Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Jakub Faryad

Experiment management system

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis:	Mgr. Tomáš Balyo
Study programme:	Computer Science
Specialization:	Programming

Prague 2012

I would like to thank my supervisor Mgr. Tomáš Balyo for all the time and good advice he's given to me.

Název práce: Experiment management system

Autor: Jakub Faryad

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Tomáš Balyo

Abstrakt: Práce se zabývá návrhem a implentací systému pro správu, spouštění a porovnávání výsledků programů pro řešení výpočetních problémů. Aplikace distribuuje výpočty na předem definované skupiny vzdálených počítačů pomocí protokolů ssh a scp. Systém umožňuje více skupinám uživatelů sdílet vstupná data, nahrávat nové verze programů a naplánovat jejich spuštění. Výsledky jsou uchovávany v databázi, lze je kdykoli stáhnout v různých formátech a graficky srovnávat. Výsledkem je vícevrstvá webová aplikace na platformě J2EE.

Klíčová slova: webová aplikace, správa experimentů

Title: Experiment management system

Author: Jakub Faryad

Department: Název katedry či ústavu, kde byla práce oficiálně zadána Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Tomáš Balyo

Abstract: The aim of the thesis is to design and implement a system for managing and running and comparing the results of programs for solving computational problems. The application distributes the experiments on predefined computer pools through the ssh and scp protocols. The system allows multiple user groups to share input date, upload new versions of programs and schedule their execution. The results are stored in a database, they can be downloaded at any time in various formats and graphically compared. The result is a multi-layer web application based on the J2EE platform.

Keywords: web application, experiment management

Contents

Introduction

Developing algorithms and improving heuristics for solving complex computational problems requires frequent running of the developed programs with a large number of inputs, comparing the results and analyzing in what aspects they improved. Performing this task manually can be very time consuming and it drives away from the original research work. The proposed solution to this problem is an experiment management system, called "Experimenter".

The aim of the thesis

- The main goal of the thesis is to implement a computational experiment management system, that would enable the users to quickly schedule new experiments, manage available input data and pools of available machines to run the computations on and view the results in the most convenient way.
- A secondary aim is to create a modern, easily maintainable an extendable web application using the current industry best practices for developing quality software.

Structure of the thesis

The text is split into 3 chapters. The first chapter contains the specification of the implemented system and analyses the requirements in order to propose the most proper solution. The second chapter explains the usage of the system from a user point of view. The last chapter describes the architecture and implementation details of the application. In the conclusion, the result is evaluated and possible future improvements are suggested.

1. Analysis

This chapter describes what characteristics and features the system should have, whether there are any available alternatives and what decisions where made in the design phase of the development.

1.1 Requirements

In terms of features, the system must meet the following requirements:

User management. It must be possible to add and remove users, group them into user groups and only allow them to see and manipulate objects (projects, experiments, connection pools) that belong to their user group.

Problem type definition. The user can define a new type of problem, assign input data specific to that problem and create projects for solving that kind of problem.

Input management. Input files specific to one problem type can be grouped into reusable input sets that can be assigned to experiments. The experiment then executes the tested program for all inputs in the assigned input sets.

Connection pools. It must be possible to create connection pools consisting of remote computers the user has access to through ssh and assign theses pools to individual experiments.

Load balancing. The application must try to distribute the execution of the experiment for individual inputs to the computers in the connection pools evenly.

Program versioning. It must be possible to run experiments for all uploaded versions of a program, unless they are deleted manually.

Result parsing. The programs themselves have to measure the characteristics of interest and print them to the standard output. The results of an experiment should then be extracted from the standard output of the executed program following specific rules on how to parse the output.

Result download. The results of all finished experiments should be available for download in the from of a CSV file and a zip archive containing raw output from the executed programs.

Result charts. It must be possible to generate charts in a vector graphics format comparing results of different experiments for every of the characteristics of interest measured in the experiments.

Usability. The system must be easy to use, frequently repeated operations must be simple to perform.

System requirements. The application must run on a UNIX system and must be build on the J2EE[3] platform.

1.2 Alternatives

Before starting to design the system, available alternatives have to be evaluated. It would be of no use to build a system, that has been already implemented elsewhere and fits the described usage scenario. However, no comparable system that would meet at several of the listed requirements was found. Given the field of usage, it is probable that there are some custom made solutions, like our system is, but they are probably used internally by computer science departments at universities for example and are not publicly available. Up to now, the intended users of the Experimenter had to manage, run and evaluate the experiments manually from the command line.

1.3 Requirement analysis and design

Used technologies

The main decision to be made is the correct technology stack to choose. Given the purpose of the system and the requirement on the platform and operating system, some design decisions are straightforward. For example:

- The web application will run in a web container or application server.
- Some lightweight database will be used, as there are no heavy load of the database to be foreseen.
- The standard solution for architecture of a Java-based web application is a multi-layered architecture with separated front end and back end.

The most important decision to be made is to choose the technology stack to use. The two major options are:

- The official Java EE specification: JSF + EJB + JPA + an application server like the GlassFish Server
- The popular alternative: Spring + Hibernate[4] + a lightweight web container + one of many available front end frameworks.

Since the second alternative is much more flexible, lightweight and the author already had experience with the mentioned technologies, the choice was made for using the alternative technology stack. For the same reason, Apache Wicket[5] is chosen as the front end framework. It is a modern, pure MVC framework and similar to the rest of the chosen frameworks, it has an active developer community, which plays an important role when looking for some technical help.

Load balancing

The requirement on load balancing addresses two separate concerns:

- The remote systems should not be overloaded, or else the execution times can grow substantially, the other users of the remote systems will suffer from great performance decrease and the results for characteristics that depend on the remote system performance (like experiment runtime) can be corrupted.
- Too frequent connections may trigger some sort of denial of service protection actions. Most systems, if receiving too many subsequent connections, will start to refuse them.

The proposed solution is to introduce configurable restrictions on the maximum number of experiments running on a computer as well as retry intervals in case a connection is refused or there are no available unoverloaded computers. Further restrictions can be configured on the size of the experiment thread pool.

Usability

The most typical use case of the application for a user is:

- uploading a new version of a program
- choosing and cloning an older experiment, preserving most of the settings (input data and connection pools don't change that often), setting it up against the new version
- running the experiment
- waiting for it to finish, then downloading the results or comparing them graphically to previous versions or the newest versions of other programs.

To make this process as quick as possible, the user interface has to be designed with this in mind. Ideally, after the user uploads the new version of the program, each step in the described process should take exactly one mouse click.

2. User documentation

This chapter explains the usage of the main features of the system by going over the standard use cases from the point of view of the client user.

2.1 System requirements

The system requirements for running the Experimenter application are a unix system with the Java Runtime Environment[1] installed and an application server or a web container, like Apache Tomcat[2]. A more detailed description of the installation and configuration procedure is provided in the appendix B. Once the application is installed, the only requirement for using it is a modern JavaScript enabled browser. Thanks to that, the Experimenter is operating system independent.

2.2 Users

Before the user can start using the system, he has to register. Since the target usage of the application is not to be accessible as a public service but rather as an internal system owned by a smaller number of users, it was decided, that there would be no automatic self-registration procedure available in the current versions of the program. Thus, the user has to ask an administrator to perform the registration.

