just fine, so he decided to let him continue managing the group G. Therefore, he yielded his delegation for G, because he did not want to pose a security threat.

When Alice came back, she found out Carl was administering her group. She did not delegate any rights to Carl, but she had administrator rights for G, so she revoked Carl's delegation and invited him to a dinner.

They got married and had a bunch of kids, but that is a completely different story...

2.3.4 Implementation

All security-related methods are implemented by the `SecurityManager` class. Since these methods are always required and there is no need for class inheritance it is implemented as a static class. The security manager requires the session to be initialized before itself, because it uses the session to store login information. The `SecurityManager::intitialize()` must be called before any other method.

Rights are always stored as an 8-bit unsigned integer (0..255) where 0 always stands for `RIGHT_NONE` and 255 for `RIGHT_ADMIN`. Other values are defined as constants in the `SecurityManager` class. It is strongly recommended to use these constants, instead of hard-coded numeric values. Integer representation of rights was chosen so that rights can be easily compared and the gaps between levels were left so that new rights may be defined in the future.

The security manager also logs important events into a security log. Important events are:

- A user logs in.
- Failed login attempt.
- Delegating rights.
- Revoking delegations.

For more details about the implementation, see the source-code reference.

3 Services

3.1 Services Overview

Sometimes users need to perform some task via automated script (eg. for such task needs to be executed periodically). Web user interface is not quite suitable for this approach so the services were introduced to CodEx.

CodEx services represents special entry points for automated scripts. They have stateless character and simple well-defined interface so the client script need not implement complicated protocol.

3.1.1 Protocol Overview

Services are accessed via HTTP protocol. Client performs a HTTP request with properly defined parameters, service performs it's task and returns a XML file with results. Therefore action oriented services might just do the job and then return simple XML with result code and datamining services may choose to assemble complex XML file with any data they need to return.

3.1.1.1 Client Query

Client performs a HTTPS query on '`service.php?service=`*`service-name`*' instead of accessing CodEx via '`index.php`'. The '`service-name`' is unique service identifier. This identifier is used in then name of service php file and the service class. The client SHOULD not add any other GET parameters in the URL. Client MUST NOT insert any `Accept` HTTP headers nor any unnecessary headers. Please be as much conservative as you may.

All query parameters must be encoded in HTTP POST data (therefore the query must use POST method) as if they were sent from a HTML form. Client must always include `login` and `passwd` fields filled with login and password of an user who authorizes the service actions.

Since client sends his/her login and password, it is obligatory to used HTTP protocol over SSL channel (ergo HTTPS). It is also recommended to use separate user's account for accessing services that has minimal necessary rights.

3.1.1.2 Service Response

The result is always encoded as XML file (the `text/xml` content) with UTF-8 encoding. Since the format may be slightly different for each service, no common XML schema is provided.

The file has `codex_service_result` root element and may contain following elements in defined order:

- `code` - This element is always present and contains a code with numerical representation of result status. Following codes are common for all services.
  - `-1` - Internal error. This result is returned only if something really bad happened (such as unhandled exception, wrong CodEx configuration, serious server problems etc.).
  - `0` - Operation succeeded.
  - `1` - Given service does not exist.
  - `2` - Invalid combination of login name and password.
  - `3` - Access to codex is restricted only for some users and user with given login name is not on the list.
  - `10` - User with given login name has not sufficient rights to use the service.
  - `11` - The query has invalid parameters.

  Other codes are service dependent.

- `message` - Error message that is ment to be shown to the user or entered into a log. This message is present only if error has occurred.
- `data` - Voluntary section that is present only if the service needs to return any data and the retrieving operation succeeded. Format of this element is service dependent.

3.1.2 Services Implementation

All services uses script '`service.php`' as a bootstrap instead of '`index.php`'. This file has some similarities as the '`index.php`' such as component initialization. However web-oriented components (eg. session) are not initialized.

Dispatching mechanism is also different. It is handled by `ServiceManager` class that finds selected service, checks user login information and executes the service. Gathered data and result code are used for assembling result XML data.

Services are stored in '`services`' subdirectory of the web-root directory. Each service has it's own file named '*`service-name`*`.php`'. Code is encapsulated in one class named `Service_`*`service-name`*. This class must be derived from `ServiceAbstract` class and implement it's interface. The most important is an `execute` method that implements the service job. For more details see code reference manual.

Service may also provide a XML schema that is placed in '`services/schemas`' directory. Details are described at each implemented service.

3.2 Creating New Services

Following text is a simple tutorial for creating new CodEx services. It is also recommended to incspect at least one of implemented services for getting better insight into the problem.

1. Create new service file in '`services`' directory. The file must be named `varservice-name.php`. The file must contain service controller class that is named `Service_`*`service-name`*. The class must be derived from `ServiceAbstract` class and implement abstract method `execute`.
2. Expected service parameters (that are sent via POST) must be listed in `$paramFilters` member variable. The list is taken by `ParamFilter` class so it must hold format of param filter rules. Mandatory parameters must be listed in `$mandatoryParams` member variable. This array contain only parameter names (as values). For more complex behaviour you may override the `filterParams` method. However if you do and you do not call inherited implementation, you need not fill in `$paramFilters` and `$mandatoryParams` since they become useless. See Section 4.1.5 [ParamFilter], page 39
3. If special rights are required for the service execution, you should override method `checkUserRights`. This method obtains the user object as an argument and returns true if the user has sufficient rights. It is strongly recommended to use `SecurityModel` class to do so. The `checkUserRights` method is always called **after** `filterParams` method therefore you may use POST parameters for rights verification.
4. Finally you need to write the service body in `execute` method. The method performs service task and returns true if the operation succeeded. If the method fails, a false value must be returned and member variables `$errorCode` and `errorMessage` must be set. The service should use positive numerical code that do not collide with general codes to report errors. Error message is not mandatory, however it is strongly recommended that the message is filled.

   Reference to simple XML element `data` from the result XML file is given to the `execute` method as an argument. If the service needs to return any data, it must format them into a XML fragment inside data element. If the final XML is somewhat difficult, the XML schema may be provided. The schema should be in XML Schema format stored in '`services/schemas`' directory in '*`service-name`*`.xsd`' file.

5. After the service is implemented and tested, service documentation should be added into this document. The documentation should contain full specification of query parameters, service functionality and returned data.

In the implementation you may use all libraries and modules implemented in CodEx, however it is forbidden to user web-specific code especially the session module.

3.2.1 Debugging and Security Matters

Debugging is more difficult since services has no "screen" to write to. Therefore all debugging messages and errors must be written to the log. New services must be also written more carefully since they are more likely to be exploited. All important modifications to the internal CodEx data or security decisions should be notified in security log.

3.3 Implemented Services

Following instances of CodEx services were implemented sofar.

3.3.1 `add_user` Service

Simple service that creates new user account. All user's information must be passed to the service in query parameters (see below). The service creates simple user account with no rights (common user account).

User provided for authorization must have at least `RIGHT_CREATE` general right for users.

3.3.1.1 Query Parameters

Except for general `login` and `passwd` parameters used for authorization process following parameters are required.

`user_login`
: Login of newly created user. The login must meet CodEx-login criteria and it must not be already used in CodEx.

`user_passwd`
: Password to the new user account. The password is given in plain text (not encrypted). Password may contain only alphanumeric characters.

`user_name`
: First name of the user (max. 50 chars), must be filled.

`user_middle_name`
: Middle name (or names) of the user (max. 100 chars).

`user_surname`
: Surname of the user (max. 50 chars), must be filled.

`user_email`
: Valid user's e-mail address.

All parameters are mandatory except the `user_middle_name`.

3.3.1.2 Results

Except for general result codes this service may also return:

- `100` - Given login already exists.
- `101` - Unable to save results into CodEx database.

If the operation succeeds, the service returns data section with single element `user_id`, which holds CodEx ID of newly created user.

3.3.2 `edit_user` Service

This service may be used for remote modification of selected user account. The query must contain parameters that identifies target account and list of attributes that should be modified.

User provided for authorization must have at least `RIGHT_CREATE` general right for users.

3.3.2.1 Query Parameters

Except for general `login` and `passwd` parameters used for authorization process following parameters are required.

`user_id` ID of the user account that should be modified.

`user_login`
: Login of the user whose account should be modified.

`set_passwd`
: If this parameter is present, the user's password will be changed to it's value.

`set_name` If this parameter is present, the user's name will be changed to it's value.

`set_middle_name`
: If this parameter is present, the user's middle name will be changed to it's value.

`set_surname`
: If this parameter is present, the user's surname will be changed to it's value.

`set_email`
: If this parameter is present, the user's e-mail address will be changed to it's value.

Exactly one of `user_id` and `user_login` parameters must be filled so the target user account may be identified. At least one of `set_` parameters must be present for the service will not be called in vain.

3.3.2.2 Results

Except for general result codes this service may also return:

- `100` - Target account does not exist. Invalid user ID or login were given.
- `101` - Unable to save results into CodEx database.

The service does not return any additional XML inside data element.

3.3.3 `delete_user` Service

This service may be used for deletion of individual user accounts. The query must contain either ID or login of the user account being deleted. User provided for authorization must have at least `RIGHT_DELETE` general right for users and the deleted account must not have any generic rights at all.

3.3.3.1 Query Parameters

Except for general `login` and `passwd` parameters used for authorization process following parameters are required.

`user_id` ID of the user account that should be deleted.

`user_login`
: Login of the user whose account should be deleted.

Exactly one of `user_id` and `user_login` parameters must be filled so the target user account may be identified.

3.3.3.2 Results

Except for general result codes this service may also return:

- `100` - Target account does not exist. Invalid user ID or login were given.
- `101` - Unable to delete the record from the CodEx database.

The service does not return any additional XML inside data element.

3.3.4 `change_membership` Service

Service for membership manipulation. It allows adding and removing user to/from a group. Restrictions and consequences of granting/revoking memberships are same as in GUI (e.g. user can not become a member of his/her own group). User provided for authorization must have at least `RIGHT_CREATE` general right for users and `RIGHT_EDIT` rights for selected group.

3.3.4.1 Query Parameters

Except for general `login` and `passwd` parameters used for authorization process following parameters are required. All parameters are mandatory.

`user_id` ID of an user, which is being added/removed to/from the group.

`group_id` ID of a group to/from which the user is being added/removed.

`operation`
: Type of operation performed. Value `0` stands for granting membership, value `1` stands for revoking.

3.3.4.2 Results

Except for general result codes this service may also return:

- `100` - Selected user does not exist.
- `101` - Selected group does not exist.
- `102` - The user is already member of selected group. This error is relevant only when granting membership.
- `103` - The user is not member of selected group. This error is relevant only when revoking membership.
- `104` - The user is owner of selected group.
- `110` - Unable to save results into CodEx database.

The service does not return any additional data.

3.3.5 `list_group_results` Service

Service for remote access to selected group members' results. It allows client to retrieve list of assigned tasks, all group members and all points and bonus points of these members.

User provided for authorization must have at least `RIGHT_READ` right for selected group.

3.3.5.1 Query Parameters

Except for general `login` and `passwd` parameters used for authorization process a `groupId` parameter is required. This parameter must contain an ID of a group from which the results are listed.

3.3.5.2 Results

This sevice does not have any special result codes of it's own. All retrieved data are stored in XML `data` element in following format.

First all the tasks are listed. Each task is encapsulated in one `task` element and has `id` attribute. This attribute holds the task ID from the database with '`t`' prefix (see example). Task contain three obligatory subelements in fixed order: `caption` and `obligatory`. Element `caption` holds a string with caption of the exercise. Obligatory points is an integer value which corresponds to `tasks.obligatory` database column.

List of users follows after the tasks. Every user is encapsulated in `user` element with `id` and `login` attributes that holds user's ID and login respectively. User has following subelements presented in fixed order. The first subelement is `name` that contains full name (name, middle name and surname) of the user.

The name is followed by list of `task_points` elements. Every element holds information about points that were scored to the user for one specific task. Element has attribute `id` that points out to the task in question and optionaly an attribute `bonus` with amount of task bonus points scored. Element content is an integer with amount of scored points. Only tasks solved by the user are listed here. The list is terminated by `task_sum` element with total sum of all points and task bonus points from all tasks scored for the user.

After task points the bonus points are listed (meaning external bonus points – not task bonus points). Format is very similar to task points. Each bonus points item is presented in `bonus_points` element. The element has `caption` attribute that identifies the bonus points. List is terminated with `bonus_sum` element that holds total sum of all bonus points scored for the user.

At the end there are elements `total` holding total sum of both task and bonus points and `done` element with boolean flag that signals whether or not the user has meet group requirements. The bool value is represented by `0` and `1` values.

For more information see the XML Schema for this service and following example of possible result.

```
<data>
    <task id="t1">
        <caption>Find The Minimum</caption>
        <obligatory>0</obligatory>
    </task>
    <task id="t2">
        <caption>Hippo's New Fence</caption>
        <obligatory>3</obligatory>
    </task>
    <user id="1" login="FOX">
        <name>Smart Fox</name>
        <task_points id="t1">10</task_points>
        <task_points id="t2" bonus="3">5</task_points>
        <task_sum>15</task_sum>
        <bonus_points caption="Extra Homework">4</bonus_points>
        <bonus_sum>4</bonus_sum>
        <total>19</total>
        <done>1</done>
    </user>
    <user id="19" login="TURTLE">
        <name>Slow Turtle</name>
        <task_points id="t1">4</task_points>
        <task_sum>4</task_sum>
```

```
            <bonus_sum>0</bonus_sum>
            <total>4</total>
            <done>0</done>
        </user>
        <user id="42" login="HIPPO">
            <name>Lazy Hippo</name>
            <task_sum>0</task_sum>
            <bonus_points caption="Overslept">-5</bonus_points>
            <bonus_sum>-5</bonus_sum>
            <total>-5</total>
            <done>0</done>
        </user>
    </data>
```

3.3.6 `sis_group_results` Service

A service specifically designed for SIS (Student Information System) of Charles University in Prague. This service provide basic information about student accomplishments in groups (point summary).

User provided for authorization must have at least `RIGHT_READ` right for every selected group. It is expected that this service has special SIS-agent account with general `RIGHT_READ` for all groups.

3.3.6.1 Query Parameters

Except for general `login` and `passwd` parameters used for authorization process a `groups` parameter is required. This parameter is an array of SIS-specific group identifiers that are matched against `sisGroupId` column of `groups` table. Proper encoding into application/x-www-form-urlencoded MIME is expected. Therefore the array is in fact encoded in multiple items with prefix `groups` and suffix `[id]`, where `id` is array key. Since keys are ignored, it is recommended to use standard numerical sequence (0, 1, ...).

For example groups `array('ABC', 'XYZ');` are encoded as attributes `'groups[0]' = 'ABC'` and `'groups[1]' = 'XYZ'`.

3.3.6.2 Results

This service does not have any special result codes of it's own. All retrieved data are stored in XML `data` element in following format.

Groups with provided SIS identifiers are listed, each in its own element. Since SIS IDs need not to be present (in CodEx), some (or even all) groups may be missing. Each group is represented by one element `group`, with attribute `id` that contains SIS group ID.

Group has arbitrary number of `user` subelements, each representing results of single member. One user may appear in multiple groups, since membership is not restricted in that way. User have attribute `login` which holds user's login. Logins of users from Charles University are the same as their personal numbers, therefore SIS may identify them easily. User has exactly three subelements in fixed order (each containing only a literal value):

- `name` – full name of the user
- `points` – total sum of points earned in the group
- `done` – bool value (`0` or `1`) indicating, whether the user has successfully completed all obligatory tasks.

For more information see the XML Schema for this service and following example of possible result.

```
<data>
<group id="sis_id_1">
<user login="1234">
<name>John Doe</name>
<points>120</points>
<done>1</done>
</user>
<user login="2345">
<name>Jane Smith</name>
<points>90</points>
<done>0</done>
</user>
</group>
<group id="sis_id_2">
<user login="2345">
<name>Jane Smith</name>
<points>150</points>
<done>1</done>
</user>
<user login="3456">
<name>Billy Johnes</name>
<points>42</points>
<done>0</done>
</user>
<user login="4254">
<name>Lucy Lue</name>
<points>54</points>
<done>1</done>
</user>
</group>
</data>
```

4 Libraries

4.1 MVC (Model-View-Controller) Framework

4.1.1 MVC General Overview

The MVC framework is represented by a set of classes for processing, routing and dispatching requests for displaying pages and performing actions. It implements the Model-View-Controller and Front-Controller design patterns.

This framework is partially based on the Zend Framework and its principles, however it does not share Zend's routing system.

4.1.1.1 Processing Requests

All the client's HTTP requests (except those for static data such as CSS, JavaScript, image files etc.) are routed to the `index.php` script. This script is also called 'bootstrap' and it is responsible for the initialization of all the necessary modules (such as log, session, database etc.).

After initialization, the request is analysed, routed to the proper page and dispatched.

We recognize two types of requests. In general they correspond to HTTP methods `GET` and `POST`. Every page must be ready to handle `GET` requests, since the main purpose of a page is to display it. Some pages can also handle `POST` requests, which are called *actions*.

4.1.1.2 Bootstrap (`index.php`)

The *bootstrap* script is mainly responsible for loading, initializing and configuring various subsystems. The bootstrap script performs the following steps, respectively:

- Includes of the most important library scripts.
- Loads configuration from the `config.ini` file.
- Ensures that HTTPS is used or blocks access to the CodEx application if these items are required by the configuration file.
- Initializes the error log and the security log.
- Opens or creates a session.
- Initializes the database module (however, a DB connection is not created until needed).
- Initializes the remaining minor modules (e.g. the mailing system or the Eval queue module).
- Prepare `PageManager` and `Dispatcher`, route and dispatch the request.

Refer to the corresponding manual sections for details on the various steps and modules.

4.1.1.3 Page Representation

Every page consists of two main parts – the *controller* and the *template*. These are implemented by a pair of dedicated PHP scripts in the page directory, along with other page data. (The data for each page reside in a separate directory).

The controller is represented by a single class derived from `Page_Abstract`. The controller encapsulates all operations and other things related to the page (such as rights testing, URL filtering etc.). The controller is also responsible for preparing all the necessary data for the template. Furthermore, it contains a method for every action that the page handles (except for component actions – described later). See Section 4.1.4 [PageAbstract], page 37

The template is simple script containing mostly of HTML with fragments of PHP. The PHP fragments insert the computed values, hide conditional sections or iterate some parts of the HTML code. The template expects that all dynamic data are prepared and sanitized by the

controller. However, it can modify or generate some data itself (e.g. code fragments generated by `HtmlHelper` or URLs). The template can also use the `tr` function for translating literals.

Data for insertion into the template are held in the `View` class. The `View` class acts as a carrier and proxy for the data. When user-defined textual data are stored in the view object, they are also converted from plain-text form to HTML from (escaped, sanitized). Failing to do so would have undesirable effects and would make the system prone to XSS (script injection) attacks.

Pages can use components – standalone parts of code (both PHP and HTML) that are meant to be used many times. See Section 2.2 [Components], page 15

4.1.1.4 Routing and Dispatching

The term 'routing and dispatching' stands for the mechanism of finding out to which page the request should be routed and performing the proper action on this page. This entire process is handled by the `Dispatcher` object.

The dispatcher checks the GET parameters first and routing parameters. According to these parameters it finds the proper page controller class and creates an instance. Then the page object is initialized and the proper action is executed. See Section 4.1.3 [Dispatcher], page 36.

4.1.2 PageManager

The `PageManager` is a singleton class which is responsible for managing paths and all things related to pages and their components. All the belongings of a page are stored in the directory of that page. Namely `controller.php`, which contains the controller class of the page and `template.php` with its HTML template.

