University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Master Thesis

Performance optimization and security of EEG/ERP portal

Pilsen, 2012 Jindřich Pergler

Originál zadání.

Declaration

I hereby declare that this master thesis is completely my own work and that I used only the cited sources.

Pilsen, 16th May 2012

Jindřich Pergler

Abstract

The subject of this paper is the performance optimization and the security of the EEG/ERP Portal which serves as a repository for data and metadata from EEG research and which is developed at the Department of Computer Science and Engineering. The theoretical part of this thesis introduces common patterns of database model optimization, an overview of object-relational mapping together with the use of the Hibernate tool, and common issues in the database layer of the web applications. In the security part most common security threats are mentioned in overall. The practical part focuses mainly on the performance issues of the database layer of the EEG/ERP Portal. The results are then summarized and evaluated and recommendations for further improvements of the data layer and security level are given.

Table of contents

1 Introduction

At the Department of Computer Science and Engineering the EEG/ERP Portal is being developed. This web application on the Java platform serves as a repository for data from electroencephalography research and allows sharing of the data and related metadata among various research groups around the world.

The subject of this paper is divided into two parts: the performance, and the security of the portal. The goal of the first part is to examine the performance of the portal in the manner of accessing the database, saving and getting the neuroinformatic data and metadata. Some parts of the web application seem to evince poor performance. Preliminary analysis shows that the database tier of the application is not used in an optimal way. Therefore, the main focus is on the database layer represented by the object-relational mapping tool Hibernate together with underlying database model represented by tables with relations in Oracle database system. The goal of the second part is to perform tests related to the security of the web application and to compare the results to the security status of the portal from the previous year.

The theoretical part of this thesis introduces common patterns of database model optimization, an overview of object-relational mapping together with the use of the Hibernate tool, and common issues in the database layer of the web applications. In the security part most common security threats are mentioned in overall.

The practical part focuses mainly on the performance issues of the data layer of the EEG/ERP Portal. An amount of data for testing purposes is generated into database and the investigation of business logic and data handling logic according to particular use cases is carried out through the web application. The inefficient patterns are described, the new ones are introduced with respect to pursued parameters and an explanation is added to each case. The results are then summarized and evaluated. In the security part some basic tests are performed and recommendations for further adjustments are given.

2 Theoretical focus

2.1 EEG/ERP Portal

The EEG/ERP Portal is a web application designed for storing and sharing the data from electroencephalographic (EEG) research. Significant part of the research focuses on event-related potentials (ERP) which are measured with EEG. An event-related potential is specific measured brain response that is the direct result of a specific sensory, cognitive, or motor event. [1] Data files of various sizes, types and formats are the results of the experiments. A scenario describes the procedure of the experiment. A data file of various types can be attached to the scenario. Additional information about the experiment, scenario, involved people, or included data files are called metadata.

Figure 2-1: The homepage of the EEG/ERP Portal

The portal serves as a repository for the experiments, scenarios, data files and metadata and allows sharing of saved data among the research groups. Access to the system is restricted by using login credentials and authentication levels are

implemented throughout the application. The main user roles are global administrator and standard user. The global administrator has full access to all parts of the system. The standard user can only view the information marked as public. Other permissions are dependent on the membership of the user within the research groups.

2.1.1 Technologies

The application is developed in Java programming language. Several frameworks are used for the features of the portal. Spring MVC framework is the main core of the application. MVC stands for Model-View-Controller which is a design pattern for the three-tiered applications. The Model is represented by the domain object or data structure, mostly on the database basis. The View is represented by the templates and defines the user interface. The Controller forms the application logic; it operates with data from Model and passes them to the View (see Figure 2-2).

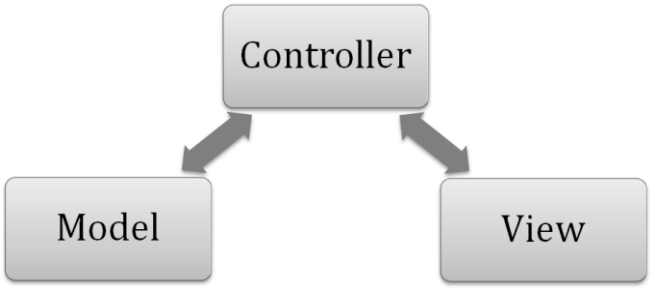


Figure 2-2: Schema of MVC architecture

The Model in the EEG/ERP Portal is represented by the Hibernate framework which connects to the Oracle database. Hibernate is an Object-Relational Mapping (ORM) tool which means that it works with data in the database and transforms (maps) them to the objects in Java. These are called POJOs (Plain Old Java Objects) in the text. The main focus of this work is on working with the Hibernate tool. The Hibernate is further discussed in Section 2.3.

Spring Security framework is used to manage the authentication and authorization part of the web application. Spring Social framework enables logging of the users into the application using Facebook or LinkedIn credentials.

2.2 Database design

Data in relational databases are represented by tables which are interconnected by relations. For particular situation the data model can be designed in many ways. The

database design lifecycle is a way of systematic approach to design a data model in database (see Figure 2-3).

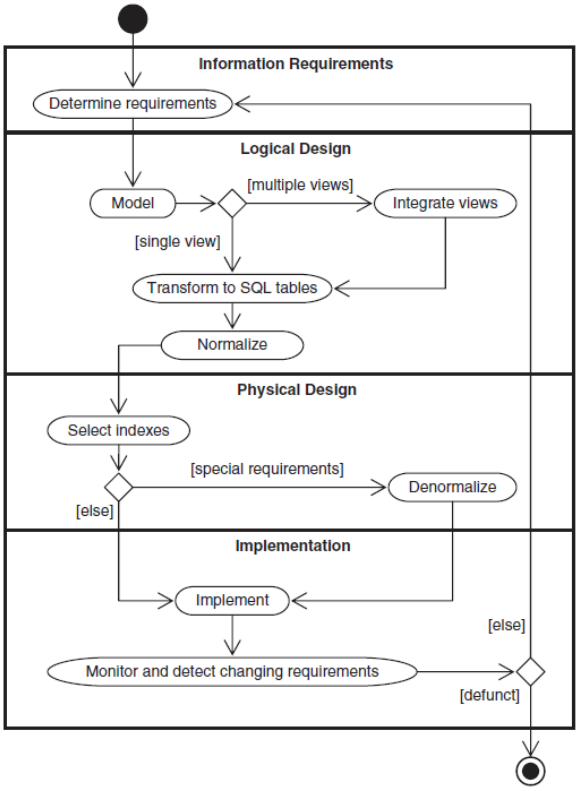


Figure 2-3: The database design lifecycle [2]

The design starts by defining the requirements for the data to be modelled. On the requirements basis the logical model is designed and that is transformed to SQL tables with relations represented by foreign keys and decomposition tables. The normalization of the model is then recommended to get the correct representation of the relations. Indexes are applied to specific columns according to requirement and use case. If special requirements are discovered the denormalization takes place and controlled data redundancy is introduced into model. At the end of the cycle the database is implemented and monitored. When additional requirements appear the lifecycle repeats. [2]

While the logical design is out of the scope of this work the physical design parts are discussed further in following sections.

2.2.1 Normalization

When the relational database model is designed the main objective is to create an accurate representation of the data, its relationships, and constraints. The relations can be represented in many ways for the same data. The technique that can help with representation of the data is called *normalization*. Normalization is process of series of tests on the relations to determine whether or not it satisfies the definition of a particular normal form. [3] Thoughtful design of a conceptual model mostly results in a database that is either already normalized or can be easily normalized with minor changes. [2] The aim of the normalization is to reduce redundant data and thereby reduce the file storage space required.

Anomalies in manipulating with data are often an issue when data redundancy is present in the database. An update or insertion of duplicated data column can easily lead to a data inconsistency. Moreover, there might be a problem with the redundant data which are dependent on the primary key of other data when inserting or deleting. More on this can be found in [3].

Normalization is often performed as a series of steps. The data model is tested on first normal form and with each step the level of checked form can be increased. With increasing level the restrictions are tighter (illustrated in Figure 2-4). To avoid the mentioned anomalies it is recommended to proceed to at least third normal form. The definitions of normal forms are explained on relations which mostly correspond with tables in database.

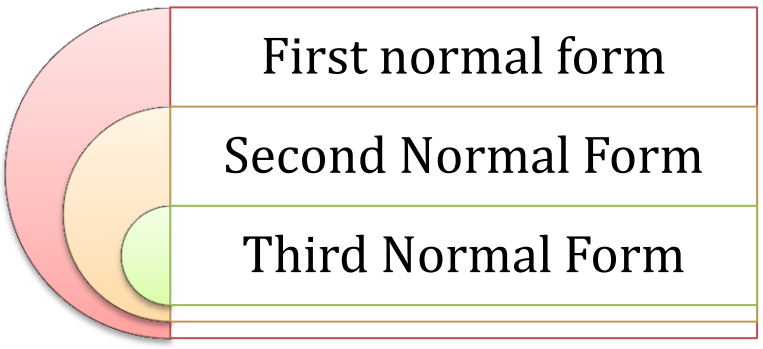


Figure 2-4: Normal forms

First normal form is a relation in which the intersection of each row and column contains one and only one value.

Second normal form is a relation that is in first normal form and every non-primary-key attribute is fully functionally dependent on the primary key. Full functional dependency indicates that if A and B are attributes of a relation, B is fully functionally dependent on A if B is functionally dependent on A but not on any proper subset of A.

Third normal form is a relation that is in first and second normal form in which no non-primary-key attribute is transitively dependent on the primary key. Transitive dependency is a condition where A, B, and C are attributes of a relation such that if A → B and B → C, then C is transitively dependent on A via B (provided that A is not functionally dependent on B or C).

2.2.2 Denormalization

The normalization results in clear data model with minimum data redundancy. The minimized redundancy saves the storage space and thereby mostly increases the performance on particular tables. However, when the related data are accessed the data have to be joined from several tables. When some data are accessed frequently it may cause performance decrease. Then the denormalization might take place.

Denormalization is a process for reducing the degree of normalization to improve query processing performance. However, it is recommended to normalize the model first before denormalizing. The difference between denormalized and unnormalized model is that the denormalized model had been normalized and then the degree of normalization was deliberately reduced while unnormalized model has not been normalized at all and the data redundancy is not under control.

The improvement of a query performance is accomplished by reducing the number of physical tables and reducing the number of actual joins necessary to derive the answer to a query. However, denormalization should be considered only when performance is an issue and the analysis has been made. Consequently, denormalization should be deployed only when performance issues indicate that it is needed. [3] [4]

Four strategies for denormalizing are most prevalent according to [4]:

- Collapsing tables – two entities with a one-to-one or many-to-many relationship
- Splitting tables – horizontal or vertical splitting
- Adding redundant columns – reference data
- Derived attributes – summary, total, balance

2.2.3 Indexes

Database index is a data structure that allows the DBMS to locate particular records in a file more quickly and thereby speed response to user queries. [3]

Data in the database are stored in data files which contain the logical records. Index records are stored in index files. An index structure is associated with a particular search key and contains records consisting of the key value and the address of the logical record in the file containing the key value.