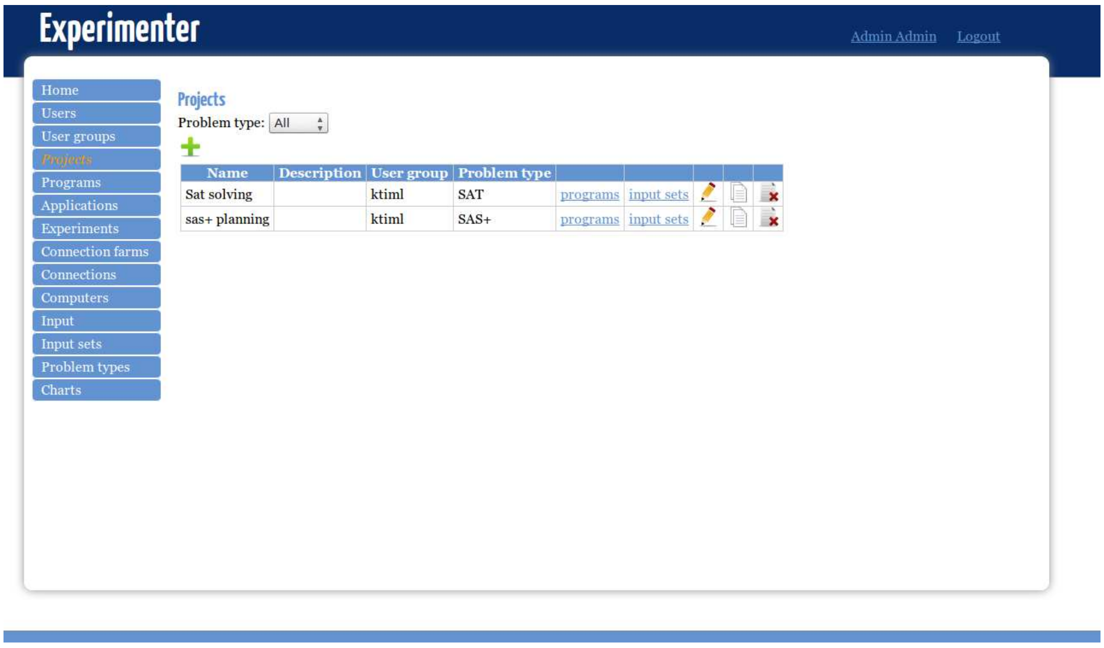
User roles

There are two different user roles available in the system:

- A **standard user** can create new projects, upload data files, executable programs, configure connection pools, launch experiments and view and download results. He only sees objects belonging to the user groups he himself belongs to, however, he can't manage his user groups.
- An **administrator** has all the privileges a standard user has. On top of it, he can create, edit the information about other users including administrators, delete their accounts in the system and assign them to user groups.

Logging in

Once the user is registered, he can connect to the system in his web browser. Any access to the application by an unauthenticated user results in being redirected to the log in page. By checking the checkbox "Remember me", a cookie will be saved in the browser, ensuring that the user will be able to access the system without having to authenticate again, unless the cookie is deleted or the server is restarted and it's work directory is cleared.



Problem type: All

Name	Description	User group	Problem type					
Sat solving		ktiml	SAT	programs	input sets			
sas+ planning		ktiml	SAS+	programs	input sets			

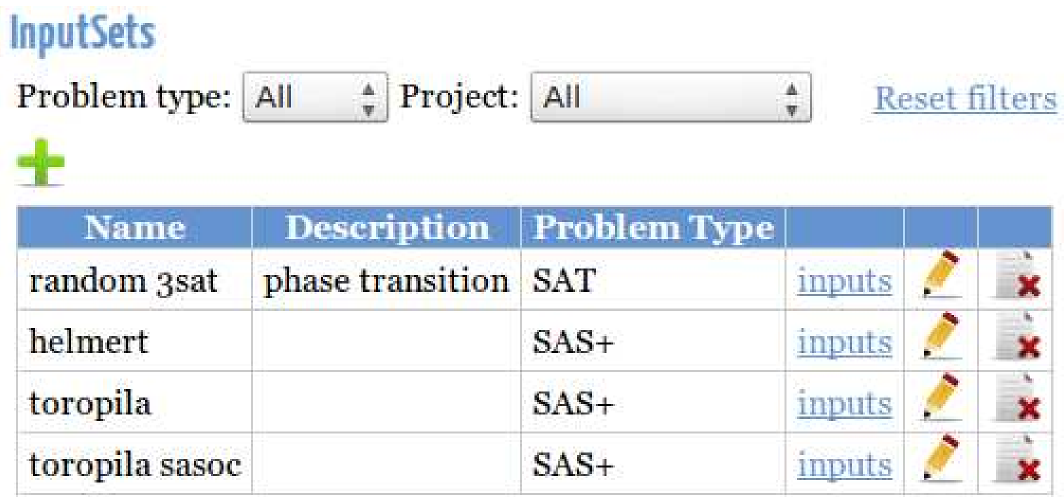
Figure 2.1: Menu and the basic page layout

2.3 Structure of the user interface

The purpose of the system is to run computation tasks and display their results. However, in order to be able to launch an experiment, certain configuration has to be done. The user has to create a new project, specify the problem type, upload data etc. Each of theses actions corresponds to setting up a domain object and has it's own configuration page. These pages can be accessed using the menu on the left. The menu is present on every page, only the area on the right changes (fig. 2.1). Individual pages will be described in the usual order the user has to access them before he can run an experiment. However, since they look very alike in terms of functionality and used controls, these shared controls will be described first.

2.3.1 Tables

All the pages meant for configuration of the domain objects are in the from of data tables, with rows representing individual objects and columns their most important properties (fig. 2.2). What objects are displayed depends on whether the user has permission to see them (they might belong to another user group) as well as on the currently selected values in the filters above the table. The filters will be described in more detail later in this chapter. The table can contain links to other pages. For example, if one clicks on the link "inputs" in a row in the table of input sets, he will be redirected to the Input page, where only the input files assigned to the selected set are displayed. This way, even though every domain object has it's own configuration page, the user can easily navigate through related domain objects.



InputSets

Problem type: All Project: All Reset filters

Name	Description	Problem Type			
random 3sat	phase transition	SAT	inputs		
helmert		SAS+	inputs		
toropila		SAS+	inputs		
toropila sasoc		SAS+	inputs		

Figure 2.2: Data table with filters and links to related domain objects

2.3.2 Icons

The following clickable icons are present on almost every page. Icons specific for one page will be described together with that page

Add

Clicking the add icon opens a dialog with a HTML form where the user can put in all the required as well as the optional information about the new domain object.

Edit

The edit icon is present on every row. Clicking it brings up the same form as the one used for creating a new object, but in edit mode some of the input fields are disabled. For example, it is not possible to reassign a program to another project.

Clone

Some object are created often, repeatedly with very similar setup. In these cases it is convenient to quickly copy over an already existing instance and only change some values. Therefore there is a "Clone" icon in every row on some selected pages, where it is most useful, e.g. Program, Application and most of all Experiment.

Delete

All objects can be deleted by clicking on the "Delete" icon. To prevent the user from deleting an object by mistake, an additional confirmation dialog has to be confirmed before the delete action is executed.