`PageManager` assembles path to page's directory from given module and page name according to internal settings. It also performs the same job for the components Furthermore, the page manager is able to load the script containing the code of the page controller and create instances of page controller objects.

4.1.2.1 Page Paths and Files

All pages are stored in the '`pages`' subdirectory. Similarly, components are stored in the '`components`' subdirectory. The path to the page directory is assembled as:

`pages/`*`module`*`/_page name/`

For example: For `module=groups/tasks` and `page=submits` the controller script is situated in

`pages/groups/tasks/_submits/controller.php`

and the template is in

`pages/groups/tasks/_submits/template.php`

Additional files can be present in the page directory. CSS styles and JavaScript code are can be stored in the files '`style.css`' and '`script.js`', respectively. They are automatically attached to the constructed XHTML document using `<link>` and `<script>` tags. Only those scripts and style-sheets that cannot be stored on the global level should be situated here. Note that the style-sheet '`style.css`' does not reflect which style is selected in the configuration. Therefore, only style-independent CSS rules should be placed there.

4.1.2.2 Component Paths and Files

A component resides in a directory derived from its name. For example, the `menu` component has its controller and template stored in:

`conponents/menu/controller.php`

```
components/menu/template.php
```

Components cannot define their own CSS and JavaScript files. However, they can store these files under the global `style` directory. See Section 2.2 [Components], page 15

4.1.2.3 Generics

Generic scripts are stored in the '`generics`' directory which has two subdirectories: '`controllers`' and '`templates`'. Generic controllers are controller parent classes that can aggregate some useful methods used on several pages. Generic templates represent parts of the page that are often displayed such as a header, a footer, error messages etc.

Access to these scripts is also provided by `PageManager`, which assembles the path to a given generic controller or template.

4.1.3 `Dispatcher`

Dispatcher encapsulates the most important feature of the MVC framework – routing and dispatching requests. Requests are processed by page controllers and the dispatcher is responsible for finding the controller and calling its proper method. Which page should be displayed or which action should be performed is determined by three GET parameters (which are part of the URL): '`module`', '`page`' and '`action`'.

`module` Defines the logical part of the application, where the pages belongs to. Modules are effectively implemented as subdirectories. The module name can only contain letters, digits and slashes, thus it can refer to subdirectories. For example, the module '`groups/tasks`' will have all pages stored in the '`pages/groups/tasks`' directory.

If the module is not defined, the default module is used. The default module does not have a name and all its pages are stored directly in the '`pages`' directory.

`page` Defines which page in the module will be displayed. The page name can only contain letters and digits. All source files belonging to a page are stored in a directory named '`pages/module/_page`'. If the specified page does not exist, the default page called '`index`' is used. It is presumed that every module has a default page.

`action` Defines the action which should be performed on the page. Actions are only valid for POST requests. If the action is empty or the request is a simple GET, the page is only displayed. Otherwise the proper action method is found and executed. When an action is completed the script will send an HTTP redirect response (code 302) to the client so the browser will not cache any POST requests in history.

The `action` can only contain letters and any method attending an action must be named after the action with the `Action` suffix (i.e. for a '`delete`' action the method `deleteAction` will be called).

Since the routing system must be modular, an action can also be dispatched to a page *component*. In this case the attribute `action` must be encoded as an array with one field. The key of this field is the ID of the component and value is the name of the action. For example, an action '`update`' for a component with ID 2 will be encoded in the URL as '`action[2]=update`'.

Routing is quite simple since it only loads routing parameters described above and verifies them. In case the values are not valid, the default values are used. After routing, the parameters can be manually overridden using the methods `setModule()`, `setPage()` and `setAction()`.

Dispatching the requests consists of finding the proper page controller (using `PageManager`), loading its code and creating an instance of the controller class. After that the page is initialized. If initialization fails (i.e. because of insufficient rights), the dispatcher tries to load a *fallback*

page. For each page there is another page specified as its fallback page. The dispatcher follows the hierarchy of fallback pages until it reaches the '`index`' page in the default module. If this page does not exist, the dispatching mechanism fails.

If any action is defined, it is dispatched. Otherwise the page template is loaded and the page is rendered using this template.

4.1.4 PageAbstract

`PageAbstract` is the default parent class for all page controllers. Each controller must also hold up to a strict naming policy that is implied by `Dispatcher`. A controller name is composed as:

`Page_<`*`module name`*`>_<`*`page name`*`>`

Since the slash character cannot be used in class names, all slashes are replaced with underscores in the module name. For example, the controller class for page '`index`' in module '`exercises/solutions`' must have the `Page_exercises_solutions_index` identifier.

4.1.4.1 Initialization and Rights Verification

Initialization consists of the following steps:

- *Pre-initialization* routine – this normally does nothing. However, some derived controllers can make use of it.
- Create component controllers and related objects.
- Filter GET parameters and add default values for missing parameters.
- Check whether the user has sufficient rights to display the page or perform the requested action. The check is performed by the `checkRights` method that can also be called separately (but only after the GET parameters are filtered).
- Initialize all components that have been created.
- *Post-initialization* routine – this normally does nothing. However, some derived controllers can make use of it.

If any of these steps fails, the whole initialization fails and the dispatcher will try the fallback page. The fallback page (and its module) is specified by the member variables `fallbackModule` and `fallbackPage`. If any of these parameters is `null`, the current module or page is used respectively (this could be used to only set the fallback page, but stay in the current module). All pages should point transitively to the '`index`' page in the default module and this page must never fail to initialize, or else the dispatching system breaks down.

4.1.4.2 Filtering GET (URL) Parameters

The controller defines a list of GET parameters that the page accepts. The list is represented as an array stored in the member variable `$getParams`. The array is indexed with parameter names and each value is a filtering rule specifying a data type and optionally some constraints. Actually, the format of `$getParams` corresponds to the format of filtering rules for the `ParamFilter` class. See Section 4.1.5 [ParamFilter], page 39

The page can also define additional parameters accepted for individual actions (in the `$getActionParams` member variable). The list of action GET parameters is indexed with the name of the action and each value has the same format as the `$getParams` field. If any action is performed, the list of accepted general GET parameters is merged with the list of GET parameters for that action (where the action parameters have higher priority).

Since the whole system is modular, GET parameters from all components are added into `$getParams`. Each component has parameters in the form `C`*`component ID`*`_`*`parameter name`*. Translation from internal name into full name provides components method `getGetParamName`. When a component is registered all its parameters are translated into full name and added

into `$getParams`. Therefore it is forbidden for common page GET parameters to contain an underscore character since it may cause a collision with some components.

The filtering is performed by the `sweepGetParams` method, which is provided with an array of parameters (usually the `$_GET` array) name of the current action and a Boolean value `defaults`. If the defaults flag is `true`, it adds the default values for missing parameters. If it is set to `false`, it will remove the default values from the list. Otherwise (if not set or if set to `null`, no such modifications are performed).

4.1.4.3 Creating URLs and Redirects

A situation often arises when one needs to create a URL for a specific page within the framework and set up the proper GET parameters. Some parameters need to be added, some need to be removed or changed. URLs are manufactured by the `createUrl` method of the page controller. This method only creates links for the page that corresponds to the controller object on which the method is called. Therefore, if one needs to create a URL for any page, the controller object for that page must be created first.

The method `createUrl` is completely configurable. It can create URLs for actions, add or remove GET parameters and it can create URLs both for HTML (encoded to special HTML entities) or plain URL (i.e. for using in the `Location` HTTP header). For more information see the code reference documentation.

Before the parameters are formatted into a URL, they are filtered using standard page filtering rules. This filtering removes or modifies invalid parameters. Parameters having the default value are removed so the URL is not cluttered with irrelevant information.

Besides creating URLs, the page controller also handles redirections. The `redirect` method takes the very same parameters as `createUrl`. Its purpose is to create a URL and insert it into the `Location` HTTP header. This method also sets the HTTP response to 302 '`Redirect`'. After the header is modified, the script is stopped so it will not produce any other output.

The mechanism of redirects is used when a restart of the script is required for some reason from a deeply nested method (thus creating another mechanism would be complicated) or after a POST request so the browser will not hold any POST data in history.

4.1.4.4 CSS and JavaScript

Cascading style sheets and JavaScript code files are important part of the page. Therefore, their management is incorporated into the page mechanisms. Every page controller has the `$css` and `$js` member variables containing arrays with lists of CSS and JavaScript files that will be attached to the page. The generic template `header.php` automatically inserts `<link>` and `<script>` tags into HTML headers for each item in the `$css` and `$js` arrays, respectively.

Cascading-style-sheet files are stored in the '`style/`*`style name`*`/css`' directory. Corresponding images used for page design are in '`style/`*`style name`*`/pic`'. The *style name* can be configured with the configuration option `application.style` in `config.ini` and its default value is '`default`'. This concept is planned to be modified in the future so the user will be able to select a style individually.

All JavaScript files are stored in the '`js`' directory and do not implement any configurable concept as the CSS files.

The components also require some CSS and JavaScript files. Therefore, they also contain their own lists that are merged with the lists on the page.

4.1.4.5 Page Rendering

Rendering of a page is performed by the `renderView` method. This method expects the name of the template to which the page should be rendered. The entire process consists of the following steps:

1. Preparation of a `View` object. This part is handled by the `createView` method that creates a view object and initializes its most important member variables. This method should not be changed by derived classes.

   Method `prepareView` is called at the end of `createView`. This method is meant to be redefined by descendants who can initialize additional view attributes.
2. Rendering of page headers. Every page has the member variable `$headers`, which contains a list of generic templates and component objects displayed before the main page. Generic templates are situated in the '`generics/templates`' directory and rendered using the same view object as the main page. Components are situated in their own directories and rendered using their own views. See Section 2.2 [Components], page 15
3. Rendering of the main page. This operation is performed by the `render` method of the view object.
4. Rendering of page footers. Every page has the member variable `$footers`, which contains a list of generic templates and component objects displayed after the main page. The principle is the same as with the headers.

4.1.4.6 Error and Informational Messages

Every page has a general way to display error and informational messages. Messages are stored in the '`_errors`' and '`_infos`' arrays in the page session storage. These messages are cleared every time the user sends a POST request, since page storage is cleared as well.

If the error or informational messages should be displayed, the Boolean GET parameters `_error` or `_info` must be present, respectively. This precaution is used because the session must not be changed by a GET request (since GETs should be idempotent). These parameters will not collide with other GET parameters since normal parameters must not contain an underscore. These parameters are automatically set when a new message is added into the session. They are also not added into any links unless they are explicitly enumerated.

4.1.5 `ParamFilter`

The `ParamFilter` class provides an engine for filtering and validating parameters passed to the PHP script via HTTP requests. It operates on an associative array (usually on `$_GET` or `$_POST` superglobals). In the following text the term *parameter* shall stand for one name-value pair in an associative array.

Each filter has a set of filtering rules. The rules describe which parameters are allowed and the constraints on the value of each parameter. The *ruleset* is stored in an associative array in which the keys are parameter names and the values are *constraints* for the value of each parameter.

Constraints are represented as numbered arrays and each field represents one constraint *attribute*. The first *attribute* (index 0) is always the data type of the parameter (all types are represented as `ParamFilter` constants `ParamFilter::PARAM_`*type*). The meaning of other *attributes* depends on the data type and will be described below. These *attributes* are all optional.

- `ParamFilter::PARAM_INT` – the parameter must be an integer.
  - [1] – lower bound (if the value is less than the lower bound, it is clamped to fit).
  - [2] – upper bound (if the value is greater than upper bound, it is clamped to fit).
  - [3] – default value (if not defined, '`0`' is used).
- `ParamFilter::PARAM_FLOAT` – the parameter must be a floating-point value in decimal notation.
  - [1] – lower bound (if the value is less than the lower bound, it is clamped to fit).
  - [2] – upper bound (if the value is greater than the upper bound it is clamped to fit).

[3] – default value (if not defined, '`0.0`' is used).

- `ParamFilter::PARAM_STRING` – the parameter is treated as a string.

  [1] – maximum length (if the string is longer, it is trimmed).

  [2] – regular expression (in POSIX extended format) against which the string is tested. If not defined, the validation is skipped.

  [3] – default value (if not defined, empty string is used).

- `ParamFilter::PARAM_DATE` – the parameter must be date-time string in application-specific format. The value is converted into a UNIX timestamp during validation.

  [1] – minimum timestamp (the oldest allowed time value).

  [2] – maximum timestamp (the newest allowed time value).

  [3] - default value (if not defined, the current timestamp is used).

- `ParamFilter::PARAM_BOOL` – the parameter must be a Boolean (represented as '`0`' or '`1`').

  [1] – default value (if not defined, `false` is used).

- `ParamFilter::PARAM_OBJECT` – the parameter must be an ID of an existing object in the database. The object is loaded from the database during validation. If the ID is not valid, `null` is loaded instead. Object are loaded using table gateways. See Section 4.3.2 [Table Gateways], page 49.

  [1] – the name of the table gateway that will be used to retrieve the object. This attribute cannot be omitted.

- `ParamFilter::PARAM_UNCHECKED` – the parameter has unspecified type and should not be validated.

  [1] – default value (if not defined, empty string is used).

Default values can have any type (the type is not bound by the type of the parameter). This can be used for detecting missing parameters. For example: We have an *int* parameter with the default value `null`. If the parameter was set, the filter ensures it will be converted to an integer and its bounds verified. Otherwise, it just provide the default value (`null`).

As a convenience, the rule can be represented as a scalar value (the data type) if it has just one field. In other words,

'`'parameter_name' => array(ParamFilter::data_type)`'

is equivalent to

'`'parameter_name' => ParamFilter::data_type`'

Example:

We are going to define a page with the following GET parameters:

`amount` – an integer from 10 to 50 inclusive (the default being 20).

`people` – an unchecked parameter (the default value being an empty array).

We shall also define additional parameters for the `delete` action:

`id` – user object (represented by its ID).

`code` – an integer without explicit bounds.

`filter` – string containing just letters (max. 64 chars).

The corresponding rulesets will be defined as follows:

```
$this->getParams = array(
    'amount' => array(ParamFilter::PARAM_INT, 1, 50, 20),
    'people' => array(ParamFilter::PARAM_UNCHECKED, array())
);
```

```
$this->getActionParams = array(
    'delete' => array(
        'id' => array(ParamFilter::PARAM_OBJECT, 'users'),
        'code' => ParamFilter::PARAM_INT,
        'name' => array(ParamFilter::PARAM_STRING, 64, '^[a-zA-Z]+$')
    )
);
```

4.1.6 `View` Class

The `View` class is derived from `Zend_View` and it is used for page rendering. Its main purpose is to carry data from the controller to the template where the data are displayed. Some data are not directly stored in the view object, but nevertheless they can be transparently accessed using the object's getters.

The `View` object is always attached to a specific page or component. It always contains the following attributes:

- Contents of all items in the `info` section of the '`config.ini`' file. For example '`info.title`' is accessible through `$view->title`.
- The rest of the configuration is accessible through `$view->config`.
- `picDir` holds path to directory with pictures in the current visual style.
- '`pageName`' holds the translated name of the current page.
- The parameters '`module`', '`page`', '`components`', '`css`', '`js`' of the page controller are readable through the view object as well. See Section 4.1.4 [PageAbstract], page 37
- If the view object is used for rendering a component, the reference to the component controller is stored in the `component` attribute.

When the view object is used for rendering a page or a more complex component, nested components can be rendered and inserted into the output stream by calling `renderComponent`. This method must be provided with a valid component ID.

The `View` class is also used for rendering URLs. It contain the methods `createUrl` and `createPageUrl` that make use of the page URL creating and filtering mechanism to only generate URLs with valid parameters. The first method is used for creating self-referring URLs (to the current page but possibly with different parameters), the second method creates a URL to any page. Both methods can also merge the GET parameters passed to them with the existing ones.

4.1.7 Session

Since HTTP is a completely stateless protocol, session information must be held somewhere. CodEx sessions are based on classic PHP sessions and cookies.

4.1.7.1 Implementation Details

Sessions are implemented using `Zend_Session` and `Zend_Session_Namespace`. Our own static class `Session` builds upon these. Session initialization is called from the bootstrap before any other module needs to access session data.

The *session ID* is stored on the client side as a cookie. Since this cookie could be stolen (even though CodEx normally use secured HTTP and we prevent XSS attacks), additional verifications and security precautions are taken.

- A *validator* cookie is stored in the client's browser. It has a random value which is also stored in the session. Each time the session is re-open, the value of the cookie is compared with the value stored in the session. Furthermore, the value of this validator changes every time the user performs a POST request. If the validator does not match, the session is destroyed.

  This cookie may seem redundant, but an attacker could easily forget to steal it along with the ID cookie.
- The *IP address* of the client is also stored in the session and verified each time the session is re-open. However, since we live in an age of NATs and dynamic IPs, this precaution is not as strong as it could be. If the address does not match, the session is destroyed.
- Session storage has also *expiration* time limit. If the user does not perform any action within this time limit, the session is destroyed.

Session verification failure is considered to be a security incident and so it is written into the security log.

The session also has a versioning system. The current version number is specified directly in the code of the `Session` class. This number is also copied into the session when it is created. If the version does not match version stored in the session, the session is destroyed. This is useful for complex upgrades which require logging out all active CodEx users.

4.1.7.2 Data Namespaces

No module accesses session storage directly. Instead they use the abstraction of a *session namespace* provided by the Zend Framework. Namespaces should only be created via the `getNamespace` method of `Session` since this method caches the created `Zend_Session_Namespace` objects.

All data stored in the session are somehow related to the current user. Therefore, when the current user logs out, the session is destroyed. Namespaces are used to hold the following data:

- Each page has its own namespace that has the same name as the controller class of that page. This namespace is designed only to hold temporary data. Every POST request deletes all namespaces of this kind.
- Each page has also has a persistent namespace. It has the same name the controller class of the page, prefixed with '`Persistent_`' (i.e. '`Persistent_Page_index`'). Data in these namespaces are permanent, they are kept until they are explicitly deleted or the whole session is destroyed.
- Some data about the current user (the user who is logged in) are stored in the `CurrentUser` namespace. The most important field stored here is ID of the user. If no user is logged in, this value is missing.
- The *data access library* uses the session to as a cache for database access. All namespaces used for this purpose are prefixed with `"Cache_"`. See Section 4.3 [Data Access Library], page 49

The default namespace and the `security` namespace are reserved for internal use. w

4.1.8 Other parts

4.1.8.1 Application Configuration and Constants

Application configuration is stored in one file named `config.ini` in the source root directory. All configuration parameters are described right there. The configuration file is loaded into an instance of the `Zend_Config` class.

Application parameters that are not meant to be changed by the deployer or CodEx administrator are held in the `Constants` class. The purpose of this aggregation is to simplify code tweaking by keeping all important parameters in one place.

4.1.8.2 Log and Error Reporting

Error logging is performed by the `Zend_Log` class. The configuration of the logging system is stored in `config.ini`. It is possible to enable up to three independent writers (back-ends).

- *Output log* that adds messages directly to the PHP output. This writer is used mainly for debugging purposes.
- *File log* that appends messages into the CodEx log file.
- *E-mail log* that sends messages to a given e-mail address.

At least one of these writers must be active. Each writer has a severity filter that defines which errors will be sent to that writer. For example, it is possible to log all errors except debug messages and to also send severe errors via e-mail.

The default PHP error-handling routine is overridden so that PHP error messages are not inserted directly into the output stream unless the Output log is activated. Furthermore, almost the entire code is wrapped in the main try-catch block that handles and logs unhandled exceptions. When the script encounters and unexpected situation that could mean a severe problem (such as being unable to create a temporary file), the script raises an exception that gets caught in this try-catch block. When such exception is caught, it is logged and a general error message is displayed to the user.

