Università
della
Svizzera
italiana

Faculty
of
Informatics

**Bachelor Project**

January 22, 2021

# Experimental Apparatus

For a Digital Health Literacy Experiment

Robert Jans

*Abstract*

In this project we implement a software toolkit for conducting an experiment planned by the *Institute of Communication and Health (ICH)*. The institute is part of the *Faculty of Communication Sciences* at the *Università della Svizzera Italiana*. Health communication is a relatively new and multidisciplinary field, which includes the study and use of communication to inform and affect decision-making at the individual and community level in order to improve the quality of healthcare [18]. The purpose of the planned experiment is to investigate the digital health literacy of participants in the area of sleeping disorders.

This project seeks to provide the experimental apparatus which will allow the research team to record data and test its hypotheses. The project's main tasks are indexing a predefined set of websites, creating a user interface similar to common search engines that can be configured to selectively show different subsets of the corpus according to the experimental conditions under investigation and setting up an experimental environment allowing to conduct controlled experiments and record salient data such as search logs and click-stream.

**Advisor:**
Prof. Marc Langheinrich
**Co-advisor:**
Prof. Peter Schulz

# Contents

## List of Figures

# 1 Introduction

## 1.1 Motivation

Since the beginning of modern science, researchers have needed special tools in order to conduct experiments. These tools consist mainly of devices for taking measurements and triggering phenomena under controlled conditions. Whereas in the past the experimental apparatuses comprised almost exclusively physical devices, nowadays increasingly more software is involved. The experiment for which the result of my project is going to be used is a case where the process of setting up the experimental conditions and collecting the result data depends heavily on the software toolkit. It is therefore essential for the software to be robust and reliable.

My personal motivations come from two sides: I have a general interest in science and the scientific method, but in practice, rather than as a scientist, I'd see myself as a technician, who in the area of software development seeks to build useful applications. In that sense, this project perfectly fits my interests, as it involves the creation of software to be used in the context of scientific research.

# 2 Requirements

Below is a description of the requirements as defined by the advisor. The requirements include three main tasks as well as eight milestones; the milestones are divided into the categories *must have*, *should have*, and *nice to have*. While working on the project, some additional requirements were defined, which are described in section 2.3.

## 2.1 Main Tasks

1. Spidering (i.e. creating a full or partial local copy of) a predefined set of websites that provide sleeping disorder information as an experimental corpus.

2. Creating a Google-like search interface to the corpus that can be configured to selectively show/rank different sets of corpus sites, according to the experimental conditions under investigation.

3. Setting up an experimental environment (e.g. using the *SafeExamBrowser*, or using a proxy server) to conduct controlled experiments using the corpus (e.g. to prevent participants from accessing non-corpus sites) and to record salient data (e.g. search logs, click-stream).

## 2.2 Milestones

1. **(Must have)** A website simulating a search engine that lets users enter keywords into a search form and returns results (including snippets) from a predefined corpus of websites/links, which can then be clicked on / followed.

2. **(Must have)** A result generator and a simple way to configure it (e.g. using a text file) on a per-group basis (i.e. participants in Group 1 receive results from lists R1, R2, and R3; Group 2 participants receive results from R4, R2, and R3).

3. **(Must have)** A report describing the system's installation, setup, and architectural design.

4. **(Should have)** A result generator that can detect identical, repeated queries (or minor variations of otherwise identical queries, detectable via stop word removal and stemming) upon which it will generate the same response.

5. **(Should have)** A detailed log engine that allows the experimenter to track key experimental results for each participant, such as search terms entered, the time spent on a given result list, and any clicks on results (when, which order).

6. **(Should have)** A visual presentation of the search entry and result section that mimics a known search engine.

7. **(Nice to have)** A result generator that can handle non-related searches using a pass-through to a real search engine.

8. **(Nice to have)** A web interface to the log engine that allows for convenient inspection, analysis, and export of experimental results.

## 2.3 Additional Requirements

- It must be possible to embed the application into a survey created using the *Qualtrics* platform [16].
- Instead of linking directly to the original web page, a click on an item in the search result list should lead to a page containing its own navigation bar, so users can easily return to the search interface.
- As a result for the first query, all participants will receive the same predefined result list (configurable per test group). Only subsequent queries will be processed by the search engine.

# 3 Project Design

## 3.1 General Structure

Given the close relationships between the experiment configuration, conduction, and analysis, I decided to include most of the implied functionalities into a single web application, called *HSE (Health Search Engine)*. Given that the planned experiments are going to be conducted with Italian-speaking participants and involve web documents written in Italian, the user interface is available both in English and Italian, and document corpora can be defined in both languages.

The user management functionality of HSE allows participants to access only a search interface, while experimenters have access to pages for managing corpora and experiments. If needed, participants can be prevented from accessing non-corpus sites by restricting the browser to a whitelist based on the corpus used for a given experiment. The following subsections briefly describe the usage modalities and the related user interfaces.

## 3.2 Typical Usage Workflow

The typical workflow includes four steps: preparing the document corpora, setting up an experiment, running the experiment, and finally evaluating the resulting data. **Figure 1** shows this usage scenario. To each step corresponds a dedicated user interface.

For preparing the corpora the experimenter provides text files containing the web URLs of the chosen documents. After setting the names for the corresponding document collection, the application takes care of downloading the contents and creating the inverted indices needed for retrieval. Setting up an experiment involves defining test groups and assigning participants to each group. Moreover, for each test group, it is needed to set the document collections from which the retrieval mechanism will select the results to be displayed to the participants. The groups can be defined either manually or by uploading a configuration file. The U.I. for experiment execution includes a control button for starting, stopping, or resetting an experiment, as well as a tabular display for real-time monitoring, showing the current number of queries and clicks for each participant. Starting an experiment enables the participants to log in, and initiates the data collection mechanism; after stopping the experiment, the participants are logged out, and transient data is saved. The experiment evaluation interface allows for quick inspection through data summaries and visualizations. Moreover, it allows exporting both the raw data and the summaries as files.

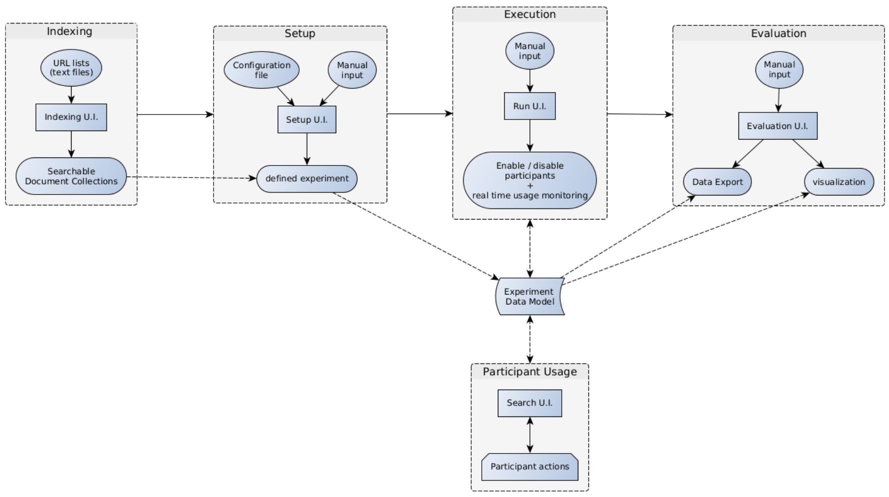
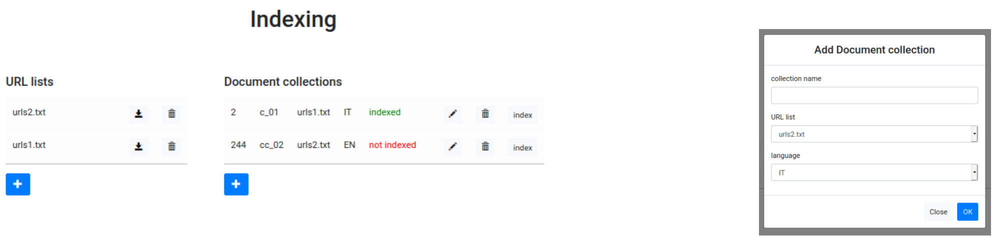


**Figure 1.** Typical usage workflow

## 3.3 Defining the Document Corpora

In order to define the document collections (corpora) to be used during subsequent experiments, an experimenter can upload text files containing lists of web URLs. The files are stored, so they can be reused for defining multiple document collections. Via a popup menu, a new document collection can be defined by providing a name, the collection's language, and the related URL list. Clicking on the "index" button initiates the indexing process, which includes data download and the creation of an index data structure for retrieval. The details of the indexing process are explained in section 6.1. **Figure 2** shows the relevant parts of the interface.


Indexing

URL lists

| | |
|---|---|
| urls2.txt | |
| urls1.txt | |

Document collections

| | | | | | |
|---|---|---|---|---|---|
| 2 | c_01 | urls1.txt | IT | indexed | index |
| 244 | cc_02 | urls2.txt | EN | not indexed | index |

Add Document collection

collection name:

URL list: urls2.txt

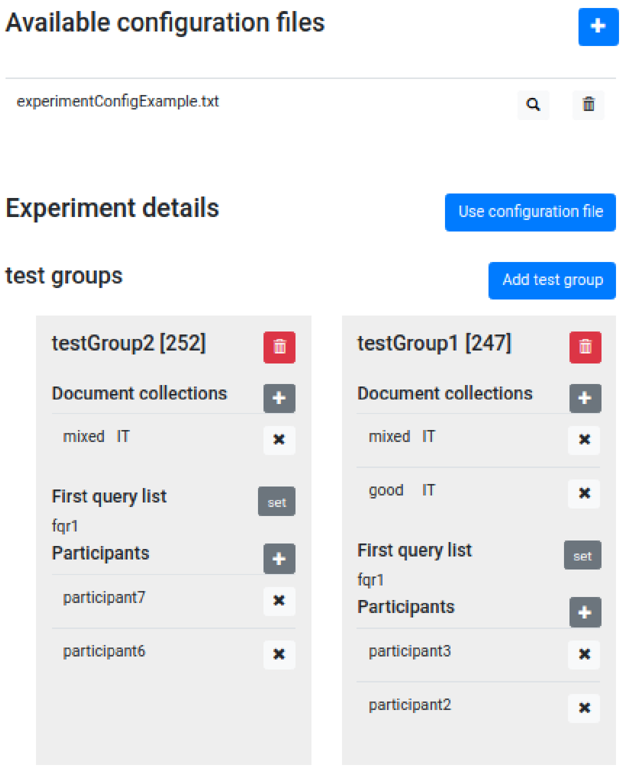
language: IT

**(a)** indexing u.i **(b)** popup for defining document collections

**Figure 2.** Indexing UI page

## 3.4 Setting up an Experiment

The UI for experiment setup allows defining the details of an experiment to be carried out. This step involves creating test groups with associated participants and document collections. Groups can be defined either manually or by using an uploaded configuration file. In both cases, the test group configuration can later be edited manually. The interface allows linking each group to a set of previously indexed document collections. Optionally a document collection can be set for each test group as predefined result list to be returned in response to the first query. If the experiment is to be executed in the context of a *Qualtrics* survey, the participants don't need to be specified, as they are defined while they take part in the survey. **Figure 3** shows this UI.


Available configuration files

experimentConfigExample.txt

Experiment details

test groups

testGroup2 [252]

Document collections

mixed IT

First query list

fqr1

Participants

participant7

participant6

testGroup1 [247]

Document collections

mixed IT

good IT

First query list

fqr1

Participants

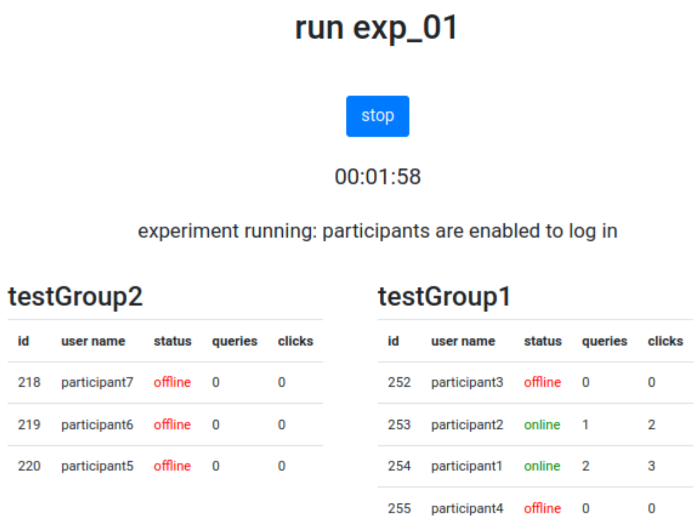
participant3

participant2

**Figure 3.** U.I. for experiment setup

## 3.5 Running an Experiment

The UI for experiment execution includes a start/stop/reset button, a timer, and a tabular display showing the current participant activities. Clicking the start button starts the timer, enables the participants to log in, and initiates the data collection process. While the experiment is running, all queries and click carried out by the participants are stored as database records including a timestamp, user id, group id, and query/document related data. The details of the data collection mechanism are described in section 4. When the stop button is clicked the participants are logged out and all transient data is saved to the database. After the experiment is complete the related evaluation page becomes available. In case something goes wrong, the experiment can be reset. This causes all collected data to be deleted, while the experiment's configuration is preserved. **Figure 4** shows the related user interface.