2.3.3 Filters

Comparing to the number of domain objects created in the application, the display area provided by a single computer screen is very small. In order to prevent the user from having to scroll through long data tables, most pages offer the possibility to filter the displayed data. For example, when listing through available input sets (sets of grouped input files), the number of displayed items can be narrowed down by selecting a problem type, or a specific project, that they have to be assigned to (fig. 2.2). By default, the filters on a page are set to "All", except for the cases, when the user is navigated to that page from another page by following a relation from some domain object. In that case, some values might be preselected, depending on where the user was navigated from. For example, if the user selects a project on the project page and clicks the link "programs", he will be taken to the program page, with the selected project already preselected in the filter. If there are multiple filters on the page, there is always also a "reset filters" link present, that enables quick restoring of all filters to the default "All" value. Selected filter values are taken over and pre-filled in the add/edit form if the user clicks on "Add" or "Edit" while having a value selected in a filter.

2.4 Pages

Problem types

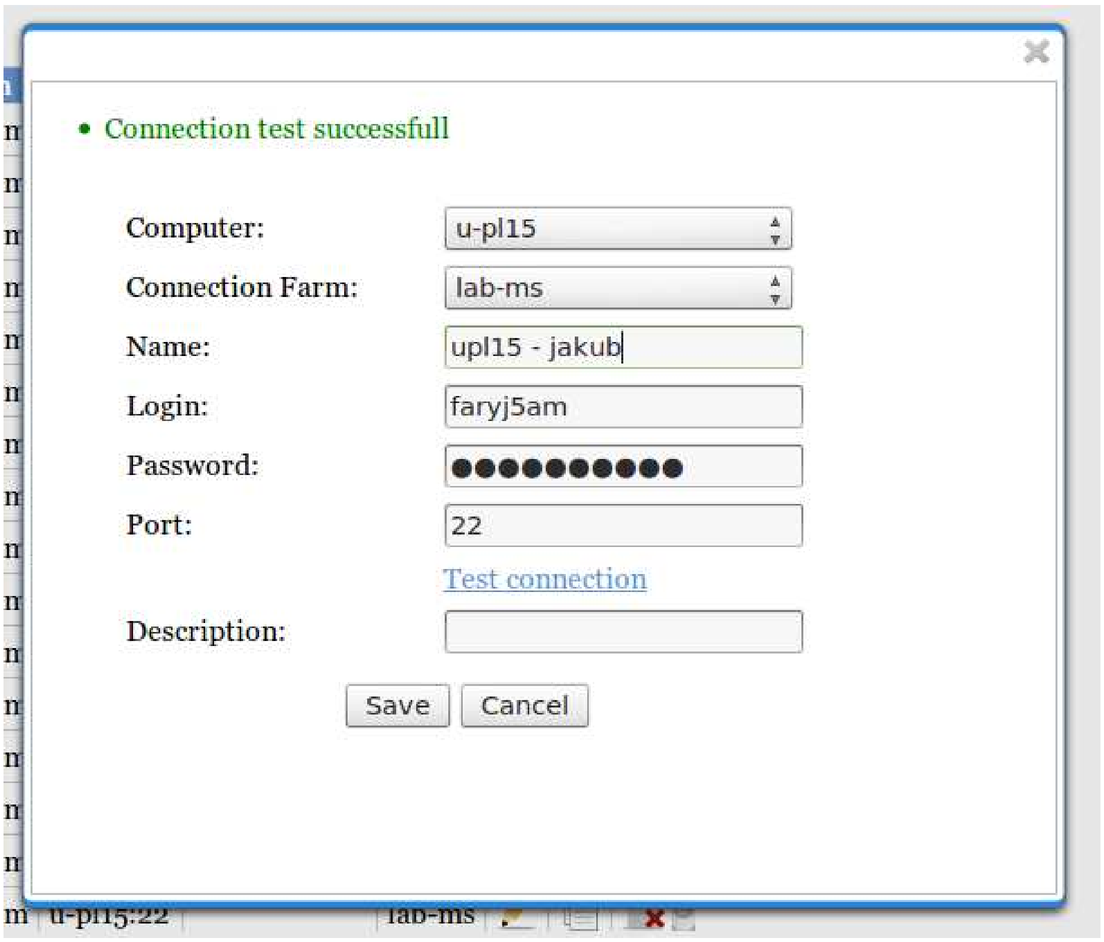
The first step is to define the type of the problem, the user will be working on. The problem types serve mainly as a means to group input data. This way, it is easy to filter for example possible input data for a specific type of problem. A problem type is defined globally and is visible even from outside of a user's group.

Computers

A computer represents one physical machine. The experiments are run on remote machines, so the users have to provide authentication information on those systems. Different users can use the same machine with different accounts. If the tasks are to be load balanced however, running jobs have to be maintained not on user account level, but on physical machine level. That's why a computer is also shared among all user groups. A computer object thus holds only the address/hostname of the machine, login information is managed on the Connections page. To see all connections assigned to one computer, one can click the "connections" link in the given computer's row in the data table.

Connections

After defining the computers, the user has to provide login information for them. This can be defined on the Connections page. Every connection is assigned to exactly one computer and one connection farm. The Connection object holds information about the authentication tokens as well as the port number, in case the standard port 22 can't be used to connect through ssh. It is the user's responsibility to ensure, that he has permission to execute the program on the remote host



- Connection test successfull

Computer:	u-pl15
Connection Farm:	lab-ms
Name:	upl15 - jakub
Login:	faryj5am
Password:	●●●●●●●●●●
Port:	22
	Test connection
Description:	

Figure 2.3: Connection edit form

as well as that all the required libraries are installed there. A connection always belongs to one specific connection farm and thus to only one user group. Users that don't belong to that group won't have their experiments make use of this connection. There are cases, especially in university computer labs, where a user uses the same connection information on multiple machines. To avoid having to manually insert this information for every combination of computers and connection farms, a connection object can be cloned. The user can then repeatedly clone the connection, changing only the computer or the connection farm dropdowns select controls (fig. 2.3). In the edit dialog (either when editing an existing connection or creating a new one), it is possible to test the connection and discover potentially misspelled authentication information or connectivity problems. The connection data table can be filtered by computer or by connection farm.

Connection farms

A connection farm is merely a convenience grouping of multiple connections. It represents a pool of available connections and can be assigned to a specific experiment. Most of the time, the user will want to ensure that all the computers assigned to a connection farm transitively through a defined connection are equal in terms of hardware, especially if runtime is one of the parameters he want's to measure. A connection farm always belongs to one specific user group. The connections belonging to a connection farm can be displayed by clicking on the ”connections” link in the data table.

Projects

A project defines a group of programs that naturally belong together. For example, it could contain a group of different algorithms for solving some type of problem. In order to compare them, they will have to be run against the same input data. A project serves to link a problem type, a user group and some sets of available input data. By selecting some input sets on the project level, the user restricts all programs and experiments belonging to that project to only use a subset of those input sets. The project table provides links to programs belonging to a project as well as to the restricted input sets. A project has substantial data dependent on it: different versions of programs and all the experiments as well as their results. Deleting a project deletes recursively all this data as well. That’s why only an administrator is permitted to delete projects. A project is user group private.

Programs

A program represents one versioned executable file or archive of files. It actually only serves to group multiple versions of one executable file. The Programs page is then mostly only a crossroad when navigating from projects to experiments.