4.1.8.3 Translations

CodEx is a multilingual application. Therefore, it must have a concept for text translations. The default language is English and we have also completed a Czech translation.

The translation API is provided by a `Zend_Translate` object which loads the message catalog from CSV files. All string literals in the application are written as arguments of the global function `tr()`. (Except for a few spots where this was not possible). We use a global function since its calling syntax is short and it can be used almost anywhere.

The language of the application is set with the configuration option `application.language`. The available values are '`english`' and '`czech`'. If the value is empty or the configuration is missing, the default language (English) is used. This setting is global and cannot be overridden on a per-user basis, since the specifications of exercises and other data in the database are always in a single language, anyway.

The translations of messages from a PHP source file reside in a file named '`czech.csv`' (or possibly '`<language>.csv`' if there are more translations) in the same directory. The translations of all messages need not be present, if the messages are already translated elsewhere. Thus, the translations are spread throughout the source tree and generally located fairly close to the place where they are used.

The script `deploy/lang.php`, usually run through the makefile, collects the messages from all `csv` files and collects them into a single message catalog for each language in a file named '`src/lang/<language>.csv`'. This file is then processed by the translation engine. Thus, if you change any translation, you should run '`make`' to re-build the message catalog(s).

There are also simple tools available for helping with writing and managing translations in the `tools` directory. '`xl <file1.php> [<file2.php> ...]`' extract messages from the specified source files and appends new messages to the '`czech.csv`' file in the current directory, '`xlall`' is a shorthand for '`xl *.php`'. They both shorthands for '`xlang`', which is the general tool having several options to control its behavior.

Finally, '`xspell`' runs a spell-check on the message catalog (that must be built first with `lang.php`) using the PHP `pspell` extension and outputs a list of words that are not in the dictionary (and not in one of the white-lists) and also words that have been blacklisted. (The word lists are located in the directory '`tools/wordlist`'.

4.1.8.4 HtmlHelper

`HtmlHelper` is static class that aggregates methods for rendering XHTML code fragments. It is meant to be used in templates as well as in other core, where XHTML fragments must be created (i.e. forms).

Main functions that `HtmlHelper` provides are:

- Creating URLs for pages, CSS files, pictures etc.
- Formatting UNIX timestamps into strings, points, difficulty levels and other specific attributes.
- Creating hyperlinks and action buttons (trivial forms with one button).
- Creating vertical-oriented texts. Texts are created as `<img>` tags with reference to general '`vertical_text.php`' script that renders texts as images.

4.2 CFI

This section describes the Class-based Filesystem Interface, or CFI for short. The CFI is a file-system-like abstraction designed to allow exposing remote-file access capability to CodEx users in an elegant, secure and efficient manner. It is primarily employed by the File Manager component for primary storage, archiving and extracting files. In the more narrow sense CFI refers just to the set of ICFI_xxx interfaces and in the broader sense also to the virtual file-system drivers and associated classes.

4.2.1 Introduction to CFI

CFI was designed with respect to the specific requirements of remote-file access in CodEx. Most of the time files are just copied or moved around, filtered, packed or unpacked. Copy is by far the most frequent operation. Operations on directories with exercise data require versioning. Security is of prime importance as the commands come from (and pass through) untrusted environment.

For the purpose of this manual, we will denote file system exposed by the operating-system interface the 'real file system' and the file-system-like object exposed through CFI the 'virtual file system'.

4.2.2 CFI Interfaces

CFI is sort of an objective interface to a file-system driver, with transaction support. The primary interfaces are: `ICFI_Filesystem`, `ICFI_Entry`, `ICFI_Directory` and `ICFI_File`.

4.2.2.1 File-system Interface

To allow a user to remotely access files on the server a class implementing `ICFI_Filesystem` is created. This interface represents a directory tree. It might correspond to a subtree of the real file system, an archive, an XML file etc. The operation (method) `getRoot` returns an `ICFI_Directory` object representing the root directory of the file system.

The file system is most often used in one of two ways: either as read-only, or in transaction-mode. In read-only mode one simply obtains the root directory and then proceeds to examining the file system. Any write-operations are forbidden and would yield undefined results.

In transaction mode a transaction is open first with `ICFI_Filesystem::begin`. Then the file system is examined and modified and finally the transaction is either committed with `commit`

or rolled-back with `rolllback`. It is also permitted to start in read-only mode and then begin a transaction. However, no object obtained before starting the transaction may be used afterwards.

In the typical case of versioning, the `begin` operation typically creates a temporary copy of the previous file-system version and `commit` inserts the temporary copy as a new version.

4.2.2.2 Directory Interface

The `ICFI_Directory` interface allows getting the list of entries in that directory (a list of `ICFI_Entry` objects), getting the parent directory (unless its the root directory), creating a subdirectory and creating a file.

One of the most notable operations is `ICFI_Directory::createFile`. For files in CFI can only be created or replaced, but not modified. Files are thus, essentially, immutable. The `createFile` operation takes two arguments: name of the file to create and an `ICFI_File` object, representing another file (a sequence of bytes). The newly created file's contents are an exact replica of the data read from the second argument.

An file object can be obtained either from a directory entry (allowing copying of files) or 'from outside' by creating a file object that takes its data from elsewhere (such as from the real file system, from a string, from a compressed file etc.)

4.2.2.3 Entry Interface

A directory entry (`ICFI_Entry`) has properties such as a name, size (in bytes), time of last modification and corresponds to either a file or a directory. The `getObject` operation returns a file or directory object in accordance with the type.

The directory entry can be either renamed (obtaining a new name but remaining in the same directory), moved into another directory (keeping its original name) or deleted. In all cases if the entry's object has been obtained earlier, it is now invalid and must not be used anymore.

4.2.2.4 File Interface

A file is actually no more than a tuple (fixed-length sequence) of bytes. An `ICFI_File` objects provides in addition a position-pointer. The main two operations are reading some bytes from the current position (`read`) and rewinding the file to the beginning (`rewind`).

The `read` operation is especially optimized for allowing efficient implementation of contracting filters without read-ahead (See Section 4.2.6 [File Filtering], page 47). It has one argument `length`. If any bytes are left in the file, it reads 1 up to `length` bytes from the file. It may *and will* read less than `length` bytes even if more are available in the file.

There is also a special operation `getRealPath` for efficiency's sake. It is used when a copy of the file in the real file system is required, such as when it needs to be passed to an external utility (e.g. `tar`). This operation creates a copy of the file in the real file system (unless there already is one) and returns its pathname. In many cases, however the file object is actually backed by a file in the real file system and in this case the method simply returns its name. (As files are immutable, it is not necessary to make a copy). This sort-of lazy copy is one of the tricks that makes CFI more efficient.

4.2.3 CFI Semantics and Security

4.2.3.1 File-system Semantics

While the semantics are essentially dependent on the individual file system drivers, some characteristics are recommended by the interface, such as: a `createFile` operation shall succeed even if the designated file already exists. In such case the existing file will be deleted and replaced

with the new rile. A `rename` or `move` operation shall fail if a file or directory with the destination name exists.

Also, as code using the CFI is likely to use '/' as a path-component separator (keep in mind that CFI does not deal with paths), it is not recommended to allow '/' in names.

Implementations are free to define any additional restrictions on allowed operations they see fit. It is possible, for example, to define quotas, restrictions on filenames or levels of directory nesting. It is possible to forbid creating directories or even make the file system read-only. The file system has a chance to give a very rough hint to its caller about which operations are allowed and which are not through the method `CFI_Filesystem::getFlags`. This method, defined by the implementation returns a combination of flags such as `CAN_MODIFY` (whether the file system is writable at all), `CAN_CREATE_FILES` etc.

4.2.3.2 File-system Security

The individual file-system implementations are responsible for keeping a sandbox-like secure environment. This is usually not very complicated, however, due to the simplistic nature of the CFI interface. It mostly suffices to check filename syntax and not return the parent directory of the file-system root to keep the filesystem safe.

4.2.4 Filesystem-based Driver

The filesystem-based driver (or filesystem), `CFI_FS_Filesystem` (FSFS) provides an implementation of the CFI interface backed by a directory (i.e. a subtree) of the real file system.

4.2.4.1 File-system Versioning

The file system operates in one of two modes. When the constructor is invoked with one parameter, pathname of a directory, it operates in *in-place mode*. In this mode all modifications are made directly on the specified directory and transactions are essentially ignored. When the constructor is invoked with two parameters, *srcDir* and *destDir*, the file system operates in *versioning mode*. In this mode the contents of *srcDir* are copied to a temporary directory upon `begin`. Any modifications are made in the temporary directory. Issuing `commit` moves the temporary directory to the final location specified by *destDir*.

The temporary directory is provided by class `TempDir` and should reside on the same file system as *srcDir* and *destDir*. If so, the files are not copied during `begin`, but hardlinked, and upon `commit` the move operation is atomic. This means it takes little time to create a new version and also preserves a significant amount of disk space.

Obviously one must be aware of the interlinking of the different versions. If somebody modified a file in one version, he would change the file in other revisions of the file system and that would be unacceptable. However, a CFI versioned file system is only ever modified by CFI and CFI files are immutable. When a file is to be replaced, it is first unlinked and then a new file is created in its place. This is sufficient to ensure that other revisions of the file systems are never affected.

4.2.4.2 Filename Checking and Security

If some external tools used by CodEx encountered filenames containing internationalized or special characters, it could potentially lead to problems. For this reasons FSFS restricts characters in filenames to the set recommended for GNU software: alphabetic characters (a-z, A-Z), digits (0-9), dot, hyphen and underscore. Any attempt to create a file or directory with an invalid name triggers an error.

The file system does not allow retrieving the parent directory of the file-system root and it does not produce symlinks. Therefore, as long as no symlinks are introduced to the file-system

from outside (through an error in other CodEx module, for example) there's no way the user could access any files outside of the virtual file system.

FSFS currently does not impose any additional restrictions (quotas, directory nesting level etc.)

4.2.5 Archive-based Drivers

The archive-based driver `CFI_Tar_Filesystem` and `CFI_Zip_Filesystem` provide the illusion of operating with files directly inside a tar or zip archive.

In reality, these drivers are only used to extract all files from an archive, or to create a new archive from a set of files and directories. They are optimized for this case, not for the case of random-access (which would be very difficult, anyway).

The implementations simply unpack the contents of the archive to a temporary directory, construct an instance of `CFI_FS_Filesystem` in *in-place* mode and hand the caller the caller the root directory of this filesystem. Upon `commit` the file system simply packs the temporary directory into a new archive.

The constructor for both file-system classes has two parameters, the first specifying the input archive and the second specifying the name of the output archive. If the input archive file is not specified, the file system starts empty. If the output file is not specified, the file system is read-only.

These file systems are not used by the File Manager for primary storage, only for packing and unpacking files for upload/download. Stringent filename checking as in `CFI_FS_Filesystem` is not necessary. When files from the primary storage are packed for download, they already have valid filenames. Conversely, if the uploaded archive contains files with unwanted filenames, the copy to the primary storage will fail. If these file systems were ever used for primary storage, stricter filename checks would have to be added.

Still, some security precautions are taken. Archives, coming from the outside, may contain malicious data. For both tar and zip the list of files they contain is first check to ensure there are no invalid paths (such as paths containing '..' components). For tar archives care is taken not to extract file ownership, permissions or symbolic links.

4.2.6 File Filtering

A file filter (`ICFI_Filefilter`) is an object that, being passed a file object, returns another file object. Filters can be passed to functions as parameters and collectively applied to sets of files. In CodEx filters are used for gzip and bzip2 compression and decompression of files (`GzFileFilter`, `Bz2FileFilter`, `UgzFileFilter` and `Ubz2FileFilter`) and for canonizing the format of text files (`TextFileFilter`).

In practice the filter class is just a wrapper that instantiates a *filtered-file class*. A filtered file class is a class implementing `ICFI_File` that has a one-parameter constructor taking another file object as the parameter. The filter interface is just a formalization of this that allows filters to be passed as parameters to functions (and instantiated multiple times).

4.2.6.1 Contracting Filters

Simply put, a filter is contracting if it ever produces less bytes than it consumes. A prototypical example is the CRLF to LF conversion implemented in the `TextFileFilter` class.

If the `read` operation was bound to return exactly `length` bytes whenever they were available, a perhaps not critical, but unwelcome inefficiency would be introduced. Consider a file containing solely of CRLF pairs and an operation `read(n)`. If the filter was not allowed to read-ahead (although there is no such case in CodEx), it would have to issue a `read(n)` on its source file, too. It would get exactly `n/2` bytes, after which it would have to issue a `read(n/2)` to read

the remaining bytes, then `read(n/2)` etc. When 'short read' is allowed, the number of read operations will be *logn* times less.

This optimization is not cricial in CodEx, as read-ahead is acceptable.

4.2.6.2 Compression and Decompression Filters

The filters `GzFileFilter`, `Bz2FileFilter`, `UgzFileFilter` and `Ubz2FileFilter` implement gzip and bzip2 compression and are implemented through PHP's gzip and bzip2 extensions. Decompression occurs on-the-fly with low buffering. When compressing, the original file is completely compressed to a temporary file first and then the compressed file object is backed with this temporary file, as the compression interface only allows writing the compressed data to a file, but not reading it directly.

The main use of these filters is to decompress files uploaded by the user and compress files send to the user for download.

4.2.6.3 Text File Filter

The text filter is used for 'canonizing' the encoding of text files uploaded by the users. The canonical format is a plain-text file in ASCII or UTF-8 format, with newline denoted by LF and no byte-order mark (BOM, which is not standard).

The filter primarily deals with text files coming from Microsoft Windows(R), that use CRLF as a line-terminator and may start with a BOM (or, in the worst case, even contain BOMs in the middle, in case of concatenating more files).

The filter removes any CR characters and any byte-order marks encountered in the file. It keeps read-ahead to a minimum.

4.2.7 CFI Utilities

The CFI Utilities is set of classes built upon the CFI interface that allow recursive copying, deleting and moving of whole subtrees (akin to UNIX '`cp -r`', `rm -r`, '`mv`') plus a multi-rename operation.

The classes `CFI_Util_Copy`, `CFI_Util_Move` and `CFI_Util_Delete` all have a very similar interface and structure. Each contains an *operation*`Tree` and *operation*`DirContents` method (where *operation* = copy, move, delete). *operation*`Tree` operates on a subtree (i.e. on a directory and all its contents, recursively), while *operation*`DirContents` only operates on the contents of a directory.

A CFI file filter can be specified to `CFI_Util_Copy`, that will be applied on the contents of each file being copied. The move operation always tries to move whole directories in one step. If that fails (because the destination directory exists) it proceeds to recursively merge the contents of the source and destination directory.

Finally, the `CFI_Util_Rename` provides two operations. `rename` finds a file with a given name in the specified directory and renames it to a new name. `renameMulti` takes a directory and a rename-list. The rename list is literally a list of *old_name-new_name* pairs. The `renameMulti` method will try to perform all the renames 'simultaneously' whenever it is possible. It is not possible if and only if either *new_name* conflicts with an existing filename that will not be moved or when two *new_names* conflict. In such case `renameMulti` will rename as much as possible (without overwriting anything).

The method only uses simple rename operations and a single temporary name (and only if necessary). This way it is very fast, robust and can operate even on file systems with extremely restricted operations. (Note that a CFI file system might not allow creating a subdirectory, for example). It needs to solve conflict chains and conflict cycles. The algorithm for doing this is

based on the solution of the *Great Wall Parking Problem* from IOI 2000, Beijing. See comments in the source code for details on the problem and algorithm.

4.2.8 CFI Helper Class

The helper class `CFI_Helper` is intended to concentrate code commonly used by file-system implementations. Currently the only method is `tempCopyFile`, that takes a CFI file as an argument, creates its temporary copy in the real file system and returns its filename. It is frequently used to implement the method `ICFI_File::getRealPath()`.

4.3 Data Access Library

CodEx stores data both on the file system and in a database. An efficient abstraction of database access is complex by itself, but synchronization with the file system makes it even more complicated. The resulting library uses three levels of abstraction:

1. Select queries from multiple tables with automatic joining
2. Table gateways and file storage
3. Objects representing rows in database tables

The central point of all these abstraction is the *Database* class. This singleton is aware of the relations between tables, creates table gateways and also contains methods to start a transaction (with optional locking), commit it or roll it back.

4.3.1 Select Queries

The elementary abstraction is a simplification of the SELECT query. Most of the needed functionality was already implemented in *Zend_Db_Select*. Class *Select* adds the method `addJoin()` that simplifies joins between tables: you can specify just the name of table, the ON clause is generated automatically.

The class requires that relations between tables are defined in the file '`lib/database/relations.php`' and that the database structure meets the following criteria:

1. Every table has a primary key called '`id`'
2. Foreign keys always reference this primary key
3. No table has two foreign keys referencing the same table

The third requirement is not satisfied by table `delegations`, so tables `delegations` and `users` must be joined manually.

There could be more than one way how to join a set of tables, so the order of adding tables matters: `addJoin()` tries to join the newly added table only with last one added. If joining is not possible, an exception is thrown.

4.3.2 Table Gateways

Almost every table in the CodEx database has its respective gateway. The gateway knows all the columns in the table and can retrieve an entire row by the value of its primary key, and also insert, update or delete rows.

Table columns are declared using instances of descendants of class *DefaultColumn*, for example *IntColumn*, *StringColumn* or *EnumColumn*. The main purpose of these classes is data validation (range of an integer, length of a string, etc.). But the gateways do not perform any validation themselves, the validation routines are used only by entity objects (see Section 4.3.4 [Entity Objects], page 51).

A special case is the so called virtual columns (class *VirtualColumn*): columns somewhat related to a table but which are, in fact stored in another table (e.g., the name of an owner of a group). Virtual columns can be used for better performance, but only when it is really needed, because they can cause an inconsistency: their values are not updated when the value of a foreign key changes.

Instances of table gateways are created by the *Database* class. They are constructed upon request for the first time and then shared for performance reasons.

Data retrieved from the database are cached in the user's session to reduce the database load. This cache is not synchronized between users, co inconsistencies can occur for a short time.

Rows read from a table are returned as an associative array or as entity objects (gateways work as factories for entity objects, see Section 4.3.4 [Entity Objects], page 51).

Gateways are defined in '`models/database/gateways.php`'.

4.3.3 File Storage

Certain CodEx data like test inputs are too large to be stored in the database, so they are stored in the file system instead. Unfortunately, the file system supports no such thing as a transaction. and there's also no way to atomically replace a non-empty directory. To make things worse, changes to the file system must be synchronized with changes to the database.

In CodEx, this problem is solved using versioning and garbage collection. Data directories are immutable. When a directory needs to be modified, it must be cloned (by creating a hard link of every file) to a temporary directory. After the needed modifications are performed, the temporary directory is moved to file storage as a new version. The temporary directory and file storage must reside on the same file system, otherwise the move could not be performed atomically.

When a new object that makes use of file storage is saved, its associated data directory needs to be created. The name of the data directory is the ID of the object. But the ID is only known after performing the INSERT. If the creation of the directory fails then, the database would be inconsistent with the file system. Therefore, the INSERT must be performed in a transaction, that can be rolled back if something goes wrong.

Versioning is even more complex, because two running processes may choose to increase the version simultaneously. Versions are subdirectories of the object data directory, so the full path to the object data is:

'`$storage_root/$object_class/$object_id/$object_data_version`'

The procedure used to store a new version of the object data is as follows:

1. Read the current version of object data (0 if no data were stored yet).
2. Increase the version number by 1 and try moving the new data directory to the path constructed as described above.
3. Repeat the previous step until it succeeds (the target directory did not exist) or exit if too many retries have been attempted.
4. Store the new version number in the database.