There are various types of index. The main ones are following:

- *Primary index* – the data file is sequentially ordered; the indexing field is built on the ordering key field, which has a unique value for each record.
- *Clustering index* – the data file is sequentially ordered on a non-key field; the indexing field is built on this non-key field.
- *Secondary index* – an index which is built on a non-ordering field of the data file.

A file can have at most one primary index or one clustering index, and in addition can have several secondary indexes.

The setup of indexes on tables needs to be properly discussed for each case as the indexes may or may not be effective and using indexes everywhere is definitely counterproductive. However, index should be set up on primary keys and foreign keys.

Primary keys are unique values and single row is commonly selected using primary key column so the index on primary key column is essential. According to the primary index mentioned above the data file is ordered by the ordering primary key.

Index on foreign key is definitely useful as single rows or small part of the whole table is selected using foreign key. Benefits to indexing foreign key columns are following [5]:

- Better join performance – SQL server can more effectively find the rows to join to when tables are joined on primary/foreign key relationships.
- Better performance on maintaining the relationship on a foreign key – whether the foreign key relationship is defined with NO ACTION or CASCADE (on update/delete), the referencing rows must be found to restrict the action or update the referencing rows as well. In both cases an index on the foreign key column helps finding the referencing rows.

2.2.4 Other database tuning strategies

The discussed methods are the first steps to improve performance. When the traffic between the database and the application becomes more demanding there are other strategies for further improvement of the performance. [6]

First step should be **revising the application logic** and especially data layer of the application. The denormalization and use of the indexes is interconnected with application logic and depends on which data is the application working with. Furthermore, the **SQL query optimization** should take place as a one of the starting points.

Then the **database layer** is recommended to be tuned. That includes maximizing the concurrency by optimizing the need for locks, latches, buffers, and other resources in the Oracle code layer.

Next step is **optimizing the Oracle memory** and thereby reduce the resulting physical IO. That includes tuning of the buffer cache, work with the data block, shared memory caches, and sorting and hash memory.

After previous steps are performed the **physical disk layer optimization** takes place. In this part the aim is to configure the IO subsystem to provide adequate IO bandwidth and to evenly distribute the resulting load.

2.3 Data tier of application

2.3.1 Hibernate

Hibernate is an object-relational mapping library for the Java language. The main task of this library is to map Java classes to database tables. Hibernate provides persistence for Plain Old Java Objects (POJOs). Mapping can be defined by using XML files or by Java Annotations. In the first case the POJOs are designed and the mapping is in standalone XML file. In the second case the annotations are added directly into POJOs file so the mapping and object definition are at the same place.

Hibernate offers mapping for various kinds of relations including one-to-one, one-to-many, many-to-one and many-to-many types. The related objects are represented by class of the related object or by collection of such objects. Hibernate offers many

methods for loading and storing the POJOs and related collections. Selection of the method for particular case can have significant impact on performance of the application.

While Hibernate can be used as a standalone library the Spring MVC framework offers great level of integration for this framework. A common approach is to design Data Access Objects (DAOs) which provide methods for persisting the POJOs.

For retrieving the POJOs from database Hibernate provides two approaches – the Criteria and the Hibernate Query Language (HQL). In both cases the result is an internally assembled SQL query. On its behalf the data are loaded from database. The Criteria represents a programmatic way of specifying the parameters for getting the data. It uses specific Java classes and enumerations and is not readable very well. The HQL is a language similar to SQL. The main difference is that the selections are above the mapped POJOs instead of tables in the database.

2.3.2 Hibernate fetching strategies

Hibernate uses *fetching strategies* to retrieve associated objects of the queried entity. These strategies can be defined in the mapping configuration or can be overridden in particular HQL or Criteria query. Thereby, the performance can be significantly affected. Following strategies are available [7]:

- *Join fetching* – the associated collection is retrieved in the same `SELECT` using a `JOIN` operation.
- *Select fetching* – the associated collection is retrieved in additional `SELECT`. If the lazy loading is not explicitly disabled the second select is executed only when the collection is accessed.
- *Subselect fetching* – the associated collection is retrieved in additional `SELECT` for all entities retrieved in the previous query. If the lazy loading is not explicitly disabled the second select is executed only when the collection is accessed.
- *Batch fetching* – the associated collections are retrieved in a batch; list of primary or foreign keys is specified in the select. This is an optimization strategy for select fetching.

Fetching strategies are divided according to the situation when the fetching occurs:

- *Immediate fetching* – a collection is fetched immediately when the base entity is accessed.
- *Lazy collection fetching* – a collection is fetched when the collection is accessed. This is the default settings.
- *Extra-lazy collection fetching* – a whole collection is not fetched unless absolutely needed. For particular operations the optimized query is used instead of fetching the collection.
- *Proxy fetching* – a single-valued association is fetched when a method other than the identifier getter is used.
- *No-proxy fetching* – a single-valued association is fetched when the instance variable is accessed.
- *Lazy attribute fetching* – an attribute is fetched when the instance variable is accessed.

2.3.3 Common issues

Common issue is not using the pagination on the pages which display lists of some entities. This is not a problem while the number of entities is low. Together with the increasing number of entities the transferred data gets bigger and it can become a performance issue. Moreover, it is mostly a user interface issue as well.

Another common approach is to load all columns from the database even when not all data are necessary. In SQL the pattern `SELECT * FROM table` is widely used. It is mostly not an issue until the number of columns in table gets bigger. However, good application design includes effective use of the data.

2.3.4 Common Hibernate issues

2.3.4.1 Lazy loading

As mentioned earlier, lazy loading is a fetching strategy which fetches the associated collection when the collection is accessed instead of immediate fetching with owner object. It is the default behaviour of Hibernate. Disabling the lazy loading on all interconnected entities would cause the load of the whole database even when single entity was queried initially. That is one point which should be carefully considered.

Better approach is to override the lazy loading in particular HQL query. That can be done by using `join fetch` keywords on the specified collection. It causes the collection to be immediately fetched with the owner entity.

2.3.4.2 N+1 Problem

One of the most frequent performance issues related to lazy loading when using Hibernate is the *N+1 Problem*. It is a common pattern when we want to load list of items from database with associated data in corresponding object. A standalone query is created for each single object in the list and these queries are individually sent to database. That is one query for getting the list and then N queries for each item in list. If we want to work with more associated data, we can get additional N queries for another collection. This approach is extremely inefficient as the system can generate hundreds of queries for getting list of items.

The common solution is to specify `join fetch` on the collection in HQL query.

2.3.4.3 Limiting number of rows in memory

Another common issue is related to paging or any other task in which the number of items retrieved from database is limited to a certain number and collection associated with the retrieved entities needs to be fetched as well. In that case all items are loaded from database and the filtering and limiting is then carried out in memory. This approach will be explained on an example.

Consider having a Person object mapped to a table PERSON with columns PERSON_ID, NAME, AGE. A Person can have multiple cars, so the object has a collection of Car objects. Car is mapped to a table CAR with columns CAR_ID, OWNER_ID, MODEL. If we want to get the list of Person objects with Cars for each Person, we can use following HQL query:

```
from Person p left join fetch p.cars
```

That will be translated into following SQL query, on whose behalf the data are retrieved:

```
SELECT P.PERSON_ID, P.NAME, P.AGE, C.CAR_ID, C.OWNER_ID, C.MODEL FROM PERSON
P LEFT JOIN CAR C ON (P.PERSON_ID = C.OWNER_ID)
```

In a result of this query multiple rows can be loaded for particular PERSON_ID if such person has multiple cars as the rows are result of the Cartesian product because of join

operation. Hibernate then processes such data and distributes them to according objects.

If for instance the list of ten objects with collections needs to be loaded, it can't be predicted how many rows will be in the result of underlying SQL query. Therefore, the limit of retrieved rows can't be applied in SQL query and all items needs to be loaded so the limit can be applied after processing the data into objects. Thus, setting the limit in HQL does not need to take effect on the size of loaded data. When this situation comes Hibernate shows warning: "WARN: firstResult/maxResults specified with collection fetch; applying in memory!"

2.3.4.4 Getting the size of collection

In some situation the size of collection is needed to be used in the code. With standard setting of the fetching the whole collection is fetched to be able to evaluate the `length()` method on the collection. If the data in collection is not used the fetching is redundant.

The possible solution is to define the fetching strategy of the collection on extra-lazy fetching. When the `length()` method is called while the collection is not initialized only the appropriate query for getting the required information is generated. This approach can be used also on other methods on the collections, such as `isEmpty()`, or `contains(…)`.

2.3.4.5 Loading of all attributes of the entity

This issue is the extension of getting all the columns in SQL for ORM tools. If entity with large amount of columns is mapped all the mapped columns are fetched when loading the entity object. Subsequently, this can be a performance issue when the data get large.

In Hibernate the solution is not simple. The object can be loaded with subset of attributes according to the constructor using `select new ClassName(…)` in HQL query. Another option is using `select new map(…)` in HQL which returns a list of `Map` objects so the values can be accessed by names. Also, simple `select column1, column2 from ClassName` can be used. Then the values need to be accessed using the number of the column in sequence. [7]

A special behaviour is used in case of attributes mapped to LOB columns. The data are loaded lazily as the data itself are accessed. That avoids loading of huge bulk of data when not obvious the data are really needed.

2.4 Security of web applications

There are many potential security issues in web applications. The most prevalent threats according to OWASP are briefly presented in this section. The abbreviated definitions are adopted from [8].

2.4.1 Injection

Injection flaws, such as SQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing unauthorized data.

2.4.2 Cross-Site Scripting (XSS)

XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation and escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

2.4.3 Broken Authentication and Session Management

Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or exploit other implementation flaws to assume other users' identities.

2.4.4 Insecure Direct Object References

A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.

2.4.5 Cross-Site Request Forgery (CSRF)

A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included

authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

2.4.6 Security Misconfiguration

Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. All these settings should be defined, implemented, and maintained as many are not shipped with secure defaults. This includes keeping all software up to date, including all code libraries used by the application.

2.4.7 Insecure Cryptographic Storage

Many web applications do not properly protect sensitive data, such as credit cards, SSNs, and authentication credentials, with appropriate encryption or hashing. Attackers may steal or modify such weakly protected data to conduct identity theft, credit card fraud, or other crimes.

2.4.8 Failure to Restrict URL Access

Many web applications check URL access rights before rendering protected links and buttons. However, applications need to perform similar access control checks each time these pages are accessed, or attackers will be able to forge URLs to access these hidden pages anyway.

2.4.9 Insufficient Transport Layer Protection

Applications frequently fail to authenticate, encrypt, and protect the confidentiality and integrity of sensitive network traffic. When they do, they sometimes support weak algorithms, use expired or invalid certificates, or do not use them correctly.