00:01:58

experiment running: participants are enabled to log in

testGroup2

| id | user name | status | queries | clicks |
|---|---|---|---|---|
| 218 | participant7 | offline | 0 | 0 |
| 219 | participant6 | offline | 0 | 0 |
| 220 | participant5 | offline | 0 | 0 |

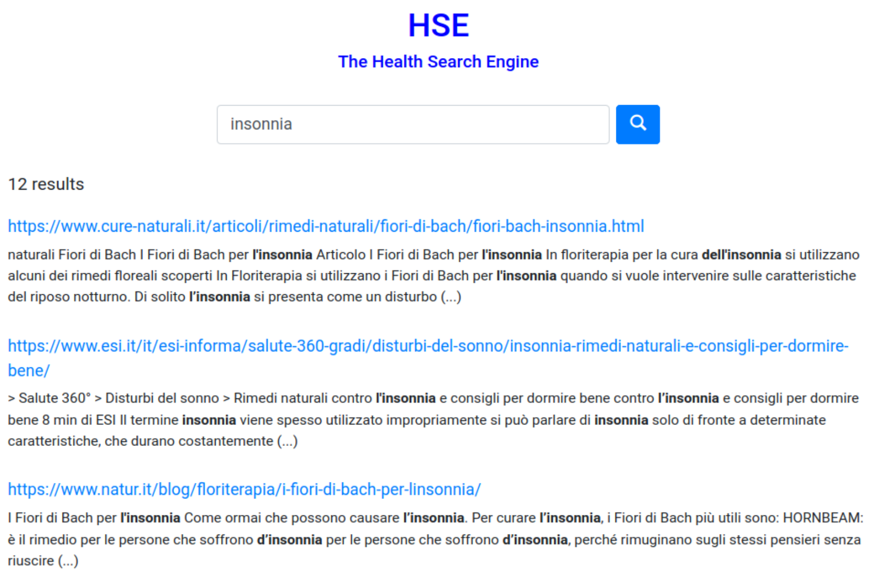
testGroup1

| id | user name | status | queries | clicks |
|---|---|---|---|---|
| 252 | participant3 | offline | 0 | 0 |
| 253 | participant2 | online | 1 | 2 |
| 254 | participant1 | online | 2 | 3 |
| 255 | participant4 | offline | 0 | 0 |

**Figure 4.** UI for experiment execution

## 3.6 Search Interface Available to Participants

The interface available to the participants looks similar to the main page of most known search engines. It simply includes a search text bar and a button for entering queries and displays the results as a list of links accompanied by short summaries (snippets) with highlighted query terms. **Figure 5** shows the UI after a query has been entered.



12 results

https://www.cure-naturali.it/articoli/rimedi-naturali/fiori-di-bach/fiori-bach-insonnia.html

naturali Fiori di Bach I Fiori di Bach per **l'insonnia** Articolo I Fiori di Bach per **l'insonnia** In floriterapia per la cura **dell'insonnia** si utilizzano alcuni dei rimedi floreali scoperti In Floriterapia si utilizzano i Fiori di Bach per **l'insonnia** quando si vuole intervenire sulle caratteristiche del riposo notturno. Di solito **l'insonnia** si presenta come un disturbo (...)

https://www.esi.it/it/esi-informa/salute-360-gradi/disturbi-del-sonno/insonnia-rimedi-naturali-e-consigli-per-dormire-bene/

> Salute 360° > Disturbi del sonno > Rimedi naturali contro **l'insonnia** e consigli per dormire bene contro **l'insonnia** e consigli per dormire bene 8 min di ESI Il termine **insonnia** viene spesso utilizzato impropriamente si può parlare di **insonnia** solo di fronte a determinate caratteristiche, che durano costantemente (...)

https://www.natur.it/blog/floriterapia/i-fiori-di-bach-per-linsonnia/

I Fiori di Bach per **l'insonnia** Come ormai che possono causare **l'insonnia**. Per curare **l'insonnia**, i Fiori di Bach più utili sono: HORNBEAM: è il rimedio per le persone che soffrono **d'insonnia** per le persone che soffrono **d'insonnia**, perché rimuginano sugli stessi pensieri senza riuscire (...)

**Figure 5.** User interface available to participants

### 3.7 Experiment Evaluation

After an experiment has been conducted, the related evaluation interface becomes available. From this page, experimenters can inspect the experiment's results and export the complete raw data or preprocessed data summaries.

The raw data can be exported either in *CSV* or *JSON* format and consists of a list of all user actions that occurred during the experiment. Each record has a timestamp, a user id, and a group id. Query event records include the query text and the proportions in which the data collections are represented in the result list. Document click event records include the document id, its URL, and the collection which the given document belongs to.

The data summaries include overall experiment statistics and per-group statistics. Moreover, the individual user histories can be exported in the same formats as the raw data. Per-experiment statistics include the total count of clicks and queries as well as averages, medians, and standard deviations for queries per user, clicks per user, clicks per query, time per query, and time per click. Per group statistics include the same metrics as the per-experiment statistics, plus totals, averages, medians, and standard deviations for clicks per document collection. Details on the raw data format and the computed statistics are described in 4.

# 4 Raw Data Generation and Computed Statistics

### 4.1 Raw Data Records

The usage tracking system is based on generating records whenever a participant performs a relevant action. The application distinguishes three kinds of usage events: session events (log in / log out), query events (generated when a participant submits a search query), and document click events (generated when a participant visits a page by clicking on an item in the search results list). All usage events include the following data fields:

- A unique id.
- A timestamp indicating the precise time when the event occurred.
- The id of the participant who triggered the event.
- The id and name of the test group which the participant belongs to.
- The event type (one of "SESSION", "QUERY", or "DOC_CLICK").

Query events contain the following additional fields:

- The query string entered by the participant.
- The total number of results retrieved.
- The number of results retrieved from each of the available document collections.

Document click events contain the following additional fields:

- The URL of the corresponding web page.
- The document's id assigned during indexing.
- The id and name of the document collection the given document belongs to.
- The document's rank (i.e. its position within the search results list).

The raw data records can be exported via the experiment's evaluation page in *CSV* or *JSON* format in order to be used for statistical analysis. Some basic metrics are computed by the application, as described in the following subsection, and displayed on the evaluation page.

## 4.2 Computed data

For each experiment the following quantities are considered:

- The total number of query events occurred.
- The total number of document click events occurred.
- Mean, median and standard deviation of the number of queries per user.
- Mean, median and standard deviation of the number of document clicks per user.
- Mean, median and standard deviation of the number of clicks per query.
- Mean, median and standard deviation of the time spent per document
  (duration between a document click event and the next usage event performed by the same participant)
- Mean, median and standard deviation of the time spent per query
  (duration between a query and the next query or logout).

The same metrics are available for each test group, allowing for interesting comparisons. Moreover, for each test group, the distribution of documents visited and time spent over the document collections from which the results are drawn is represented.

## 5 Software Architecture and Employed Frameworks

The project requirements implied building a web application including a text information retrieval system. Both aspects are highly complex and it would not be reasonable to build such an application completely from scratch, so I needed to rely on appropriate libraries and frameworks. As a general framework I chose *SpringBoot* (a *Java* framework for web applications) [8], since I had used it in past projects and knew that it includes very useful features for handling the main issues related to serving web pages, interacting with a database, and providing endpoints for *Ajax* calls and *WebSocket* services. For the information retrieval part, I chose *Apache Lucene* [1] in conjuction with the *Tika* content analysis toolkit [2] (used for text extraction). *Lucene* includes all functionalities needed for indexing and retrieval, and since it is a *Java* library, it can be easily integrated into a *SpringBoot* application. As a database management system I chose *MySql* [7]. For managing the dependencies and configuring the build process of the SpringBoot application I used the *Apache Maven* project management tool, which allows a simple configuration based on a single file (*pom.xml*). For an easy and portable deployment I configured *Maven* to create a *Docker* [5] image when the application is built; this makes the configuration and system requirements on the on the production server as simple as possible. For the client-side I used the *jQuery*[6] library in order to keep the *JavaScript* code simple and having an easy way to perform *AJAX* (Asynchronous JavaScript and XML) calls. I also employed the *Bootstrap* [4] toolkit for the graphical aspects of the front-end interfaces. The next two subsections explain how the different parts of the system interact.

### 5.1 Overall Architecture

The core of the system is the *SpringBoot* application, whose tasks include serving web content (HTML, CSS and JS files) upon HTTP requests, managing users and authentication (users with different roles have access to different interfaces), updating database records upon form submissions and *AJAX* calls, managing *WebSocket* services, and providing interfaces for uploading and downloading files. **Figure 6** summarizes the overall system architecture; the following sections explain why various interactions are required and how they are implemented using the chosen frameworks and libraries.

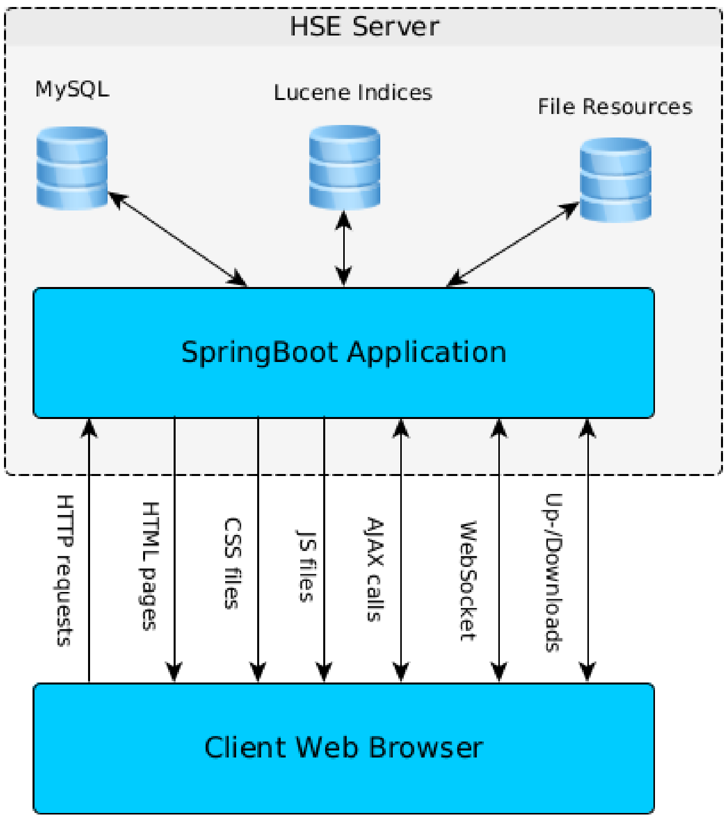


**Figure 6.** General System Architecture

### 5.1.1 **Serving Web Content**

For serving web content, the framework leverages the *Thymeleaf* [12] template engine and uses so called *Controller* classes for reacting to *HTTP* requests. Within a controller class, methods with the annotation *@RequestMapping*, are defined for responding to requests for specific *URL* paths. These methods can return either an object of the class *ModelAndView*, allowing to inject variables into the template, or simply a string representing the name of a template.

*Thymeleaf* template files consist mainly of *HTML*, but allow also syntax for useful operations to be performed server-side, such as including variables (which can be complex *Java* objects), selectively include *HTML* elements depending on conditions, include multiple *HTML* elements by iterating over a list, or defining reusable elements (fragments), which can be imported in multiple other template files. *JavaScript*, *CSS*, and other static files can be included using classical *HTML* syntax. The template engine automatically finds and loads the templates as well as static files, as long as they are placed in the project's *resources* directory.

### 5.1.2 **REST Endpoints for *Ajax* requests**

In several parts of the application, the front-end code needs to send information to the server (e.g. when defining experiments and test groups), or retrieve information from the server (e.g. the current state of an experiment). This is done via *Asynchronous JavaScript and XML* (*Ajax*). The main advantage of *Ajax* is that it allows interactions between server and client that are independent from displaying a web page [13]. This makes the application much more efficient, because the alternative would be reloading the same page whenever some data must be sent or retrieved. There are several possible implementations for *Ajax*, but the one most frequently used nowadays is based on *XMLHttpRequest* and transfers data in *JavaScript Object Notation* (*JSON*).

Systems that use *Ajax* for transferring data and consider the semantics of the *HHTP* request's method (GET, POST, PUT, DELETE and others) are often referred to as *REST API* or *RESTful* service, where *REST* stands for "Representational State Transfer" [14]. Implementing the server-side of such a system can be done quite easily in *SpringBoot*, since it includes a mechanism designed specially for this purpose: the *Jackson ObjectMapper* converts *Java objects* to *JSON* strings and vice versa, so when writing a *Controller* class, all data is represented in *Java*, while the front-end sends and receives *JSON* [3].

### 5.1.3 **WebSocket for interactive communication**

Though *Ajax* works perfectly for pulling information from the server to the client or pushing it from the client to the server, the actions can only be triggered on the client-side. But in our application there are a few situations in which it must work the other way around. One such case is updating the participant actions displayed to experimenters on the experiment's run page during execution: when a user performs a query or clicks on a link the corresponding event is stored on the server, and the experimenter's page must be somehow notified that an event occurred.

The classical way to solve this issue (before the *WebSocket* protocol existed) was to poll for the current state. In our case this would have meant performing an *Ajax* call at regular and possibly short intervals (e.g. once a second) in order to retrieve the current state of the experiment (including the participant's action counts) and update the displayed data. This approach works, but is quite inefficient, since information is transferred and computations are done even if nothing has changed.

The *WebSocket* protocol offers an elegant alternative [15]: it provides full-duplex communication channels over a single *TCP* connection by using a modification of the *HTTP* protocol. Including the *WebSocket* functionality in a web application is quite simple. In *SpringBoot* it requires adding the dependency for *spring-boot-starter-websocket*, creating a class which implements the *WebSocketMessageBrokerConfigurer* interface in order to configure the channels, and send messages by using the *@SendTo* annotation or an instantiation of *SimpMessagingTemplate*. On the front-end, the *SockJs* library provides functions for connecting and subscribing to specific channels, and it is straight-forward to execute any *JavaScript* code whenever a message is received.

### 5.1.4 **File Upload and Download**

In several situation my application must enable uploading files and storing them on the server or downloading files generated on the server to the client system: experimenters need to upload *URL* lists for creating document collections as well as configuration files for defining test groups, and they must be able to download the data collected during an experiment.

File upload can be done in *SpringBoot* by setting a parameter of type *MultipartFile* for a controller method. This object has a method *getInputStream()*, which returns a standard *Java* stream that can be written to the file system e.g. using the *java.nio.file.Files* API. File download is equally simple: the *Java InputStream* from a file can be copied to the *OutputStream* associated with a *HttpServletResponse* object (passed as parameter to a *Controller* method) e.g. using

*FileCopyUtils* [11]. On the front-end one can upload files via a form including an *HTML* input tag with `type="file"`. Files can be downloaded using a link.

#### 5.1.5 Database interactions with *Spring Data JPA*

The application heavily relies on storing and retrieving structured data: user data needs to be stored for enabling access control, metadata about document collections and experiment settings must be stored for regulating the experiment's execution process, and during execution the participant's actions must be logged. The best way for implementing this kind of functionality is by using a *Database Management System (DBMS)* which allows for efficient storage and retrieval. I chose *MySql* for this purpose because I already had some experience with it, and there is plenty of documentation online on how to interact with it from a *SpringBoot* application [9] [10].