If some object does not need to modify its data, versioning is not used - this is the case of Submits. Exercises use versioned storage.

A **garbage collector** is run periodically, that deletes unreferenced objects (those deleted from the database or left over after an aborted transaction) and different versions of objects (old versions or abandoned recent versions). But it cannot delete them immediately, because some running script could still be using them, so a *grace period* is employed. Unreferenced objects and newer versions are removed when they are older than the grace period, and older versions are removed when the current version is older than the grace period.

The garbage collector also checks that all submits that might reference an older version of an exercise were evaluated. If not, it does not run and informs the administrator.

4.3.4 Entity Objects

The ultimate data abstraction are the entity objects. Every object type is related to a single table in the database and optionally uses the file storage. Entity objects are always created by a table gateway. The gateway either loads an object from the database (method `load()` or `multiLoad()`) or creates an empty object (method `create()`).

Objects are identified by an ID, which is the value of the primary key in the table the object comes from. The gateway caches instances of objects with the same ID, so that `load()` with the same ID always returns the same instance. The ID of a newly created object is *null*.

All columns from the table the object comes from are properties of the object. They have a getter function, that may perform a conversion, and a setter function, that performs validation of the new value. Invalid value is reported by throwing a *ConstraintException*.

Additional property is added for every property that represents a foreign key column. Reading from this property returns an instance of the object referenced by the foreign key. The property can also be assigned to. Names of these additional properties always end with `"Object"`.

The `save()` method of an object stores its properties to the database: it either inserts a new row and sets the object ID, or updates an existing row. Although the column values are validated, the save can still fail, because only the data format is verified, not correct values of foreign keys or uniqueness.

Objects also unify database access with access to file storage: if an object uses file storage, it contains a method to get the path to the data directory or set a new data directory. The `save()` method then updates data in the file storage and makes sure all update operations are performed in the right order, so that the database and the file system always stay synchronized.

Individual objects are defined in '`models/database/objects.php`'.

4.4 Mailing

4.4.1 Generic Mailer Class

4.4.1.1 `Email`

The `Email` class is a generic mailing class used by CodEx (indirectly through `CodEx_Mail`) to send e-mail messages. It is defined in the file `lib/email/email.php`, which also contains the helper class `Email_Encode` and two transfer classes `Email_Funmail` and `Email_Dump`. An instance of `Email` represents one message that can be composed part-by-part and then sent.

`Email` is a fairly standard (although somewhat simplified) mail transfer class. It supports a plain-text message body and an arbitrary number of attachments. Both the body and address fields can contain international characters, the default encoding is set to UTF-8.

The class sports a classical interface for adding various address fields (To, Cc, Bcc, Reply-To). An alternate transfer class can be selected, the default being `Email_Funmail`. Some advanced functionality is not supported, however, e.g. multiple sender addresses, HTML message bodies and related functionality (inlined images etc.).

4.4.1.2 `Email_Encode`

This is a helper class used internally by mail-transfer classes to encode messages to the Internet Message Format as specified by RFC 2822, 2045, 2046 and 2047.

4.4.1.3 Email_Funmail

This is the default mail-transfer class for `Email` messages. It encodes the message using `Email_Encode` and sends it using the built-in PHP function `mail()`. This function, in turn, usually runs `sendmail` to transfer the message to a mail server.

4.4.1.4 Email_Dump

A 'fake' transfer class. It does not send the message, instead it converts the message to a textual (human-readable) form and appends it to a file. This is very useful for debugging.

4.4.2 CodEx Mailer Class

The class `CodEx_Mail`, derived from class `Email`, allows the CodEx system to send e-mail messages to its users. It adds CodEx-specific features to `Email` and it is configurable through the CodEx configuration file.

The `addUser()` method allows adding a CodEx user to an address field. The first argument is a user object or ID, the second optional argument specifies the address field type (such as To, Cc, Bcc etc.). The `setNoReply()` method marks the message as an automatic message, setting the *From* field to a *no-reply* address (configured with `email.noReplyAddr`).

The message body can be created from a template with `setBodyTemplate()`. The first argument is the name of the template. The template is loaded from the corresponding template file '`generics/emails/`*`language/templateName`*`.txt`'. The second argument is an associative array of values that will be substituted into the template. The template is a text file containing tags in the form '`{{`*`variableName`*`}}`' that are replaced when the template is instantiated.

Finally, if the configuration option `email.debugFile` is non-empty, all messages use `Email_Dump` as a transfer class by default, configured to write in the specified file. As a result, messages are not sent to the mail server, but their dumps are logged in that file instead.

4.4.3 Event Notifier

The Event Notifier (the `Notifier` class) is responsible for sending event *notifications*. Notifications are e-mail messages sent when certain events occur (e.g. a new task is created, a user is removed from a group etc.). Any user wishing to receive these notifications must enable their reception by checking this on his Personal Settings page. This will set the appropriate flags in the `notifications` field of his user record.

The `Notifier` takes on most of the job of sending notifications. It composes the message text, determines the list of recipients and whether the message should really be sent.

If a page performs an action that can trigger a notification (e.g., if it modifies the properties of an exercise), it calls the appropriate `Notifier::`*`eventType`*`Event()` method and passes the object(s) involved as an argument.

The notifier composes the text of the message using an e-mail template with the name '`notification/`*`event_type`*'. It determines the potential recipients and selects only those who have enabled reception of the corresponding notification type). Also, for exercises, notificationd are not sent if the exercise is locked. Finally, the notification is transmitted to the mail server.

5 Database Description

5.1 Database Issues

Database schema is depicted on attached figure. Arrows on this figure represent foreign key relations. The schema use following abbreviations:

- `PK` - Primary Key
- `inc` - autoincrement
- `FK` - Foreign Key for id in another table (with arrow pointing to that table)
- `U` - Unique index constraint
- `N` - Nullable value
- `ts` - UNIX timestamp
- `T` - value updated only by trrigger (application may only read this value)
- `hash` - hashed item (password)
- `flags` - the item has integral value which is in fact separated to bits.
- `%` - the value is in permille (range 0-1000).

5.2 Database Implementation

5.3 Table `users`

The `users` table stores all user records in the CodEx system. The user with `id == 1` is treated as the almighty *admin* (similar to user *root* on UNIX systems).

- `login` - Unique login name that is assigned by the system. We presume that the student personal identification number from the SIS database will be used here.
- `loginAlias` - Alias login name. This is also unique and it must not collide with any `login` either. The login alias is set by the user. In the authentication process both `login` and `loginAlias` can be used. The login alias can only contain alpha-numerical characters.
- `passwd` - User's password. The password is stored hashed along with a salt. The salt is composed of 8 random hexadecimal digits. It is prepended to the password before hashing and after hashing it is prepended to hash. The hash is computed by the SHA-1 function and stored as 40 hexadecimal digits. The structure of the stored password is thus: `<SALT:8><SHA-1(<SALT:8><SHA-1(<PASSWORD>)>):40>`

  In order to explain the process of encoding the password more clearly, we present here a simple algorithm for creating or verifying a password:
  - Obtain the 8-digit salt. When encoding a new password, the salt is randomly generated. Otherwise, it is obtained by taking the first 8 digits of the encoded password.
  - Hash the password with the SHA-1 function.
  - Prepend the salt before the hashed password.
  - Hash (salt `+` hashed password) again, with SHA-1.
  - Finally, prepend the salt before the result again.
- `passwdExpiration` - UNIX timestamp of the instant when the password expires. An expired password is no longer valid and the user cannot use it for authentication. If it is `null`, the password never expires.
- `name` - User's first name.
- `middleName` - User's middle name. This field may be empty or it may also contain all user's middle names.

- `surname` - User's surname.
- `email` - User's e-mail address. All notification messages from CodEx as well as messages from administrators of groups where user belongs are sent to this address.
- `notifications` - Bit mask, a combination of flags determining which notification messages should be sent to the user (via e-mail). The meaning of the bits is following:

  `bit 0` Notify when new global news is created.

  `bit 1` Notify when new news for a group is created and user is member of that group.

  `bit 2` Notify when new task is assigned to any group in which the user has membership.

  `bit 3` Notify when an exercise changes and the user may see the exercise (or a task created from that exercise).

  `bit 4` Notify when a submit of the user is evaluated.

  `bit 5` Notify when membership status to any group has changed for the user.

  `bit 6` Notify when new comment for an exercise is created and user has `RIGHT_ADMIN` to that exercise.

  `bit 7` Notify when new comment for any exercise that user may see is created.
- `studyProgram` - Study program to which the student is assigned to. The program may be `M` for Mathematics, `F` for Physics and `I` for Informatics. If the program is unknown this field is `null`. This value is extracted from SIS during registration. The user may change it.
- `studyYear` - Year of user's studies. The expected value is from 1 to 8. This value is extracted from SIS during registration. The user may change it.
- `studyGroup` - Number of user's study group. This value is extracted from SIS during registration. The user may change it.
- `groupRights` - General rights to group objects. See Section 2.3 [Security], page 21
- `exerciseRights` - General rights to exercise objects. See Section 2.3 [Security], page 21
- `usersRights` - Rights to users table. See Section 2.3 [Security], page 21
- `newsRights` - General rights to news objects. See Section 2.3 [Security], page 21
- `created` - UNIX timestamp of the instant when the user's account was created.
- `lastLogin` - UNIX timestamp of the instant when the user has logged in for the last time. This field is `null` if the user has not logged in yet.

5.4 Table `groups`

A group is an entity that binds together users and a set of exercises. Users in a group can solve those exercises and obtain points for their solutions. The group operator can see the members' results.

A group object can represent, for example, a group of students from one course, group of competitors from a coding competition or a group of tricky exercises that can be solved by talented coders for practice.

- `name` - General group caption.
- `comment` - More detailed description of the group.
- `created` - UNIX timestamp of the instant when the group was created.
- `owner` - Foreign key referring to the users table, ID of the owner of the group. The owner has administrative rights on the group. If the owner is deleted, the ownership of the group is transferred to the administrator (user with `id == 1`).

- `public` - A flag determining whether the group is public. Public groups are visible for all users and any user may join the group. Private (non-public) groups are strictly managed by the group owner and they are not visible for non-members.
- `discreet` - A flag indicating whether members may see results of other members. In a discrete group each member can only see his own results.
- `pointLimit` - Minimum amount of points a member needs to satisfy the group demands (i.e. how many points a student needs to obtain credit). If the point limit is zero, the group will not display nor check this limit.
- `sisGroupId` - SIS specific identifier of corresponding group in Student Group Roster application. If the ID is not NULL, SIS is able to acquire points via `sis_group_results` service and show them along with other students' information.

5.5 User-group Relation Tables

5.5.1 Table `users_in_groups`

Since one group can contain any number of users and one user may be member of any number of groups, an M to N relationship must be defined. This table is a membership list connecting the `users` and `groups` tables. The deletion of either a user or a group will result in a cascaded deletion of corresponding rows from this table.

- `userId` - Foreign key of the user who is a member of the group.
- `groupId` - Foreign key of the group where the user belongs to.
- `points` - Total sum of points (includin submit and task bonus points) that user obtained. This field is maintained by triggers on table `task_points` and should not be modified by application directly.
- `bonusPoints` - Total sum of bonus points assigned for current user in current group in `bonus_points` table. This field is maintained by triggers on `bonus_points` and should not be modified by application directly.
- `tasksDone` - A flag that indicates whether the user has obtained necessary amount of points (defined in `tasks.obligatory`) for every task in the group. Value of this flag is in fact result of logical OR over all corresponding `done` values in `task_points` table. This field is maintained by triggers on `task_points` table and should not be modified by application directly.

5.5.2 Table `bonus_points`

This table keeps extra points assigned by the group owner. They can be used for giving the user credit for things like attendance or extra homework.

This table has a *unique* index on `groupId`, `userId` and `comment`. Therefore, one user in one group must have distinct comments for his bonus points. This precaution is taken because bonus points are displayed aggregated by the comment.

- `userId` - Foreign key of the user to whom the points belong to.
- `groupId` - Foreign key of the group to which the points belong to.
- `comment` - Granter's comment giving reason for the assignment of points. Bonus points are grouped together by their comment when displayed to the users.
- `created` - UNIX timestamp of the instant when the points were assigned.
- `points` - Number of points assigned.

5.6 Table `exercises`

List of all exercises available in the CodEx system. An exercise can not be solved by users directly. They may be, however, assigned to groups by creating tasks. The abstract exercise can be considered as a class (template) and the task can be considered an instance of that class.

- `name` - Name of the exercise.
- `comment` - Short motivation text introducing the exercise. This text is visible for all users.
- `note` - Instructions, hints and other important information for group owners that are creating a task from this exercise.
- `taskComment` - Default information and hints for users that solve this exercise. When a task is created, this field is used as the default value for `tasks.comment`. However, the group owner (who is creating the task) may redefine it.
- `textId` - Foreign key referring to the `texts` table which contains an XHTML specification of the exercise. There can be more than one version of the specification and the `textId` field points to the current (head) version.
- `author` - Foreign key to the `users` table, ID of the author of the exercise. The author has administrative rights to the exercise. If the author is deleted, the administrator (the user with `id == 1`) will take over his exercises.
- `created` - UNIX timestamp of the instant when the exercise was created.
- `changed` - UNIX timestamp of the instant when the exercise was last modified. time.
- `keywords` - List of keywords related to the exercise. Keywords are separated with commas so they look like a standard SQL set. Keywords can contain spaces (they are in fact phrases).
- `difficulty` - Difficulty level. This field can contain an integer value from 0 to 10. Value 10 stands for an extremely difficult exercise, while 1 stands for a trivial one. Zero value means that the difficulty of this exercise is not specified.
- `public` - A flag that indicates whether this exercise is visible for other users (with proper rights) or only to the author (and administrator).
- `locked` - A flag that indicates whether this exercise is locked. Locked exercises cannot be assigned as tasks nor submitted. Exercises that are not finished, broken or are being modified should be locked.
- `extensions` - Set of extensions (programming languages) that are allowed for this exercises. Tasks may restrict this set further.
- `acceptThreshold` - Default value for `tasks.acceptThreshold`. See Section 5.7 [Table tasks], page 57
- `version` - Version of related data files stored in the file system. This number grows incrementally each time the data are changed.
- `tasksCount` - Total number of tasks that has been assigned from this exercise. This field is maintained by triggers on table `tasks` and should not be modified by application directly.
- `submitsCount` - Total number of submits (with non-null points) for this exercise. This field is maintained by triggers on table `exercise_ratings` and should not be modified by application directly.
- `submitsPoints` - Total sum of best points from all users who solved this exercirse. The value is kept in permille and if divided by submitsUsers, it gives avarage success rate. This field is maintained by triggers on table `exercise_ratings` and should not be modified by application directly.
- `submitsUsers` - Number of distinct users who have submitted at least one submit (and the submit has been evaluated) for this exercise. This field is maintained by triggers on table `exercise_ratings` and should not be modified by application directly.

- `ratings` - Total sum of all ratings assigned in `exercise_ratings` to current exercise. Value of `ratings` divided by `ratingsCount` gives avarage ratings. This field is maintained by triggers on table `exercise_ratings` and should not be modified by application directly.
- `ratingsCount` - Number of non-null ratings from `exercise_ratings` for current exercise. This field is maintained by triggers on table `exercise_ratings` and should not be modified by application directly.
- `oldRatings` - Total sum of all rating values that have been deleted from `exercise_ratings` (therefore are no longer summarized in `ratings` field). The ratings should be computed as `(ratings + oldRatings) / (ratingsCount + oldRatingsCount)`. This field is maintained by delete trigger on table `exercise_ratings` and should not be modified by application directly except for complete reset.
- `oldRatingsCount` - Number of rating values added to `oldRatings`. This field is maintained by delete trigger on table `exercise_ratings` and should not be modified by application directly except for complete reset.

5.6.1 Table `texts`

Texts are exercise specifications in XHTML that are displayed to the user. Each exercise has a reference (`exercise.textId`) to its current specification called the *head version*. The head version is visible to all users (who may see the exercise).

However, the exercise can have more versions of this specification. Non-head versions are also stored in the `texts` table and they have an `exerciseId` foreign key that points to the exercise to which they belong to. When displayed, versions of the specifications are sorted by their `created` timestamp. Therefore, the table has a *unique* index on `exerciseId` and `created`. This index is not necessary, however, it helps the DBMS sort the specification versions.

Multiple versions of a single specification are kept in database so that if severe modifications are made (and a new version created), the users that solve the exercise can see and compare the changes.

- `exerciseId` - Foreign key referring to the exercise to which this specification belongs to.
- `created` - UNIX timestamp of the instant when this text was created.
- `changed` - UNIX timestamp of the last modification of this text.
- `content` - The XHTML content of the specification. This field is inserted directly into the generated XHTML when the page is rendered.

5.7 Table `tasks`

A task is an exercise instance. It takes a general exercise and attaches it to a group so that group members can solve this exercise. It also specifies additional parameters and restrictions for the exercise.

- `groupId` - Foreign key referring to the group where this task is presented.
- `exerciseId` - Foreign key referring to the exercise from which this task is created.
- `fromTime` - UNIX timestamp of the instant when it became or becomes visible for users.
- `toTime` - First task deadline stored as a UNIX timestamp. Submits commited before this deadline have point limit set in `points` field. If this value is `null`, no deadline is applied.
- `toTime2` - Second task deadline. Submits commited after first deadline and before the second one have point limit set in `points2` field. If this value is `null`, no second deadline is applied and any code submitted after first deadline is rated by `points2`. If `toTime` is `null`, this value is also `null`.
- `comment` - Hint for users that solve this task from the task creator. The default value for this field is taken from `exercises.taskComment`.

- `points` - Number of points that are assigned to user if he/she submits a completely correct solution before first deadline.
- `points2` - Number of points that are assigned to user if he/shw submits a completely correct solution after first and before second deadline. If there is no second deadline, this amount of points is used for submits after first deadline.
- `submits` - Limit for submits. User may not submit more solutions of this task than specified in this field.
- `extensions` - List of allowed extensions (programming languages). The user may submit his/her solutions only in languages on this list.
- `obligatory` - How many points are obligatory for this task. The user must obtain at least the number of points specified here in order to satisfy the group requirements.
- `acceptThreshold` - Minimum permille that any solution must obtain from evaluating system. If the solution has not sufficient permille (`submits.points`, results from this solution are completely ignored (user does not get any points from it). This precaution can be used in cases when the user could obtain some points just by guessing correct answers. This value is taken by default from `exercises.acceptThreshold`).

5.8 Table `task_points`

This table holds intermediate redundant values that are often required by appliaction, however they need to be optimized for reading. These values sumarizes results of each user for each task. It also contains special field where bonus points for particular task are assigned.

When user joins a group, record for each task in that group is created in this table. Also, when task is assigned to a group, record for each member of that group is created. Records are deleted when user leaves group or task is removed. Creating and deleting operations are performed only by triggers and cascade constraints.

- `userId` - Foreign key referring to the user to whom the points belong to.
- `taskId` - Foreign key referring to the task to which the points are assigned.
- `groupId` - Foreign key referring to the group where task (referred by `taskId`) belongs to. This field is properly set by a trigger just before the record is inserted.
- `bestSubmitId` - Foreign key referring to the best submit (the one which is awarded with the most points with respective to task deadlines and other limits) by current user to current task. If there are no submits yet, this value is `null`. This field is maintained by triggers on table `submits` and should not be modified by application directly.
- `points` - Number of points assigned to best submit (which is also amount of points which user recieves for current task). This field is maintained by triggers on table `submits` and should not be modified by application directly.
- `bonusPoints` - Number of bonus points assigned to current user for current task. Unlike best points for submits, these bonus points are added to always best submit (not to particular submit).
- `submitsCount` - Total number of submits current user sent for current task. This field is maintained by triggers on table `submits` and should not be modified by application directly.
- `done` - A flag which signals whether the user has sufficient amount of points to satisfy demands of corresponding task. This field is maintained by triggers on table `submits` and should not be modified by application directly.