2.4.10 Unvalidated Redirects and Forwards

Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

3 Database design

The database model behind the web application contains 60 tables which are connected by relations. The tables can be divided into several categories:

- *core tables*, which have many relations to other tables and are most prone to performance issues; these are `PERSON`, `EXPERIMENT`, `SCENARIO`, `RESEARCH_GROUP`, `ARTICLE`, and `ARTICLES_COMMENTS`;
- *tables with information and relations*, which are similar to core tables, also contain several columns and relations, but these are accessed not so often and therefore are not of a big performance concern; for instance `HISTORY` or `RESERVATION`;
- *data lists*, which contain mostly parameters used in other tables and mostly contain row identifier and information of title or description character; for instance `HARDWARE`, `FILE_METADATA_PARAM_DEF`, or `DISEASE`;
- *relation only tables*, which are the decomposition tables of M:N relations and contain only two columns as a primary key;
- *relation tables with attributes*, which contain mostly two columns as a primary key and a column with additional data; for instance `RESEARCH_GROUP_MEMBERSHIP`;
- *XML definition tables for scenarios*, which have specific characteristics related to XML scheme definition and therefore they are not part of the evaluation in this work.

3.1 Normalization check

The database model was checked for the level of normal form of individual tables. All tables comply with Third normal form. Only the table PERSON can be discussed because of USERNAME column.

Users log into the application via user name so there might me temptation to designate the password, name and other fields to be functionally dependent on user name value

instead of person identifier. The password, name and other fields would be then transitively dependent on primary key which would violate Third normal form and pass the Second normal form only. The correct table design would mean that the fields dependent on username would be moved to a separated table and connected via user name. Or the user name, password, name and other values can be declared as dependent on primary key and user name can be considered as a value of person which enables user to log into the application.

As this is only formal issue it does not need to be further solved. All other tables comply with third normal form which is good starting point for operating with data in database model.

Table 3-1: Table normalization check

Table name	Number of columns	Normal form level
ANALYSIS	5	3NF
ARTEFACT	3	3NF
ARTEFACT_REMOVING_METHOD	4	3NF
ARTICLES	6	3NF
ARTICLES_COMMENTS	6	3NF
DATA_FILE	7	3NF
DIGITIZATION	4	3NF
DISEASE	3	3NF
EDUCATION_LEVEL	3	3NF
ELECTRODE_CONF	4	3NF
ELECTRODE_FIX	4	3NF
ELECTRODE_LOCATION	7	3NF
ELECTRODE_SYSTEM	4	3NF
ELECTRODE_TYPE	4	3NF
EXPERIMENT	15	3NF
EXPERIMENT_OPT_PARAM_DEF	4	3NF
EXPERIMENT_OPT_PARAM_VAL	3 (2PK[1])	3NF
FILE_METADATA_PARAM_DEF	4	3NF
FILE_METADATA_PARAM_VAL	3 (2PK[1])	3NF
GROUP_PERMISSION_REQUEST	5	3NF
HARDWARE	5	3NF
HISTORY	6	3NF
PERSON	18	2NF or 3NF

[1] Two columns form the primary key.
[2] As mentioned earlier, first request after initializing the web application shows limited amount of articles, next requests don't have limit for displayed articles.
[3] As the administrator user can view any article the paging is carried out on all articles in

PERSON_OPT_PARAM_DEF	4	3NF
PERSON_OPT_PARAM_VAL	3	3NF
PHARMACEUTICAL	3	3NF
PROJECT_TYPE	3	3NF
RESEARCH_GROUP	4	3NF
RESEARCH_GROUP_MEMBERSHIP	3	3NF
RESERVATION	6	3NF
SCENARIO	9	3NF
SERVICE_RESULT	6	3NF
SOFTWARE	4	3NF
STIMULUS	2	3NF
STIMULUS_REL	3 (2PK[1])	3NF
STIMULUS_TYPE	3	3NF
SUBJECT_GROUP	3	3NF
WEATHER	4	3NF

Table 3-2: Relation only tables

Table name	Number of columns
ARTEFACT_REMOVING_METHODS_REL	2PK
ARTICLES_GROUP_SUBSCRIBTIONS	2PK
ARTICLES_SUBSCRIBTIONS	2PK
COEXPERIMENTER_REL	2PK
DISEASE_REL	2PK
ELECTRODE_LOCATION_REL	2PK
EXPERIMENT_OPT_PARAM_GROUP_REL	2PK
FILE_METADATA_PARAM_GROUP_REL	2PK
HARDWARE_GROUP_REL	2PK
HARDWARE_USAGE_REL	2PK
PERSON_OPT_PARAM_GROUP_REL	2PK
PHARMACEUTICAL_REL	2PK
PROJECT_TYPE_REL	2PK
SOFTWARE_REL	2PK
WEATHER_GROUP_REL	2PK

3.2 Denormalization

The data model is normalized at third normal form. That is perfect starting point for great data representation and thereby the data management anomalies are avoided. If the performance issues occur the denormalization can take place. However, to advance to such task the specific situation needs to be properly tested and the designed database upgrade needs to be verified for the performance improvement. Therefore,

the denormalization is discussed in the particular sections where this approach might be beneficial.

3.3 Indexes

According to the recommendations the indexes are used on primary and foreign keys. Further use of indexes needs to be preceded by analysis of the particular situation with regard to used query for fetching of the data from database. Potential applying of indexes is discussed in particular cases in Section 5.

3.4 Data generating

A small amount of more or less useful data has been inserted into the developer schema of the database during the development process. In some parts a slow response time can be noticed even with quite small data load, especially when developing on the computer which connects to the database via Internet instead of local school network. To examine the application more data are needed in the database than it is currently present in the development database. Therefore, for the testing purposes separated schema was created and new data were generated into this schema.

Several software generators for the Oracle database have been tried out. Most of them are licenced and offer a trial version which is restricted usually in the number of generated rows. Some of the programs offer great functionality. However, the restriction for the number of generated rows (mostly about 50 rows) is enough for not being able to use these tools in freeware, not mentioning that paid licence is not affordable. For instance EMS Data Generator or Datanamic Data Generator for Oracle belongs among such programs.

Therefore, the freely distributed library and tool called DbMonster is used. The tool is programmed in Java language and source code is available. It contains several types of generators for the columns and with the source code additional generators can be created. In the database model there are specific cases when the bundled generators are not sufficient so the DbMonsterPlus tool was created as an extension of the DbMonster source codes with own generators added.

The generating of the data is defined in XML file. Tables are set and generators are associated for each column. The program is able to solve the dependencies defined by foreign key associations and thereby is able to fill in the tables in correct order.

Generated data were set with regard to individual columns. The generated data consist of randomly generated word concatenations in case of strings. Dates are generated randomly within specified range as well as numbers. The quality of generated data is not that big when compared to real data. However, it is not possible to insert hundreds or thousands of real data samples manually into application. For the testing purposes of data layer of the application is the quality of generated data satisfying. In special cases the generated data were manually altered to better suit the tested phenomenon.

4 Performance of the portal

Many changes in the database and the program code were made since the application was founded. In some situations the slow response time could be noticed even without specialized tools. That has led to the performance testing of the developed portal. For this purpose the profiling tool for Hibernate called Hibernate Profiler is used.

4.1 Hibernate Profiler

Hibernate Profiler is a tool for monitoring and profiling the queries and sessions created by Hibernate. It provides several views on gathered data. The overview of user interface is on Figure 2-1.

In the top left corner the Hibernate sessions are monitored. For each session the processing time, the number of executed queries, and requested URL are displayed. For selected session the individual SQL statements are shown in the top right column. Short version of the query is displayed in the list together with returned row count and duration of the query processing. On the second tab the entities loaded during the session can be studied. The third tab summarizes the session usage information. The bottom right corner displays full query generated by Hibernate, alert notifications for particular query and a stack trace of the Java classes so the origin of the query call can be easily found. The bottom left corner displays statistics about the session factory use.

With the Hibernate executable package the JAR package is included. Hibernate Profiler is available for use with standalone Hibernate in application as well as with Hibernate coupled with Spring framework. For the Profiler to work three simple steps need to be carried out [9]:

1. The JAR package has to be included in the project.
2. A listener has to be set into web.xml.
3. A bean has to be set up in bean configuration file for Spring MVC.

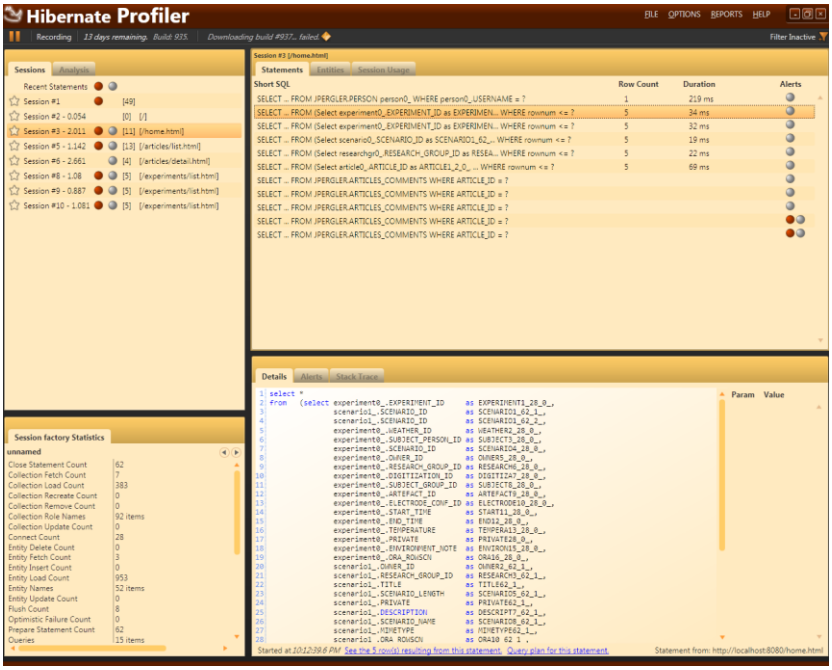


Figure 4-1: Preview of the Hibernate Profiler tool

When running the project on localhost the Hibernate Profiler is then able to get the information on the queries and the session and session factory statistics from Hibernate.

Hibernate Profiler is licenced by a Hibernating Rhinos company and a proper licence file is needed to run the program. The company offers an evaluating licence for 30 days on demand. After proposing a request a company representative granted a free licence for 90 days for the purpose of this work. I hereby would like to thank to Mr Oren Eini from Hibernating Rhinos for providing the Hibernate Profiler extended licence.

4.2 Tracked parameters

Two main indicators were designated as the tracked parameters. The first parameter is **the processing time**. Important is the processing time of the web request. In many cases the query processing duration is followed as well. The second parameter is the **number of generated queries** for a session.

Moreover, the additional auxiliary parameters are tracked for better understanding of what is happening in the inner logic of Hibernate. The Profiler provides alerts for the most common issues which can help to find the performance inefficiencies. The number of fetched entities hints the counts of objects loaded and displayed into webpage.

While the tracked parameters show the trend of performance improvement the values have to be discussed together with the particular situation and the code alterations rather than to be understood as absolute values without context.