Application

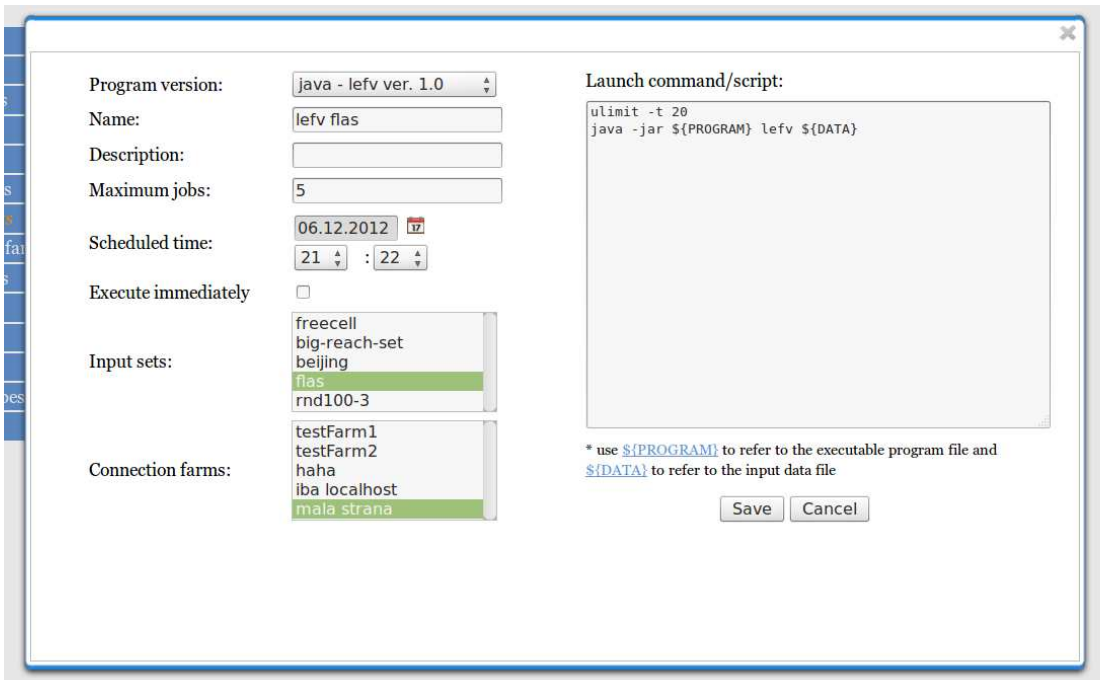
One concrete version of a program is called an application. On the Applications page, one can upload new versions of programs and edit/re-upload the existing ones. For every version, there are links to all experiments using it.

Input Sets

Every computation needs input data. The standard use case is to run one version of a program (representing one specific heuristic or algorithm) with a large number of different inputs, and then another program (another algorithm) again with the same input set. This way, the algorithms can be compared by numerous parameters. An input set is such a group of inputs, meant to be assigned to an experiment. A new input object can be created by uploading a zip archive of input files in the add/edit dialog. After an input set has been created, the user can manually go over individual input files in the set and add/remove them.

Input

An input is just one file, that can be passed as an argument to a program. It is not possible to upload or edit inputs individually on the Input page. They are always created by uploading them to an input set. Instead, they can be viewed by downloading the files one by one for every input. If the same file is uploaded multiple times in different input sets, it is recognized (by calculating the checksum) and the input sets then reference the same file.



Program version:	java - lefv ver. 1.0
Name:	lefv flas
Description:	
Maximum jobs:	5
Scheduled time:	06.12.2012 21 : 22
Execute immediately	☐
Input sets:	freecell big-reach-set beijing flas rnd100-3
Connection farms:	testFarm1 testFarm2 haha iba localhost mala strana

Launch command/script:

```
ulimit -t 20
java -jar ${PROGRAM} lefv ${DATA}
```

* use ${PROGRAM} to refer to the executable program file and ${DATA} to refer to the input data file

Figure 2.4: Experiment edit form

Experiments

Only after all the setup on the other pages has been done, an experiment can be created and scheduled. On the Experiments page, the displayed rows can be filtered by project, program or application. A new experiment can be scheduled either by clicking on the add button and creating one from scratch, or by cloning an existing experiment. Either way, the user ends up with the add/edit dialog, where he has to set some required values. Apart from the standard name and description fields, there are several mandatory parameters (fig. 2.4):

- **Maximum jobs.** Sets a limit to how many concurrent jobs must be running on the computers belonging to some of the experiment's connection farms to stop being considered as free for further scheduling. It is the primary means to load–balancing the experiment execution. The running jobs are being calculated globally, for all users and user groups. If the field stays empty, there will be no limit to the number of currently running jobs on one machine and all new jobs will be directly sent to the least loaded machine, no matter what the current number of jobs is.

- **Schedule time.** the exact time, when the experiment should start can be configured as well. This is practical, if the experiments need to be run at night on computers that serve as normal workstations during the day. If no delay is necessary, the user can check the "execute immediately" check box and the experiment will be scheduled for the first minute after submitting the form.

- **Input sets.** The input sets to run experiments against. The experiment will run the program once for every input from the union of all selected input sets.

- **Connection farms.** The selected connection farms determine what computers to run the experiment on. The application will try to parallelize calculations for individual inputs as much as possible. For every input, it will look for the currently least loaded computer (not connection) belonging to any of the selected farms, and run the computation there.

- **Launch command.** Usually, the executable file has to be executed on the remote machine with some of the inputs from the input set. In order for the application to be as generic as possible, the user has the option to define his own launching script/shell procedure. This script will be copied over to the remote host into a file called "startScript.sh" into one folder together with the executable file and the input data file and it will be executed like this: `sh startScript.sh &>output.txt` redirecting all standard and error output into the output file. Since this has to work for every input file, the user cannot use concrete file names in the script. That's why he can use variables ${PROGRAM} and ${DATA} to refer to the executable file and the input data file in the script. An example launch script might look for example like this:

  ```
  utime -t 30
  java -jar ${PROGRAM} berkmin ${DATA}
  ```

  That will run the given program for the supplied input with the selected heuristic "berkmin", but for seconds at most.

If the program is written in a non cross-platform language, the user can upload an archive with sources instead of the executable program. Then he can put all the commands necessary to unpack the archive and compile the program on the target system into the launch script.

Experiment status and results.

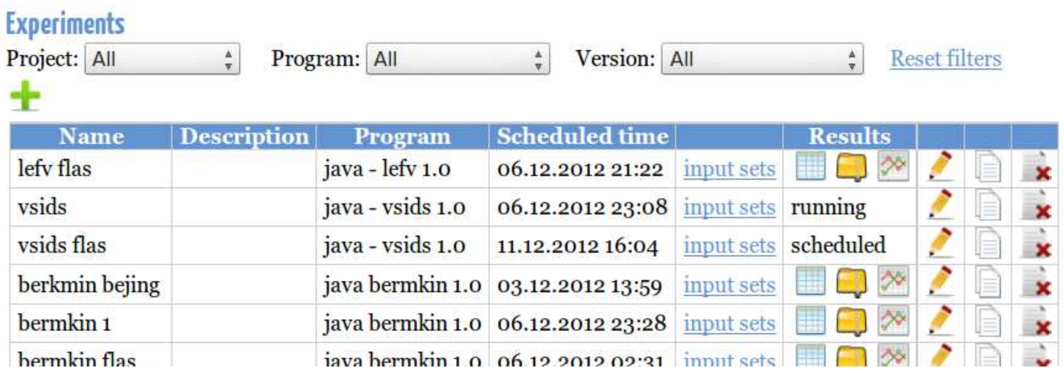
After the form is submitted a new experiment is added to the table and it's state (displayed in the "Results" column) is set to "scheduled". The state is refreshed every 10 seconds, so when the time comes for the experiment to start, the displayed state will switch to "running" (fig. 2.5). Once the experiment is finished, three new icons will be displayed in the column instead of the state information:

Download as CSV

A link that downloads a CSV (comma separated values) file containing the results for all inputs of the experiment.Every row represents one input file and every column one specific measured parameter

Download zip

Clicking this link triggers a download of an archive of files containing raw output for every input in the experiment. As was already mentioned, the files contain all standard as well as error output from the computations.