In fact *SpringBoot*'s *Data JPA* features allow to define entities as *Java* classes without ever using the *DBMS* directly. The library automatically generates the corresponding tables: classes are mapped to tables, and data members are mapped to fields. Also advanced functionalities such as one-to-many or many-to-many realationships can be defined in pure *Java*; even inheritance structures are mapped automatically from *Java* to the *DBMS*. The details can be configured using annotations. Data access is managed via repositories which are defined as *Java* interfaces.

#### 5.1.6 Integrating *Lucene* indexing and retrieval

The *Lucene* library can be easily added to the project by specifying it as a dependency in the *Maven* configuration. Once the library is available, its classes can be instantiated and methods can be called. Both indexing and retrieval can be implemented based on classical *Java* data structures. Details on how I used the library for the specific needs of my application are described in section 6.

## 5.2 Internal Architecture of the *SpringBoot* Application

*SpringBoot* does not enforce a particular architecture or package structure, but a pattern which I often encountered is a combination of a layered structure in which each layer communicates only with the layer directly below, and a package-per-feature structure, where semantically related classes are placed in the same package. Our application roughly follows this approach, using the following layers:

**Web Layer** Top layer of the hierarchy, containing *Controller* classes with methods to handle web requests and API calls. This layer contains a single package named "endpoints" and uses the functionalities provided by the service layer.

**Service Layer** This layer provides abstractions for lower level functionalities such as file access, database interactions, user management, and implementations of custom features like indexing, retrieval, and experiment configuration / evaluation. Also this layer consists of a single package named "services", containing classes for accessing the specific functionalities provided by the data access and processing layer below.

**Data / Processing Layer** This layer contains the actual implementation of the features exposed by the service layer, and is divided into packages grouping related functionalities. The following packages belong to the implementation layer: "db" with sub-packages "entities" and "repositories" (providing access to the *DBMS*), "storage" (containing classes for reading and writing files in several formats), "indexing" and "retrieval" (for using the features of the *Lucene* library).

Functionalities which are used application-wide are to be considered outside the layered structure. These include custom exceptions and an exception-handling mechanism (grouped in a package named "exceptions"), as well as global configuration classes (grouped in a package named "config"). **Figure 7** shows the layered architecture of the application: green rectangles represent packages, blue ovals represent classes, solid lines represent connections through instantiation or method calls, dashed lines indicate other interactions. In the following subsections, each layer is described in more detail.

## 5.3 The Web Layer: Controller Classes for Handling *HTTP* requests

The *Web Layer* contains the single package `endpoints`; functionalities are split into classes depending on the user interface from which they are supposed to be called and the services they depend on. The `endpoints` package contains the following classes: `AdminController`, `AuthController`, `ExperimentsController`, `FromSurveyController`, `IndexingController`, and `SearchController`.

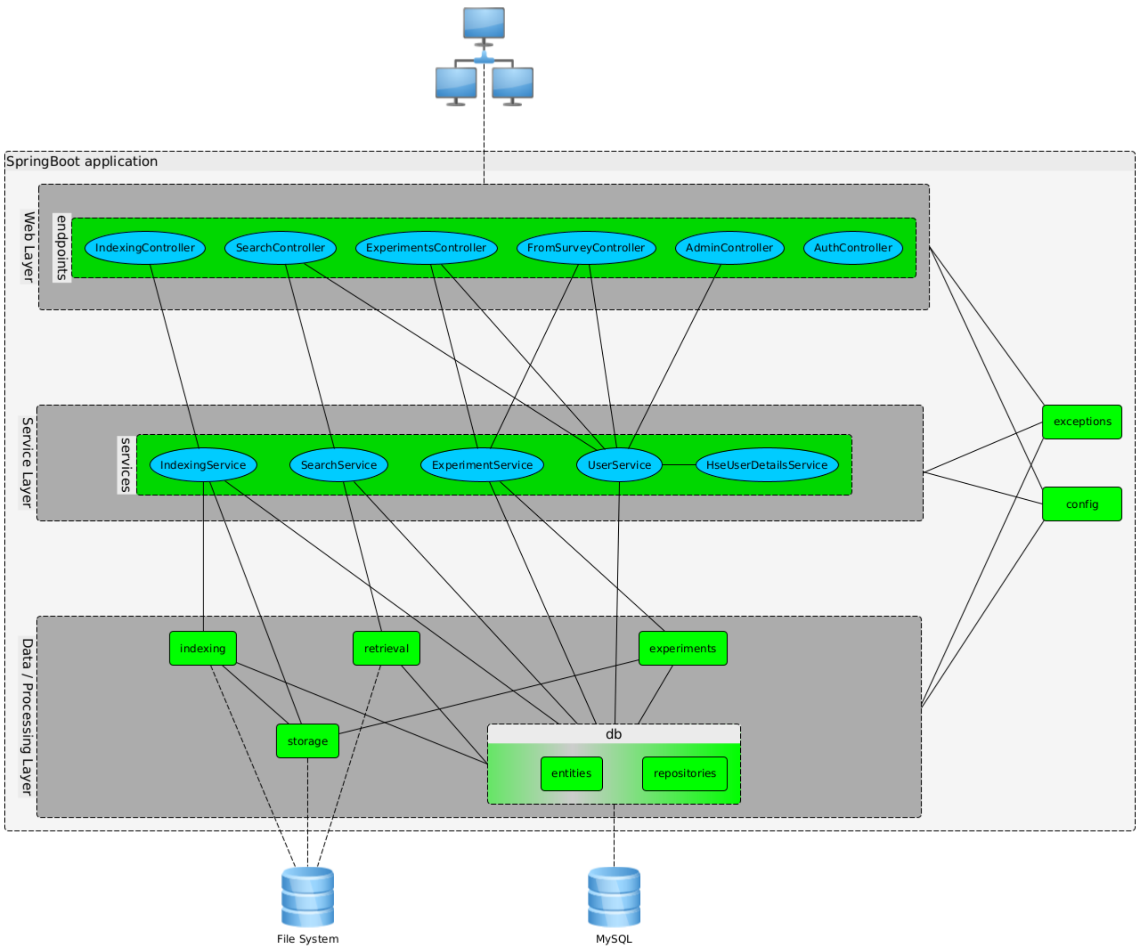


**Figure 7.** Layered architecture of the back-end application

### 5.3.1 AdminController

The class `AdminController` handles requests to *URLs* with prefix “`/admin`” available to users with `Administrator` role. The functionalities of this controller include serving the `admin` *UI* page and reacting to *Ajax* requests coming from it. The purpose of this interface is creating, updating or removing administrators, experimenters, and participants by using methods of the `UserService` class from the service layer.

The class contains the following public methods:

**`getAdminUi`** (annotated with `@GetMapping("/ui")`)

serves the `admin` *UI* page. It takes no explicit parameters and returns an object of class `ModelAndView`. The lists of all users, separated by role, are added to the returned object as template variables.

**`postAdministrator`** (annotated with `@PostMapping("/administrators")`)

takes a parameter of type `Administrator`, which is passed as request body in an *Ajax POST* request. It adds the corresponding new administrator to the database by calling the `UserService` and returns a `ResponseEntity` containing the newly created user.

***POST* mappings for `Experimenter` and `Participant`** analogous to the `postAdministrator` method described above, but for handling users with `Experimenter` or `Participant` role.

**`putAdministrator`** (annotated with `@PutMapping("/administrators")`) takes a parameter of type `Administrator`, which is passed as request body in an *Ajax PUT* request. It updates the corresponding user in the database (e.g. changing the user name or password) and returns the updated object wrapped in a `ResponseEntity`.

***PUT* mappings for `Experimenter` and `Participant`** analogous to the `putAdministrator` method described above, but for handling users with `Experimenter` or `Participant` role.

**`deleteAdministrator`** (annotated with `@DeleteMapping("/administrators")`) takes a parameter of type `Administrator`, which is passed as request body in an *Ajax DELETE* request. The corresponding user is deleted from the database. The method returns a `ResponseEntity` containing the deleted user.

***DELETE* mappings for `Experimenter` and `Participant`** analogous to the `deleteAdministrator` method described above, but for handling users with `Experimenter` or `Participant` role.

**delteAllExperimenters** (annotated with `@DeleteMapping("/experimenters/all")`)

takes no explicit parameters and removes all users with `Experimenter` role from the database. The method returns a `ResponseEntity` containing a string to confirm that the operation has succeeded.

**delteAllParticipants** Same as the method described above, but for deleting all users with `Participant` role.

### 5.3.2 AuthController

This is a minimal controller class containing a single method for serving the login page.
The method returns an object of class `ModelAndView`.

### 5.3.3 SearchController

The `SearchController` is in charge of serving the main search engine *UI* available to participants, and redirecting them to a specific search result when they click on a link or to a logout page if the experiment is terminated. These actions are performed by calling the `SearchService` and `UserService` from the service layer through the following methods:

**`getSearchUi`** (annotated with `@GetMapping("/")`)

This method serves the main search engine *UI*, optionally including the results of the last query submitted. It takes two parameters: an object of class `org.springframework.security.core.userdetails.User` which is generated by the framework and is needed for identifying the calling participant, and optionally a `String` representing the query entered in the search form.

If the query string is empty and no query has been submitted before by the given user, the search *UI* with empty results list is returned; if a query was submitted before, the last results list is added to the `ModelAndView`. If the query string is not empty, the method checks whether it is the first query (in which case if a predefined first-query result list for exists it is used, otherwise the query is processed by the retrieval system and the returned result list is added to the `ModelAndView`). If the query is not the first one and has different content from the first one, the query is processed and the result list is added to the `ModelAndView`.

These operations are performed by calling the methods `handleFirstQuery`, `handleRepeatedQuery`, and `handleNewQuery` from the `SearchService` (see section 5.4.3).

**`getResultDoc`** (annotated with `@GetMapping("/doc")`)

This method serves the result page that is displayed upon clicking on a link in the results list. It take the following two parameters: the *URL* of the original web document and its file type (*PDF* documents are displayed using a slightly different mechanism than the one used for *HTML* documents). These parameters are added to the returned `ModelAndView`.

**`postBrowseEvent`** (annotated with `@GetMapping("/doc")`)

This method is to be called via an *Ajax POST* request in order to generate and store a document click event. It takes the `SearchResult` object (see section 6.2) corresponding to the clicked link and the `User` object corresponding to the participant as parameters.

### 5.3.4 IndexingController

The class `IndexingController` implements all interactions needed for managing *URL* lists and document collections. It serves the indexing *UI* page and provides endpoints for handling file upload / download and *Ajax* requests related to the creation, modification, and deletion of document collections by responding to *HTTP* requests with *URL* prefix `/indexing`. It uses the `IndexingService` from the service layer and contains the following public methods:

**`getIndexingUi`** (annotated with `@GetMapping("/ui")`)

serves the indexing *UI* page. The file names of all saved `URL` lists and the objects corresponding to all stored document collections are added to the returned `ModelAndView`.

**`postUrlList`** (annotated with `@PostMapping("/urlLists")`)

takes an object of type `MultipartFile` representing a text file as parameter. The file is stored by calling the `addUrlList` method of the `IndexingService`.

**`deleteUrlList`** (annotated with `@DeleteMapping("/urlLists")`)

takes the name of the *URL* list to be deleted representing a as parameter and deletes the corresponding file by calling the `removeUrlList` method of the `IndexingService`.

**`downloadUrlList`** (annotated with `@GetMapping("/urlLists/dl")`)

takes the `HttpServletResponse` (provided by the framework) and the name of a *URL* list as parameters and makes the corresponding file available for download. It does so by getting the file as *Java* `InputStream` from the `IndexingService` and copying it to the response's `OutputStream`.

**`postDocCollection`** (annotated with `@PostMapping("/docCollections")`)

responds to an *Ajax* request containing a `DocCollection` object as request body. It takes the object as parameter, persists it in the database using the `IndexingService`'s addDocCollection method, and returns the saved object wrapped in a `ResponseEntity`.

**`updateDocCollection`** (annotated with `@PutMapping("/docCollections")`)

responds to an *Ajax* request containing a `DocCollection` object as request body. It takes the object as parameter, updates it's persistent version in the database using the `IndexingService`'s updateDocCollection method, and updated the saved object wrapped in a `ResponseEntity`.

**`deleteDocCollection`** (annotated with `@DeleteMapping("/docCollections")`)

responds to an *Ajax* request containing a `DocCollection` object as request body. It takes the object as parameter, deletes it from the database using the `IndexingService`'s removeDocCollection method, and updated the deleted object wrapped in a `ResponseEntity`.

**`buildIndex`** (annotated with `@PostMapping("/buildIndex")`)

responds to an *Ajax* request containing a `DocCollection` object as request body. The indexing process for the given collection is initiated (see section 6.1), and an object of type `IndexingResult`, which summarizes the operations results, is returned wrapped in a `ResponseEntity`. The process is performed by calling the `buildIndex` method from the `IndexingService`.

### 5.3.5 **ExperimentsController**

The class `ExperimentsController` handles requests to *URL*s with prefix "`/experiments`" available to users with `Administrator` or `Experimenter` role. Its purposes are serving the *UI* pages for creating, configuring, running, and evaluating experiments, and handling the related *Ajax* requests. For performing these operation, the class uses the functionalities provided by the `ExperimentService` and `UserService` from the service layer.

**Serving The *UI* Pages** The pages for managing experiments are served via the methods `getExperimentsUi()`, `getExperimentsSetupUi()`, `getExperimentsUi()`, `getExperimentsRunUi()`, and `getExperimentsEvalUi()`. The first method takes no explicit parameters, while the others require the experiment's id to be passed as a `URL` parameter in the `HTTP` request because the corresponding pages relate to a single experiment at a time. All four methods return an object of class `ModelAndView`.

**REST API for creating, updating and deleting experiments** Experiments can be created, updated and deleted via *AJAX* calls using the `HTTP` methods *POST*, *PUT*, and *DELETE* and providing the experiment object as request body.

The methods responsible for these actions are `postExperiment`, `putExperiment`, and `deleteExperiment`, each taking a parameter of type `Experiment` and returning an object of the same type wrapped in a `ResponseEntity`.