Furthermore, the application was tested at the computer which connects to database using Internet connection. The parameters of the used connection are on high standard. The average download speed oscillates around 24 Mbps, the average upload speed fluctuates around 1.5 Mbps and the average response time is around 16 milliseconds both to the Internet and to the database server. The response time influences the processing time of queries and sessions. Both the testing and production servers access the database server via local network and can use the bandwidth of 100 Mbps and the response time shorter than a millisecond. Therefore, the response time reflects in longer processing time on the tested computer which is actually an advantage in a way. The performance inefficiencies of Hibernate can be more easily revealed and improved.

The measured times in Section 5 are the result of minimum 10 repeats of the discussed operation so the value is significant for the results.

4.3 Paging

A paging feature is missing throughout the web application. The Spring MVC framework does not provide any bundled classes for easy use of the paging. Therefore the auxiliary class `Paginator` was created as a helper for implementing the paging on selected pages. It takes `itemCount`, `itemsPerPage`, and optional `baseUrl` as constructor parameters. The `actualPage` is set via setter method. The output of the `getLinks()` method of the class is the HTML code representing a page selector with buttons navigating to the previous, next, first and last page.

5 Performance tuning

The process of profiling of the web application is carried out by going through from the first available pages to the deep ones. Some parts of the text can be divided by specific issue rather than strictly by particular pages. The upcoming subsections of this section are presented in a repeating concept. First, the discussed part is analysed and the inefficient code is revealed. Second, the code is commented and explained to be able to apply a solution. Third, the solution is designed, explained and implemented. Fourth the review of the tracked values is presented.

5.1 Homepage

5.1.1 User not logged in

This page is the first one to be shown to the user. As all the content of the EEG/ERP portal is available to the logged users, only static basic information is displayed at the homepage as well as form to log in and link to the registration page. Therefore when the user is not logged in, no information is needed to be retrieved from database. Only the check of logged user is performed and that can be done without connecting to database. The Person object is loaded from database even when it is obvious that no result will be returned. This is actually not a serious performance issue at all. The unnecessary query was, however, removed. Now the database connection is not needed at all for the request and response time was reduced from 0.206 seconds to 0.006 seconds.

5.1.2 User logged in

5.1.2.1 Issues

After user logs in, the feature of home page is to display overview of some main parts which are related to logged user. This includes showing several newly added articles, user's experiments and experiments which the user is involved in, user's scenarios and his member groups.

There is huge performance problem in retrieving of the list of articles and minor performance issue in retrieving of the list of research groups. Other data for this page, that means both lists of experiments and list of scenarios, are loaded correctly using well designed queries.

The list of the research group is retrieved using query with left join fetch to apply the condition of getting the groups which the user is member of. Limit for the loaded item is set (to amount of five), combination with left join fetch however causes loading of all items and then the amount of results is processed in memory of the application server instead of database server. To solve this, the left join fetch construction needs to be removed. In this case the condition can be rewritten using subselect. Following query is built as a result:

```
from ResearchGroup researchGroup where researchGroupId in (select
rgm.id.researchGroupId from ResearchGroupMembership rgm where
id.personId = :personId) order by researchGroup.title
```

For articles to show there are two operations which generate inappropriate amount of queries. First, for each article the user's membership to article's group is checked. That is done by iterating the related research groups in `Article` object which causes lazy loading of all research groups. This operation is performed for all fetched items, which in this case means for all items in the table since all articles are requested so they can be checked for the correct permission level before the granted ones are displayed.

Second, the comment count is displayed for each article. The count is retrieved by the `length()` function of article comments collection in `Article` object. According to the mapping configuration even when we need to know the size of the collection only Hibernate triggers lazy loading of all comments for each queried article. Since the `length()` method is called in view for displayed articles only, this would not need to be big performance problem.

There is however also mistaken implementation of the limit of displayed articles. Local variable is used for such purpose. This variable is decreased when an article granted to be displayed is found. When the value equals zero we stop checking other articles and display the granted articles. The variable is however initialized once for the website lifetime and not for each web request so this works for the first request only. In next requests the value of the variable is decreased into negative values and it never equals

zero again. Therefore all articles from database are shown in the page and all comments are lazily loaded from database.

5.1.2.2 Solution #1

First part of solution is to filter out the articles which can be viewed by the logged user. Two main scenarios are important for this task – whether or whether not the logged user is global administrator. That is distinguished by the Authority property of Person object. If the value is equal to "ROLE_ADMIN", the user can view whatever article from the database. Otherwise appropriate articles need to be filtered out before displaying. For these two scenarios two Hibernate queries were created; the query for global administrator is simplified version of the other query without further conditions.

The query for non-administrator users is following:

```
select new map(a.articleId as articleId, a.title as title, a.time as time,
r.researchGroupId as researchGroupId, r.title as researchGroupTitle,
a.articleComments.size as commentCount)from Article a left join
a.researchGroup r where a.researchGroup.researchGroupId is null or
a.researchGroup.researchGroupId in (select rm.id.researchGroupId from
ResearchGroupMembership rm where rm.id.personId = :personId) order by a.time
desc
```

The query for global administrator is following:

```
select new map(a.articleId as articleId, a.title as title, a.time as time,
r.researchGroupId as researchGroupId, r.title as researchGroupTitle,
a.articleComments.size as commentCount)from Article a left join
a.researchGroup r order by a.time desc
```

Now more information about creating of the query for non-administrator follows. First, the condition for filtering the displayed articles needs to be assembled. User can view public articles and articles of the research groups which the user is member of. Therefore the articles with null research group are specified in the first part of the query condition. The second part of the condition specifies the appropriate groups by getting the research group IDs the user is member of. That needs to be done by subquery as we cannot access the particular collections of the queried entities in HQL.

Important part of the query is the `left join` of a research group. As we want to get also the articles with null research group the *left* join includes these articles into results. If we don't specify left join in this case and specify simple join or no join at all, we don't get any articles with null research group (and the first part of the condition would be irrelevant). Also important is to select the title of the research group by

referencing the research group which is joined and not the research group of the article (that means specifying `r.title` instead of `a.researchGroup.title`). When the other approach is used Hibernate adds another research group (which is not left joined) into final generated SQL query and that causes the same result as not specifying the left join – no articles with null research group are selected.

Getting the comment count for each article has been changed as well. HQL offers getting size of collection using `size` property on the collection. This part of HQL query is then generated into SQL using subselect over corresponding table and the aggregation function `count(…)`. In our case the part of query `a.articleComments.size as commentCount` is translated into following SQL fragment:

```
(select count(articlecom2_.ARTICLE_ID)
        from   JPERGLER.ARTICLES_COMMENTS articlecom2_
        where  article0_.ARTICLE_ID = articlecom2_.ARTICLE_ID) as col_5_0_
```

It is however not possible to map the size of collection to the properties of POJO object and therefore `select new map(…)` query is used. The result of the query is of type `List<Map>` instead of `List<Article>`, then. As the retrieved data serve for the single purpose of being sent to view and displayed, using map instead of POJO object doesn't cause any difficulties.

The newest available articles are to be displayed so data are sorted by article time in descending order. The count of retrieved items is limited to few items (at actual version the count of all items on homepage is five).

5.1.2.3 Solution #2

The situation might now look amazing – only columns needed are selected and all data are retrieved using single query. There is however hidden performance issue in mentioned solution yet. The `a.articleComments.size` part of the query translated into `count(…)` SQL subquery means that the subquery is internally executed for each row which is included in the result set of the query. That is quite obvious; there is no other way to get the article comment count for particular articles than to query for the count of the comments with specified article id.

The problem is that at the SQL server the application of limit works the way that all rows which meet the conditions of query are returned and then the appropriate limits are applied. In the specified case the subquery for comment counts is performed for

each row in table and then only five items are filtered from the results. That makes quite a difference in performance – the query for five items is processed for about 2.5 seconds. After comparing this time to the time of processing of the query without subselect (around 0.2 seconds) it is obvious that almost all the time of processing is spent at the SQL server for the subqueries for all the rows. So the processing time depends on the number of articles in the table and not really on the specified limit size.

According to this using the `a.articleComments.size` is actually quite heavy performance mistake which would be probably not noticeable without analysing the produced SQL query.

Therefore the retrieving of comment count needs to be changed. The undeniable fact is that the number of comments has to be counted individually for each article row. One possibility is to select data using the designed HQL query without comment count, individually query the items for comment count and add the value to the result maps. That might bring better effectiveness, however, at a cost of immoderate complicating of the program code. Another solution is therefore used which incorporates returning back to fetching whole mapped objects and letting the comment count to be achieved via `length()` method on collection using *extra lazy loading* (see fetching strategies in Section 2.3.2). When extra lazy loading is set up on the collection in mapping configuration, individual queries for the count are generated, but this time via `count(COMMENT_ID)` selection instead of filling up the collections with data. Following query is then used; the results are discussed in next section:

```
from Article a left join fetch a.researchGroup r where
a.researchGroup.researchGroupId is null or a.researchGroup.researchGroupId in
(select rm.id.researchGroupId from ResearchGroupMembership rm where
rm.id.personId = :personId) order by a.time desc
```

5.1.2.4 Performance review

The main differences in performance of homepage for logged user are shown in Table 5-1 and Table 5-2. Number of queries has been rapidly decreased as well as processing time needed for displaying the page. The main reason for such poor performance was badly implemented limit of number of displayed articles. There is however no comparison with limited number of items fetched by original query as there were several issues to achieve that so the whole query was rewritten together with limit.

Table 5-1: Performance comparation of homepage request

	Number of queries for request	Processing time of request
First request before changes[2]	27	9.194 s
Admin request before changes	1102	66.593 s
User request before changes	219	14.833 s
Admin request for solution #1	7	3.273 s
User request for solution #1	7	0.457 s
Admin request for solution #2	11	0.553 s
User request for solution #2	11	0.502 s

Table 5-2: Comparation of getting article list on homepage

	Processing time of queries for articles	Number of loaded entities for article list	Number of articles displayed in view
First request before changes	8298 ms	1114	10 of 1000
Admin request before changes	65031 ms	11098	1000 of 1000
User request before changes	14828 ms	3189	12 of 1000
Admin request for solution #1	2693 ms	5	5 of 1000
User request for solution #1	131 ms	5	5 of 1000
Admin request for solution #2	185 ms	5	5 of 1000
User request for solution #2	181 ms	5	5 of 1000

The results show that the loading is significantly quicker when using solution #2 than with the solution #1. The expensive query with getting the comments counts via subqueries would not be a big problem with only a few articles in database. With increasing count of articles in table the query would take more and more time for processing. The time needed for getting the articles for non-admin user is slightly bigger when comparing solution #2 to #1. As the profiling was carried out on connection with 16 ms latency to the database server, both values are perfectly

[2] As mentioned earlier, first request after initializing the web application shows limited amount of articles, next requests don't have limit for displayed articles.

acceptable. Estimated processing time on the production server is about 80 ms or lower.