Experiments

Project: All Program: All Version: All Reset filters

Name	Description	Program	Scheduled time		Results			
lefv flas		java - lefv 1.0	06.12.2012 21:22	input sets				
vsids		java - vsids 1.0	06.12.2012 23:08	input sets	running			
vsids flas		java - vsids 1.0	11.12.2012 16:04	input sets	scheduled			
berkmin bejing		java bermkin 1.0	03.12.2012 13:59	input sets				
bermkin 1		java bermkin 1.0	06.12.2012 23:28	input sets				
bermkin flas		java bermkin 1.0	06.12.2012 02:31	input sets				

Figure 2.5: Experiment status

Create result charts

A link to the Charts page which preselects the given experiment in the list of experiments to compare.

Result format

In order for the application to be able to parse the result of an experiment, the output of the executed program has to contain two special lines: a line starting with ”`experimenter-results-head`” and then containing a semi-colon separated list of words defines the measured parameters of the experiment for example:

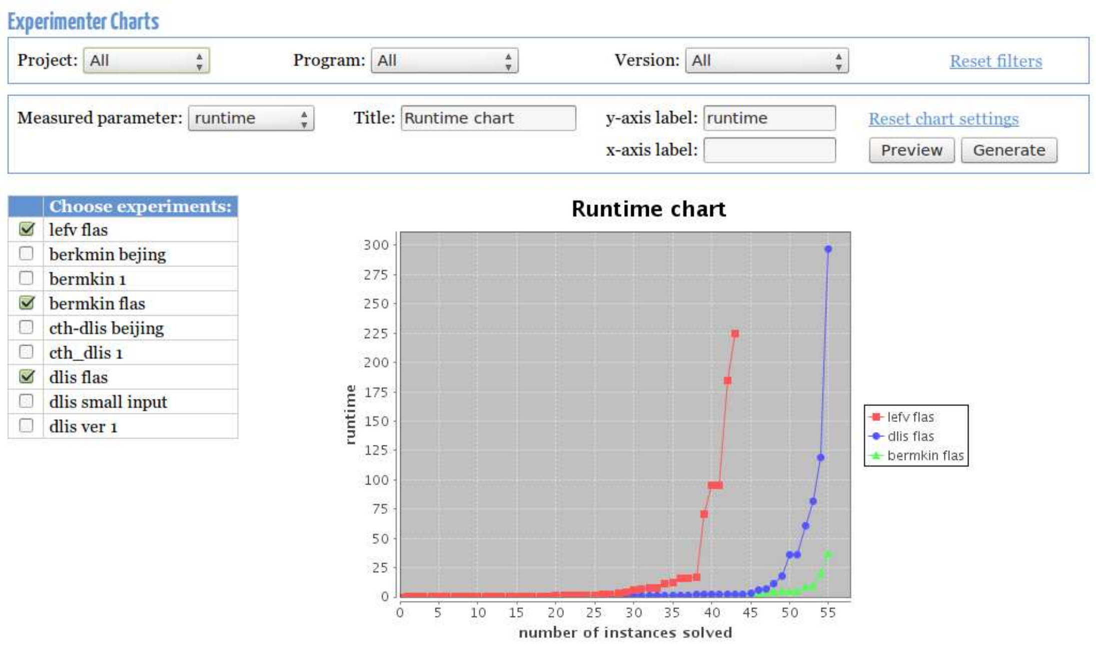
```
experimenter-results-head;param1;param2;param3
```

A line starting with ”`experimenter-results-data`” and then containing a semi-colon separated list of decimal numbers defines the values where value at the nth position in the results data line corresponds to the parameter at the same position in the results head line.

If the output doesn't contain these two lines, there will be no results stored in the database for that particular input.

Charts

The Charts page allows graphically displaying the results of the experiments for different measured parameters and comparing them with other experiments (fig. 2.6). The user can either enter the page through the menu, or follow a link in the ”Results” column on the Experiments page. In the later case, the selected experiment will be selected in the experiment list and the project filter will be set to the experiment's project. It is possible to filter the experiment list using two more filters: programs and applications. On the left hand, next to the menu a list of experiments is displayed. Every experiment has a checkbox, where the user can select whether to include that experiment's results int he chart. After selecting the experiments, the measured parameter has to be selected as well. The ”measured parameter” dropdown is populated dynamically with all available parameters for the given experiments. The user has to select the one to display.


Experimenter Charts

Project: All Program: All Version: All Reset filters

Measured parameter: runtime Title: Runtime chart y-axis label: runtime x-axis label: Reset chart settings

	Choose experiments:
✓	lefv flas
☐	berkmin bejing
☐	bermkin 1
✓	bermkin flas
☐	cth-dlis beijing
☐	cth_dlis 1
✓	dlis flas
☐	dlis small input
☐	dlis ver 1

Figure 2.6: Experiment status

He can optionally fill in the labels that will be used in the chart to annotate the axis or the main title.

The graph always represents the values of the measured parameter per input, where inputs are sorted by ascending parameter value on the x-axis. That means, that the curve is always nondecreasing.

By clicking the Preview button, a small version of the graph is displayed on the page, right below the chart settings. If the user is satisfied, he can click Generate to download a SVG image of the file.

3. Programming documentation

This chapter describes the technology stack, the architecture and the most important implementation details of the Experimenter application. The programming documentation assumes general knowledge of J2EE patterns and technologies as well as experience with the used (or similar) frameworks. It is not the purpose of this chapter to describe all packages and classes, which is done by the generated Javadoc documentation available on the attached CD, but rather to point out the important parts of the design and implementation.

3.1 Technology stack

Java

Java is a platform independent object oriented programming language and is practically a standard solution for client-server web applications.

Spring Framework

Spring is an open source inversion of control container for Java applications with a very high level of integration with other frameworks. In the Experimenter, Spring is used for life cycle management of back end Java beans, transaction management and dependency injection in the back end as well as in the front end.

Hibernate

Hibernate is an object-relational mapping framework, that connects the object model of the Java application with the relational model of the database.

Apache Wicket

Wicket is a model-view-controller web application framework. Wicket is component based, every page consists of a tree of components, that have their own markup and a model.

Apache Maven

Maven[6] is a tool for managing dependencies between libraries and automating the build process.

HyperSQL database

HyperSQL is a lightweight relational database, written in Java. It can be run in standalone server mode or as an in-memory database, which is useful for unit and integration testing.