### 5.3.6 **FromSurveyController**

This controller handles the interaction required for integrating the application into a *Qualtrics* survey; it responds to *HTTP* requests with *URL* prefix "/from_survey". It provides a method for serving the search *UI* while automatically logging in a participant, as well as a *REST API* which allows for creating participants and checking if a given experiment is running. These operations are performed by calling the `UserService` and `ExperimentService` from the service layer. Below is a brief description of each method:

**getSearchUiFromSurvey** (annotated with `@GetMapping("/")`)

Serves the search *UI* page for participants from a *Qualtrics* survey. takes a user id and the survey *URL* as parameters. The id is needed for logging in the corresponding participant and load the experiment data, while the *URL* is stored for later redirecting the participant back to the survey.

the method performs the following actions:

- The user data corresponding to the given id is retrieved from the database via the `UserService`.
- The experiment is loaded using the experiment id stored in the user data.
- If the given experiment is not running, the user is redirected to an error page by returning the corresponding `ModelAndView` object.

  This is a fallback for a situation that should not occur, since the survey is supposed to check whether the experiment is running before redirecting the user to the application (see **appendix B.6**).
- The participant is logged in and the experiment is updated. If these operations succeed, a *WebSocket* message is sent for updating the experiment's run *UI* and the user is redirected to the search *UI* page by returning the corresponding `ModelAndView` object. Otherwise an exception is returned in the *HTTP* response.

## 5.4 The Service Layer: Service Classes Providing the Core Functionalities

The service layer acts as an interface between the data / processing layer and the web layer. It is divided into classes which group related functionalities:

### 5.4.1 UserService

The `UserService` allows for creating, updating, and removing users by interacting with the database repository interfaces from the layer below (package `db`. Following are its public methods:

**`long userCount()`** returns the total number of saved users.

**`long administratorCount()`** returns the total number of saved administrators.

**`long experimenterCount()`** returns the total number of saved experimenters.

**`long participantCount()`** returns the total number of saved participants.

**`boolean userExists(int id)`** returns `true` if a user with the given id exists, false otherwise.

**`boolean userExists(String name)`** returns `true` if a user with the given user name exists, false otherwise.

**`List<HseUser> allUsers()`** returns all saved users.

**`List<Administrator> allAdministrators()`** returns all saved administrators.

**`List<Experimenter> allExperimenters()`** returns all saved experimenters.

**`List<Participant> allParticipants()`** returns all saved participants.

**`HseUser findUser(int id)`** returns a saved user given its id.

**`HseUser findUser(String userName)`** returns a saved user given its user name.

**`Administrator findAdministrator(int id)`** returns a saved administrator given its id.

**`Administrator findAdministrator(String userName)`** returns a saved administrator given its user name.

**`Experimenter findExperimenter(int id)`** returns a saved experimenter given its id.

**`Experimenter findExperimenter(String userName)`** returns a saved experimenter given its user name.

**`Experimenter findParticipant(int id)`** returns a saved participant given its id.

**`Participant findParticipant(String userName)`** returns a saved participant given its user name.

**`Administrator addAdministrator(Administrator administrator)`** adds a new administrator to the database.

**`Experimenter addExperimenter(Experimenter experimenter)`** adds a new experimenter to the database.

**`Participant addParticipant(Participant participant)`** adds a new participant to the database.

**`Participant addParticipant(Participant participant)`** adds a new participant to the database.

**`Participant addSurveyParticipant(int groupId)`** adds a new participant to the database and sets its user name and password to defaults.

**`Administrator updateAdministrator(Administrator administrator)`** Updates the saved administrator with matching id.

**`Experimenter updateExperimenter(Experimenter experimenter)`** Updates the saved experimenter with matching id.

**`Participant updateParticipant(Participant participant)`** Updates the saved participant with matching id.

**`void removeUser(int id)`** removes the save user with the given id.

**`void clearExperimenters()`** removes all saved experimenters.

**`void clearParticipants()`** removes all saved participants.

**`void removeAdministrator(Administrator administrator)`** deletes the given administrator.

**`void removeExperimenter(Experimenter experimenter)`** deletes the given experimenter and its experiments.

**`void removeParticipant(Participant participant)`** deletes the given participant.

### 5.4.2 IndexingService

The main purpose of the `IndexingService` is providing an interface to the *Lucene* based functionalities functionalities implemented in the data / processing layer by using classes from the `indexing` package. It also uses classes from the `storage` package for saving and loading *URL* lists, as well as classes from the `db` package for updating metadata related to document collections. Following are its public methods:

**`List<String> savedUrlLists()`** returns the file names of all saved url lists.

**`void addUrlList(MultipartFile file)`** adds a new url list file.

**`void removeUrlList(String fileName)`** removes a stored url list given its file name.

**`InputStream getUrlListFile(String fileName)`** returns the `InputStream` of a save url list file (for making it availablefor download).

**`List<DocCollection> docCollections()`** returns all defined document collections.

**`DocCollection addDocCollection(DocCollection docCollection)`** adds a new document collection to the database.

**`DocCollection updateDocCollection(DocCollection docCollection)`** updates an existing document collection.

**`void removeDocCollection(DocCollection docCollection)`** removes an existing document collection.

**`IndexingResult buildIndex(DocCollection docCollection)`** starts the indexing process for the given document collection.

### 5.4.3 SearchService

The `SearchService` provides access to the *Lucene* based search functionalities implemented in the `retrieval` package. It also uses classes from the `db` package in order to gather test-group based information about which document collections the results are to be retrieved from, as well as metadata about the given document collections. Following are its public methods:

**`SearchResultList handleNewQuery(String query, Participant participant)`** processes the query string using the retrieval system, based on the document collections associated with the given participant's test group. If the query string is equal to the first query entered and a predefined result list is defined, the predefined result list is returned.

**`SearchResultList handleRepeatedQuery(String query, Participant participant)`** to be called when the query string is the same as in the immediately previous query. This situation occurs if the search *UI* page is reloaded or the participant gets back to it after having visited a document.

**`SearchResultList handleFirstQuery(String query, Participant participant)`** to be called when a participant submits the very first query. If a predefined result list is defined, the predefined result list is returned, otherwise the query is processed by the retrieval system.

**`void addDocClickEvent(SearchResult searchResult, Participant participant)`** stores a document click event in the database. This method is to be called from the web layer when a participant visits a link in the results list. (see section 5.3.3).

### 5.4.4 **ExperimentService**

The `ExperimentService` provides all functionalities related to experiments and test groups, including their definition, configuration, execution, and evaluation. The class interacts mainly with the `experiments` and `db` packages. Following are its public methods:

**`List<Experiment> allExperiments()`** returns all defined experiments.

**`Experiment findExperiment(int id)`** returns a saved experiment given its id.

**`TestGroup findTestGroup(int id)`** returns a saved test group given its id.

**`List<Experiment> findByExperimenter(Experimenter experimenter)`** returns all saved experiments belonging to the given experimenter.

**`Experiment addExperiment(Experiment experiment)`** adds a new experiment to the database.

**`Experiment updateExperiment(Experiment experiment)`** updates an existing experiment.

**`void deleteExperiment(Experiment experiment)`** deletes an existing experiment.

**`TestGroup addTestGroup(TestGroup testGroup)`** adds a new test group to the database.

**`TestGroup updateTestGroup(TestGroup testGroup)`** updates an existing test group.

**`void deleteTestGroup(TestGroup testGroup)`** deletes an existing test group.

**`Experiment configureTestGroups(Experiment experiment, String configFileName)`** defines the test groups and participants for the given experiment, based on the configuration file specified by its file name.

**`List<DocCollection> getIndexedDocCollections()`** returns all indexed (therefore usable) document collections.

**`List<String> savedConfigFiles()`** returns the names of all available configuration files.

**`void addConfigFile(MultipartFile file)`** stores an uploaded configuration file.

**`void removeConfigFile(String fileName)`** deletes a stored configuration file given its file name.

**`InputStream getConfigFile(String fileName)`** returns the `InputStream` from a configuration file given its name (making it available for download).

**`Experiment startExperiment(Experiment experiment)`** starts an experiment's execution. The related participants are enabled to log in and the experiment's data fields is updated.

**`Experiment stopExperiment(Experiment experiment)`** starts an experiment's execution. The related participants are logged out and the experiment's data fields is updated.

**`Experiment resetExperiment(Experiment experiment)`** All related usage events are deleted and the experiment's state is set to READY.

**`InputStream rawResultsCsv(Experiment experiment)`** makes the experiment's collected raw data available as `InputStream` for download in *CSV* format.

**`InputStream rawResultsJson(Experiment experiment)`** makes the experiment's collected raw data available as `InputStream` for download in *JSON* format.

**`InputStream summaryJson(Experiment experiment)`** makes a summary of the experiments data available as `InputStream` for download in *JSON* format.

**`InputStream userHistoriesCsv(int groupId)`** makes the individual usage histories of each participant belonging to the given test group available as `InputStream` for download in form of a *.zip* directory containing per-user *CSV* files.

**`InputStream userHistoriesJson(Experiment experiment)`** makes the usage histories of all participants in the experiment available as `InputStream` for download in *JSON* format.

**`ExperimentSummary experimentSummary(Experiment experiment)`** returns a summary of the experiment's collected data as *Java* object.

### 5.4.5 HseUserDetailsService

`HseUserDetailsService` is an auxiliary service class needed for the security configuration in the `config` package. Its purpose is to define the relation between the custom user class `HseUser` and the user management system internal to the *Spring* framework. It implements the framework's interface `UserDetailsService` with the method `loadUserByUsername(String userName)`.

## 5.5 The Data / Processing Layer: Implementation of the Core Functionalities

### 5.5.1 Interactions with the data base (package `db`)

As mentioned in section 5.1.5, access to the *MySQL* database is managed via *SpringBoot*'s *Data JPA* features. It's general working principle is to define entities as *Java* classes annotated with `@Entity` and allow transactions via *Java* interfaces annotated with `@Repository`, extending the `CrudRepository` interface defined in the framework. I grouped the entities in the sub-package `db.enities` and the interfaces in the sub-package `db.repositories`.

Below is a list of all defined entities, grouped according to the feature sets they are related to:

**Entities for user management** the following classes define the different user types and roles

- `HseUser`: super-class for the user class hierarchy. It contains fields pertinent to all user entities (administrators, experimenters, and participants). The fields include a unique id, a user name, a password a boolean indicating whether the given user is active, as well as a one-to-many relation to entities of type `Role`.
- `Administrator`: subclass of `HseUser` with no additional fields.
- `Experimenter`: subclass of `HseUser` with a one-to-many relation to a set of `Experiment` entities.
- `Participant`: subclass of `HseUser` with a many-to-one relation to a `TestGroup` entity and several data fields needed during experiment execution and evaluation. The fields include the id and name of the experiment and the test group which the participant belongs to, a boolean indicating whether the participant is currently online, counters for submitted queries and visited documents, and strings storing the first and last query submitted. There is also a field which is used only if the experiment is conducted in the context of a *Qualtrics* survey, which stores the *URL* to which the user is going to be redirected at the end of the experiment.
- `Role` Roles are defined as separate entities, so if needed, multiple roles can be assigned to the same user.

**Entities for managing experiments** the following classes represent entities related to experiments:

- `Experiment`: this entity contains fields for the experiment's metadata and relations to test groups (which in turn are related to participants), usage events, and the assigned experimenter. The data fields include a unique id, a title, a status (enum with values NOT_READY, READY, RUNNING, or COMPLETE), a mode (enum with values STAND_ALONE, or QUALTRICS), the date when the experiment was created, the date when it was conducted, its start and stop time, and its duration.
- `TestGroup`: an entity representing a test group within an experiment. It has a many-to-one relation to an experiment, a one-to-many relation to a set of participants, a one-to-many relation to the associated document collections, and several fields for holding relevant metadata. The fields include a unique id and a name.

- `DocCollection`: an entity for holdin data related to document collections. Its data fields include an id, a name, a language (IT or EN), the file name of the *URL* list the collection is based on, a boolean indicating whether the collection has been indexed, and the name of the directory where the associated *Lucene* index is stored.

**Entities for storing usage events** the following classes represent entities needed by the usage logging mechanism:

- `UsageEvent`: super-class for the usage event class hierarchy. It contains a many-to-one relation to an experiment and data fields which are common to all sub-classes. The fields include an id, a type (enum with values SESSION, QUERY, or DOC_CLICK), a timestamp, the id of the user who generated the event, as well as the id and name of the test group which the user belongs to.
- `SessionEvent`: sub-class of `UsageEvent` for representing participant's log-in and log-out actions.
- `QueryEvent`: sub-class of `UsageEvent` for representing queries submitted by participants. This entity stores the submitted query string and the number of results retrieved for the given query. It also has a one-to-many relation to entities of class `QueryStat` which hold information about the distribution of the search results over the available document collection.
- `QueryStat`: auxiliary entity holding information about the number of results retrieved from a particular document collection upon a given query. It has a many-to-one relation to a `QueryEvent`, as well as data fields for storing the collection's id and name and the number of retrieved results.

### 5.5.2 Interactions with the file system (package `storage`)

Access to the file system for storing and retrieving *URL* lists and configuration files is implemented in the class hierarchy shown in **Figure 8**. Following are brief descriptions of the classes and their public methods.

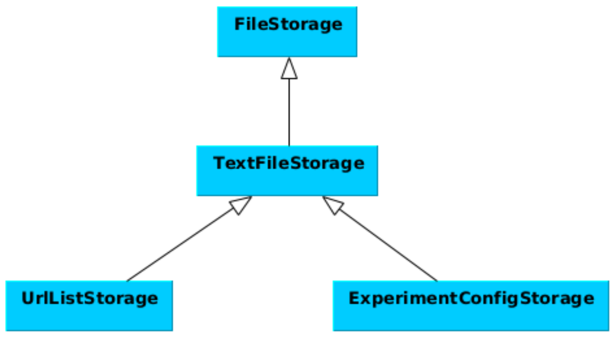


**Figure 8.** Class Hierachy for File Storage

**FileStorage** super-class at the top of the hierarchy, implementing general I/O operations for reading, writing and deleting files or directories by using the `java.io` and `java.nio` packages. It includes the following public methods:

- `void store(MultipartFile file, Path dirPath)`
  stores the given `MultipartFile` in the directory specified by `dirPath`.
- `void store(InputStream inputStream, Path filePath)`
  stores the contents of the given `InputStream` in a file specified by `filePath`.
- `void store(String content, Path filePath)`
  stores the contents of the given string in a file specified by `filePath`.
- `void store(MultipartFile file, String dirPathString)`
  stores the given `MultipartFile` in the directory specified by `dirPathString`.
- `void delete(Path filePath)`
  deletes the file at the specified path.
- `InputStream getInputStream(Path filePath)`
  returns the file at the specified path as `InputStream` (to be used for download).