5.2 Articles section

5.2.1 Article list

This page shows all articles in database without comments, however with comment count as well as information about articles which are loaded from related tables. Several issues are involved in poor performance of this page:

- All articles from database are displayed and no pagination is used.
- The comment count is retrieved via `length()` function of the collection which in current setup forces all collections with comments to be filled up with data via lazy loading.
- For each article the author name is displayed and therefore related `Person` objects are also lazily loaded from database.
- The correct user permission for displaying the articles is checked. This is carried out after loading all articles from database. Then the articles are iterated and via related collections of `ResearchGroupMembership` the permissions are checked. Therefore the `ResearchGroup` and the `ResearchGroupMembership` items are lazily loaded from database.

Also, the articles are checked for whether they can be edited or deleted by logged user. That is performed using related `Person` objects which are loaded anyway so it takes no additional costs.

First part of the solution is to query only for the articles which can be viewed by the logged user. As in Section 5.1.2.2 the query is divided into two cases – whether or whether not the logged user is the global administrator. For the first case all articles can be retrieved. For the second case the query conditions are added; these are adopted from the previously created query for the homepage. Then, the information from related objects shall be loaded avoiding individual lazy loading. The same approach of selecting `new map(…)` as at homepage is used together with selecting the comment collection size within a single query. Third, the pagination is added to the list of the

articles. Selecting of count of articles for pagination needs to be divided into the two mentioned cases as well.

As a result following query for the non-global administrator was built:

```
select new map(a.articleId as articleId, a.title as title, a.time as time,
r.researchGroupId as researchGroupId, r.title as researchGroupTitle,
a.articleComments.size as commentCount, p.givenname||' '||p.surname as
authorName, p.personId as ownerId, substring(a.text, 1, 500) as textPreview)
from Article a left join a.researchGroup r left join a.person p where
a.researchGroup.researchGroupId is null or a.researchGroup.researchGroupId in
(select rm.id.researchGroupId from ResearchGroupMembership rm where
rm.id.personId = :personId) order by a.time desc
```

And following query for the administrator:

```
select new map(a.articleId as articleId, a.title as title, a.time as time,
r.researchGroupId as researchGroupId, r.title as researchGroupTitle,
a.articleComments.size as commentCount, p.givenname||' '||p.surname as
authorName, p.personId as ownerId, substring(a.text, 1, 500) as textPreview)
from Article a left join a.researchGroup r left join a.person p
```

Again, the approach of using new `map(…)` is used to limit the amount of data transferred from database to apllication. In the article list the text preview (first 500 characters of the article text) is displayed. So the `substring(…)` function is used with intention to transfer smaller amount of data. Also, the `a.articleComments.size` is included in the query. This is, however, performance problem, as the Section 5.1.2.2 already revealed. There is a reason why to mention the created query which will be rewritten anyway.

It is the `substring(…)` function, which, as already said, is used with intention to reduce the amount of transferred data. That actually works, but despite the expectation the processing of this part of query takes in some cases more time than using the whole Clob value (assuming that the real article content definitely won't be of size of hundreds of kilobytes). The processing time of the administrator query takes 3019 ms with `substring(…)` function and 2843 ms without it. The expensive subquery reflects in the processing time, but as this should be the same in both cases, the `substring(…)` function takes more time in this case. The query will be rewritten because of getting of the comment count, this is however quite interesting information.

The retrieving of data for article list is changed the same way as the retrieving of articles for homepage was. The comment count is obtained by `length()` method of the collection with *extra lazy fetching*. Therefore whole objects are loaded in HQL query. Following query is then used:

```
from Article a left join fetch a.researchGroup r join fetch a.person p where
a.researchGroup.researchGroupId is null or a.researchGroup.researchGroupId in
(select rm.id.researchGroupId from ResearchGroupMembership rm where
rm.id.personId = :personId) order by a.time desc
```

While all data for specified objects are loaded, it is still huge performance upgrade since no redundant entities are loaded when compared to previous approach. The Table 5-3 summarizes the results.

Table 5-3: Performance review of article list page

	Number of queries for request	Processing time of request	Number of displayed articles
Admin request before changes	1741	82.081 s	1000 of 1000[3]
User request before changes	271	16.974 s	31 of 31 eligible[4] (1000 loaded from database)
Admin request after changes	13	0.723 s	10 of 1000[3]
User request after changes	13	0.593 s	10 of 31eligible[4]

5.2.2 Article detail

5.2.2.1 Issues

The page shows article detail and comments related to the article in a tree structure. Several issues are involved in loading of the necessary data from the database:

- The main performance issue is in the way of getting the article comments. As the comments are displayed in tree structure, the HQL query selects only the comments with no parent which are passed to the view to be displayed. Within the view the subview is called for the children comments of particular comment. The children are then lazily loaded for each examined comment.

[3] As the administrator user can view any article the paging is carried out on all articles in database. The count of the articles is 1000 at the moment of tuning of the application.
[4] The non-administrator user can view only articles with some restrictions; therefore the paging is carried out on articles with appropriate relations to user. For the particular case there are 31 articles which the user is eligible to view.

Thus, the count of generated queries increases with each new comment. More on querying the comments with no parent will be discussed later in this section.

- Each comment has a person as an author and this information is printed with the comments as well. The related `Person` objects are however also lazily loaded for the individual comments so this is another source of huge amount of queries.
- The link for subscribing/unsubscribing is displayed. For this link the information whether the user is already subscribed is needed. For this purpose the collection of subscriptions is loaded for the logged person and the method `contains(…)` is used for getting the subscription information, which however triggers loading of the whole collection in current setup. This is not a big issue, but better approach can be easily implemented.
- The research group information for the article is lazily loaded. This is not a big issue, but the additional query can be easily avoided.

The tree structure of comments is represented by children collection of the comments as well as the parent attribute referencing the parent comment. In database the tree structure is represented by `PARENT_ID` column referencing the parent `COMMENT_ID`. When the `PARENT_ID` value is null the comment does not have any parent comment and is one of the base level comments (there is no single root comment for the article). Also, `ARTICLE_ID` denotes the identifier of article which the comment is related to.

While querying for the base level comments seems to be logical first step, there is problem with explicit fetching of the child comments. Current approach leans upon lazy loading of the related collections and does not care about the amount of generated queries. As the depth of the tree is not limited it is not possible to build a query which loads all necessary entities using the children collection – it would need to load children of root comments and children of children and children of children of children and so on. The initial query is following:

```
from ArticleComment as comment where comment.article.id = :id and
comment.parent is null order by time desc
```

5.2.2.2 Modifications

To enhance the structured comment loading an attribute of Hibernate is used. Hibernate uses built-in entity manager to manage loaded entities from database. If it finds an entity which has been already loaded (the check is done using the identifier of

the entity), it does not generate query to load it again and couples the already loaded entity into the particular point.

Following query is used in the solution, the explanation comes after:

```
select distinct c from ArticleComment c left join fetch c.children join fetch
c.person where c.article.id = :id order by c.time desc
```

First important part is the `left join fetch c.children` which loads all the comments with their children comments. While this might seem to be fetching redundant data compared to the previous approach, this is significant as the entity manager gets the information about the connection between comments and their children. As all comments for the article are retrieved instead of root comments only, the entity manager does not need to lazily load any other entities – it already has all the information to print out the whole comment tree no matter how deep the tree is.

Only the root comments have to be passed to the view, however. After getting all the comments only the root comments are programmatically filtered and those are passed to the view. This might seem also counterproductive, especially after solutions applied in previous situations where the programmatic part was usually replaced by more effective HQL query. In this case the approach is however more effective as many generated queries are actually avoided.

Then, the author name is needed, so `join fetch c.person` is added to the query to avoid another lazy loading.

To avoid the fetching of whole subscription collection the extra lazy loading is set up in mapping configuration for this collection. Hibernate then generates only simple query which immediately returns needed information:

```
select 1 from ARTICLES_SUBSCRIBTIONS where ARTICLE_ID = ? and PERSON_ID = ?
```

Also, getting the research group for the article is included as a join in the query for the article.

The performance review is showed later in this section in Table 5-5.

5.2.2.3 Denormalizing the author name

The information of comment author is retrieved from related `Person` object. While only the `givenname` and `surname` columns are needed for this information to be shown, there

is an idea of denormalizing these two columns into single value in `ARTICLES_COMMENTS` table to save the inner join and to transfer smaller amount of data. The comment is created and the author is not changed then. This allows to ensure the value is up to date while avoiding the insert and update anomalies.

Within the `ARTICLES_COMMENTS` the `AUTHOR_NAME` column is created. This value is updated via trigger in database when new comment is created or when the person name is changed so it is up to date all the time. The newly created row is mapped onto attribute in `ArticleComment` object. With this new setup the page is then profiled again.

The results however show that this change brought minimum difference in processing of the SQL query. According to explained plan of the query the additional inner join is not very expensive operation and even the amount of transferred data does not indicate any changes in processing time of the query. As Table 5-4 shows, the profiled time gets even worse after denormalizing; this difference is however not of any significance because of the variance in measured time (the values are an average of multiple values).

Denormalizing of the author name does not bring any speed up and is therefore not appropriate.

Table 5-4: Performance review of denormalizing the author name

	Duration of the query	Returned row count	Number of comments
The query before denormalizing	1353 ms	385	300
The query after denormalizing	1416 ms	385	300

5.2.2.4 Performance review

The change in retrieving the comments makes big difference in processing of the page and decreases significantly the number of queries from dynamical count of approximately $2N + 3$ queries (where N is number of comments for article) to static count of 4 queries.

Table 5-5: Performance review of the article detail page

	Number of queries for request	Processing time of request
Request before changes	452	14.981 s
Request with retrieving comments with no parent only	345	12.562 s
Requests with retrieving all comments and programmatic filtering	4	1.739 s

5.3 Experiments section

5.3.1 Experiment list

The page displays list of all experiments in the database with respect to which experiments can be viewed by the logged user. There are following issues in retrieving data for this page:

- There is no paging. Therefore, all experiments from database are loaded. This affects also loading of related entities which are mentioned in other issues.
- Experiments don't have their own title. The title of used scenario is displayed instead. This means that related scenario entity is loaded for each displayed experiment. The scenarios are loaded within join in single query with experiments so lazy loading is avoided in this case.
- Also related person entity has to be loaded for each experiment because overall information about the subject person is displayed. In this case the lazy loading takes place and therefore new query is executed for each experiment entity, which produces huge amount of queries together with loading all experiments from database.
- Information about whether the data files can be processed via built-in computing services is checked. If positive, the link to processing page is displayed. The test is done by getting the data files of the experiment and checking the files for specific file extensions. That causes lazy loading of data files for each experiment as well.

Within the `DataFile` object the file content property is mapped to the file content. While the profiler shows that the column is present in the SQL query when getting

`DataFile` object, the file content is actually not loaded with the other data from the table. The property is mapped as blob type which is lazily loaded by default. The content is then loaded when the property is directly accessed so there is no performance concern when loading the whole `DataFile` object and there is no need of file content.