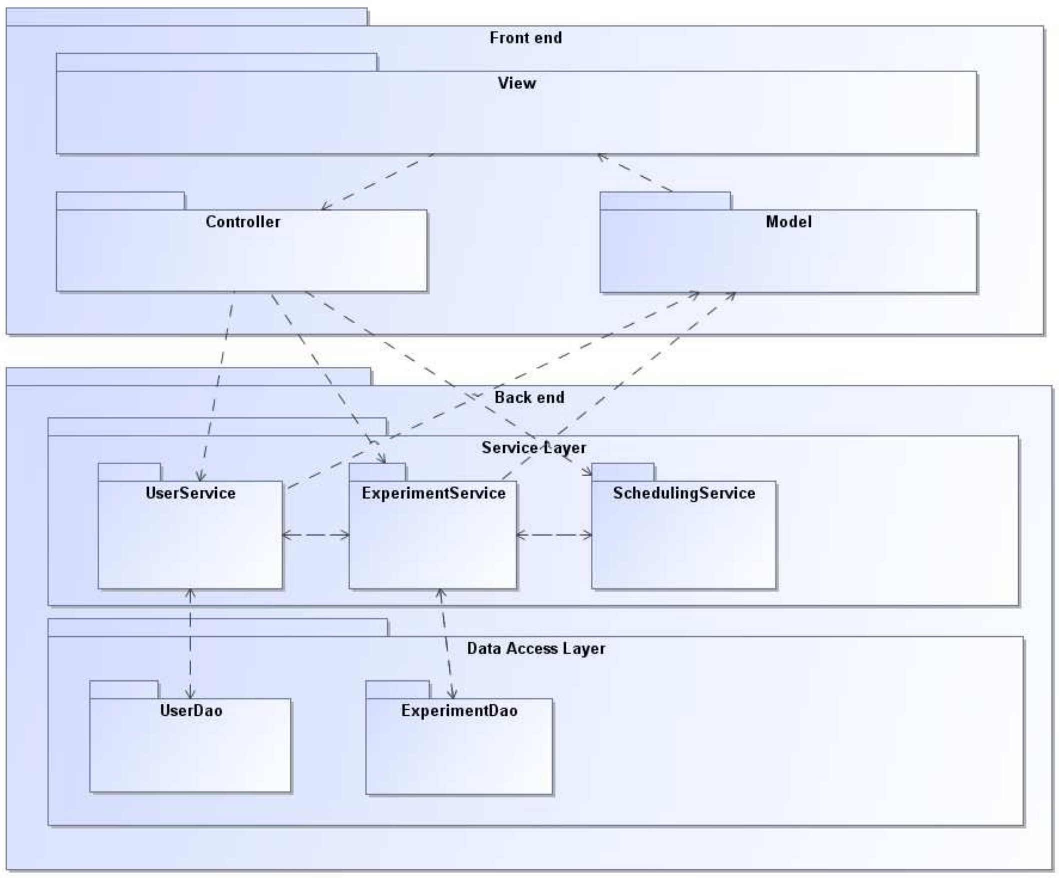


Figure 3.1: Experimenter - layered architecture

3.2 Architecture

The application is divided into two maven modules: the back end is called experimenter-repository and the front end is called experimenter-web and has a strict layered architecture (fig. 3.1). That means, that the communication between the database and the UI always passes through the service layer and the data access object layer (DAO). Although this sometimes leads to unnecessary code redundancy and a lot of "boilerplate" code, it allows easy tracking of the data flow or managing transactions and promotes loose coupling.

3.2.1 Back end

Data model

The data model practically maps the domain objects described in chapter1 to database tables. Tables `USER` and `USERGROUP` are in a many-to-many relation, whereas tables `PROGRAM` and `APPLICATION` are of course in a one-to-many relation, as one would guess by the relations of the objects on the UI. All domain object tables have artificial keys, since the business relations are not enough to uniquely identify an object. On top of the tables for business objects, there are additional tables for result data and the current experiment status. The data model is specified in the schema.sql file on the attached cd disk.

Entities

Entities are POJOs (simple Java objects, without any logic, containing only properties and their getter/setter methods) mapped by Hibernate to the underlying database tables

Data access layer

The data access layer is a layer of data access objects (short DAOs). Every domain object has it's own DAO, that provides a basic create-read-update-delete access to the entity. It also provides methods for calling individual SQL statements. The purpose of the DAO layer is to separate the upper layers from the database. This way, if any implementation details in the database or in the ORM change, the rest of the application will not be heavily impacted. This has been confirmed in the early stages of implementation, when the ORM solution was switched from SQL Processor to Hibernate. The implementation of the data access objects changed, but the interfaces stayed almost intact.

Service layer

The service layer provides an interface between the front end and the repository. It contains most of the business logic. While the front end serves mostly as a means to transform user actions into method calls to the back end, the logic of managing threads with experiments, load balancing, parsing results and generating charts and other output data is implemented in the services. Also, all database transactions start and end by entering and leaving a method in the service layer. If anything in a service call goes wrong, the whole transaction is rolled back, thus maintaining atomicity of all service operations.

3.2.2 Front end

The front end consist of only one layer - the Wicket application. Wicket is a pure **MVC** (Model - View - Controller) framework. MVC is a very popular design pattern for modern web applications. It divides the front end logically into three parts:

The **Model** provides data. Every time when data is requested from the model, the model knows how to fetch up-to-date data from the back end. An example of a model is for example the `AvailableExperiments` class. When asked for data, it calls the appropriate service method and always returns only those experiments, that belong the the current user's user groups.

The **View** provides a user interface for the user. It consists of "dumb" components, without business logic. The components in the View layer know only the model they ask for data. For example, the data table on the Experiments page knows, that when rendered, it should request the list of experiments from the `AvailableExperiments` model. The table knows how to display the experiments, but it knows nothing about the business constraints on the displayed data. Another function of the View is to propagate user actions to the Controller.

The **Controller** receives calls from the View about the actions the user has taken. It knows what to do with them and how to propagate them to the back end. It then notifies the Model that the data has changed, and that it needs to be reloaded. In the Experimenter, the Controller is implemented in form of behaviors (mostly Ajax behaviors) attached to the components. For example, an "onClick" ajax behavior attached to the "Delete" icon in a row of the table knows that it should call the appropriate method in the service layer to delete the given entity. By detaching the `AvailableExperiments` model and adding the data table component to the list of components to be re-rendered after the Ajax call it makes sure, that at the end of the request, the data table will be rendered correctly without the deleted row.

The chosen framework has a crucial impact on the structure and the logic of the whole View layer. Although it is not split into multiple sub-layers like the back end is, the Wicket concept imposes a logical separation of the user interface into UI components (pages, panels, tables, labels, etc) and their "models", which contain the logic of fetching the right data from the back end for their respective components.

3.3 Implementation

Context initialization

The web application is started by the web container (e.g. Apache Tomcat), which by reading the web.xml file knows, that it should start the Spring context loader. Everything else is taken care of by the Spring container.

IoC container

The application is initialized and run in the Spring inversion of control container. The container takes care of initializing the data source, the hibernate session factory, the transaction manager and all the singleton service and DAO beans and linking them all together. The dependency injection design pattern is used to provide dependencies between beans at runtime. All the beans and their dependencies are defined in the applicationContext.xml file. Dependency injection means, that the programmer doesn't have to construct the beans manually in one centralized place in the code, since they are created by the container and have all their dependencies provided. For more information on how dependency injection works, see [7]. Some configurable values (for example thread pool parameters, database authentication information etc) are specified in the experimenter.properties file. Spring will load them at startup.