5.9 Table `submits`

List of submitted solutions from all users. Each submit has some attached data in the file system (the submitted source code, result log etc.). See Chapter 6 [Data in Files], page 65

- `userId` - Foreign key referring to the user who submitted this solution.
- `taskId` - Foreign key referring to the task to which the submit applies.
- `exerciseId` - Foreign key referring to the exercise from which task (refered by `taskId`) is derived. This key is redundant and set properly by before-insert trigger on `submits` table.
- `submitted` - UNIX timestamp of the instant when this solution was submitted.
- `enqueued` - UNIX timestamp of the instant when this submit was moved into the *eval* queue. If it is `null`, the submit was not moved into the queue (this may be a signal that something went wrong). This field is also updated when the submit is re-evaluated.
- `evaluated` - UNIX timestamp when this submit was taken for evaluation or when the evaluation finished. The meaning depends on the `points` field. When `points == null`, the submit has not been evaluated yet and this timestamp holds the time when the evaluation started. Otherwise this field represents the time when the evaluation ended. Also, if this field is `null`, the submit is still in the queue waiting for evaluation.
- `extension` - Extension of the submitted solution. Extension uniquely identifies the programming language used.
- `comment` - User's internal note.
- `points` - Points assigned by the evaluating system. If `null`, the solution was not evaluated yet. The points representing relative correctness of the solution and are stored in permille. A value of -1 signifies a compilation error.
- `bonusPoints` - Additional points assigned by group owner for an excellent (or terrible) implementation, out-of-the-box solution, etc. Bonus points are represented as absolute points (not permille) and they can also be negative (negative bonus means a penalty).

5.9.1 Table `solutions`

Table of exemplary and testing solutions of the exercise. The exercise author and other users (with sufficient rights) can submit their solutions here. These solutions are tested exactly the same way as normal submits. However, these submits are also available for reading by other users and their evaluation is prioritized.

Data associated with each solution are stored in the file system (the source code, evaluation log, etc.). See Chapter 6 [Data in Files], page 65

- `exerciseId` - Foreign key referring to the exercise to which this solution belongs.
- `author` - Foreign key referring to the user who created this solution.
- `submitted` - UNIX timestamp of the instant when the submit was received by CodEx.
- `enqueued` - UNIX timestamp of the instant when this solution was moved into the *eval* queue. If it is `null`, the solution was not moved into the queue (this may be a signal that something went wrong). This field is also updated when the solution is re-evaluated.
- `evaluated` - UNIX timestamp of the instant when this solution was taken for evaluation or when the evaluation finished. The meaning depends on the `points` field. When `points == null`, the solution has not been evaluated yet and this timestamp holds time when the evaluation started. Otherwise this field represents the time when the evaluation ended. Also, if this field is `null`, the solution is still in the queue waiting for evaluation.
- `comment` - Comment for the solution. The author can include some details here such as time and space complexity or implementation tricks.
- `public` - A flag that indicates whether other users (with sufficient rights to the corresponding exercise) may see this solution.
- `extension` - The extension identifies the programming language of the solution.
- `points` - Points assigned by the evaluating system. If `null`, the solution hasn't been evaluated yet. Points represent the relative correctness of the solution and are stored in permille. A value of -1 signifies a compilation error.

5.10 Table exercise_comments

Table of users' comments for exercises. These comments are ment to evaluate the exercises, inlcude additional information and share best practices. Comments are designed only for users with operator rights.

- `exerciseId` - Foreign key referring to the exercise to which this comment belongs.
- `author` - Foreign key referring to the user who has added this comment.
- `caption` - The headline of the comment.
- `content` - Text content of the comment. It might contain some XHTML elements aswell.
- `created` - UNIX timestamp that holds date and time when the comment was created.
- `type` - Enum that defines type of the comment. The default value is '`common`' and it indicates that the comment should be displayed the default way. Value '`private`' defines that the comment will be displayed only to users with `RIGHT_ADMIN` for given exercise. Finally the '`important`' value means that the comment will be displayed emphasized.

5.11 Table exercise_ratings

This table holds ratings of solvers of this exercise. Ratings may provide feedback for author of the exercise and for group operators who wish to assign a task from an exercise. It also hold some cached values that are mainatined by triggers so they can be easily read without complicated join of tables.

Records in this table are created only by a trigger when user sends it first submit for a particular exercise. Records are deleted by cascade constraint when user or exercise is removed or when last submit (by particular user to corresponding exercise) is deleted and no rating is specified.

- `userId` - Foreign key referring to the user to whom the ratings and cached values belongs to.
- `exerciseId` - Foreign key referring to the exercise to which the ratings and cached values are assigned.
- `rating` - Ratings assigned by common users to exercises. This value is in permille (although application uses only 1-10 scale) where 0 is worst and 1000 is the best rating. If no rating has been assigned, this value is `null`. When this record is deleted, rating is permanently added to `oldRatings` field in `exercise` table.
- `bestPoints` - The greatest value of points assigned to any submit for this exercise. This field is maintained by triggers on table `submits` and should not be modified by application directly.
- `submitsCount` - Total number of submits for this exercise. This field is maintained by triggers on table `submits` and should not be modified by application directly.

5.12 Table news

News are announcements for the users. News items inform users about changes and other important events in CodEx. They can be restricted to the members of one group or to users with a specified minimum level of general rights.

- `groupId` - Foreign key referring to a group. Only members of this group are allowed to see this news item. If it is `null`, the news item is intended for all users.
- `author` - Foreign key referring to the user who is the author of this news item.
- `created` - UNIX timestamp of the instant when the news item was created.
- `expiration` - UNIX timestamp of the instant when the news item expires and will be deleted.

- `caption` - Title of the news item.
- `content` - The XHTML content of the news item.
- `groupRights` - The minimum level of general group rights (`users.groupRights`) the user must have to see this news item.
- `exerciseRights` - The minimum level of general exercise rights (`users.exerciseRights`) the user must have to see this news item.
- `usersRights` - The minimum level of user rights (`users.usersRights`) the user must have to see this news item.

5.13 Table `delegations`

Delegations are rights granted for individual objects (groups or exercises). *Delegation* is given by a *granter* (user with sufficient rights) to a a *trustee* (a user who gains the rights). Delegations for groups and exercises are completely disjoint even though they are stored in the same table. See Section 2.3 [Security], page 21

- `groupId` and `exerciseId` - Foreign keys referring to a group or exercise object to which the rights are delegated to. One of these two fields must be `null` and the other not null. See Section 2.3 [Security], page 21
- `trustee` - Foreign key referring to the user who received this delegation.
- `granter` - Foreign key referring to the user who delegated the rights.
- `rights` - Level of rights granted. See Section 2.3 [Security], page 21

5.14 Views

A *view* is a query stored in the database, which can be used the same way as an ordinary table (only for reading). The CodEx libraries do not distinguish between tables and views, so, for example, the `Select` class supports automatic joining with views.

5.14.1 View `group_interested`

This view is used to obtain the list of users who are interested in information about a certain group (news and notifications). It contains group members (defined in `users_in_groups`), its owner and trustees with any rights. Note that it can contain duplicate values!

- `userId` - ID of a user.
- `groupId` - ID of a group where the user is a member, the owner or a trustee.

5.15 Stored Procedures

Following stored functions and procedures are embeded into the database. They are used by triggers and some automated SQL scripts.

- `FUNCTION compute_points(points, pointLimit, threshold)` – function that computes real points from permille (points) by given limit and acceptance threshold
- `FUNCTION get_exercise_bestPoints(uId, eId)` – returns best points (in permille) for selected user and exercise; if no points exists, `null` is returned
- `FUNCTION get_realSubmitPoints(tId, ts, p)` – retrieves data of selected task (tId) and computes real points for submit with given timestamp (ts) and points in permille (p)
- `FUNCTION find_bestSubmitId(uId, tId)` – returns ID of best submit for given user (uId) and task (tId); if there are multiple submits with the same number of real points (with respect to deadlines), ID of submit which has been commited first is returned
- `FUNCTION find_bestPoints(uId, tId)` – compute best real points for given task (tId) and given user (uId)

- `PROCEDURE recount_task_points(uId, tId)` – recount all trigger-managed values of a `task_points` record identified by `userId` and `taskId`
- `PROCEDURE recount_task_points_table()` – recount all values in entire `task_points` table

5.16 Triggers

Triggers maintain data integrity for redundant database fields. These fields were inserted into the structure in order to reduce time required by complicated join operations (mostly when result points are retrieved).

5.16.1 Triggers Objectives

Here are main objectives that should be ensured by triggers. These objectives are in fact a cross reference to trigger description below.

- `task_points`
  - when user joins/leaves a group, create/remove a record for every task in the group
  - when task is inserted, create a record for every user in the group (note that deletion of these records is ensured by CASCADE constraint)
  - when submit is inserted/updated/deleted, update `points` accordingly
  - when submit is inserted/deleted, increment/decrement `submitsCount`
  - when task is updated so that it might affect results, recount `points` and `bestSubmit` reference
- `users_in_groups`
  - when `task_points` record is updated, recount `points` accordingly
  - when `bonus_points` is inserted/updated/deleted, recount `bonusPoints` accordingly
  - all trigger-managed values must be recounted when an `exercise` is deleted since cascade deletion of `task_points` does not trigger any procedures
  - when `task` is deleted, all trigger-managed values of affected group must be recounted since cascade deletion of `task_points` does not trigger any procedures
  - when `task` with obligatory points is inserted, all users in affected group must have the `tasksDone` value set to `false`.
  - before a new membership record is inserted, the `tasksDone` value is computed (it is false iff there are some tasks with obligatory points)
- `exercise_ratings`
  - when submit is inserted and no corresponding record is present, create new one
  - when submit is inserted/updated/deleted, recount `bestPoints` and update `submitsCount`
  - when `submitsCount` reaches zero and there is no information in `rating` field, remove the record
  - before user is deleted, delete also all exercise_ratings for this user (this manual cascade efect ensures that delete triggers are executed properly)
  - when task is deleted, all values affected by cascade deletion of corresponding submits are recounted
- `exercises`
  - when new task is created/deleted, update `tasksCount`
  - when `exercise_ratings.submitsCount` is modified, update `submitsCount`

  - when `exercise_ratings.bestPoints` is modified, update `submitsPoints` (and `submitsUsers`)
  - when `exercise_ratings.rating` is modified, `update` ratings and `ratingsCount`
  - when `exercise_ratings` is deleted, update `ratings`, `oldRatings` and corresponding "count" fields

5.16.2 Triggers Overview

Description of each trigger in the database. We use insert, update and delete triggers. In MySQL each trigger may be executed before or after the event. In following text we expect that the trigger is executed after the event unless specified otherwise.

- `task_points`
  - *before insert* – ensures correct value of `groupId` foreign key (so it corresponds with `taskId`)
  - *update* – actualize `points` and `tasksDone` values in `users_in_groups`
- `bonus_points` – *insert*, *update* and *delete* triggers maintain `bonusPoints` value in `user_in_groups` table
- `exercise_ratings`
  - *insert* – adds summarized statistics (about an exercise) to `exercises` table
  - *update* – updates corresponding statistics in `exercises` table
  - *delete* – updates corresponding statistics in `exercises` table and adds ratings into `oldRatings` column
- `submits`
  - *before insert* – ensures correct value of `exerciseId` foreign key (so it corresponds with `taskId`)
  - *after insert* – update `bestPoints` and `submitsCount` in `exercise_ratings` (or create corresponding record if it does not exist yet) and update `task_points`
  - *update* – updates `exercise_ratings` and recount `task_points` if necessary
  - *delete* – updates `exercise_ratings` (even delete the record if `submitsCount` reaches 0 and no ratings is set) and recount `task_points` if necessary
- `tasks`
  - *before insert* – ensures that second deadline is not set if the first one is `null`
  - *after insert* – increment `tasksCount` of corresponding exercise, create a record in `task_points` for each member of the group in which the task belongs to and set `tasksDone` to `false` for each member of the group if the task has any `obligatory` points
  - *update* – if the update may affect assigned points, all corresponding records in `task_points` are recounted
  - *delete* – exercise `tasksCount` is decremented, `users_in_groups` and `exercise_ratings` values affected by cascade deletion of submits are recounted.
- `users_in_groups`
  - *before insert* – ensures correct value of `tasksDone` by scanning for obligatory tasks in corresponding group
  - *after insert* – create a record in `task_points` for new member and each task in the group
  - *delete* – remove all corresponding `task_points` records
- `exercises`
  - *delete* – recount all trigger-managed values in `users_in_groups` since cascade deletion of `task_points` does not execute the triggers

- `users`
  - *before delete* – delete all `exercise_ratings` of deleted user (this way we ensure that delete triggers are executed properly)
- `groups`
  - *before delete* – delete all `tasks` from deleted group (this way we ensure that delete triggers are executed properly)

6 Data in Files

6.1 Directory structure

Apart from the database CodEx also uses the file system to store its data. All CodEx files reside in directory called the Data Root. The location of this directory is configurable, so that multiple CodEx installations can coexist on a single server.

In order for the CodEx front-end to work, the Data Root and all its subdirectories must be writable by the user under which the PHP scripts run (typically '`apache`' or '`www-data`').

Caution: the whole subtree of the CodEx Data Root must reside on a single file system, because CodEx depends on atomic moves between directories under the Data Root.

Names of subdirectories in the Data Root can be changed in '`config.ini`'. We will now describe the typical directory layout assuming the names of the directories are set to their default values.

'`log`' Contains CodEx logs.

'`codex.log`'
: The main CodEx log. Messages of different priority (including PHP errors and exceptions) are recorded here. The minimum priority of messages that are logged is configurable.

'`security.log`'
: The CodEx security log. Events such as a user logging in or out and possible hacking attempts are recorded here.

'`queue`' Queues for communication with Qman. These directories can be regarded as a data structure shared between the front-end and back-end. Access to the entries in this queue (the subdirectories) is governed by a special protocol (see Section 7.1 [Protocol], page 73).

'`in`' Queue Manager input queue. The front-end puts jobs (copies of submits and solutions) to be evaluated here.

'`out`' Queue Manager output queue. Qman puts evaluated jobs here.

'`storage`' Data related to CodEx Entity objects. See Section 4.3.3 [File Storage], page 50.

'`exercises`'
: Data directories of the *Exercises*. Contains a subdirectory for every exercise (named with the exercise ID) which in turn contains one or more subdirectories with different versions of the exercise data. Typically only the most recent version of the data is present, but sometimes more version can be present, like when there are some pending submits dependent on the older version, or before the garbage collector manages to delete the obsolete version.

  The directory with exercise data typically contains the following files:

  '`config`' Exercise configuration. This file is required for the evaluation of submits and solutions. It is usually edited indirectly using a form-based interface in the front-end, but it is also directly accessible to the user. See Section 6.2 [Exercise Configuration], page 67 for the specification of its format.

  '*ID*`.in`' Input data for test named *ID*. The name of a test can be any non-empty string consisting of alphanumeric characters. It

should be a text file with UNIX line endings, but some exercises can use binary files. If the exercise specifies multiple input files, '`<ID>.in`' is a directory instead of a file.

'*ID*`.out`' Reference output for test named *ID*. This file can actually contain any data (not necessarily data for comparison) and is not mandatory. The output can be validated in any other way, for example, if there are multiple correct answers.

'`submits`' Data directories of the *Submits*. If contains a subdirectory for every submit (named with the ID of the submit) which in turn contains the following files:

'`source.`*ext*'
Source code submitted by user. Extension of this file differs, the actual extension is stored in database (`submits.extension` or `solutions.extension`, see Chapter 5 [Database Description], page 53).

'`metadata`'
This file is only present if the submit has already been evaluated. It contains a description of the submit for the back-end and the back-end appends here the results of the tests. See Section 6.3 [Metadata], page 68.

'`eval.log`'
This file is only present if the submit has already been evaluated. It contains a log of messages produced by the back-end during evaluation. (Such as compiler error, a summary of test results, etc.) The front-end does not process this file in any way, it just displays it to the user.

'`solutions`'
Data directories of the *Solutions*. The structure is identical to that of the directory containing the *Submits*.

'`texts`' Data directories of the *Texts*. It contains a subdirectory for every exercise text (named with the ID of the text). These directories contain the attached files for the exercise specification text. They are directly exposed by the web server, so the user's web browser can access them. The author of the exercise can put any files (subject to certain limitations) and reference them from the text of the exercise specification.

All these directories are managed by the `Storage` class and the associated garbage-collector script (see Section 4.3.3 [File Storage], page 50) and their contents are managed through CFI (see Section 4.2 [CFI], page 44) using the `CFI_FS_Filesystem` driver.

The CFI driver imposes several limitations on the contents of these directories in order to prevent possible filesystem-based attacks and other possible problems. Namely, the name of a directory entry is not allowed to begin with a period ('`.`') and it can only contain alphanumeric characters, periods, hyphens and underscores. There can be no symbolic links and also no filenames in the same CFI tree link to a single file or a file outside of the storage.

On the other hand, the exercise data is versioned. If a file is unchanged between two versions of the data, it is hardlinked (in order to save space and processing time). Thus, entries with the same name in the different versions can link to a

single physical file. Thus, you should never modify the contents of any file directly, unless you are absolutely sure what you are doing.

'`temp`' The CodEx temporary directory. New files or directories are typically created here and then moved to the *Storage* or to the *Queue*. The system-wide temporary directory '`/tmp`' is not used, because it could reside on a different file system and CodEx needs to move directories between the temporary directory and the storage atomically.

6.2 Exercise Configuration

Most of information about exercise is stored in CodEx database, but the specification of tests is needed for evaluation (see Section 7.2 [Evaluation Process], page 74). Workers cannot access CodEx database, so the specification must be stored in a file. The format of this file is described here.

The exercise configuration resembles a Bash script: each line is either empty, a comment (if it begins with '`#`') or defines a parameter. Parameter definition looks like this:

```
PARAMETER_NAME='parameter value'
```

Allowed parameters are:

'`TESTS`' Identifiers of tests separated by spaces, for example '`1 2 3`'.

'`IN_TYPE`' Defines how the evaluated program should read input, possible values are:

- '`stdio`' Program reads input from *stdin*.
- '`file`' Program reads input from file, `IN_FILE` must be specified.
- '`dir`' Program uses multiple input files.

'`OUT_TYPE`'
Defines how the evaluated program should write output, possible values are:

- '`stdio`' Program writes output to *stdout*.
- '`file`' Program writes output to file, `OUT_FILE` must be specified.

'`IN_FILE`' Used only when `IN_TYPE` is `file`. Defines the name of the input file.

'`OUT_FILE`'
Used only when `OUT_TYPE` is `file`. Defines the name of the output file.

'`OUTPUT_FILTER`'
Output of evaluated program is piped through this command. It's used in CodEx to remove comments from program output.

'`OUTPUT_CHECK`'
Defines command which is used to compare program output with correct output.

'`TIME_LIMIT`'
Time limit in seconds (floating-point number). The evaluated program is killed if it runs longer than this time.

'`MEM_LIMIT`'
Memory limit in kilobytes. The evaluated program is killed if it tries to allocate more memory than this limit.

'`POINTS_PER_TEST`'
Number of points assigned if the program passes a test. The value is in permille, so total should be 1000.