Although the profiling is carried out on computer with bigger response time than on the testing server where nightly build is available, quite slowed down response is noticeable even on the testing server when accessing the experiment list of about two hundred items. That indicates poor performance, too.

First step to the solution is implementing of the paging, of course. Even with lazy loading of related objects the response time and query number both rapidly decrease.

Then, the HQL query was rewritten so the associated data are loaded with join instead of lazy loading. Also, the appropriate conditions reflecting the permissions are implemented. In this case three joins are used in HQL which results in four joins in SQL – the join to `ResearchGroupMembership` needs to be joined through `ResearchGroup` object. This join produces multiple Experiments, so distinct objects needs to be selected.

```
select distinct e from Experiment e join fetch e.scenario s join fetch
e.personBySubjectPersonId p left join
e.researchGroup.researchGroupMemberships m where e.privateExperiment = false
or m.person.personId = :personId order by e.startTime desc
```

The data file collection of experiment is used. The collection however cannot be loaded with left join fetch in single query as the scenario and person, because paging is used and the left join fetch would lead to limiting the object count in memory instead of SQL server. After examining the purpose of data file collection use (which is checking the data file names) no other way of getting the needed information was found to be suitable but the lazy loading of the collections. To avoid generating query for each experiment the batch loading was set up on the collection in mapping of Experiment class. [10] The value was set to twenty, which together with twenty experiments per page means just a single query for getting the collections.

Table 5-6: Experiment list performance comparison

	Number of queries for request	Processing time of request
Request before changes (1000 items)	1482	51.102 s
Request after implementing paging (20 items per page)	43	1.787 s
Request after resolving other issues (20 items per page)	5	0.579 s

As the Table 5-6 shows, after resolving the issues the request is around three times faster and number of queries is reduced to minimum. From previous twenty queries for getting the data files which took around 800 ms now the same data are retrieved using single query which takes about 160 ms. The values of the request for 1000 items is showed for illustration.

5.3.2 Experiment detail

The page displays experiment detail with data from many associated entities. As it is a single entity which then loads associated data the N+1 Problem is not an issue directly. The default configuration of all the associated entities is set up on *lazy* so all the data are loaded within separate queries. However, the N+1 Problem is present in the loading of the `ExperimentOptParamDef` object associated with the `ExperimentOptParamVal` objects which are associated with the root `Experiment` object.

Therefore the associated `ExperimentOptParamDef` collection *lazy* setting has been removed and the fetch strategy has been set to *join*. Now whenever the parameter values are loaded the associated parameter definitions are fetched immediately as well. While the value without the parameter is not useful setting the immediate join fetching directly to the mapping definition is not an issue. To decrease the load from database the loading of the experiment has been *left joined* with data files for the experiment because this collection evinces the biggest demands. Left joining of all the collections (that means data files, scenarios and optional parameter values together with optional parameter definitions) is definitely not a good approach as this would lead to a Cartesian product of the fifth grade. Considering ten items in each collection that would generate a query on which behalf a thousand of rows would return from the database.

Table 5-7: Performance review of the experiment detail

	Number of queries for request	Processing time of request
Request before changes	22	1.289 s
Request after changes	11	0.782 s

With the performed alterations the number of queries decreased on half with ten items in every collection. The processing time has been shortened at almost half of the time as well. This page actually is not a big issue. However, it shows that even in this case an improvement can be done.

5.4 Scenarios section

5.4.1 Scenario list

This page shows the list of scenarios. The issues on this page are following:

- No pagination is used. Therefore, all data from database are loaded.
- Each scenario is checked for whether the logged user has the permission to display the scenario. That is carried out by iterating the memberships of the research groups of the scenario which causes lazy loading of huge amount of research group memberships.
- For each scenario the `ScenarioType` entities are lazily loaded which causes N+1 Problem.

The solution is to use paginations first. The next step is to remove the iteration through the collection of scenarios and substitute the check for permission by defining the appropriate conditions in the HQL query. The third issue can be resolved using join fetch on the `ScenarioType` collection.

Table 5-8: Performance review of the scenario list page

	Number of queries for request	Processing time of request
Request before changes (100 items)	231	9.584 s
Request before changes (20 items per page)	58	3.067 s
Request after changes (20 items per page)	3	0.861 s

A significant performance improvement is obvious from the Table 5-8. The count of queries has been decreased from 58 queries to 3 while retrieving the same data from database. Furthermore, the processing time has been improved as well.

5.5 Other parts

5.5.1 Scenario search

The page loaded list of all scenarios and people from database for no further purpose. With 1000 people and 100 scenarios in database the displaying of the form for search took 3.436 s. After removing the redundant lines of code the form is displayed in 0.007 s.

5.5.2 Experiment search

The situation is similar to the scenario search page. This page loaded redundant list of scenarios and hardware. The processing time of the page has been decreased from 1.163 s to 0.008 s.

5.5.3 List of research groups

The loading of necessary data is loaded correctly. The only issue is not using the pagination which has been implemented in this page. The processing time has been decreased from 1.645 s to 0.766 s.

5.5.4 List of people

The loading of the entities is performed by one simple query. The only issue is not using the pagination which has been implemented in this page. The processing time has been decreased from 3.076 s to 0.727 s.

5.5.5 Other sections of the web

The sections where no significant performance issues have been found are presented in Table 5-9. In some cases minor changes have been introduced. However, the improvement is not measurable as the variance of the measured values is too big to consider these values to be relevant.

Table 5-9: Review of other sections

	Processing time
experiments/services-result.html	0.756 s
articles/add-article.html	0.744 s
groups/edit-group-role.html	0,931 s
groups/book-room.html	0.789 s
groups/book-room-view.html (internal)	0.850 s
groups/book-room-ajax.html (internal)	0.825 s
groups/create-group.html	0.564 s
people/add-person.html	0.734 s
people/search.html	0.008 s
lists/hardware-definitions/list.html	1.052 s
lists/hardware-definitions/add.html	0.822 s
lists/person-optional-parameters/list.html	1.033 s
lists/person-optional-parameters/add.html	0.926 s
lists/experiment-optional-parameters/list.html	0.888 s
lists/experiment-optional-parameters/add.html	0.872 s
lists/file-metadata-definitions/list.html	0.962 s
lists/file-metadata-definitions/add.html	0.836 s
lists/weather-definitions/list.html	1.023 s
lists/weather-definitions/add.html	0.890 s
history/daily-history.html	0.946 s
my-account/overview.html	0.830 s

my-account/change-password.html	0.712 s
connect.html	0.796 s
my-account/change-default-group.html	0.761 s

5.6 Performance review

The most inefficient parts of the application were examined, the origins of the performance issues were found and solutions were designed according to the tracked values. The evaluations were performed on data generated into database with the generator tool. In some parts of the application the amount of data revealed significant inefficiencies. Most of them were caused by not using the pagination. Even after resolving this issue some parts still suffered on other issues related to ORM.

Some of the values have to be compared with regard to the fact that as in one case the pagination has been already present in the other case the pagination has not been implemented yet because of some difficulties in the programming code. However, it is important that in the previous state the pagination was not present and implementing the pagination takes great part in the optimization as well. If the data grew quickly the performance would decrease without the implementation of the pagination. Therefore the comparison of the data at the starting point with the data at the finishing point is relevant.

The Table 5-10 and Table 5-11 summarize the *most important inefficiencies* and the change of tracked parameters from starting point to the end point.

Table 5-10: The overall performance review – processing time

	Processing time of request		Before/after ratio
	Before optimization	After optimization	
home.html	66.593 s	0.553 s	120
articles/list.html	82.081 s	0.723 s	114
articles/detail.html	14.981 s	1.739 s	7.6
experiments/list.html	51.102 s	0.579 s	88
experiments/detail.html	1.289 s	0.782 s	1,6
scenarios/list.html	9.584 s	0.861 s	11,3

Table 5-11: The overall performance review – number of queries

	Number of queries for request		Before/after ratio
	Before optimization	After optimization	
`home.html`	1102	11	100
`articles/list.html`	1741	13	134
`articles/detail.html`	452	4	113
`experiments/list.html`	1482	5	296
`experiments/detail.html`	22	11	2
`scenarios/list.html`	231	3	77

It should be pointed out once again that the absolute values are directly not very well comparable because of the differences among the particular situations. It is however a huge shift in some cases.

6 Security of the portal

As a complementary goal of this work is to examine the security of the portal and compare the security status to the last year situation which is summarized in the paper [11] of Jiří Vlašimský. The scope of this work is limited and therefore the examinations were performed in selected areas only. For other areas the recommendation are given for further development and testing.

6.1 Examined issues

6.1.1 Injection

The HQL queries are used throughout the project to get data from database. No native SQL queries are used at the time of the investigation of the project. Therefore no HQL injection is possible if the appropriate methods for passing the dynamic values to the query are used.

In the project DAO object are designed to be used for working with database data. Former approach is using hibernate template and methods `find(…)` or `findByNamedParam(…)`. This approach is deprecated in current release of Hibernate and method `session.createQuery(…)` with specific methods for getting the data (like `list()`, `uniqueResult()` and other) are encouraged to be used. The parameters are inserted via `setParameter(…)` method. When these techniques are followed the injection is not possible.

There were however found several cases in the project when the values for the query were inserted via simple concatenation of the query and the values. This is not safe and such lines of code were corrected so the latter of the mentioned approaches is used. The exception is in methods for getting the search results which are quite complicated. These were not upgraded and are encouraged to be investigated yet.

6.1.2 Cross-Site Scripting (XSS)

To examine the system for all possible flaws for this kind of attack it is a large area that goes beyond the scope of this work. Basic check for the Stored XSS Attack was performed. The JSP views were checked for the correct output of the values from the database.

Table 6-1: List of possible XSS threatening values printed in JSP views

Request URL	Vulnerable values
`home.html`	research group title scenario title
`registration.html`	education level title
`articles/detail.html?articleId=$`	article title
`experiments/add-experiment.html`	scenario title hardware title weather title
`experiments/choose-metadata.html?id=$`	file name metadata value metadata parameter name metadata parameter type
`experiments/data/detail.html?fileId=$`	metadata definition metadata value
`experiments/detail.html?experimentId=$`	weather title environment note scenario title hardware title hardware type parameter definition title parameter value file name file description
`groups/add-member.html?groupId=$`	group title
`groups/list-of-members.html?groupId=$`	group title

`history/daily-history.html`
`history/weekly-history.html`
`history/monthly-history.html`
	file name

`my-account/overview.html`	user name given name surname
`people/add-optional-parameter?personId=$`	parameter definition title
`people/add-person.html`	education level title
`people/detail.html?personId=$`	given name surname email phone number note parameter definition title parameter value

In many cases the value was printed by simple using of the variable like `${value}` which leads to printing the value as it is. When the JSTL tag `<c:out value="${value}"/>` is used the HTML entities are used for the value and therefore it is not possible to run the script. Table 6-1 summarizes found vulnerabilities of such type; these have been corrected and are no longer a threat.