Hibernate session

In order to be able to access the database, a Hibernate session has to be open. A session holds references to loaded persisted entities and their state and provides methods for executing queries in the database. Every time some information has to be fetched from the database, a session must be opened. It is important to choose a correct strategy for when to open and close the session. In our case, the Open Session In View (OSIV) strategy has been chosen. The session is opened and closed with every user request. This solution has many disadvantages, as it means, that the View layer may be working with entity proxies instead of real entities and some entity associations are lazily fetched only when rendering the page at the end of the request. This may lead to performance issues and is wrong also on the conceptual level: the front end layer becomes dependent on the lowest back end layer. On the other hand, this strategy is very simple as it is always clear when a session is open. The programmer doesn't have to take care of opening it manually and no complicated session factory has to be set up. Since performance problems are rather negligible in the anticipated usage of the Experimenter, we decided to prefer simplicity and thus maintainability of code over a performance-tuned and conceptually flawless design. The whole session management for user requests is taken care of by Spring by specifying the OpenSessionInView filter in the web.xml file.

Hibernate session for background threads

As was mentioned, the Open Session In View pattern closes the session with the end of the user request. However, background threads, that are actually executing the experiments need to communicate with the database as well. To allow them to open a session a Hibernate interceptor is used. Using Aspect Oriented Programming, ExperimentExecutor instances (which is the class responsible for launching an experiment) are advised with the interceptor which results in their method calls being wrapped with opening and closing a hibernate session.

Entities

All entities implement the interface Entity, which means, they all have an Integer id property. Mapped entities are: `User`, `UserGroup`, `Computer`, `Connection`, `ConnectionFarm`, `Project`, `Program`, `Application`, `Experiment`, `Input`, `InputSet`, `ProblemType` and `Result`. Mapping classes to tables is done via annotations. As the identity of a persisted entity is determined by an artificial id, the `equals()` and `hashcode()` methods have to be modified to reflect this logic.

Data access objects

The `BaseDao` interface defines basic CRUD operations with entities. These are implemented in the `AbstractBaseDaoImpl` class, which all other DAO classes extend and add their own methods as specified in their respective interfaces. For example, ExperimentDao declares methods for retrieving experiments for a given user or retrieving only those experiments, whose schedule date is still in the future.

Services

Service classes follow the same pattern as the DAO classes. Again there is the `EntityService` interface, that defines methods common for all services, like `findById`, `delete`, `findByExample` etc. These methods are implemented in a common parent class of all entity service classes. Again, individual services add methods specific for their entities. These methods either only delegate the call to the DAO, or call other services and perform more complex business operations.

The `SchedulingService` and the `RepositoryService` are exceptions, as they are not entity services.

SchedulingService

The `SchedulingService` is called every time an experiment is created or updated on the UI. It creates an `ExperimentExecutor` instance, and schedules it to be called at the right time. The scheduler implementation used is Quartz[10], mainly because of good integration with Spring. Spring provides a basic task scheduler as well, which might be sufficient for the current purpose of the application, however, in regard to possible future requirements, the flexibility of Quartz may be of use.

StorageService

Uploaded files and experiment results have to be stored somewhere. The possible solutions were to store binary data in the database, or directly on the filesystem. The later option was chosen for simplicity and better manual administration in case of database corruption or other state inconsistencies. On the other hand, storing data on the filesystem and referencing it in the database breaks transactionality, as the two repositories have to be synchronized. There are solutions for this problem: content repositories like for example Apache JackRabbit, that provide transactional storing on the filesystem, but these solutions were found to be too complex for the simple use case of the Experimenter application. That's why storing files in the filesystem has been implemented without the use of any external frameworks and possible inconsistencies between the database and the file storage have to be counted with. The location of the storage directories as well as the location of temporary folders to use can be configured in experimenter.properties

3.3.1 Experiment execution process

As was mentioned, when the user saves an experiment, it is added to the Quartz job store. Quartz then ensures, that it is executed at the right moment. The `ExperimentExecutor.execute()` method is invoked with the Hibernate interceptor taking care of opening a session. The process of executing the experiment is as follows:

- `ExperimentExecutor` loads all the necessary information about the experiment: what program to execute, what are the assigned input sets and connection farms.
- The experiment is marked as "running".
- It iterates over the union of all inputs from all input sets and tries to eagerly find the least loaded machine from the specified connection farms. If the least loaded machine has more jobs running on it, than is allowed by the experiment, the thread sleeps for a configurable amount of time and then tries it again.

- When a suitable computer is found a new `ExperimentJob` instance is be created with all the necessary information including the selected input and computer.
- The `ExperimentJob` is submitted into the thread pool (which is again configurable in the experimenter.properties file.) If the thread pool is full and the job is rejected, the process will sleep for a while (configurable) and try again.
- If the thread pool is not freed up in a certain time limit, the job will be discarded.
- After all jobs have been added to the thread pool, the `ExperimentExecutor` thread will be blocked on a `CountDownLatch` and wait for the jobs to finish, or the timeout to elapse.

Most of the parameters in this algorithm are configurable : thread pool size, queue size, maximum number of jobs per computer, the timeouts as well as the retry intervals. All these variables play a role in the system's performance and stability. If wrongly set up, more than necessary jobs might be discarded because of in unavailable resources, or the remote computers, that the experiments are to be run on, might consider the big number of connections as some sort of denial of service attack and refuse them.

ExperimentJob

The `ExperimentExecutor` creates instances of `ExperimentJob`, which are `Runnable` classes taking care of executing the program with a concrete input on a concrete computer. When started,

- `xperimentJob` retrieves the input file and the application file from the `StorageService`.
- Then it takes the launch script of the experiment and uploads it as a file together with the input and executable data to the remote computer.
- There it executes the script redirecting all output into a new file.
- Once the remote program finishes, the output file is downloaded
- If all the previous steps passed, the output file is handed over to the Result-Service, which takes care of parsing it, extracting results and then passing the file to the `StorageService` to be stored in the storage.

If the execution fails because of refused or dropped connections, `ExperimentJob` will repeatedly try to run it again after some time for a configurable number of tries.

All communication with the remote computer is done via the JCraft JSch library [8].

Mutlithreading

The concurrent execution of jobs means more concerns about thread safety. Since the application is not designed to run in a cluster, synchronization is possible on multiple levels, mainly in the database or Java. The second option has been chosen, for being more straightforward: the methods manipulating experiment execution status information or the number of currently running jobs per computer are marked as synchronized and the data they retrieve is not held in the hibernate session cache.

Storing results

After a job finishes and calls the `ResultService` to save the result, the `ResultParser.parseRes` method is invoked, the special lines in the output file are parsed and for every measured parameter of the experiment specified in the "`experimenter-results-head`" line, a new entry is saved in the `RESULT` table (uniquely identified by experiment id, input id and the parameter name.

Creating charts

The class `ChartUtil` is responsible for creating the charts on the fly upon user request. There are two different formats it can create the charts in:

- SVG file : high resolution chart, used for downloading the result chart.
- BufferedImage: small chart used for displaying the preview on the Charts page.

The library used is JFreeChart[9]. One of the pre-defined chart-types is used, with only a little extra configuration.

3.4 Front end implementation

Structure

The front end keeps the standard Wicket structuring of files. .java files are kept together in folders with their respective .html files.