- `void clearDirectory(Path dir)`
  removes all contents of the given directory, including sub-directories.
- `void removeDirectory(Path dir)`
  recursively removes a directory and all its contents, including sub-directories.

**TextFileStorage** Intermediate class, specialized for handling text files. It includes the following public methods:

`void storeTextFile(MultipartFile file, Path directory)`
stores a `MultiprtFile` in the given directory.

`void deleteTextFile(String fileName, Path directory)`
deletes the file with the given name and location.

`List<String> getTextLines(String fileName, Path directory)`
returns the content of a text file as list of strings.

`List<String> listTextFiles(Path directory)`
returns a the names of the files in the given directory as list.

`InputStream getTextFileAsStream(String fileName, Path directory)`
returns the file at the given name and location as `InputStream`.

**UrlListStorage** Sub-class specialized for handling the `URL` list files used in the application. All files are stored and retrieved from the directory for `URL` lists defined in the application's configuration. It extends `TextFileStorage` and contains the following public methods:

- `void storeUrlList(MultipartFile file)`
  stores the given `MultipartFile` in the `URL` lists directory.
- `void deleteUrlList(String fileName)`
  deletes the file with the given name from the `URL` lists directory.
- `List<String> getUrlLines(String fileName)`
  returns the contents of the file with the given name as list (one string per line).
- `List<String> listUrlFiles()`
  returns the names of all available `URL` list files as list.
- `InputStream getUrlFileAsStream(String fileName)`
  returns the file with the given name as `InputStream`.

**ExperimentConfigStorage** Sub-class specialized for handling the experiment configuration files used in the application. All files are stored and retrieved from the directory for this kind of files defined in the application's configuration. It extends `TextFileStorage` and contains the following public methods:

`void storeConfigFile(MultipartFile file)`

stores the given `MultipartFile` in the defined directory.

`void deleteConfigFile(String fileName)`

deletes the file with the given name.

`List<String> getConfigLines(String fileName)`

returns the contents of the file with the given name as list (one string per line).

`List<String> listConfigFiles()`

returns the names of all available configuration files as list.

`InputStream getConfigFileAsStream(String fileName)`

returns the file with the given name as `InputStream`.

### 5.5.3 Document indexing functionalities (package `indexing`)

The process of obtaining a searchable inverted index from a `URL` list is handled by the classes in the `indexing` package. The `IndexingService` from the service layer calls the `IndexBuilder`, which uses the other classes in the `indexing` package via composition in order to accomplish the sub-tasks of downloading the documents, extracting their text content, building the actual index and storing the relevant metadata in the `DocumentCollection` object which represents the index at application level. **Figure 9** shows the involved classes which are briefly described below; for details on how the indexing process is implemented see section 6.1.

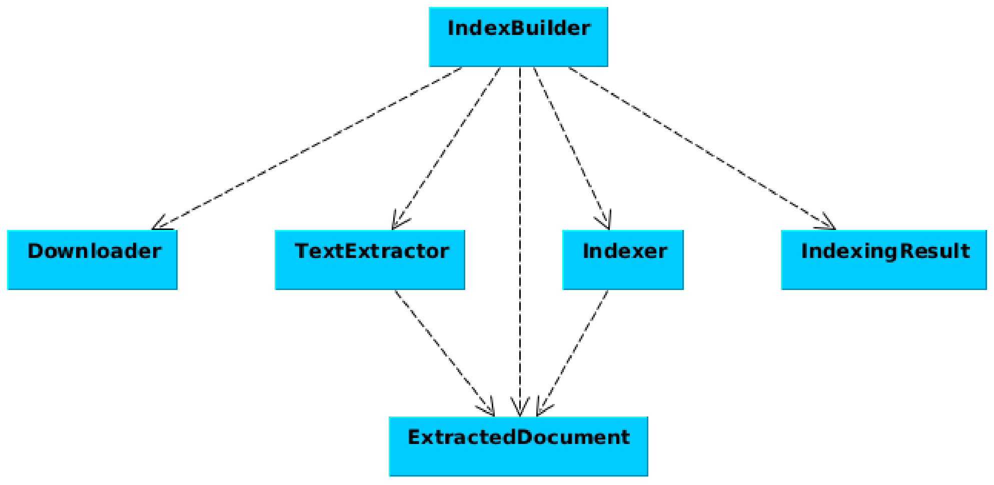


**Figure 9.** Classes in the indexing package

**`IndexBuilder`** This class handles the indexing process by instantiating the other classes and calling their methods in a loop for each document to be indexed. It has the single public method `buildIndex`, which takes a `DocumentCollection` object (see section 5.5.1) as parameter, processes the related documents, and returns a `IndexingResult` object which summarizes the the actions performed.

**`Downloader`** Is a very simple class containing a single method for downloading a document from the web. The method called `fetch` takes a `URL` string as parameter and returns an `InputStream` containing the document data. It works by calling the `openStream()` method of the `java.net.URL` class.

**`TextExtractor`** extracts the text content from `HTML` or `PDF` files by leveraging the features of the *Apache Tika* library. it exposes two public methods: `extractHtml` and `extractPdf`. Both take an `Inputstream` containing the document's raw data and a string representing the document's `URL` (to be added to the returned object), and return an instance of `ExtractedDocument`, which contains the extracted text as string along with some metadata to be added to the index.

**`Indexer`** creates the actual inverted index by using the features of the `Lucene` library. It contains the following public methods which are called from the `IndexBuilder`:

- `void setUp(Path dir, String language)` Initializes the index directory with the given path, sets up an `Analyzer` for the specified language (EN or IT), and an `IndexWriter`.
- `void addDocument(ExtractedDocument source)` adds a document to the previously initialized index. This method is called repeatedly from the loop in the `IndexBuilder`.
- `void tearDown()` is to be called when all documents belonging to the collection have been indexed. Its purpose is to close open files and streams by calling the `close()` methods of *Lucene*'s `IndexWriter` and `Directory` classes.

**`ExtractedDocument`** is a data class containing fields for a document's extracted text content, its *URL*, file type (`HTML` or `PDF`), and metadata generated by the *Tika* text extraction.

**`IndexingResult`** Is a data class representing a summary of what happened during the indexing process. It contains the names of the document collection and the *URL* list used, the number of processed, indexed, and skipped `URLS`, and the total execution time.

### 5.5.4 Document retrieval functionalities (package `retrieval`)

In general the text based retrieval process consists of finding documents that are related to a query by searching over an inverted index and returning them sorted by relevance; *Lucene* provides a straight forward way for implementing this functionality. A peculiarity of my application is that results must be retrieved from multiple document collections (and therefore multiple *Lucene* indices) and combined into a single result list. Moreover an experimenter must be optionally able to define a fixed result list to be displayed in response to the first query submitted by each user. Therefore I needed to build some structures around the features provided by the library. **Figure 10** shows the involved classes, which are briefly described below. Details on how the retrieval process is implemented are described in section 6.2.

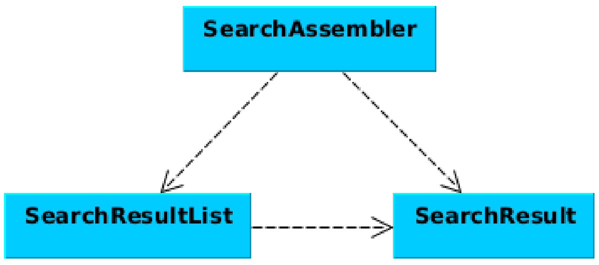


**Figure 10.** Classes in the retrieval package

**`SearchAssembler`** is the main class to be called from the service layer. Its contains the following public methods:

- `SearchResultList getSearchResults(String queryString, List<DocCollection> docCollections)`
  searches for documents relevant to the query string among the indices corresponding to the document collections provided as parameter. and returns the results wrapped in an instance of the `SearchResultList` class described below.
- `SearchResultList getFirstQueryList(DocCollection firstQueryCollection, String queryString)`
  returns all documents from the given collection (to be used for providing the predefined results for the first query).
- `void updateIndexAccess()`
  is to be called when an index is modified or replaced, which happens if a document collection is updated or re-indexed.

**`SearchResult`** is a data class representing a retrieved document. It contains the id and name of the document collection the result is drawn from, the document's id (generated during indexing), its score (determined by the *Lucene* retrieval system and used for sorting), its document's rank within the result list, a summery consisting of text snippets from the content, with highlighted query terms, and its file type (`HTML` or `PDF`).

**`SearchResultList`** represents the list of documents retrieved. An instance of this class is returned when the to the service layer after a call to the `SearchAssembler`'s `getSearchResults` or `getFirstQueryList`. It contains a list of `SearchResult` instances as well as the query string that was processed.

### 5.5.5 Experiment management (package `experiments`)

The `experiments` package contains one class named `ExperimentConfigurer` which is used for defining test groups and participants based on a configuration file, as well as several classes related to extracting and processing the data collected during an experiment.

`ExperimentConfigurer` contains the single public method `configureTestGroups`, which take an `Experiment` and the name name of a configuration file as parameters, and returns the updated `Experiment` after having parsed the contents of the file and set the experiment's test groups and participants accordingly.

The classes dealing with data extraction and processing are shown in **Figure 11** and briefly described below.

**`EventDataExtractor`** Is in charge of retrieving usage event data from the database and computing statistical metrics. It provides public methods for accessing information such as the number of queries or documents visited per experiment, test group, or user. Many of these methods return instances of the class `DataStats`, which are used for representing statistics over a list of numeric values.

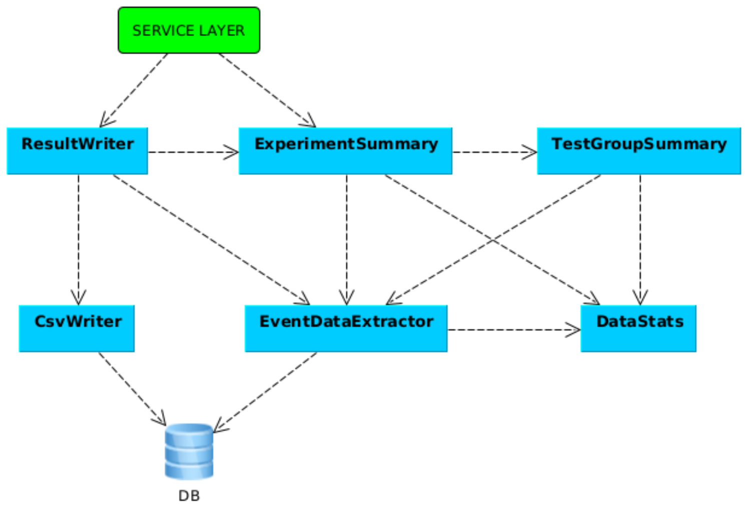


**Figure 11.** Classes related to data extraction and processing

**`ExperimentSummary`** contains relevant information about an experiment, which is displayed on the experiments evaluation *UI* page and made available for export through the `ResultWriter`. The data fields include the experiment's title, the date it was conducted, its duration, the names of its test groups, and statistical metrics obtained from the `EventDataExtractor`. It also contains a list of `TestGroupSummary` objects which contain detailed information about the results collected for each test group.

**`TestGroupSummary`** contains information analogous to `ExperimentSummary`, but computed on a per test group basis.

**`DataStats`** has the purpose of computing and storing often used statistical metrics. Objects of this class can be constructed from a list of `double` values (alternatively values can be appended later on). The accessible data fields are the number of values in the list, their total, mean, median, and standard deviation.

**`CsvWriter`** translates usage events into *CSV* format. It returns the data as a single string which is later processed by `ResultWriter`. It contains a method named `writeExperimentData`, which processes all usage events that occurred within a given experiment, as well as a method named `writeUserData`, which processes all usage events triggered by a given participant, thus representing its usage history.

**`ResultWriter`** is used for exporting data in *CSV* or *JSON* format. Following are its public methods, which all return an `InputStream` to be used for download:

- `InputStream rawDataCsv(Experiment experiment)`
  returns the complete data set related to the given experiment in *CSV* format.
- `InputStream rawDataJson(Experiment experiment)`
  returns the complete data set related to the given experiment in *JSON* format.
- `InputStream summaryJson(Experiment experiment)`
  returns the data contained in the `ExperimentSummary` generated for the given experiment in *JSON* format.
- `InputStream userHistoriesCsv(TestGroup testGroup)`
  returns the individual user histories of each participant in the give test group in *CSV* format grouped in a `.zip` directory.
- `InputStream userHistoriesJson(Experiment experiment)`
  returns the user histories of all participants in the given experiment in *JSON* format.

# 6 Implementation of the Information Retrieval System

## 6.1 Indexing

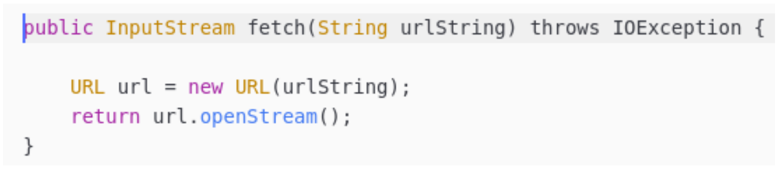
This section describes some technical details regarding the process of building a *Lucene* index starting from a *URL* list uploaded by an experimenter. The process consists in a number of operations to be performed for each *URL* and can be summarized in the following steps:

1. download the raw data received when pointing to the given address on the internet.
2. extract text content and additional information from the downloaded data.
3. analyze and persist the obtained representation using features of the *Lucene* library.

The *URLs* are stored one pr line in a text file and are made available as a list of strings via the file access feature provided in the `storage` package (see section 5.5.2). At a top level, the indexing process is implemented as a loop over the list of strings. If anything goes wrong at some step (e.g. an address not being available on the network or an I/O operation failing), the given *URL* is skipped and the loop continues with the next one. The number of documents skipped as well as the number of successfully indexed documents are stored in counter variables. After all *URLs* in the list have been processed, execution exits the loop and a summary containing relevant information such as the values of the counters and the processing time is returned to the caller as an instance of the class `IndexingResult` (see section 5.5.3. The following subsections describe the details of each step within the main loop.