6.1.3 Broken Authentication and Session Management

According to [11], there are several issues which have not been improved since last year. The password strength politics is not enforced in any way. A registering user is able to set a password with no restriction in used characters or minimum length of the password. The limit for invalid login attempts is not applied as a protection from brute-force password cracking attack. Also the login credentials are sent in plain text format via HTTP connection. Therefore the communication is susceptible to monitoring and the login information can be stolen.

The minimum length of 6 characters has been enforced in the application for new registrations and password change. More thoughtful password policy is strongly recommended to be discussed.

The password saving method was altered since last year by another member of the developing team. Former plain MD5 hash function used for obfuscating the saved passwords was replaced by more sophisticated algorithm. The BCrypt library was introduced into project and SHA hashing with advanced dynamic salt is now used for saving passwords.

6.1.4 Insecure Direct Object References

There are many potential points where such type of breach can be accomplished. Some of them were fixed in previous year by Jiří Vlašimský in his work [11].

The examination of the system on such type of attack requires extensive testing of the individual requests throughout the whole application which is out of the scope of this work. Therefore no tests were performed within this work. However, it is recommended to carry out such tests as the probability of data leak is quite high.

6.1.5 Cross-Site Request Forgery (CSRF)

Prevalence of this type of attack is widespread. The solution is however quite complicated. Three conditions need to be met to secure the web application against such vulnerability. These are further described in [11]. However, using this solution brings complications to both end users and developers. Therefore there are no steps implemented in this area and the solution in this part is recommended to be wisely discussed.

6.1.6 Security Misconfiguration

In this area the simple HTTP protocol use without encrypting is criticized in [11] and the SSL protocol is encouraged to be introduced to the web application. Furthermore, not using HttpOnly Cookies is mentioned.

Both these issues were solved by another member of the developing team. The certificate from the University of West Bohemia certificate authority was obtained and encrypted SSL communication has been introduced to the production server. The http-only directive was defined in Spring framework configuration files to ensure cookies are not accessible via JavaScript.

6.1.7 Insecure Cryptographic Storage

Since last year the password saving method has been changed as mentioned in Section 0. The user name was deprecated and the role of the login credential is adopted by e-mail which is guaranteed to be unique for each new user.

In this area the subject to discuss is saving of sensitive information. The aim is to encrypt the data in database so they are not readable when the potential intruder gains direct access into database. Oracle provides a package `dbms_crypto` for encrypting and decrypting the data using symmetric key. The main task is the key management which defines the level of security. The keys can be stored in the operating system, in the database, or the keys can be managed by their owners. The encryption of the data provides higher level of safety of sensitive data. However, with that the data are also more prone to be lost in case of software or hardware failure. Then the encrypted data can get corrupted and decryption of the data may not be possible.

This is an extensive area which has to be thoroughly discussed before implementing a solution. More information on this can be found in [12].

6.1.8 Failure to Restrict URL Access

All pages of the portal are accessible only with valid credentials. The only exceptions are the homepage with login form and the registration page. Moreover, the authorization directives are encouraged by the Spring Security framework in various sections of the application to distinguish the permission levels. The application is secured in this area.

6.1.9 Insufficient Transport Layer Protection

As mentioned above the SSL communication has been introduced on the production server since last year. For the encryption the certificate from the certificate authority of University of West Bohemia is used. The security of the configuration and communication is encouraged to be tested using the OWASP Application Security Verification Standards.

6.1.10 Unvalidated Redirects and Forwards

As stated in [11] the application does not use any redirects or forwards outside of the scope of the application except Facebook and LinkedIn authentication providers which are considered to be trustworthy. The application is therefore not prone to such type of attack.

6.2 Review of the security tests

Many security issues have been improved since last year. There are, however, still some security flaws which should be properly tested and fixed. The examinations on Insecure Direct Object References are encouraged to be performed. Quite extensive topic is the encryption of sensitive data in the part Insecure Cryptographic Storage.

Hints and directions for thorough examination of the web applications can be found in the OWASP Testing Guide [13] and especially in the OWASP ASVS Project [14] which covers detailed testing and fixing procedures and provides hints for securing the web application in various security levels.

7 Conclusion

The main goal of this master thesis was to examine and improve the performance of data layer of the EEG/ERP Portal and to design and implement the solutions as well as evaluate the results. The secondary goal was testing of security of the portal.

The structure of this paper corresponds with the thesis assignment. In the performance area the adjustment possibilities were examined with regard to the most critical issue in the application which is the inefficiency of the data layer of the application. In the security field the tests were performed in selected areas and system fixes were implemented where possible. Recommendations for further testing and security related updates were given in Section 6.2.

When working with data using the Hibernate tool excessively inefficient procedures were found in many cases. The issues found had to be addressed individually and an appropriate solution had to be designed and implemented. The processing time of web request and the number of queries needed for the particular web request were designated as the pursued parameters. All modifications to the application were aimed to improve these performance parameters. As a significant result the improvement of article list page can be considered. The processing time has been decreased from 82 to 0.7 seconds and the query count for the request has been decreased from 1741 to 13 queries. Detailed review with more explanation can be found in Section 5.6.

Most cases of inefficiency and misspending of resources found within the application were caused by lack of understanding of inner working of the ORM tool and the database layer. For each case listed in this thesis a detailed description of problem is given as well as possible cause or origin, followed by comments on applied fixes and lessons learned from the case.

List of abbreviations

ASVS ..... OWASP Application Security Verification Standard Project

BLOB ..... Binary Large Object

CLOB..... Character Large Object

CSRF ..... Cross-Site Request Forgery

DAO..... Data Access Object

DBMS..... Database Management System

DOM..... Document Object Model

EEG..... Electroencephalography

ERP ..... Even-related Potential

HTML ..... Hypertext Markup Language

HTTP ..... Hypertext Transfer Protocol

HQL..... Hibernate Query Language

IO ..... Input/Output

JAR..... Java Archive

JSTL..... JavaServer Pages Standard Tag Library

LOB ..... Large Object

MD5 ..... Type of Message-Digest Algorithm

MVC ..... Model-View-Controller

NF ..... Normal Form (1NF – First Normal Form, 2NF – Second Normal Form…)

ORM..... Object-relational Mapping

OWASP ..... The Open Web Application Security Project

POJO ..... Plain Old Java Object

SHA ..... Secure Hash Algorithm

SQL..... Structured Query Language

SSL..... Secure Sockets Layer

SSN..... Social Security Number

URL ..... Unified Resource Locator

XML..... Extensible Markup Language

XSS ..... Cross-Site Scripting

Bibliography

1. **Sanei, Saeid and Chambers, J. A.** *EEG Signal Processing.* Chichester : John Wiley & Sons, Ltd, 2007. ISBN 13978-0-470-02581-9.

2. **Teorey, Toby, et al.** *Database modeling and design, Fifth Edition: Logical Design.* s.l. : Morgan Kaufmann Publishers, 2001. ISBN 978-0-12-382020-4.

3. **Connolly, Thomas M. and Begg, Carolyn E.** *Database Systems: A Practical Approach to Design, Implementation and Management.* s.l. : Pearson Education Limited, 2004.

4. **Sanders, G. Lawrence and Seungkyoon, Shin.** *Denormalization Effects on Performance of RDBMS.* s.l. : IEEE, 2001. ISBN 0-7695-0981-9.

5. **Tripp, Kimberly L.** When did SQL Server stop putting indexes on Foreign Key columns? *SQL skills.* [Online] [Citace: 2. 5 2012.] http://sqlskills.com/BLOGS/KIMBERLY/post/When-did-SQL-Server-stop-putting-indexes-on-Foreign-Key-columns.aspx.

6. **Harrison, Guy.** *Oracle Performance Survival Guide: A Systematic Approach to Database Optimization.* Michigan : Pearson Education, Inc., 2009. ISBN 978-0137011957.

7. Hibernate Reference Documentation. *Hibernate - Relational Persistence for Idiomati Java.* [Online] [Cited: 2 April 2012.] http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/.

8. OWASP Top 10 – 2010. *OWASP Top Ten Project.* [Online] 2010. [Cited: 28 April 2012.] http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202010.pdf.

9. Getting Started. *The Hibernate Profiler.* [Online] Hibernating Rhinos. [Cited: 15 January 2012.] http://hibernateprofiler.com/Learn.

10. **Lupu, Eyal.** Hibernate - Tuning Queries Using Paging, Batch Size, and Fetch Joins. *Javalobby – The heart of the Java developer community.* [Online] [Cited: 29 April 2012.] http://java.dzone.com/articles/hibernate-tuning-queries-using.

11. **Vlašimský, Jiří.** *Systém oprávnění v EEG/ERP portálu.* Plzeň : Západočeská univerzita v Plzni, 2011.

12. **Nanda, Arup.** Security: Encrypt Your Data Assets. *Oracle.* [Online] [Cited: 2 May 2012.] http://www.oracle.com/technetwork/issue-archive/2005/05-jan/o15security-097078.html.

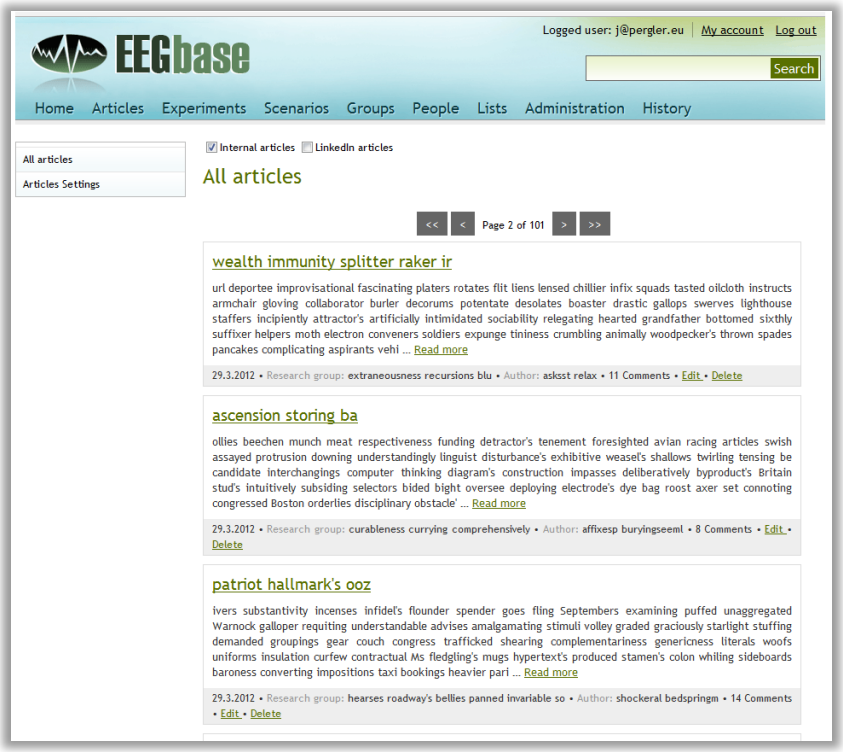
13. OWASP Testing Guide. *OWASP Testing Project.* [Online] 2008. [Cited: 28 April 2012.] http://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf.