Parameters `TIME_LIMIT`, `MEM_LIMIT` and `POINTS_PER_TEST` may be redefined for every test and extension. If a limit is specific for a test, the parameter name is prefixed by '`TEST_<id>_`', where '`<id>`' is an identifier of the test. If a limit is specific for an extension, the parameter name is prefixed by '`EXT_<ext>_`', where '`<ext>`' is the extension. Extension prefix must come before test prefix. For example a time limit for test '`3`' and extension '`.pas`' will be '`EXT_pas_TEST_3_TIME_LIMIT`'.

If more parameters apply to some combination of a test and an extension, they are considered in this order (first match is used):

1. Limit specific for this test and this extension.
2. Limit specific for this test for all extensions.
3. Default limit for this extension.
4. Default limit for all extensions.

6.3 Metadata File Format

Metadata files are used for communication between CodEx and Qman (see Section 7.1 [Protocol], page 73). A metadata file describes the submit added to the queue and also contains the results of tests after successful evaluation.

6.3.1 Generic Format

The metadata file is a line-oriented text file with UNIX line endings (LF). It uses UTF-8 encoding, but in CodEx just plain ASCII is used, which is a subset of UTF-8. The file contains definitions of attributes and their values. The same attribute can be specified more than once.

Attributes can be simple or compound. A simple attribute has a string value, which is separated from its name by a colon. The name of a compound attribute is followed by a left parenthesis, multiple lines with definitions of nested attributes and a right parenthesis.

Empty lines or lines beginning with '`#`' (hash sign) are treated as comments and ignored.

The syntax expressed as grammar in BNF form is as follows:

```
<file> = <attribute>*
<attribute> = (<simple> | <compound> | <comment>)
<simple> = <indent> <name> ":" <value> "\n"
<compound> = <open> <attribute>* <close>
<open> = <indent> <name> "(" "\n"
<close> = <indent> ")" "\n"
<indent> = (<sp>|<tab>)*
<name> = (A-Z | a-z | 0-9 | "-" | "_")+
<value> = (<any-printable-UTF8-char> | 0x09)*
<comment> = ("\n" | <indent> "#" <value> "\n")
```

Note that spaces are allowed only at a few specific places, for example there cannot be spaces around the colon that separates attribute and its value.

6.3.2 Attribute Description

CodEx puts the following attributes to Metadata file, Eval must not change them:

'`task_name`'
: Name that uniquely identifies an exercise (currently the exercise ID).

'`task_version`'
: The exercise version. Must change when exercise data (including config) changes.

'`task_dir`'
: Path to the directory containing exercise data, relative to CodEx data root.

‘`codex_type`’
: Name of table in the CodEx database, either ‘`submits`’ or ‘`solutions`’.

‘`codex_id`’
: ID of a submit or solution.

‘`source`’ Name of the submitted source file (inlcuding extension, but without path).

‘`exec`’ Full path to a script that Qman executes when evaluation is finished.

For every test executed, Eval adds a compound ‘`test`’ attribute with these nested attributes:

‘`id`’ Test ID. Non-empty string of alphanumeric characters.

‘`points`’ The number of assigned points (in permille). A non-negative integer.

‘`status`’ Status code. Two upper-case characters. The possible values and their meanings are:

‘`OK`’	test passed
‘`CE`’	compile error
‘`FO`’	forbidden operation
‘`RE`’	runtime error (exitcode is set)
‘`SG`’	killed by signal (exitsig is set)
‘`TO`’	time limit exceeded
‘`WA`’	wrong answer
‘`PA`’	partial answer
‘`PE`’	protocol error (for interactive tasks)
‘`XX`’	internal error

‘`message`’ Human-readable status message. (In English)

‘`time`’ [optional] Duration of execution in seconds. Fractional number represented in decimal or scientific notation, as specified by the %g format specifier in C.

‘`mem`’ [optional] Memory consumed by the tested program in bytes. Non-negative integer.

‘`exitcode`’
: [optional] Exit code of program. Non-negative integer.

‘`exitsig`’ [optional] Signal the program was killed with. Non-negative integer.

6.4 Exercise Export And Import

CodEx allows exercises to be exported to ZIP file for archiving and sharing purposes. These packages can be imported later (even to another instance of CodEx). The package contains all exercise data, testing data and configuration, specification with attached files and solutions. Exercise comments are NOT exported.

6.4.1 Package Format

The ZIP package contains XML contents file named ‘`content.xml`’ and other subdirectories and files that depends on version of the package. At the moment, only valid version is version 1 and its format is described here.

There are three subdirectories in the package:

- '`testdata`' – Contains exact copy of exercise storage directory. That includes test data, results for comparison and configuration file.
- '`attachments`' – Directory with attachment files for exercise specification. Each version of specification has its files in separate subdirectory which has the same name as the value of `id` attribute of corresponding text element in XML file.
- '`solutions`' – Directory with all exported solutions for the exercise. Each solution has its own subdirectory which has the same name as the value of `id` attribute of corresponding solution element in XML file. Each solution usually contains source code file, evaluation log and metadata file.

Example of package file structure:

```
[ZIP file]
 |--[testdata]
 |   |--1.in
 |   |--1.out
 |   |--2.in
 |   |--2.out
 |   '--config
 |--[attachments]
 |   |--[0]
 |   |   '--schema.png
 |   '--[1]
 |    |--schema.png
 |    '--sample.pas
 |--[solutions]
 |   |--[0]
 |   |   |--source.pas
 |   |   |--metadata
 |   |   '--eval.log
 |   '--[1]
 |    |--source.cpp
 |    |--metadata
 |    '--eval.log
 '--content.xml
```

Note that some attachment files may have been hardlinked (when specification changes, attachment files are duplicated by hardlinks), but when loaded to package, this information is lost. Therefore, when package is imported, every attachment is copied into new separate file (even if they contain the same data).

6.4.2 XML Contents File

The exercise content file is standard XML file that contains all information about the exercise. XML pseudo structure is depicted below. Version of package format is stored as attribute of root element, which must be always named '`exercise`'.

Data contents are encoded into XML entities by SimpleXML library, so even chars like '`<`' or '`>`' can be safely stored. Numeric values are stored in decadic representation and bool values can be either 0 or 1. In our case the only nullable values are numbers, thus we encode null value as a '`NULL`' string.

```
<exercise version="1">
<data>
<name></name>
<comment></comment>
```

```
<note></note>
<taskComment></taskComment>
<created></created>
<changed></changed>
<keywords>
<item></item>
...
</keywords>
<difficulty></difficulty>
<public></public>
<extensions>
<item></item>
...
</extensions>
<acceptThreshold></acceptThreshold>
<ratings></ratings>
<ratingsCount></ratingsCount>
</data>
<texts>
<text id="" [active="1"]>
<created></created>
<changed></changed>
<content></content>
</text>
</texts>
<solutions>
<solution id="">
<submitted></submitted>
<enqueued></enqueued>
<evaluated></evaluated>
<extension></extension>
<comment></comment>
<points></points>
<public></public>
</solution>
</solutions>
</exercise>
```

The XML file contains three main sections '`data`', '`texts`', and '`solutions`'. The '`data`' section stores almost all data of the exercise object from the database, except for a few fields. The '`textId`' field is filed automatically, when active text object is inserted (and its database ID is known at last). The '`author`' is set to current user (i.e. the user who performs the import). The '`locked`' field is missing since it is always set to `true` when exercise is imported. The '`version`' is set to 1, and finally all statistic values (except for '`oldRatings`' and '`oldRatingsCount`') are implicitly 0. Otherwise every element directly corresponds to a field with the same name, except for '`ratings`' that corresponds to '`oldRatings`' and '`ratingsCount`' that corresponds to '`oldRatingsCount`'. When the exercise is exported, the rating values are stored as ratings plus oldRatings.

The '`texts`' section contains all specification versions. The '`id`' attribute is generated sequentially (from 0) and it does not have any relation to original database ID of the text object. The '`active`' attribute has implied value of 1 and exactly one text element has this attribute

set. It marks the text object that is selected by '`textId`' reference of the exercise object. All subelements correspond directly to their database counterparts.

The '`solutions`' section contains public solutions and solutions that belong to the user who performed the export. The '`id`' attribute is assigned the same way as it was for text objects. Also all subelements correspond to database columns with the same name.

6.4.3 Exporting/Importing Mechanism

Exporting mechanism defines abstract class `ExerciseExporter`, which is derived by subclasses called `ExerciseExported_<version>`. Version is a number of the package version. The `ExerciseExporter` holds static variable with most recent version and uses corresponding exporter class to create exporters. The outer API is implemented in static method of `ExerciseExporter` calles `export`. It performs tasks common for all versions and invokes `preparePackage` method of active exporter.

Importing mechanism is very similar. Class `ExerciseImporter` provides API of static method `import` and defines interface for all importers with class names `ExerciseExporter_<version>`. When package is loaded, proper importer class is found (according to version specified in content XML file) and its `processPackage` method is invoked. If no such class exists, the package can not be imported.

7 Evaluation

The CodEx web front-end accepts source code to be evaluated from the users. The evaluation itself is performed asynchronously by the CodEx back-end, which consists of the Queue manager (*Qman*) and multiple workers. The Queue manager runs as a daemon, picks up jobs from the queue and assigns them to workers, which evaluate the jobs as described in Section 7.2 [Evaluation Process], page 74.

The communication between the front-end and the back-end follows the protocol described in Section 7.1 [Protocol], page 73. In the front-end this protocol is implemented by the *EvalQueue* class and by the script '`hook.php`'.

7.1 Communication Protocol

7.1.1 Definition of Terms

Although these terms are also described in the glossary, whe shall repeat them here. Their perception is slightly different from the point of view of the protocol and they are crucial for understanding the protocol perfectly.

exercise An exercise represents a problem to be solved. From the point of view of this protocol, it consists of a versioned directory containing a configuration file (`config`, test input files and output verification files. An exercise directory is immutable. When a change is needed, a new version (and thus a new directory) is created.

submit or solution
Submits and solutions are created by the users as an attempt to solve the exercise. Each submit or solution is evaluated at least once, but it can be re-evaluated any number of times. A submit or solution consists of a directory with the source file and a `metadata` file.

queue (evaluation queue)
The evaluation queue is a directory shared between CodEx and Qman.

job A job is an entry in the evaluation queue (a subdirectory). Is is a unit of work that must be processed by Qman. A job is created by the front-end when it needs a submit evaluated. It is a directory containing the source file (actually a hard link to the source file) and a `metadata` file generated on the fly.

7.1.2 Protocol Specification

1. CodEx creates a temporary directory and puts a hard link to a submitted source file inside.
2. CodEx creates a *metadata file* (see Section 6.3 [Metadata], page 68) named '`metadata`' in the temporary directory. The metadata file contains the following attributes:

   '`task_name`'
   Database ID of an exercise in the CodEx database

   '`task_version`'
   Version of the exercise data (changes when the data change)

   '`task_dir`'
   Directory containing the exercise data (relative to CodEx data root directory)

   '`codex_type`'
   '`solution`' or '`submit`'

   '`codex_id`'
   ID of the solution or submit in the CodEx database

‘`source`’ Name of the source file to be evaluated (without path)

‘`exec`’ Full path of a script that Qman must run when the evaluation finishes

The Queue manager or workers MUST NOT modify any of these values.

3. CodEx atomically moves the temporary directory to the evaluation queue (i.e. inserts a job into the queue). The exact format of the directory name is only relevant to CodEx and it is composed as ‘`$priority-$timestamp-$submitID`’.
4. The Queue manager SHOULD pick the jobs from the queue in lexicographic order. The Queue manager MAY move jobs to other directories (e.g. a work queue).
5. Worker MAY cache the exercise data. But if it does, it MUST make sure it always uses the version specified in the metadata. The version numbers can change in either direction (i.e. a newer job can use a lower version than an older job). Possible solutions are upgrading or downgrading the cached data to the version specified in metadata, or storing multiple versions in parallel.
6. Worker runs tests, appends results to the metadata and stores an evaluation log in a file named ‘`eval.log`’.
7. When the evaluation finishes, Qman moves the evaluated submit to the output queue and executes the script specified in metadata. It passes the script a single argument, the directory with the evaluated job. The script MAY be executed multiple times in parallel.
8. When the script completes, it deletes the directory given as an argument.

7.2 Evaluation Process

Evaluation itself is performed by a third-party component called MO-Eval (available at `http://mj.ucw.cz/mo-eval/`), which was only slightly modified for CodEx. Together with a simple wrapper and a script which runs MO-Eval repeatedly according to commands given by Qman, this constitutes the worker used in CodEx. There can be more copies of the worker running simultaneously. In each cycle a worker performs the following operations:

1. The worker compiles the submitted source code using the compiler appropriate fo the extension of the source file. If the compilation fails, no tests are run and the score of all tests is zero. Otherwise the next steps are repeated for every test defined in the exercise configuration.
2. If the evaluated program should read data from a file, the worker copies test input file and names it as specified in exercise configuration. If the exercise specifies multiple input files, the worker copies all files in the test input directory.
3. The compiled program is run in a sandbox, which intercepts all system calls. If the program attempts to make a forbidden system call or if it exceeds the memory limit or the time limit, it is immediately killed. Its standard output is redirected to a file. If the exercise specifies reading from the standard input, the standard input of the program is fed from the input file.
4. If the program terminantes with a non-zero exit code or if is killed by the sandbox or by a signal, it fails the test and scores 0 points. Otherwise the worker continues, checking its output.
5. The output of the program is piped through the output filter. One of two filters can be chosen in the exercise configuration. Tge first filter leaves the output as it is, the second one removes comments from it.
6. Finally, the worker checks whether the output is correct using a program called *judge*. Four different judges can be chosen in the exercise configuration:
   - Strict judge, which just compares the program output and the correct output byte by byte

- Text judge, which splits both outputs to tokens using whitespace sequences as delimiters and compares the tokens
- Float judge, that compares floating-points numbers with a defined error tollerance
- Shuffle judge, that ignores the order of tokens on a line, the order of lines in the output file or both

If the judge declared the output correct, the test scores the number of points defined in the exercise configuration. Otherwise no points are awarded.

7.3 Modifications for CodEx

The `mo-eval` system used in CodEx to evaluate the solution had to be slightly modified and extended to be more suitable for the task. Some of the original scripts (especially these used in the MO-P contest itself, especially user and installation management) have been removed.

Most of the modifications were carried in cooperation with the main developer of `mo-eval` Martin Mares and are now incorporated into the `mo-eval` system.

The main modifications include:

- Extension of the sandbox program to report time and memory usage in metadata format. These modifications (especially the memory usage reporting) are mostly by Martin Mares.
- Adding metadata information about points and test result to the evaluator script.
- Adding support for interpreted languages (C$, Python, ...), allowing specific pre-, post-, and run commands.
- Allowing custom values of all evaluator parameters for individual tests and languages (ie. special time and memory limits for C$). This was added by Martin Mares.
- New wrapper scripts (`codex-eval-wrapper`) for CodEx around the evaluator system for interaction with the job directory and for working with metadata and caching the task data (for use over the network).
- Added new judges for comparison of the tested program output with the original result. These were developed independently by Martin Krulis for CodEx (`codex_judge` and others) and by Martin Mares for the `mo-eval` system (`judge-shuff`, ...) and are both available in the CodEx system.

7.4 Security

The students and other users submitting their programs are NOT trusted, i.e. their programs are considered potentially harmful to the running system. The evaluation system internally uses a ptrace-type sandbox to prevent the tested program from harming other processes or the system.

The sandbox catches every system call the tested program makes and either blocks it (killing the program in most cases) or allows it. The main blocked actions are file manipulation outside the permitted (safe) directories, `fork()`, threading, device manipulation, manipulation with permissions and networking. The sandbox program also limits the the memory consumption and the time used either as CPU time or as a real-world (wall) time.

For details please refer to the `mo-eval` manual and to the `box` program source, which is well written and understandable.

Another good (although optional) separation is trough running the sandbox as a separate non-privileged user with disk quota. This is very easy to set up. For that it is necessary to set the `box` program to SUID to the unprivileged user and to change the permissions and ownership of the '`box/`' directory in the respective copy of the evaluator.

It is recommended to set a reasonable disk quota for every testing user (as the sandbox program does not limit the amount of data written) and to use different users for different instances of the evaluator.

These precautions should be enough to prevent the tested program from harming the system, but in a system with more strict security more separation layers should be used. These include running the evaluator on different or virtual machines or in a chroot jail. These methods are briefly described in Security section in Section 8.1 [Queue Manager Overview], page 77.

8 Queue Manager (qman)

This chapter describes the internals and interface of the queue manager used in the CodEx project.

The first section analyzes the design and the underlying problem in general. It also describes the concepts and the operation of the manager.

The second section describes how Qman is executed and all its interfaces.

The third section analyzes some implementation details of the program.

8.1 Overview

8.1.1 Introduction

For various reasons it is not desirable to call the evaluator of the submitted programs directly. Direct parallel execution of dozens of submitted programs (think about the last half hour before the deadline) would overload the evaluation machine. Many parallel executions lead to many context switches (and cache misses) which, apart from being ineffective, also disrupt time measurements.

To evaluate the submits sequentially we had to implement a system fetching the submits from a queue and distributing these to a single worker or multiple workers. We decided to make this an independent program useful for any general scenario where many similar *jobs* are submitted for some sort of evaluation and then the results are handed to some other system (possibly into another queue) or just stored for later processing.

The jobs should be parametrized (in the case of CodEx this is for example the exercise and exercise version) and should have some priority based on their type and time of submission. The workers should be independent and not considered to be reliable - a worker crashing on some job should not disturb the processing of other jobs. To improve performance, the workers should be able to run remotely on several machines at once.

All these conditions, along with high customizability, are addressed in this implementation. The state of the manager can be viewed in a global log, per-worker logs and per-job logs.

We decided to call this utility `qman` (**q**ueue **man**ager).

The program in implemented in plain C for performance and code transparency reasons. The program uses many of the Linux/UNIX system features and therefore is being developed only for that platform.

8.1.2 Submit Storage Design

When the CodEx front-end receives a submit or solution to evaluate, it should store it in a safe place for re-evaluation and for recovery after (sub)system crash. The first safe storage in the case of CodEx is the database in conjunction with text files (for storing the source code).

However, the evaluation system currently in use (*mo-eval*, described briefly in Chapter 7 [Evaluation], page 73) is also being used in another projects (mainly in Czech Mathematical Olympiad - Category Programming) and for these the need of a full-featured database system is burdensome.

For that reason, a secondary storage system for jobs awaiting evaluation was devised as an interface between the web front-end and the evaluation back-end. The system is based on file-system directories and and consists of the following parts:

- An *input queue directory* with one subdirectory for every job.
- A *working directory* with unmodified jobs being evaluated.
- An *output directory* for handling processed jobs back to the front-end.

– A *hook script* called for every job inserted into the output queue.
– A *pool of worker processes* evaluating individual jobs.

This design was chosen because it satisfied all the necessary conditions:

– It is very simple and therefore can be implemented to be robust, stable and effective.
– The front-end, the queue manager and the evaluator are separate modules and can be used independently.
– The manager and the worker processes can run on different machines sharing the data over NFS or they can run in a virtual environment.
– The recovery of a crashed `qman` consists of just moving all the submits from the working directory back to the input queue.
– It is very easy to examine the current state of all the queues and to estimate the workload. This simplifies all administrator tasks and any performance tuning.

The details of the interface and of the storage directories are described in the following sections.

8.1.2.1 Recovery

After the program crashed, all the workers are restarted by their respective init command, which should clean up any data left from previous execution and quick-check the worker consistency. It may even completely rewrite all worker data in the worker directory.

The jobs are either in the input queue (and therefore untouched), in the working directory and their copy being processed by the worker, or finished either in the output directory or in the error directory. As the job in the worker queue may only contain additional metadata and must not change any other previously present files, it is sufficient to move the job directory from working directory back to the input queue directory.