### 6.1.1 Raw Data Download

The `java.net.URL` class provides a very simple way for downloading web content via its `openStream()` method. **Figure 12** shows how I embedded this functionality in my custom `fetch` method.



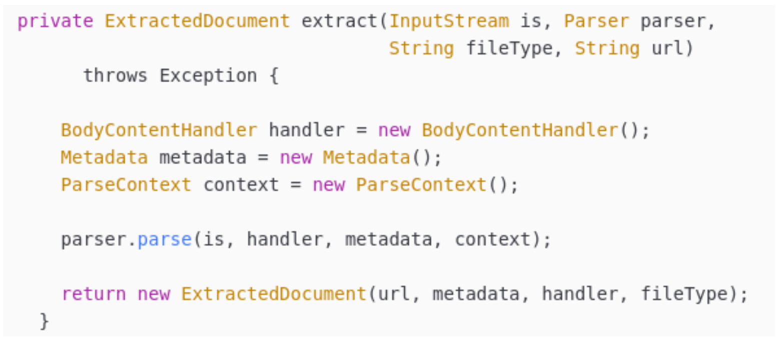
```
public InputStream fetch(String urlString) throws IOException {

    URL url = new URL(urlString);
    return url.openStream();
}
```

**Figure 12.** Method for downloading a web site

The method returns an `InputStream` to be processed in the next step. If something unexpected happens, an instance of `IOException` is thrown. If this happens, a `catch` block causes the loop to continue with the next *URL*.

### 6.1.2 Text Extraction

For the text extraction step I used some functionalities provided by the *Apache Tika* [2] library. The involved *Tika* classes are `BodyContentHandler`, `Metadata`, `ParseContext`, and implementations of the `Parser` interface (HtmlParser or PDFParser, depending on the document type). The `Parser` processes an `InputStream` and writes the results to objects of the other three classes. **Figure 13** shows my method for accessing these features.



```
private ExtractedDocument extract(InputStream is, Parser parser,
                                  String fileType, String url)
      throws Exception {

    BodyContentHandler handler = new BodyContentHandler();
    Metadata metadata = new Metadata();
    ParseContext context = new ParseContext();

    parser.parse(is, handler, metadata, context);

    return new ExtractedDocument(url, metadata, handler, fileType);
  }
```

**Figure 13.** Method for text extraction

The returned `ExtractedDocument` object converts the information from the *BodyContentHandler* to a *Java* String and the *Metadata* to a `Map<String, String>` and stores it in data fields along with the document's `URL` and file type, to be used in the next processing step.

### 6.1.3 Analysis an Storage

The core part of the indexing process is of course creating an index and adding documents to it. The involved classes from the *Lucene* library are `IndexWriter`, `Directory`, `Analyzer`, `Document`, and `Fileld`, whose interactions are shown in **Figure 14**.

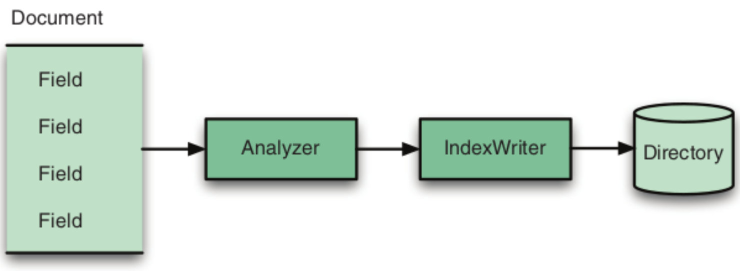


**Figure 14.** Lucene Indexing Classes

**IndexWriter** creates a new index or opens an existing one and allows adding, removing or updating documents.

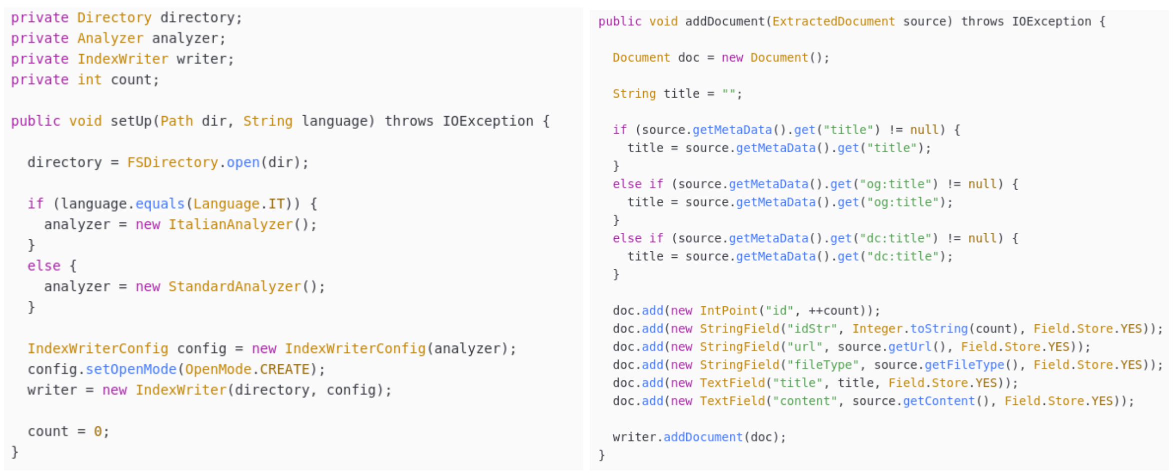
**Directory** represents the location of an index (in my case a directory in the file system).

**Analyzer** converts plain text into a sequence of tokens. This involves removing so called "stop words" (frequently used but irrelevant words) and stemming (reducing similar words to their common root). Since both stop word removal and stemming are language dependent, so a different `Analyzer` implementation for Italian document collections than for English ones was to be used.

**Document** The Document class represents a collection of fields which can optionally be stored. In my application each document contains the following fields; `id`, `url`, `fileType`, `title`, and `content`.

In my application, access to these classes is mediated by a class named `Indexer`, which exposes the methods `setUp`, `addDocument` (shown in **Figure 15**) and tearDown. before the main loop is started, the `setUp` method is called which initializes the `Directory` and `Analyzer`, and `IndexWriter` with information from the `DocumentCollection` instance to be indexed. During the loop, for each extracted document, the `addDocument` method is called, which creates the *Lucene* document and populates the fields. After the main loop terminates, the `tearDown` method is called, which in turn calls the `close()` methods of the `IndexWriter` and `Directory`.

For details on the usage of the *Lucene* indexing functionalities see chapter 1 of the book "Lucene in Action" [17].



```java
private Directory directory;
private Analyzer analyzer;
private IndexWriter writer;
private int count;

public void setUp(Path dir, String language) throws IOException {

  directory = FSDirectory.open(dir);

  if (language.equals(Language.IT)) {
    analyzer = new ItalianAnalyzer();
  }
  else {
    analyzer = new StandardAnalyzer();
  }

  IndexWriterConfig config = new IndexWriterConfig(analyzer);
  config.setOpenMode(OpenMode.CREATE);
  writer = new IndexWriter(directory, config);

  count = 0;
}
```

```java
public void addDocument(ExtractedDocument source) throws IOException {

  Document doc = new Document();

  String title = "";

  if (source.getMetaData().get("title") != null) {
    title = source.getMetaData().get("title");
  }
  else if (source.getMetaData().get("og:title") != null) {
    title = source.getMetaData().get("og:title");
  }
  else if (source.getMetaData().get("dc:title") != null) {
    title = source.getMetaData().get("dc:title");
  }

  doc.add(new IntPoint("id", ++count));
  doc.add(new StringField("idStr", Integer.toString(count), Field.Store.YES));
  doc.add(new StringField("url", source.getUrl(), Field.Store.YES));
  doc.add(new StringField("fileType", source.getFileType(), Field.Store.YES));
  doc.add(new TextField("title", title, Field.Store.YES));
  doc.add(new TextField("content", source.getContent(), Field.Store.YES));

  writer.addDocument(doc);
}
```

**Figure 15.** Methods for accessing*Lucene*'s indexing features

## 6.2 Retrieval

Retrieval consists of translating a query string into terms and finding matching documents in the index by comparing the query terms with the terms stored in the fields defined during indexing. The involved *Lucene* classes are briefly described below [17]:

**`IndexReader` and `IndexSearcher`** provide read access to the index for searching.

**`Query` (and subclasses)** encapsulate logic for particular query types. Instances of this class are passed to `IndexSearcher`'s `search` method.

**`QueryParser`** transforms a query string into a `Query` object, given the field to be considered and an instace of `Analyzer`. In most cases it is recommended to use the same type of `Analyzer` that was used during indexing, so the query terms undergo the same stop word removal an stemming procedures.

**TopDocs** Contains the results returned by `IndexSearcher`'s `search` method.

**ScoreDoc** provides access to the results in `TopDocs`.

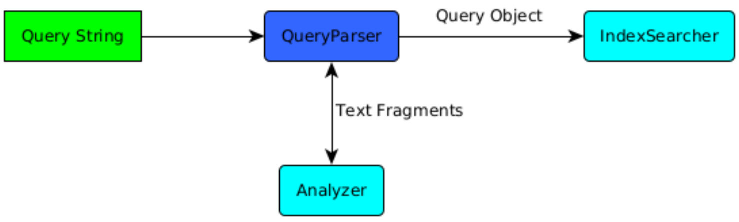


**Figure 16.** Role of `QueryParser` in the search process [17]

In our application, access to the retrieval features is implemented in class `SearchAssembler` within the `retrieval` package. When a query is submitted, each document collection available to the given test group is searched, and results are appended to a list in form of instances of the class `SearchResult`, which contains the information provided by *Lucene*'s `ScoreDoc` and `Document` classes, along with other metadata. Before the list is returned, it is sorted by score.

## 7 Conclusions and Future Work

Though the resulting application satisfies (at leas at a basic level) the requirements defined at the beginning of the project, there are a number of ways it can be improved and extended in the future. An important point is that I implemented the indexing and retrieval mechanisms using only the simplest aspects of the *Lucene* API. I suppose that an assessment of the implementation's performance in terms of precision and recall would show a large margin of possible improvement, which could affect the participant's usage experience and thus have an impact on the experiment's results. The quality of of the search engine could definitely be improved by at the level of *Lucenes* `Analyzer` class, which can be extended for a fine grained application-specific configuration. Also the graphical aspects of the user interface may be designed in a more attractive way. A future extension which I have kind of set the bases for is the support for multiple languages, both at the user interface and document corpus level, as it is now the case for English and Italian.

In conclusion I'd like to express my thanks to the professors Marc Langheinrich and Peter Schulz for having given me the opportunity to work on this interesting and in many ways challenging project, and having given me precise and clear indications on how to proceed. I hope that the result will allow to collect useful information in the context of the panned experiment, an maybe be reused in adapted forms for similar research scenarios in the future.

# Appendices

## A Configuration, Build, and Deployment (from HSE Setup and Usage Guide)

### A.1 Dependencies

The system on which you build the application must have the following software installed:

- Java JDK 11
- Apache Maven 3.6.3
- Docker version 20.10.1

If you need to run the application locally without using docker, also *MySql* is required.
The server on which the application is to be deployed only needs *Docker* and *Docker Compose*.

### A.2 Configuration Files

#### A.2.1 Maven configuration: pom.xml

The build settings used by *Maven* are defined in `/hse/pom.xml`. This file contains some general information about the project, such as name and version, as well as a list of java packages and *Maven* plugins that are downloaded and set up at build time. The File also declares two profiles ("dev" and "prod"). The "dev" profile is intended for creating a local build to be used during development, while the "prod" profile is to be used for building the deployment version. The profiles are linked to specific configuration files which contain various settings such as server ports and global constants: when the profile "dev" is selected, the application uses the file
`/hse/src/main/resources/application-dev.properties`; when "prod" is selected, `/hse/src/main/resources/application-prod.properties`.
By default the "dev" profile is selected; for using the "prod" profile, the flag `-Pprod` is to be included in the *Maven* command (e.g. `mvn -Pprod clean install`).

#### A.2.2 Specific configurations in `.properties` files

The directory `/hse/src/main/resources/` contains three files with extension `.properties`:
`application.properties`, `application-dev.properties` and `application-prod.properties`. The first one contains settings that are applied independently of the selected profile, while the other two contain profile-specific settings. The crucial settings to be considered at build time are the `spring.datasource.xxx` properties, which indicates the database which the application is going to connect to, and the `baseUrl` parameter, which indicates the prefix used at server-level. For instance, in the `application-prod.properties` as I have set it up, the data source is pointing to a *MySql* Docker container, and the base URL is set to `/hse/` since I deploy it on `http://www.robix-projects.org/hse/`. In the `application-dev.properties` file the data source points to a local *MySql* instance and the baseUrl is `/`.
The other properties indicate directory paths and should not need to be modified.

#### A.2.3 docker-compose.yml

The simplest way to run the application on a server is by using *Docker Compose*. The way in which the containers are created from the images and the internal ports used are defined in `/hse/docker-compose.yml`.

### A.3 Creating a local build

#### A.3.1 Preparing the database

In order to run the application locally, a *MySql* database service running on port 3306 is required. The service must contain a database named `hse_db` and should be accessible via username `root` and password `root`. The tables are created automatically at application startup. If you need to use other login credentials, or the service is running on another port, these parameters can be set in `application-dev.properties`.

### A.3.2 Issuing the build command

The command

```
mvn clean install
```

initiates the build process. The process involves executing several test suites, which should work without failure. In case the tests fail (e.g. due to path incompatibilities or missing files) the tests can be skipped using the `-DskipTests` flag:

```
mvn -DskipTests clean install
```

### A.3.3 Running the application

Once the application is built, it can be run in several ways. During development, it is convenient to run it from the IDE (in *Eclipse* package explorer, right-click on project → Run As → Spring Boot App). Alternatives are to run it from command-line using *Maven*:

```
cd hse/
mvn spring-boot:run
```

or using *Java*:

```
cd hse/target/
java -jar hse-0.1.jar
```

## A.4 Example deployment on *Ubuntu Server* with *Apache2*

### A.4.1 Create and transfer the *Docker* image

The first step consists of creating the application's image by issuing

```
cd hse/
mvn -Pprod clean install
```

At this point, the output of `docker image ls` should contain a line similar to:

```
REPOSITORY           TAG       IMAGE ID       CREATED             SIZE
robix82/usi.ch-hse   0.1       c2137e7cc110   37 seconds ago      758MB
```

Once the image is created it can be exported as a `.tar` file by issuing

```
docker save robix82/usi.ch-hse:0.1 > hse.tar
```

Finally the `.tar` file and `/hse/docker-compose.yml` must be copied to the server, e.g. using `scp`.