The custom extension of Wicket's `WebApplication` registers the Spring injector with the component initialization listener to enable Spring dependency injection in the front end as well.

Every page extends `BasePage`, which adds an authentication check. If the user is not signed in, he can't access any other page than the login or logout pages. After the user is authenticated, he can access pages extending from `AbstractExperimenterPage`, which adds the menu to the left.

Authentication and Authorization

The user is authenticated by comparing a hash of the provided password with the password hash in the database. The `ExperimenterSession` extends `AuthenticatedWebSessio` and defines the two available user roles to work with. This is the native Wicket solution to authentication and authorization, no other libraries have been used for this purpose. Wicket allows to either annotate a class to mark it as visible only for certain roles, or the authorization definition can be added to an already created instance. The result will be, that the given component will not be part of the generated markup. This way for example, these two lines ensure, that the Users page and the User groups pages will only be available for an admin user:

```
add(authorizedLink("userPage", UserPage.class, Roles.ADMIN));
add(authorizedLink("userGroupPage", UserGroupPage.class, Roles.ADMIN));
```

Loadable Detachable Models

A loadable detachable model is a good practice in Wicket applications. It knows how to retrieve the data and remembers only the necessary information (the identifier) between the http requests. All models and components are detached on the end of the request, that's when the model object is set to null. When the object is being retrieved again in the next request, the model loads the data again and holds it for the time of the request. All models working with entities extend `EntityModel` which has special functionality, if the given entity is not yet persisted (for example it comes from th "Add" form.) A non persisted entity is characterized by having it's id null. In such a case, the `EntityModel` will not detach the model object, as it wouldn't know how to retrieve the entity again for the next request. Since the session is serialized and written to disk between requests, entities must be `Serializable`.

Data tables

All entity datatables on all pages extend the component called `DataTablePanel`. `DataTablePanel` implements the common functionality of all the data tables, like the add, edit and clone columns and opening the add/edit form dialog.

Forms

When creating or editing an entity, extensions of `EntityForm` are used. Each of them has it's own markup, since every form is different. Like every component in Wicket, the form has always a model associated with it. In the case of `EntityForm` it's always an EntityModel. Every input field in the form is associated with a property of the entity represented by the form. So when a form is submitted, it's model object is retrieved and can be directly saved by calling the `saveUpdate` method on the correct service.

Navigating between pages

When a page is accessed in wicket, it's Java component is instantiated. Every page can take `PageParameters` as constructor argument. The `PageParameters` object can be used to pass url parameters to the page. This is used, when we want to preselect a certain value in the filter in the target page. For example, the user clicks on the link "experiments" on the Applications page, the selected application id will be passed as page parameter to the called `ExperimentPage`. This is how the Experiments page knows, what filter to preselect.

Testing

JUnit[11] is used for running tests. The back end module is almost completely covered by unit and integration tests. There is a test framework prepared to write tests of the front end module as well. Due to the character of web application, where the front end changes significantly during development, it would be time consuming to maintain tests of the user interface components. Since the application is still in development, covering the front end with tests is planned for a later phase, where most of the layout will be fixed and confirmed to be convenient by a long time usage.

Conclusion

The goal of the thesis was to design and implement an experiment management system for executing and evaluating programs for solving computational problems. The provided implementation, a multi-layered web application, meets all the functional and usability requirements. The application is stable, tested and ready to be used.

The core technologies used are Apache Wicket, the Spring Framework, Hibernate and the Hyper SQL database. The application can be deployed to the Apache Tomcat web server. Communication with remote computers is implemented using the JSch library. JFreeChart is used for chart generation. The application is build with Maven.

Evaluation

Apart from fulfilling the functional requirements, the main qualities of the system are:

- Responsiveness. Thanks to the used lightweight technologies, the application feels very fast.
- Usability. The most frequent operations are easy to perform.
- Clean code. The code is well documented and for the most part adheres to code style best practices. The back end module is almost completely covered by unit and integration tests. There is a test framework prepared to write tests for the front end module as well.
- Effective load balancing. The load balancing measures seem to work well, evenly distributing the experiments on all available computers.

Areas of possible improvement

- The synchronization between the database and the filesystem repository is a week point. Using of a more complex, transactional solution like a content repository system should be considered.
- Scalability might be an issue, if the usage of the system grows beyond the intended scale. Synchronization is currently implemented on application level, not on database level, hence clustering would not be possible. Also, if the data amount grows a lot, the user interface will have to be changed to introduce paging through large data sets.
- The load balancing could be further improved by actually checking the remote system's real load factor, thus avoiding performance losses caused by other resource heavy applications running on the remote systems.

Bibliography

[1] Schildt, Herbert. *Java The Complete Reference*. 8th Edition. McGraw-Hill Companies,Incorporated, 2011. ISBN 00-704-3592-8.

[2] Brittain, Jason and Darwin, Ian F. *Tomcat: the definitive guide*. O'Reilly & Associates, Inc., 2003. ISBN 0-596-00318-8.

[3] Perrone, Paul J. and Chaganti, Krishna. *J2EE Developer's Handbook*. Indianapolis, Indiana: Sam's Publishing, 2003. ISBN 0-672-32348-6.

[4] Murphy, Brian D. *Spring Persistence with Hibernate*. 1th Edition. Apress, 2010. ISBN 1-430-22632-3.

[5] Hillenius, Eelco. *Wicket in Action*. 1th Edition. Manning Publications, 2008. ISBN 1-932-39498-2.

[6] Sonatype Company. *Maven: The definitive guide*. O'Reilly Media, 2008. IS-BN 0-596-51733-5.

[7] Prassana, Dhanji R. *Dependecy Injection*. Manning Publications Co, 2009. ISBN 978-1-933988-55-9.

[8] JSch Library, http://www.jcraft.com/jsch/

[9] JFreeChart, http://www.jfree.org/jfreechart/

[10] Cavaness, Chuck. *Quartz Job Scheduling Framework: Building Open Source Enterprise Applications*. 1th Edition. Prentice Hall Co, 2006. ISBN 0-131-88670-3.

[11] Massol, Vincent and Husted, Ted. *JUnit in Action*. Manning Publications Co, 2003. ISBN 1-930-11099-5.

A. Contents of the attached CD

The attached cd contains these files and folders:

- **experimenter** - the root of the project sources
- **javadoc** - the generated javadoc
- **db** - folder containig the database schema and the database jdbc driver

B. Installation guide

The installation guide references the resources on the attached cd.

Building from source

- edit the configuration file

  experimenter/experimenter-web/src/main/resources/experimenter.properties to match the required settings.

- run "mvn clean package" in the root experimenter directory

- generated war is located in experimenter/experimenter-web/target/experimenter-web-1.0.0.war

Starting the database

- create a folder for the database

- copy db/hsqldb-2.2.4.jar into the created folder

- copy db/mydb.script into the created folder

- change directory into the crated folder and execute:

  ```
  java -cp hsqldb-2.2.4.jar org.hsqldb.Server -database.0 file:mydb
  -dbname.0 xdb
  ```

Deploying to Tomcat

- copy the experimenter-web-1.0.0.war file to Tomcat's webapps folder as experimenter.war

- wait for it to be autodeployed, or restart the server

- the experimenter application

The Experimenter application should be running on Tomcats default port in the /experimenter context:
for example:
http://127.0.0.1:8080/experimenter

A user with administrator privileges, with the username: admin, password: admin is created by default.