When a job is processed again for any reason its logfile is appended to or overwritten, depending on the configuration.

8.1.2.2 Input Job Queue

In the input job queue the jobs are given priorities based basically on the name of their directory. The basic (but easily customizable) order is lexicographic, so a proposed job directory name format consists of (in that order):

1. A priority category indicated by a letter (to allow more or less privileged jobs).
2. A timestamp (to process the earlier jobs sooner). Note that it may be reasonable policy to artificially increase or decrease the timestamp on insertion to express some kind of priority.
3. One or more fields briefly informing the administrator about the job and making the directory name unique.

The input queue directory re-read periodically and is also watched by the `inotify` kernel service (waking the `qman` for every new job directory).

Note that `inotify` may not work over NFS or on other (special) filesystems. In that case a different way to re-read the input directory must be used (do frequent periodic checks or implement another way to wake `qman`, ie. UDP socket waiting for any packets).

8.1.3 Job Design

Every job submitted to the manager is a directory containing a *metadata file* with any parameters for the evaluation process along with instructions for the post-processing of the evaluated job (hook scripts). This file has a fixed name (usually `metadata`). The directory also contains any data files to be processed (in the case of CodEx the source files).

After processing all the resulting files and additional metadata are added to the job directory and handed to any post-processing and delivery.

8.1.3.1 Job Metadata

The job metadata file should contain at least these attributes:

`task_name`
`task_version`
: The name and version of the task.

`task_dir`
: The directory containing the data for this task and this version.

`source`
: Name of the source file in the submitted job directory.

`exec`
: Path to the hook script to be executed when the evaluation of the job finishes. CodEx uses this to record the results into its database. This attribute may be omitted, in that case the job is left in the output directory.

In the absence of a metadata file or some necessary attributes `qman` moves the job to the error directory for debugging. The absence of a metadata may be allowed in the configuration file.

8.1.3.2 Job Flow

1. Every job, after being atomically moved to the *input queue*, wakes `qman` through the `inotify` kernel service (a service for waiting for file/directory event) and is recorded in the *inbox heap* (implemented in `inbox.c`) and in a global *job hashtable* (implemented in `job.h`). The job now has the state `JOBSTATE_INBOX`.
2. Whenever there is a worker ready to process a next job, the job with the highest priority is deleted from the inbox queue and its directory is (atomically) moved from the input directory to the working directory. The job is *assigned* to the free worker and its state changes to `JOBSTATE_WORKING`.
3. In this phase the *job logfile* is also opened and the progress is logged there. The job logfile stays in the job directory for later debugging and analysis.

   At this point, the job metadata file is read and parsed. The worker is then issued the command to process the job in the worker process.
4. After the worker finishes working on the job and writing any results to the job directory and to the metafile, the job is moved to the output directory and its status is changed to `JOBSTATE_OUTBOX`. The `qman` then synchronously calls the hook script provided in the metafile and this should report the results to wherever they should appear.
5. After the hook script finishes, the job is kept in a short status queue for the purpose of generation of an on-line status file with the list of last successful jobs. The job structures are finally deleted when there are enough newer jobs in the status queue. Then the job completely disappears from the `qman`.

The output hook command may delete the job directory from the output queue after it extracted all the needed information.

8.1.3.3 Job Errors

In the job processing there may be various errors, such as problems with file permissions (unable to move, unable to read, unable to write/append), file nonexistence (no metadata), bad format of metadata, any error in the processing of the job (these are discussed in the worker section), any problem with the worker process and finally any problem with calling the output hook command. All these errors except one permission error are handled in the same way: The error is reported to both the worker and job log, the job is moved to the error directory and the worker is restarted.

The only exception in this behavior is the error in moving the job directory to the working directory. This means that either the job somehow vanished from the input directory (in which

case the job is forgotten with a warning), or that the permissions on some of the directories (especially the input directory) are wrong and in that case the `qman` dies.

The `qman` does not analyze the actual reason for the failure. The job is always blamed and removed from processing. There is no definite way to tell what caused the failure of the processing and the job should be examined by the administrator. An automatic rescheduling the job would help only in a very limited number of cases, as at most every failure is caused either by deformed job or by misconfigured worker.

8.1.4 Worker Design

Worker is a single *shell process* running in pre-prepared worker directory. This process should run continuously and receive commands in textual form. The commands are *initialization command* to set up the worker directory, *ping command* to verify that the shell process is ready and a *work command* to process a single job.

All the workers are child processes of the `qman` process and some of them may be remote (for example `ssh`). Every child process has its own process group and therefore all its processes can be cleaned up effectively (even through `ssh`).

The worker *init command* should clean up any data left from a previous execution and quick-check the worker consistency.

The *work command* may not change any data files in the job directory and may only add new attributes to the metadata file (to allow re-processing in case of any error).

All the work on a job is processed asynchronously in dedicated child processes, each running a worker shell and processing a single job at a time. All these workers need to have separate worker directories, although some of the files (ie. binaries) may be shared.

Please note that in this section as well as in the rest of the manual, the *worker* refers to both the worker process with the worker shell and the `worker` structure in `qman` keeping the current worker status and configuration.

8.1.4.1 Worker Flow

1. When a worker is created, its basic attributes are set and its state is `WORKERSTATE_NONE`. The logfile is opened, but the subprocess is not started. The worker attributes may be changed almost freely in this phase.
2. The subprocess is started when the worker enters state `WORKERSTATE_INIT`. The subprocess redirects created pipes to the standard file descriptors 0, 1 and 2 and launches the shell command. The init command is immediately written into the shell along with appended command to echo the *init passphrase* when done. Then `qman` waits on descriptor 1 for the passphrase, descriptor 2 leads to the worker logfile.
3. After receiving the valid passphrase, worker enters state `WORKERSTATE_READY` and is immediately sent a ping command to echo the *ping passphrase*. While waiting for the passphrase, the worker is in state `WORKERSTATE_PING`.
4. When the *ping passphrase* is received, the worker changes its state from `WORKERSTATE_PING` to `WORKERSTATE_READY` and awaits the jobs.
5. After receiving a job, the worker executes the *work command* with an appended command to write the *work passphrase* and enters the state `WORKERSTATE_WORK`.
6. When the worker process indicates a successful processing of the job by sending the *work passphrase*, the worker enters the state `WORKERSTATE_READY` and is sent the ping command to verify its integrity as in step 3. The finished job is then synchronously processed by `qman` as discussed in the section Job Flow in Section 8.1 [Queue Manager Overview], page 77.

Note that while the file descriptor 1 is used to receive the passphrases confirming a successful completion of the issued command, all the output of the commands is redirected to the file descriptor 2 and then (through `qman`) to the worker log.

The ping command may be sent to the worker periodically while idle to keep the connection alive and to check that the worker is still ready.

8.1.4.2 Worker Errors

The worker may fail for many reasons, mainly because of misconfiguration of the shell or init commands, failure of the init command (because the worker directory does not contain the necessary data), init, ping or work timeout for any reason, death of the worker shell process, any failure on the side of the remote computer (in the case of distributed system), any failure in the working command (job failure) or any error in the permissions.

When the error happens, the worker process group is killed by a signal (if not dead already) and any processed job is moved to the error directory (see section Job Error in Section 8.1 [Queue Manager Overview], page 77). The worker's state is set to `WORKERSTATE_ERROR`.

The worker is killed even in the case of a job error, because it may be necessary to init/check the worker directory by the init command.

After the worker process dies, the worker is judged and, depending on the circumstances, either restarted (the child process is `fork()`'ed again and the init command is sent to the worker shell) or destroyed (freeing all the memory and closing the logfile).

8.1.5 Security

A basic stability is attained through timeouts for all worker shell commands. This prevents misconfigured tasks from blocking the system indefinitely.

The `qman` system in its basic configuration is not anyhow protected from any damage the worker processes may cause to it and therefore other measures should be applied depending on the scenario. Few simple possibilities with complexity beyond the basic distribution package of CodEx and `qman` (some of them require root access, configuration of another software and/or knowledge of the system structure):

A simple yet useful separation measure is to make the shell commands SUID (effective user change upon execution) to a less privileged user. These users may be different for every worker, separating the workers.

Even better layer of separation may be easily reached by locking the worker shell into a `chroot` jail created for this purpose. This separates the process not only by permissions, but also by filesystem.

Even stronger solution is to use a remote shell (ie. `ssh`) to start the actual worker shell on a different machine - either virtually (by the use of `qemu`, `xen` or other virtualization software) or physically. This kind of separation is practically impenetrable, but may be harder to set up and may cause small speed reduction.

8.1.5.1 Resource regulation

It may be necessary to regulate the system resources available to each worker. The basic limits may be set through `setrlimit` (memory, CPU time, number of open files) and through the *disk quota* system (these limits may need separated users).

However, more regulation may need some sort of advanced scheduler control, use of a virtual machine or a sandbox for the worker process.

The `eval` system used in the CodEx uses a sandbox approach to restrain and control the (potentially dangerous) tested programs. This is described in Section 7.4 [Eval Security], page 75.

8.1.6 Locking and Logging

Every `qman` process may place a lock on a lock file (usually `qman.lock`, but can be configured). This lock file should be placed in the root of one `qman` subtree (with queues and worker direc-

tories) to prevent multiple qman processes accessing the storages at once. This could break any job processing and eve lead to data corruption.

To inform the user and the administrator about the history and the progress of the processing, two facilities are implemented: log files and status files.

Log files are of three kinds - global log file for global information, per-worker log files for information about one particular worker and its shell, and per-job configuration files recording the whole processing of a job along with any errors.

All the log files are described in section Log Files in Section 8.2 [Queue Manager Interface], page 82.

Status files are more user oriented and provide overview of the current state of the relevant parts of a qman. The status files can be generated in more formats (currently only plain text is supported, but HTML is in preparation).

The status files are re-generated periodically and also with every relevant state change (but not too often). The status files are re-generated atomically (by a file move), so their state is always complete.

The status files include information about every worker (status, job), about every job being processed, about amount of jobs waiting in the input queue and two short lists of the latest processed and failed jobs.

Details of the status files are described in the section Status files in Section 8.2 [Queue Manager Interface], page 82.

8.2 Interface

This section describes the interaction of qman with other applications and with the user.

8.2.1 Configuration

All the configuration can be changed in a configuration file (usually qman.conf) or by command-line arguments. Most of the options change behavior of qman internals and are preset to sensible defaults. The semantics of the internal options are described in the config file and in the Section 8.3 [Queue Manager Implementation], page 88.

8.2.1.1 Configuration File

This section is partially taken from libucw documentation.

Configuration file is a plain text file consisting of configuration items. Configuration items are organized into sections. The sections form a tree structure with top-level sections corresponding to program modules.

Each configuration item belongs to one of the following classes:

1. single value or a fixed-length array of values
2. variable-length array of values
3. subsection with several nested attributes
4. list of nodes, each being an instance of a subsection
5. bitmap of small integers (0..31) or fixed list of strings
6. exceptions (items with irregular syntax; however, they always appear as a sequence of strings, only the semantics differ)

Both fixed- and variable-length arrays consist of items of the same type. The basic types supported by the configuration mechanism are:

1. 32-bit integer
2. 64-bit integer

3. floating point number
4. IP address
5. string
6. choice (one of a fixed list of strings)

Configuration files are text files that usually set one attribute per line, though it is possible to split one assignment into multiple lines and/or assign several attributes in one line. The basic format of an assignment command is

```
name value1 value2 ... valueN
```

or

```
name=value1 value2 ... valueN
```

The end of line means also end of a command unless it is preceded by a backslash. On the other hand, a semicolon terminates the command and another command can start after the semicolon. A hash starts a comment that lasts until the end of the line. A value can be enclosed in apostrophes or quotation marks and then it can contain spaces and/or control characters, otherwise the first space or control character denotes the end of the value. Values enclosed in quotation marks are interpreted as C-strings. For example, the following are valid assignment commands:

```
Database "main db\x2b"; Directory='index/'; Weights 100 20 30 \
  40 50 80                              $ a comment that is ignored
```

Numerical values can be succeeded by a unit. The following units are supported:

d=86400	k=1000	K=1024
h=3600	m=1000000	M=1048576
%=0.01	g=1000000000	G=1073741824

Attributes of a section or a list node can be set in two ways. First, you can write the name of the section or list, open a bracket, and then set the attributes inside the section. For example,

```
Section1 {
  Attr1       value1
  Attr2       value2
  ListNode {          a list and adds its first node
    Attr3     value3
    Attr4     value4
  }
  ListNode { Attr3=value5; Attr4=value6 }
                      a new node; this is still the same syntax
}
```

The second possibility is using a shorter syntax when all attributes of a section are set on one line in a fixed order. The above example could be as well written as

```
Section1 {
  Attr1       value1
  Attr2       value2
  ListNode    value3 value4
  ListNode    value5 value6
}
```

Of course, you cannot use the latter syntax when the attributes allow variable numbers of parameters. The parser of the configuration files checks this possibility.

If you want to set a single attribute in some section, you can also refer to the attribute as Section.Attribute.

Lists support several operations besides adding a new node. You just have to write a colon immediately after the attribute name, followed by the name of the operation. The following operations are supported:

```
List:clear                              $ removes all nodes
List:append { attr1=value1; ... }       $ adds a new node at the end
List:prepend { attr1=value1; ... }      $ adds a new node at the beginning
List:remove { attr1=search1 }           $ find a node and delete it
List:edit { attr1=search1 } { attr1=value1; ... }
                                        $ find a node and edit it
List:after { attr1=search1 } { ... }    $ insert a node after a found node
List:before { attr1=search1 } { ... }   $ insert a node before a found node
List:copy { attr1=search1 } { ... }     $ duplicate a node and edit the copy
```

You can specify several attributes in the search condition and the nodes are tested for equality in all these attributes. In the editing commands, you can either open a second block with overridden attributes, or specify the new values using the shorter one-line syntax.

The commands `:clear`, `:append`, and `:prepend` are also supported by var-length arrays. The command `:clear` can also be used on string values. The following operations can be used on bitmaps: `:set`, `:remove`, `:clear`, and `:all`.

8.2.1.2 Command-line Parameters

The default configuration file (`qman.conf`) is read before the program is started. You can use a `-C` option to override the name of the configuration file. If you use this parameter several times, then all those files are loaded consecutively. A parameter `-S` can be used to execute a configuration command directly (after loading the default or specified configuration file). Example:

```
bin/program -Ccf/my-config -S'module.trace=2;module.logfile:clear' ...
```

If the program is compiled with debugging information, then one more parameter `--dumpconfig` is supported. It prints all parsed configuration items and exits.

All these switches must be used before any other parameters of the program.

8.2.1.3 List of Parameters

This listing describes all the parameters customizable in `qman` config file and the command-line. This list includes also internal parameters that do not need to be modified and are included just fore completeness.

Note, that `libucw` itself defines many internal configuration options. To get more information about these, refer to `libucw` documentation.

All the paths can be either relative to `qman` current directory or absolute. All the option names are followed by their default values.

`main.logfile ''`
: This is the name of the main logfile. If empty, log goes to `stderr`.

`main.workdir ''`
: Make this the working directory before start. All the paths will be relative to this path. Empty for current.

`main.lockfile 'qman.lock'`
: Name of the file to be locked by this `qman`. Prevents accidental execution of multiple managers with the same configuration.

`main.lock-enable 1`
: Enable (`1`) or disable (`0`) the locking.

`jobs.dir-in 'queue/in'`
`jobs.dir-out 'queue/out'`
`jobs.dir-error 'queue/error'`
`jobs.dir-work 'queue/working'`
: Directories for the individual storages, see the section Submit Storage Design in Section 8.1 [Queue Manager Overview], page 77.

`jobs.metafile 'metadata'`
: Name of metafile for each job.

`jobs.logfile 'job.log'`
: Name of logfile for each job (" for none).

`jobs.no-metafile 2`
: What to do when there is no job metafile? (0-nothing, 1-warn, 2-error)

`jobs.logfile-overwrite 0`
: Overwrite job logfile when already present (ie. from aborted processing).

`jobs.error-hook '/bin/true'`
: Command called for failed jobs. It is given only single parameter - full path to copy of the failed job in `dir-out`. Use to report failed jobs to user.

`inbox.minsize 42`
`inbox.maxsize 100000`
: Minimal and maximal size of the inbox priority queue (at least 4).

`inbox.check-int-min 300`
`inbox.check-int-max 60000`
: Minimal and maximal interval between re-reading the in-dir (msec).

`workers.path 'workers'`
: Directory with all worker subdirectories.

`workers.logfile 'worker.log'`
: Name of logfile of each worker.

`workers.ping-timeout 1000`
`workers.ping-phrase 'PING42PING'`
`workers.init-timeout 60000`
`workers.init-phrase 'INIT42INIT'`
`workers.work-timeout 60000`
`workers.work-phrase 'WORK42WORK'`
: Timeouts and phrases for detecting the ending of individual worker phases.

`workers.read-buf 2048`
`workers.write-buf 2048`
: Buffer lengths for each worker.

`workers.worker`
: List of workers to be used.

`workers.worker.name 'eval1'`
: Name of the worker directory in `workers.path`.

`workers.worker.data-path '.'`
: Path to common data storage, from workers view (ie. over NFS). Given to the worker as parameter.

`workers.worker.shell-cmd '/bin/sh'`
: Shell executed on worker start and receiving all the commands for the worker. Must be a POSIX shell (ie. `sh`, `bash` via `ssh`, ...).

`workers.worker.init-cmd '/bin/true'`
`workers.worker.work-cmd '/bin/true'`
: Commands issued to init the worker and to start every work.

`status.file-plain 'qman.status.txt'`
`status.file-html 'qman.status.html'`
: File names for different status formats.

`status.update-int-min 300`
`status.update-int-max 60000`
: Minimal and maximal interval between re-writing the status files.

`status.show-out 15`
: Number of last successfully precessed jobs to show in status.

`status.show-error 5`
: Number of last failed jobs to show in status.

8.2.2 Metadata

Metadata files store information and parameters of incoming jobs and well-formatted results ofter their processing. The metadata file has a very simple text format consisting of these items:

1. One attribute of type `name:value`, all the values are arbitrary single-line texts.
2. Single-line comment.
3. Empty line.
4. Named subtree (nested attribute) containing any number of the above items and named subtrees.

Every attribute is defined on a single line, therefore neither the names nor the values may contain newlines (`'\n'`) or NULL characters (`'\0'`, ASCII code 0).

8.2.2.1 Metadata Grammar

The metadata file is rigorously best described by the following grammar:

```
<file> = <attribute>*

<attribute> = (<single> | <nested> | <comment> | <emptyline>)

<single> = <indent> <name> ":" <value> "\n"

<nested> = <open> <attribute>* <close>
<open> = <indent> <name> "(" "\n"
<close> = <indent> ")" "\n"

<comment> = <indent> "$" <value> "\n"

<emptyline> = <indent> "\n"

<indent> = (<sp>|<tab>)*
<name> = (A-Z | a-z | 0-9 | "-" | "_" | "." )+
<value> = <any-character-except-newline-and-NULL>*
```

It is strongly recommended to represent the <value> in UTF-8, however, this is not necessary and any encoding may be used provided that all the applications using the metadata files use the same.

Any unknown attributes in the metafile are ignored as these may be used by other components. Collisions in naming new attributes should be avoided by using appropriate prefixes.

8.2.2.2 Attributes for CodEx

When submitting a job, CodEx writes metadata file with *at least* these attributes:

`task_name`
: The name of the task. In the case of CodEx, this is a number.

`task_version`
: The version of the task. Also a number.

`task_dir` The directory containing the data for this task and this version.

`codex_type`
: The category of the job. Either `submits` or `solutions`.