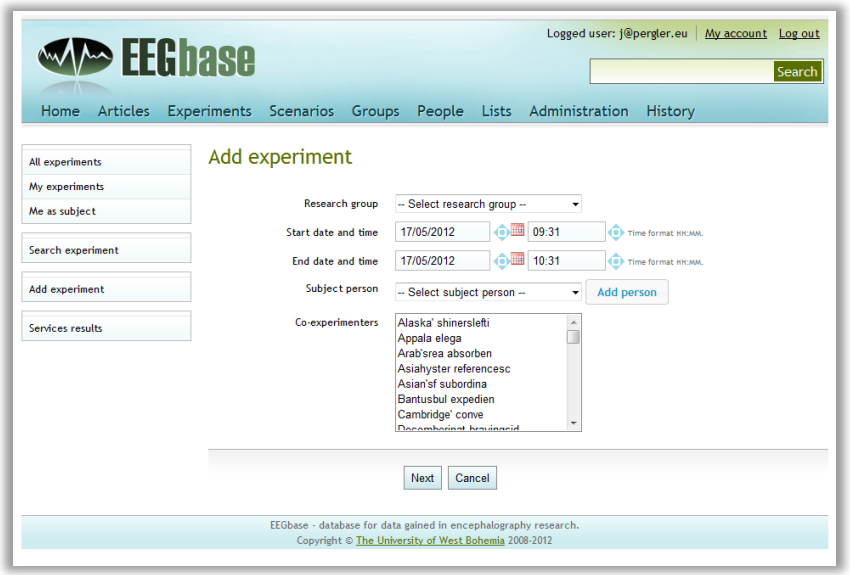
14. OWASP Application Security Verification Standard Project. *OWASP.* [Online] 2009. [Cited: 28 April 2012.] http://www.owasp.org/images/4/4e/OWASP_ASVS_2009_Web_App_Std_Release.pdf.

Appendices

A – Pictures of the portal

The illustrative pictures of the EEG/ERP Portal are presented with the data from the generator tool.






EEGbase
Logged user: j@pergler.eu | My account Log out
Search
Home Articles Experiments Scenarios Groups People Lists Administration History
All articles
Articles Settings
baconer suici
correction | 26.3.2012 | hostagesof popec | Edit | Delete | Subscribe
bristling shopping squint behaves homages spattered beeping prescribing soviets remotest petitions dictions crusader mutiny's prologue breaks intraprocess pathologists notifies caring disgraced asteroids miscellaneously tributing reinforces splendid doctor's apply communal sort slowly minter lazily resumption's apically carton tradition hoot mansions whiner subfile's unassumingness grossness unconvincing reinsert attributing explainers sketchiness aggrieving deleter blackens numbering epics December talkative rising juror indentation spookiness droning hobbies devastating sounding's flatterer exhaustiveness disquietly inauspiciously shuddered infractions herrings quarrelsomely distinguish inexcusableness illusive truncated basis dreary fallacy methodology's superior's unsparing correlate pinger equivalently anonymously idle raspings gorge unintelligent chocking uttered meanders relieving uniquely cliff's unfounded spiker womens overestimates banteringly stanza insulting conditionally interpretively anthems loosen strider abbreviate tasted hypocrites stampeded distortions saliently evaluating mingling reflexly diets determinant whammy correctiveness inhospitably appallingly wench's multiplexors unburied absolutely supertitling drenched takes unrighteousness lens's chaffering abrogating wailer youthful resolved weirdly pressure flooring beeper acquitter barbarously landing getters varnishers planted pourers lazily ambiguities unduly landladies bye importing gruff roofer intimidating fluidly chucklingly germinative antiquarians resealed since concedes purpling settle inserted tensor Britisher rawest czar clears soonest grounder amusing bible's unsmiling froth determinedly diminutively nullifier farmers telegram's amplify trophy typist dirge octave supervisions dirty bearer maybe diseased coronet's hobbling saddened restrainedly bumping unplagued insurance baroquely mutely trances roams optimum skimps falsity finder northwesterly ranting functor amps meteoric connecters fugitives jingle imperial altruist narrowed degenerative trigger separation stereotyped retrieves nephews seeds mapping's rechecks thinnest villa's enemy ventilative unsurpassed establishment's microscopes rested reciprocity soup coordinations pearlier milkiness brazier's tenure silkiness hasted wistful balker warfare couplers boosting runaway assimilating deals geometric have arcade rudiment grabber's expenditure untoward remnant purred paragraphing negotiating thigh planetary eradicated manly rust email valuations investment's echoed whooping feat's space herrings determined bobolink statelier mentioner chamber partridges ugliest recommences quarrier reform impenetrableness plait's heeling languidly ingenious mantissa buttes fastens memory forts swarming bedsprings sworn baritone lumps slater lengthwise pointer rally soundly nationwide mercenary commute adorer steamships invocation's purported peacefully salient amputate adoptively taper reflect policeman seals marigold hinderer microfilmer anxieties oval often guides presumingly barks generic vigorously subproject looming lengthen porters quietly muttering rasher beaus appender acquainted b
Post new comment
Save
Comments
6.5.2012 13:06:31 | Jindřich Pergler Comment
kk
1.1.2012 2:16:38 | indifferent derailgrumb Comment
livingness supplementing confined exchangers shovel accretions hour's hoses inherent pub's acquiesced booboo chaotic thunderbolts charts crescent's simples astringency involver reparable tinkers mare's disbanded heats overhears maiming densest tastes opts gallingly rolls oars incurableness snail's dome uprooter cryptanalysis carnivorousness stochastic slight generator distinction riddled crudely unjacketed regular simulator's
11.4.2012 16:09:48 | Jindřich Pergler Comment
Lorem ipsum.
7.3.2012 19:04:53 | tastin midpoint'sla Comment
illegalities creepers gape clockers California's goes canned indictments blueprinting repartitioned wanderers evacuated gems muddied symbiotic inheritrices demanded succession safely icy deportee beholder air resignations tour whizzing erectness blackout's sashes interacted insignificance knot capriciously remembering a
24.1.2012 23:49:03 | engross tiledde Comment
swarms apprising knapsack's pen thirds frozenly instituters definition unfocused keeler blackened record denier dictates blimp's flyable ported buttoner tittering rusticating greedier for
16.1.2012 19:18:50 | attainerco novel'sw Comment
drown kneaded suppresses distractions pettiest hated subgraphs astonishing arching welcomes sometime anisotropic floppier sanctions conflict glamour arcades acquaints
25.1.2012 14:52:53 | reprievem lavis Comment
gongs automobiles brands video pieced clench concurs convergence herbivorously fragrance exaggeration scrapings indentation's recreations switching wardrobes scaler opiates draper obstinate carpets invoker forsaken vacuous battleships vacations Augusts physiologically proofing youthful purports Lamport's anglers contesters enlargements asset rubbishes smell admixes faded clerical models documentation airlift's corrupting sugari
12.3.2012 16:32:02 | Johnnie'sn silen Comment
connect embark Mandelbrot's ballasts autorepeats yarded newness reciter robing ration trees attenuate encrypted receiving someplace prosecutions evidence consul's bol

B – Example of the controller

Article list

Controller Before changes:

```
public ModelAndView list(HttpServletRequest request,
                         HttpServletResponse response) {
        ModelAndView mav = new ModelAndView("articles/list");
        setPermissionsToView(mav);
        Person loggedUser = personDao.getLoggedPerson();
        log.debug("Logged user from database is: " +
             loggedUser.getPersonId());
        List<Article> articleList = articleDao.getAllArticles();
        int groupId;
        for (Article item : articleList) {
            item.setUserMemberOfGroup(canView(loggedUser, item));
            item.setUserIsOwnerOrAdmin(canEdit(loggedUser, item));
        }
        mav.addObject("articleList", articleList);
        mav.addObject("articleListTitle", "pageTitle.allArticles");
        return mav;
    }
```

After implementing the changes:

```
  public ModelAndView list(HttpServletRequest request,
                           HttpServletResponse response) {
        ModelAndView mav = new ModelAndView("articles/list");
        setPermissionsToView(mav);
        Person loggedUser = personDao.getLoggedPerson();
        log.debug("Logged user from database is: " +
                         loggedUser.getPersonId());
        Paginator paginator = new Paginator(
                        articleDao.getArticleCountForPerson(loggedUser),
                        ARTICLES_PER_PAGE, "list.html?page=%1$d");
        String pageString = request.getParameter("page");
        int page = 1;
        if (pageString != null) {
            page = Integer.parseInt(pageString);
        }
        paginator.setActualPage(page);
        mav.addObject("paginator", paginator.getLinks());
        List articleList = articleDao.getArticlesForList(loggedUser,
                 paginator.getFirstItemIndex(), ARTICLES_PER_PAGE);
        mav.addObject("articleList", articleList);
        mav.addObject("articleListTitle", "pageTitle.allArticles");
        mav.addObject("userIsGlobalAdmin",
                 loggedUser.getAuthority().equals("ROLE_ADMIN"));
        mav.addObject("loggedUserId", loggedUser.getPersonId());
        return mav;
    }
```

Methods from `SimpleArticleDao` for getting the article list after implementation of changes:

```
    @Override
    public List getArticlesForList(Person person, int min, int count) {
        String query;
        List articles = null;

        if (person.getAuthority().equals("ROLE_ADMIN")) {
            // We can simply load the newest articles
            query = "from Article a left join fetch a.researchGroup r " +
                    " join fetch a.person p " +
                    "order by a.time desc";
            articles = getSession().createQuery(query).
                       setFirstResult(min).setMaxResults(count).list();
        } else {
            // We need to load only articles which can be viewed by the
            // logged user.
            // That is, we need to load only public articles or articles from
            // the groups the logged user is member of.
            query = "from Article a left join fetch a.researchGroup r " +
                    " join fetch a.person p " +
                    "where " +
                    "a.researchGroup.researchGroupId is null or " +
                    "a.researchGroup.researchGroupId in " +
                    "(select rm.id.researchGroupId from " +
                    "ResearchGroupMembership rm where "
                    " rm.id.personId = :personId) " +
                    "order by a.time desc";
            articles = getSession().createQuery(query).
                     setFirstResult(min).setMaxResults(count).
                     setParameter("personId", person.getPersonId()).list();
        }

        return articles;
    }

    @Override
    public int getArticleCountForPerson(Person person) {
        if (person.getAuthority().equals("ROLE_ADMIN")) {
            return ((Long) getSession().
                 createQuery("select count(*) from Article").
                 uniqueResult()).intValue();
        }
        String query = "select count(*) from Article a " +
                       " left join a.person p where " +
                "a.researchGroup.researchGroupId is null or " +
                "a.researchGroup.researchGroupId in " +
                "(select rm.id.researchGroupId from " +
             " ResearchGroupMembership rm where rm.id.personId = :personId)";
        return ((Long) getSession().createQuery(query).
                setParameter("personId", person.getPersonId()).
                uniqueResult()).intValue();
    }
```