### A.4.2 Load the image and start the application

On the server, the image from the `.tar` file can be loaded with

```
docker load < hse.tar
```

With the image loaded, the application can be started by issuing

```
docker-compose up &
```

from the directory containing the `docker-compose.yml` file. The required *MySql* image will be downloaded and initialized automatically.

At this point, the application is reachable on port 8081 (the port can be configured in `docker-compose.yml`).

### A.4.3 Apache2 configuration

In order to make the application reachable on the server's external address with a custom prefix, it is necessary to configure a virtual host using Apache's `mod_proxy` module. For details on `mod_proxy`, please refer to https://www.digitalocean.com/community/tutorials/...

The virtual host configuration is done by placing a file (in this example `hse.conf`) in `/etc/apache2/sites-available/` and creating a symlink to it in `/etc/apache2/sites-enabled/`:

```
ln -s /etc/apache2/sites-available/hse.conf /etc/apache2/sites-enabled/
```

The content of the `.conf` file should look similar to

```
<VirtualHost *:80>

  ServerName www.robix-projects.org
  ProxyPreserveHost On

  ProxyPass /hse/ http://127.0.0.1:8081/
  ProxyPassReverse /hse/ http://127.0.0.1:8081/

</VirtualHost>
```

This configuration makes the application available under `http://www.robix-projects.org/hse`. Notice that the prefix `/hse/` must match the `baseUrl` property in `application-prod.properties` and the port (8081 in this example) must correspond to the port defined in `docker-compose.yml`.

## B Usage (from HSE Setup and Usage Guide)

### B.1 Users, Roles, and their Definition (/admin/ui)

The application distinguishes three types (roles) of users: *Administrators*, *experimenters*, and *participants*. Depending on a user's role, different UI elements are visible or accessible: participants can access only the search interface; experimenters have access to the indexing and the experiment setup interfaces; administrators have access to all interfaces, including a page for creating, updating, and removing administrators and experimenters. Participants can be defined by administrators or experimenters when an experiment is set up, or automatically if the experiment is run within a *Qualtrics* survey.

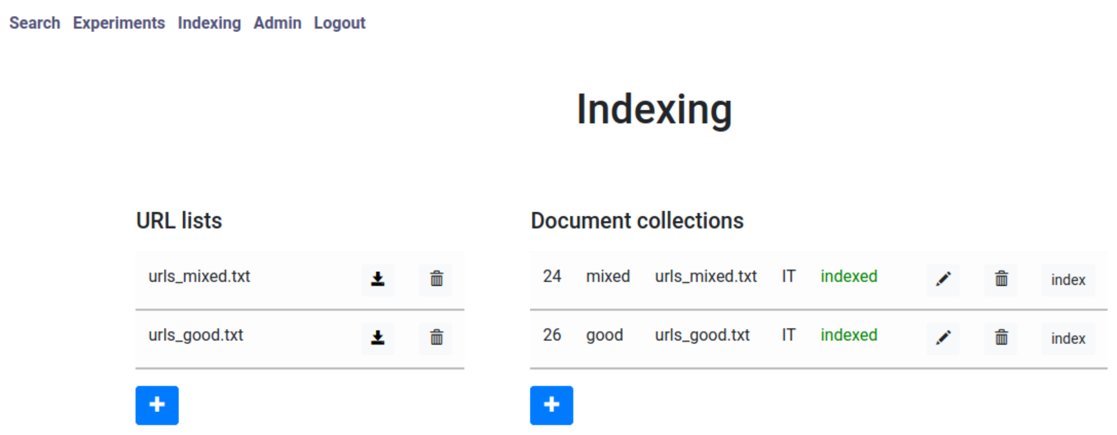
If the application is newly installed, a default user with *Administrator* role is created. This user can log in using the user name "*admin*" and password "*admin*".

### B.2 URL lists and Document collections (/indexing/ui)

Before an experiment can be set up, at least one document collection must be available. These can be created on the *indexing* UI page available to experimenters and administrators (see **Figure 17**). Creating a document collection involves the following steps:

- Upload a URL list, i.e. a text file (.txt) containing one URL per line.
- Define a doc collection by setting its name, the URL list to be used, and the language (*IT* or *EN*) of the web pages.
- Start the indexing process. This may take a long time since it involves downloading all the pages over the network; while testing I observed times in the order of one second per URL.

Document collections will later be assigned to test groups, so the search engine returns different results depending on which test group a participant belongs to. Optionally a document collection can be set as a fixed result for the first query; in this case, the first query submitted by a participant returns this entire document collection, in the order in which the URLs are set in the list used for generating it.



Indexing

URL lists

| | | |
|---|---|---|
| urls_mixed.txt | ⬇ | 🗑 |
| urls_good.txt | ⬇ | 🗑 |

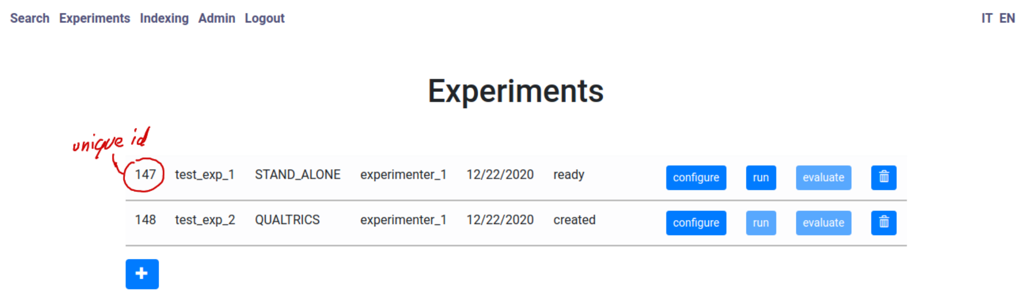
Document collections

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 24 | mixed | urls_mixed.txt | IT | indexed | ✎ | 🗑 | index |
| 26 | good | urls_good.txt | IT | indexed | ✎ | 🗑 | index |

**Figure 17.** User interface for creating document collections

## B.3 Experiment Definition (/experiments/ui)

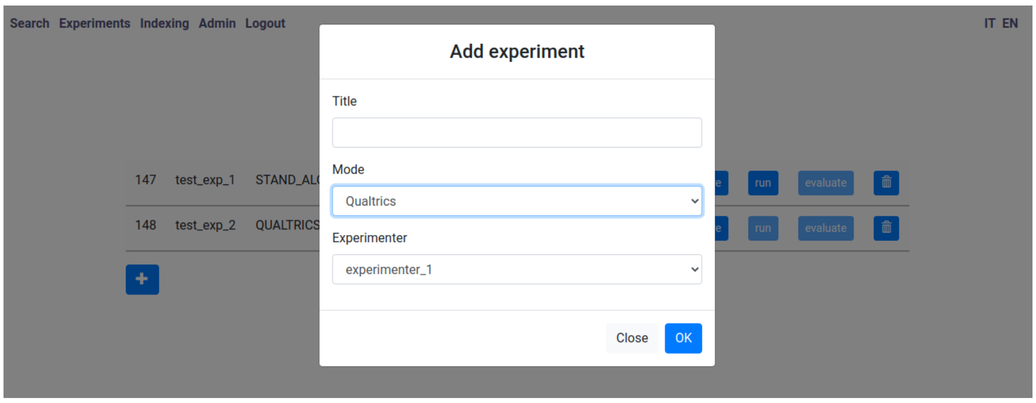
The */experiments/ui* page lists all defined experiments. Each line shows the experiment's unique id (needed for running with a *Qualtrics* survey), its title, mode (*stand_alone* or *Qualtrics*), the assigned experimenter, the date on which it was defined, its status (one of *created*, *ready*, *running*, or *complete*), and buttons leading to the UI for configuration, execution, and evaluation. The buttons are enabled or disabled depending on the experiment's status. **Figure 18** shows the experiments interface.



Experiments

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 147 | test_exp_1 | STAND_ALONE | experimenter_1 | 12/22/2020 | ready | configure | run | evaluate | 🗑 |
| 148 | test_exp_2 | QUALTRICS | experimenter_1 | 12/22/2020 | created | configure | run | evaluate | 🗑 |

**(a)** Experiments list



Add experiment

Title

Mode: Qualtrics

Experimenter: experimenter_1

**(b)** Popup for defining a new experiment

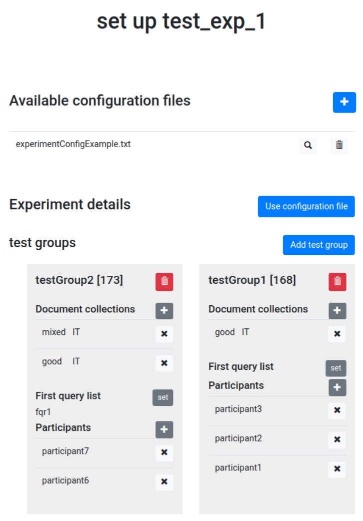
**Figure 18.** Interface for defining experiments

## B.4 Experiment Configuration (/experiments/setup/ui)

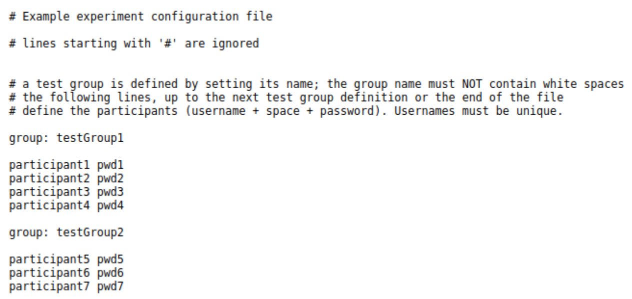
The interface for experiment configuration is reachable by clicking on the *configure* button in the experiments list UI. If the experiment's mode is *stand_alone* the configuration involves creating test groups and assigning document collections and participants to them. Moreover, a document collection can be set as a predefined result list for the first query. Participants and groups can be defined manually or loaded from a configuration file. The same applies to *Qualtrics* mode; the only difference is that participants don't deed to be defined, since they are created while they take part in the survey.

### B.4.1 Configuration in stand-alone mode

The setup UI for stand-alone experiments includes a section for uploading, inspecting, or deleting configuration files (which can be used for defining test groups and participants), as well as a section for editing the test groups. **Figure 19a** shows the page after two test groups have been added and edited; **Figure 19b** shows an example of a configuration file. For the configuration to be complete, i.e. its status being set to *complete* and the *run* UI being available, there must be at least one test group defined, and each test group must have some participants and at least one document collection.



**(a)** Experiment setup UI for stand-alone mode



```
# Example experiment configuration file

# lines starting with '#' are ignored


# a test group is defined by setting its name; the group name must NOT contain white spaces
# the following lines, up to the next test group definition or the end of the file
# define the participants (username + space + password). Usernames must be unique.

group: testGroup1

participant1 pwd1
participant2 pwd2
participant3 pwd3
participant4 pwd4

group: testGroup2

participant5 pwd5
participant6 pwd6
participant7 pwd7
```

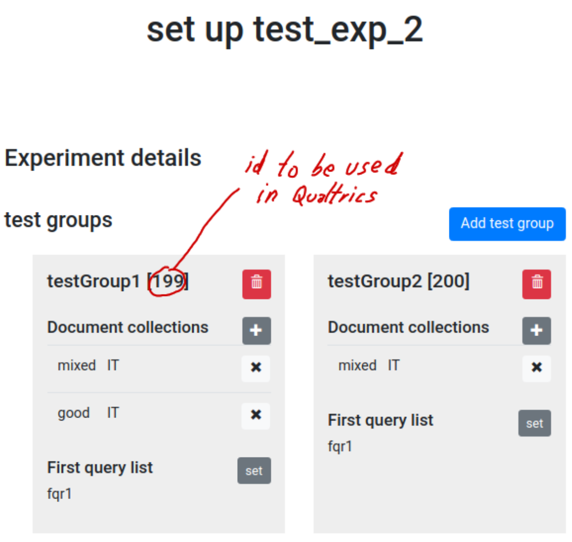
**(b)** Example of a configuration file

**Figure 19.** Experiment Configuration File

### B.4.2 Configuration in Qualtrics mode

In Qualtrics mode the setup options are similar, but the participants don't need to be specified, as they are defined during the survey. So there is no need for configuration files. For the configuration to be complete, there must be at least one test group, and each test group must have at least one document collection.

**Figure 20** shows the configuration interface after creating and editing two test groups. Notice the test group's id number: this will be needed when setting up the related Qualtrics survey.



set up test_exp_2

Experiment details

test groups

Add test group

testGroup1 [199]

Document collections

mixed IT

good IT

First query list

fqr1

testGroup2 [200]

Document collections

mixed IT

First query list

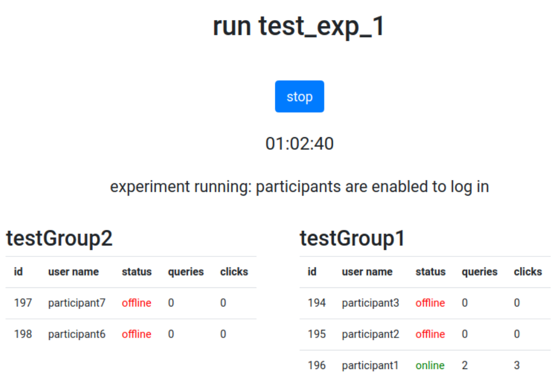
fqr1

**Figure 20.** Experiment setup UI for Qualtrics mode

## B.5 Experiment Execution (/experiments/run/ui)

If an experiment is configured, the related execution UI becomes available. The interface is quite simple: there is a button for starting, stopping, and resetting the experiment, as well as live updated information on participant's activities. In stand-alone mode, all participants are listed from the beginning, while in Qualtrics mode they appear as they log in.