`codex_id` Internal id in the CodEx database.

`source` Name of the source file in the submitted job directory.

`exec` Path to the hook script to be executed when the evaluation of the job finishes. CodEx uses this to record the results into its database.

The called `eval` system adds one nested attribute named `test` for every test in the problem. The `test` subtree contains some of the following attributes depending on the result. `id`, `points`, `status` and `message` should be always present.

`id` The name of the test, usually number.

`points` Assigned points, 0 in case of any error.

`time` Processor time consumed by the process (sec).

`time-wall`
: Total (wall) time time consumed by the process (sec).

`mem` Maximum memory consumption (bytes).

`exitcode` Program exit code (if other than 0).

`message` Human-readable string describing the test result.

`status` Two-letter status indication.

The `status` is one of the following:

`OK` Test passed successfully

`CE` Compile error (test not run)

`FO` The program tried a forbidden operation

`RE` The program exited because of a run-time error

`SG` The program was killed by a signal

`TO` Time limit exceeded

`WA` The program gave a wrong answer

`PA` The program gave only a partial answer

`PE` The program interaction/output was not in the specified format (protocol error)

`XX` Internal or unknown evaluator error

8.2.3 Log Files

The messages from the qman system are either general messages, messages about a single worker or messages about a single job.

Every message takes one line in the following format: `I 2008-08-25 13:42:45 Acquired lock on lockfile 'qman.lock'`

The first letter represents severity of the message (**D**ebug, **I**nfo, **W**arning, **E**rror, **F**atal and **!** for the fatal `die()` message). The two following fields are date and time of the message and are follower by the message text.

General messages include initialization progress, list of job-worker pairs, error messages and debug statements (if DEBUG was switched on during compilation). It is logged to the file set by `main.logfile` or to standard error output (`stderr`).

Worker messages include worker initialization, assigned jobs, any output from the worker (without the usual heading), job and worker errors and debug messages. The logfile is set by `workers.logfile` usually in the worker directory.

Job messages contain detailed progress of the job processing and also any errors with some debug messages. The logfile is present in the job directory and its name is defined by `jobs.logfile`.

8.2.4 Status Files

To provide some on-line evaluation status information, qman produces a status overview file(s) every time the relevant status changes (and also periodically).

The basic format is plain text, but other formats may be included easily (HTML status file is under development). The path to the file is set by `status.file-plain`. The status files are replaced atomically and therefore a reader won't ever see a partial file.

8.3 Implementation

qman is a program written in plain C (version C99) and uses some specifics of the gcc compiler, although these are not necessary.

The program is written to use specific functions from the Linux operating system and its portability to other systems (even POSIX systems) is therefore very limited.

8.3.1 The libucw Library

The only external library used in the program is libucw from the project *Sherlock Holmes searching engine* which is distributed under the terms of GPL. libucw is a general-purpose library with collection of data structures (heap, list, trie, red-black tree, hash-table, ...) and routines (mainly system and I/O helpers).

The single sophisticated part used from libucw is the mainloop module. It provides mechanism to register all the relevant files and processes and allows to add timers and other kinds of hooks. When the program enters the `main_loop()` function, the files, processes and timers are waited for and appropriate handlers are called for any events on them (ready to read/write, process exited, timer expired, handler should be called). The kernel of that module is a `poll()` call with an appropriate timeout.

Other structures include heap and lists, which are implemented very simply and elegantly.

A bit more elaborate structure is the hash-table, which is implemented as a header file included every time you want to make either one global hash-table or a hash-table type. The desired properties and the name prefix are defined by macros.

Another used module is the configuration reader. The definitions of sections are given as constant structures (and arrays of structures, nested) describing the configuration items (string name, variable, type) along with initialization and correctness checking functions.

8.3.2 Modules

The individual files cover their respective areas. The major structures, functions and types are described here.

For a detailed description of the individual functions please refer to the sources. Reading the source of a short function should be more useful than reading a description actually longer than the code itself. If in any doubt, please contact the authors.

Some simple functions internal to the modules are omitted in the module descriptions.

8.3.2.1 job

`job.c, job.h`

This module defines the `struct job` representing a single job in all its states and provides functions assigning a job to a worker and a function fetching the job from the worker and calling the output hook command.

The `job` structure is defined as follows:

```
typedef struct job {
  char *dirname;          /* name of the job directory */
  int state;              /* JOBSTATE_xxx, state of the job */
  struct metanode *meta;  /* the metadata read from the metafile */
  struct worker *worker;  /* the assigned worker, if any */
  FILE *log;              /* an open per-job logfile (or NULL) */
  } job;
```

The meaning of all the attributes is straightforward. The attributes `meta`, `worker` and `log` may be null, but both `meta` and `worker` are defined in the state `JOBSTATE_WORKING`.

The `dirname` and `metadata` should be allocated by `xmalloc` as they are `xfree`'d on job destruction.

A global hash-table (from libucw) is declared to store all loaded jobs (even those in the input queue and those remembered by status). This table provides a very fast lookup by the job directory name.

The `job` module defines several major functions to work with jobs:

`void job_init(void);`
: Initializes the module (mainly jobs hashtable structure).

`void job_to_error(job *j, const char *fmt, ...);`
: Marks the job as failed (as described in the section Job Flow in Section 8.1 [Queue Manager Overview], page 77), releases and kills the worker moves the job directory to the error directory and adds the job to the status error queue.

`job* new_job(const char *dirname);`
: Creates a new job for the given directory, registers it in the job hashtable does NOT add the job to the `inbox` heap.

  Returns the new job or `NULL` if a job with the same name is already present present or on error.

`int read_job_metadata(job *j,int filter);`
: Reads the job metadata. The job directory must be still in the input queue.

  May fail and move job to error, returns 1 on success, 0 otherwise.

`void destroy_job(job *j);`
: Frees loaded job metadata (if any), closes the logfile, removes the job from the job hash-table and deallocates the `job` structure.

Two important functions are provided for the interaction with the assigned worker:

`int pickup_job(void);`
: Picks the job with highest priority from the inbox heap and assigns it to a free worker. Constructs and writes the worker *work command* (the worker command may be customized in the sources).

  Return 1 if one job was picked up, 0 if none (no jobs in the inbox heap, no no free workers or some job error).

`int job_finish(job *j);`
: This is called after the worker reads the *work passphrase*. Detaches the job from worker, sends the worker a ping command, moves the job to the out directory and runs the hook output command on the job.

  Returns 0 on success, 1 on any error (job or hook error).

8.3.2.2 worker

`worker.c, worker.h`

This module defines the `struct worker` representing a single worker in all its states and provides functions manipulating and controlling the worker.

The `worker` structure is defined as follows:

```
typedef struct worker {
  cnode n;              /* libucw list element */
  char *name;           /* worker name */
  int state;            /* current state, WORKERSTATE_xxx */
  struct main_file fin, fout;  /* file objects, out goes TO the process */
  int inpipe[2],outpipe[2];    /* pipes FROM and TO the worker */
  struct main_process proc;    /* the worker process */
  struct main_timer timeout;   /* timeout object for all the actions */
  FILE *log;            /* worker logfile */
  struct job *job;      /* assigned job (if any) */
  char *write_buf;      /* buffer for writing commands */
  char *read_buf;       /* buffer for reading response */
  char *data_dir;       /* [*] global data path (relative from the worker) */
  char *shell_cmd;      /* [*] shell command (bash, ssh, ...) */
  char *init_cmd;       /* [*] init command - initial cleanup */
  char *run_cmd;        /* [*] work command - process a job */
} worker;
```

Note, that fields marked with `[*]` are `NULL` after `worker_new()` and may be set to appropriate values before worker start. If unset, `start_worker()` fills them with defaults (`getcwd()`, `/bin/bash`, ...).

The global worker list and worker counters are declared in this module.

The module defines these major functions to manipulate the workers:

`void init_workers(void);`
: Initialize the module and the worker list.

`void worker_setstate(worker *w, int state);`
: Set the state of the worker. This is provided to adjust all the worker counters (busy, ready).

`worker *worker_new(const char *worker_name);`
: Creates a worker and initializes all its values. Opens the worker logfile, opens the pipes and registers the file descriptors in the `libucw mainloop` module. Adds the worker to the global worker list.

`int worker_start(worker *w);`
: Starts (by `fork()`) a new worker process with worker shell. Send it the init command.

  Returns 0 on success, otherwise in case of any error.

`void worker_kill(worker *w);`
: Sends a `KILL` signal to the worker process group. The worker is not restarted until the worker process dies and is `wait()`'ed for by the main loop.

`void worker_destroy(worker *w);`
: Deallocates all worker data, closes the log and the pipes, removes the worker from the global worker list.

  Note that a worker should have no job assigned at this point.

`void worker_ping(worker *w);`
: Sends the ping command to a free ready worker.

`void worker_error(worker *w, const char *fmt, ...);`
: Report the worker failure, kill the worker if not dead already, blame the job (if any) and call `job_to_error()`.

`void worker_judgement(worker *w);`
: This routine considers the dead worker and either restarts or destroys it.

8.3.2.3 inbox

`inbox.c, inbox.h`

This module contains the global *inbox heap* used as a priority queue and provides functions for manipulating it and for re-reading the input queue directory. This module also contains the timer responsible for the periodic re-reading of the input directory (fallback in case `inotify` fails for some reason, ie. operation over NFS).

This module also provides an interface to the `inotify` system, that provides an open file with a read event every time some directory is moved into the input queue directory.

The *inbox heap* is a heap in a dynamic-size array from the `libucw` library.

The major functions defined in this module are the following:

`void init_inbox(void);`
: Initialize the inbox heap and timer.

`int inbox_init_inotify(void);`
: Initialize inotify input directory watching, return 0 on success.

`void set_check_inbox(void);`
: Schedule the inbox re-reading as soon as allowed.

`int inboxheap_less(struct job* a, struct job * b);`
: A function determining the heap order of the jobs, can be redefined for a different priority measure.

`int inboxheap_insert(struct job *j);`
: Heap insertion with potential heap expansion. Returns 1 on success, 0 on failure (heap size reached its upper limit).

`int reread_inboxdir(void);`
: Re-read the entire input directory, insert all the unknown jobs, return the amount of new jobs.

`struct job* inboxheap_delmin(void);`
: Delete and return the job with the highest priority from the inbox heap. Return `NULL` in case the heap is empty.

8.3.2.4 status

status.c, status.h

The module contains the two job lists (for last successful and failed jobs) and re-generates the status files (periodically and upon change).

The major functions defined in this module are the following:

void init_status(void);
: Init the module and the periodical status timer.

void status_changed(void);
: Schedule the status files update as soon as possible.

void status_update(const char *fname, int type);
: Atomically update the status file of given type (STATUS_PLAINTEXT or STATUS_HTML).

void status_add_job_out(job *j);
: Add a successful job to status list "last output jobs".

void status_add_job_err(job *j);
: Add a failed job to status list "last failed jobs".

8.3.2.5 metadata

metadata.c, metadata.h

The module contains the definition of the metadata tree and routines for loading, printing and querying them.

Refer to Section 8.2 [Queue Manager Interface], page 82 for the details of metadata structure.

The structure metanode is defined as follows:

```
typedef struct metanode {
  struct cnode m;     /* member of a list */
  u8 type;            /* type of this item, METANODE_xxx */
  char *name;         /* name of ITEM or TREE */
  char *data;         /* text of ITEM or COMMENT, (clist*) of TREE */
} metanode;
```

The major functions defined in this module are the following:

void fprint_metalist(FILE *f, const struct clist* l, int indent, int indshift);
: Print all the nodes in the list in the metadata format to the given file. The last two parameters describe basic indentation and steps for nested levels.

metanode *meta_item_findnext(metanode *list, metanode *from, const char *name);
: Find the first METANODE_ITEM named name after from (from=NULL for search from the beginning, name=NULL for any item).

  Return NULL on failure (no more such items).

char *meta_item_findnext_data(metanode *list, metanode *from, const char *name, char *def);
: Similar to meta_item_findnext() but return directly the item data pointer (or def if none found).

metanode *meta_tree_findnext(metanode *list, metanode *from, const char *name);
: Find the first METANODE_TREE named name after from (from=NULL for search from the beginning, name=NULL for any item).

  Return NULL on failure (no more such items).

The routines for metadata loading are omitted as these are internal and called only from read_job_metadata() from job.h.

8.3.2.6 conf

`conf.c, conf-read.h, conf-read.c`, generated `conf.h`

The module contains the global parameters of all the modules and routines to load them from command line an/or from a config file.

The header file `conf.h` is automatically generated from `conf.c`.

All the global variables and their configurations are described in the section List of Parameters in Section 8.2 [Queue Manager Interface], page 82.

The only relevant function `void init_conf(int argc, char *argv[]);` reads the parameters from a command line and a config file and initializes the config variables. It also sets up the list of simple `struct worker_conf` of workers to be initialized.

8.3.2.7 log

`flog.c, flog.h`

This module slightly extends the `libucw` log facility to include logging to any file.

The main provided function is `void flog(FILE *f, unsigned int cat, const char *fmt, ...);` that logs a formatted message into a `(FILE *)` with severity and date and time stamp.

See `libucw` log system for details.

8.3.2.8 qman

`qman.c`

This module contains the `main()` function. The function loads the configuration, initializes all the modules, recovers lost jobs, starts all workers defined in the configuration and enters the `mainloop` from `libucw`.

At this point, all the necessary files, timers, hooks and processes are already configured and registered in `mainloop` and the program runs by calling appropriate handlers for all files with data to be read, processes to be waited for, timers expiring and others.

Appendix A Glossary

action An operation performed by a page controller, that the page exposes by defining a public method. For the action *foo*, this method is named `fooAction`. It is activated through an HTTP POST request.

admin A special user account with the database ID 1. The admin is automatically granted maximum rights to all objects, regardless on the actual values in the *rights* fields of his user account. The account is special in that it cannot be modifed or removed. Any user possessing maximum rights (i.e. any user given the *supervisor* role) can do anything that *admin* can do.

back-end The back-end is responsible for the evaluation of enqueued jobs. It runs asynchronously from the WWW front-end and it consists of Qman and the workers (MO-Eval etc.).

CFI Class-based Filesystem Interface. An abstraction providing users remote access to files. Responsible for securing file access and allows versioning.

controller The controller (of a page or component), part of the Model-View-Controller design pattern. is responsible for responding to user actions and controlling the display of a page or component.

component
A piece of code that can be inserted into many pages (even several times into the same page). They are used for frequently repeating page elements such as tables, forms, menus etc.

config The file named '`config`' in the exercise directory containing the configuration and limits of tests. It is usually edited through a form-based interface, but also accessible directly to the users.

delegation An object bestowing the rights of one user (the granter) upon another user (the trustee).

driver (CFI)
A virtual-filesystem driver (primarily a class implementing the `ICFI_Filesystem` interface). There is a plain filesystem-based driver and archive-based drivers (Tar, Zip).

entity object
The ultimate abstraction of a database record (or records). In the application, database is accessed exclusively through these. They are built upon *table gateways*.

exercise A problem that should be (in the end) solved by the users. An exercise must be first assigned as a task (to a group) which allows the members of that group to solve it.

An exercise consists of a database record and a directory with configuration and data. From the point of view of the back-end, it is just a directory.

Eval Short for MO-Eval.

filter (CFI)
A class that provides file-filtering services, mostly used in the file manager. There are filters available for compression, decompression and newline conversion.

file manager
A component that allows remote access to files through a web-based interface. It is the primary means for the users to upload test data for exercises and attached files for exercise specifications.

front-end The WWW-based user interface written in PHP. Apart for the *hook* script, it runs synchronously with the client's requests.

gateway, table gateway
Abstraction of a database tables. Gateways serve as factories and caches for database entity objects.

granter The user who delegates his rights to another user.

group A group organizes together a set of users (its *members*) and a set of exercises (the *tasks*) they solve.

group membership
A relation describing which users belong to which groups. A user can *join* a group, but he cannot leave. He can be *added* to the group or *removed* from it by an operator.

group operator
A user who is managing (operating) a group. More generally, anyone possessing the rights to a group greater or equal to those defined by the *Group Operator* role.

job A unit of job for the evaluation back-end. The front-end generates one whenever it needs a submit or solution evaluated. Physically a job is a directory containing the source code and a `metadata` file. It is actually just a copy of the directory of a submit or solution.

metadata The file named `metadata` that is used for communication between the front-end and the various parts of the back-end. These files are generated by the front-end. They reside in the directories of the submits or solutions (and in the jobs). The parts of the back-end add the results of the evaluation to the metadata file.

MO-Eval The third-party component of the back-end that handles the evaluation itself. It is the principal part of the workers.

news, news item
A piece of text created by a user to pass information to other users (the members of a group, all users). News can be browsed by the other users and they usually have an expiry date after which they are deleted.

notification
E-mail messages automatically send by CodEx, informing about various events. In a more narrow sense, these are just the *maskable notifications* that the user can turn on or off in his user preferences.

object Short for *entity object*.

operator Either a *group operator* or, in general, any user possessing the rights of the roles *Group Operator*, *Head Operator* or higher.

output queue
A directory where Qman (the back-end) puts evaluated jobs. Qman then executes the front-end's *hook* script that picks the evaluated job up from the output queue.

page A unit of the user interface, usually displaying similar information. It is implemented as a pair of classes/scripts the *controller* and the *template*.

Qman A generic application written in C that picks up *jobs* from a *queue* and assigns them to one or several *workers* which process the jobs one at a time.

queue (evaluation queue, Qman input queue)
A directory where the front-end puts jobs for evaluation. It contains subdirectories each corresponding to a job. Qman picks these up in lexicographic order and moves them out of the queue.

rights In the general sense a permission to do something with an object in CodEx. In the more narrow sense it is a privilege level, represented as an integer from 0 to 255 (or as RIGHT_NONE through RIGHT_ADMIN) that determines the rights the user has or the rights that are required to perform a certain operation.

role One of the pre-defined templates for setting up the rights of a user. (Common User, Group Operator, Head Operator, Supervisor).

sandbox A part of MO-Eval that intercepts all system calls of a program being evaluated and prevents it from performing undesired operations.

service Special entry point that provides access to CodEx services for automated scripts. Whole chapter *Services* is dedicated to this topic.

session An object stored on the server that works around the statelessness of the HTTP protocol. Implemented by the `Session` class.

solution Created by privileged users (the author of an exercise or a user who wants to assign it), usually to test the exercise. Similar to a *submit*, it consists primarily of the source code that tries to solve an exercise and should be evaluated. A solution solves an exercise *directly*, whereas submits solve them indirectly through tasks.

storage

submit Created by common users as an attempt to solve a task. It has a structure similar to a solution. But whereas solutions solve exercises directly, submits solve them indirectly, through tasks.

supervisor A pre-defined role having maximum rights to all objects. Also any user having these rights. There can be any number of supervisors in the system. There is always at least one, the *admin*, i.e. the user with ID = 1.

task An exercise assigned to a group, where the group members are supposed to solve it. It has several attributes that can modify the properties of the exercise from which it is derived.

template A PHP script consisting mostly of HTML with some PHP fragments. It is used to compose the contents of a page or component for display, using data stored in a *view* object.

There are also e-mail templates which are simpler and only allow variable substitution.

text The text of an exercise specification. An exercise specification can have several texts, representing different revisions of the specification. One of them is always marked as the current one.

trustee The user to whom some rights were *delegated* by another user (the *granter*).

user An object usually representing a physical person registered in the system. For each user, there are associated preferences, group membership, results etc.

view An object holding data for insertion into a template. Part of the Model-View-Controller design pattern.

worker A running process (with an associated directory tree where it resides) that sequentially processes jobs. Jobs are handed to workers by Qman.

Appendix B Database Schema