The purpose of the start / stop mechanism is to enable participant access and to measure the experiment's duration. As soon as the experiment is started, participants can lo in; when the experiment is stopped they are automatically logged out. In case the experiment is defined in Qualtrics mode, the participants are redirected back to the survey. After an experiment is started, the page can be left and returned to later. **Figure 21** shows the interface for running an experiment.



run test_exp_1

01:02:40

experiment running: participants are enabled to log in

testGroup2

| id | user name | status | queries | clicks |
|---|---|---|---|---|
| 197 | participant7 | offline | 0 | 0 |
| 198 | participant6 | offline | 0 | 0 |

testGroup1

| id | user name | status | queries | clicks |
|---|---|---|---|---|
| 194 | participant3 | offline | 0 | 0 |
| 195 | participant2 | offline | 0 | 0 |
| 196 | participant1 | online | 2 | 3 |

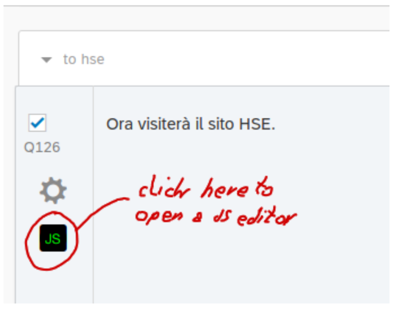
**Figure 21.** Experiments run UI

## B.6 Setting up a linked Qualtrics survey

For linking a survey to an experiment, the survey must contain a block whose "next" button redirects to HSE, a failure block to be displayed in case accidentally the experiment is not running, or there is some connection error, and some settings at the beginning of the *Survey flow*.

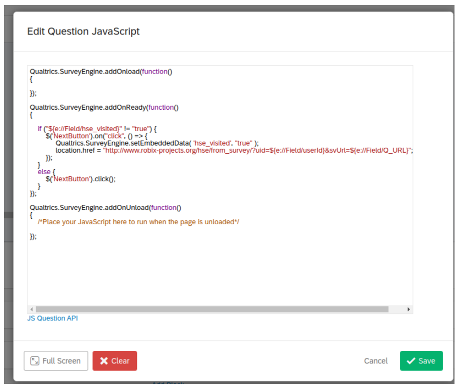
### B.6.1 Survey Questions

The redirection is set via *JavaScript* in the survey question (see **Figure 22**). The failure block needs no specific configuration.



Ora visiterà il sito HSE.

**(a)** Opening the JS editor



```
Qualtrics.SurveyEngine.addOnload(function()
{

});

Qualtrics.SurveyEngine.addOnReady(function()
{

    if ("${e://Field/hse_visited}" != "true") {
        $('NextButton').on("click", () => {
            Qualtrics.SurveyEngine.setEmbeddedData( 'hse_visited', "true" );
            location.href = "http://www.robix-projects.org/hse/from_survey/?uid=${e://Field/userId}&svUrl=${e://Field/Q_URL}";
        });
    }
    else {
        $('NextButton').click();
    }
});

Qualtrics.SurveyEngine.addOnUnload(function()
{
    /*Place your JavaScript here to run when the page is unloaded*/

});
```

**(b)** Edited JS

**Figure 22.** Link to JS Editor

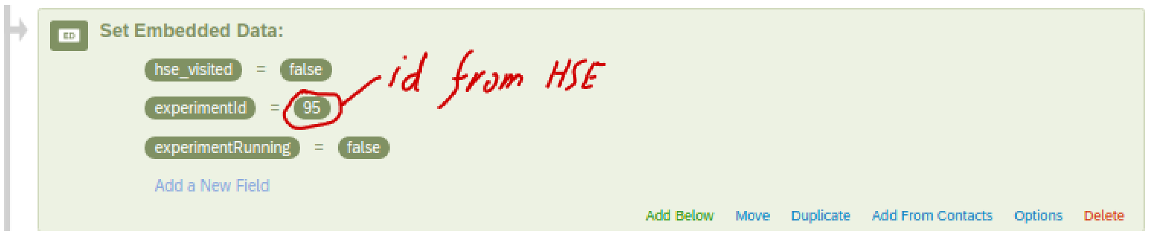
In the JS editor there are three default functions: `Qualtrics.SurveyEngine.addOnload(...)`, `Qualtrics.SurveyEngine.addOnReady(...)`, and `Qualtrics.SurveyEngine.addOnUnload(...)`. The first and third functions don't need to be modified, while the `Qualtrics.SurveyEngine.addOnReady(...)` function needs to be edited as follows:

```
Qualtrics.SurveyEngine.addOnReady(function()
{
    if ("${e://Field/hse_visited}" != "true") {

        $('NextButton').on("click", () => {
            Qualtrics.SurveyEngine.setEmbeddedData( 'hse_visited', "true" );
            location.href = "http://[hse server url]/from_survey/
                                 ?uid=${e://Field/userId}&svUrl=${e://Field/Q_URL}";
        });
    }
    else {

        $('NextButton').click();
    }
});
```

### B.6.2 Survey Flow

The following elements are to be added at the beginning of the survey flow, in the order they are presented here:

1. An *Embedded Data* element for initializing some variables:



Set Embedded Data:

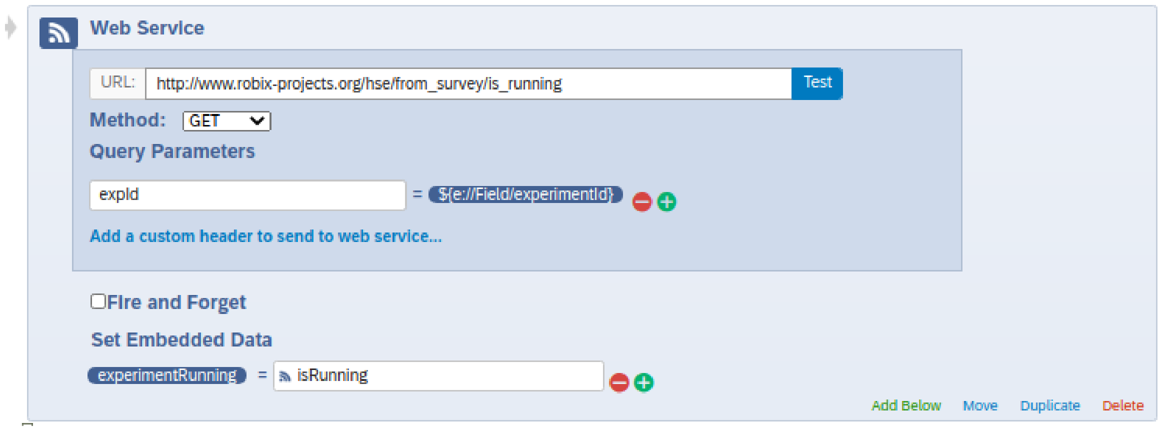
hse_visited = false

experimentId = 95

experimentRunning = false

**Figure 23.** Embedded Data Element

2. A *Web Service* element to do an API call for checking that the given experiment is actually running:



Web Service

URL: http://www.robix-projects.org/hse/from_survey/is_running

Method: GET

Query Parameters

expId = ${e://Field/experimentId}

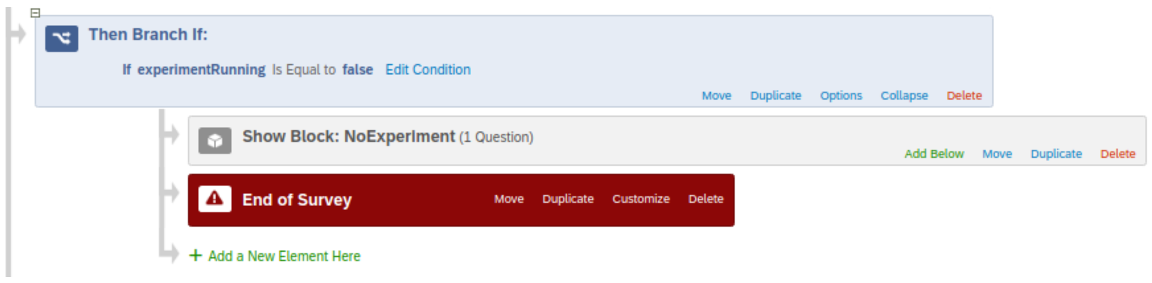
Fire and Forget

Set Embedded Data

experimentRunning = isRunning

**Figure 24.** Web Service Element

3. A *Branch* element to interrupt the survey if the experiment is not running:



Then Branch If:

If experimentRunning Is Equal to false

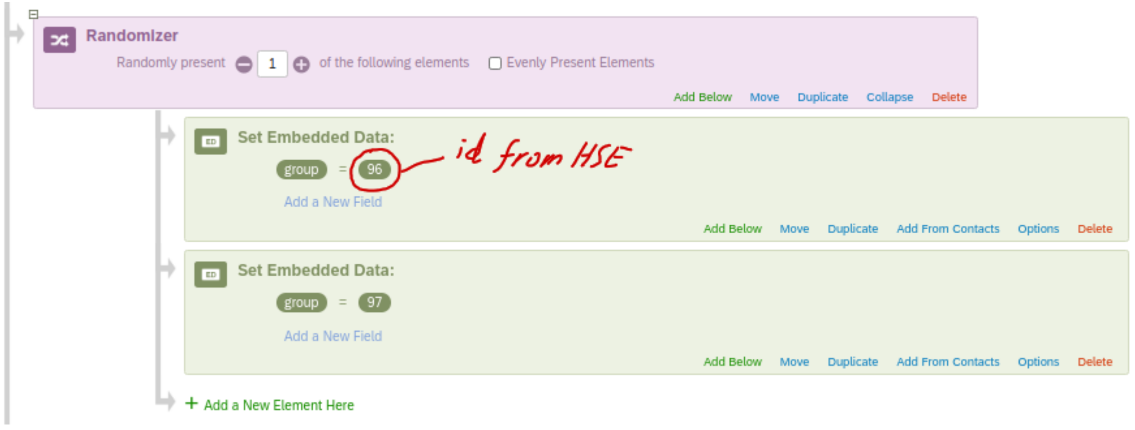
Show Block: NoExperiment (1 Question)

End of Survey

Edit Condition
Move Duplicate Options Collapse Delete
Add Below Move Duplicate Delete
Move Duplicate Customize Delete
+ Add a New Element Here

**Figure 25.** Branch Element

4. A *Randomizer* element for assigning participants to test groups:



Randomizer

Randomly present 1 of the following elements ☐ Evenly Present Elements

Set Embedded Data:

group = 96 id from HSE

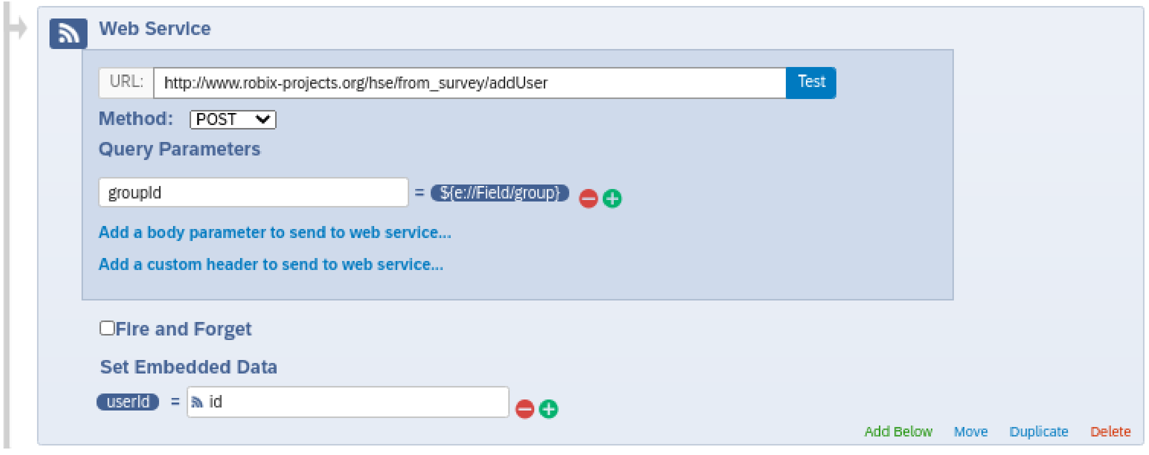
Set Embedded Data:

group = 97

Add Below Move Duplicate Collapse Delete
Add a New Field
Add Below Move Duplicate Add From Contacts Options Delete
Add a New Field
Add Below Move Duplicate Add From Contacts Options Delete
+ Add a New Element Here

**Figure 26.** Randomizer Data element

5. A *Web Service* element for initializing the participant in HSE:



Web Service

URL: http://www.robix-projects.org/hse/from_survey/addUser

Method: POST

Query Parameters

groupId = ${e://Field/group}

☐ Fire and Forget

Set Embedded Data

userId = id

Test
Add a body parameter to send to web service...
Add a custom header to send to web service...
Add Below Move Duplicate Delete

**Figure 27.** Web Service element

## B.7 Experiment Evaluation and Data Export (/experiments/eval/ui)

Once an experiment is completed, its evaluation page is accessible. The page contains several links for downloading the raw or partially pre-processed data, as well as a summary including per-user means and standard deviations for relevant derived data such as the number of documents visited, the time spent on a document, and the distribution of these measures over the different document collection.

### B.7.1 Data Representation

The data collected during an experiment consists of a list of *Usage Events*. There are three kinds of such events: *Session Events* (login and logout), *Query Events* (generated when a participant submits a search query), and *Document Click Events* (generated when a participant clicks on a link from the results list and visits a page). All *Usage Events* contain the following data fields:

- A unique id (generated by the database system).
- A timestamp of the moment when the event was generated.
- The user id of the participant who generated the event.
- The id and name of the test group the participant belongs to
- The event type (one of `SESSION`, `QUERY`, or `DOC_CLICK`)

*Session Events* contain an additional field indicating whether it was a login or logout event; *Query Events* contain the query string submitted, the total number of retrieved results, and the proportions in which the document collections are represented in the results. *Document Click Events* contain the document's URL, an id, the name and id of the collection the document belongs to, and its rank (position within the query results).

The experiment's evaluation page allows to download the entire raw data in a single `.csv` or `.json` file, or the same data split into per-user histroies (in a single `.json`, or in separate `.csv` files).

## References

[1] Apache lucene web site. https://lucene.apache.org/core/. (accessed: 06.01.2021).

[2] Apache tika. https://tika.apache.org/. (accessed: 06.01.2021).

[3] Baeldung: Intro to the jackson objectmapper. https://www.baeldung.com/jackson-object-mapper-tutorial. (accessed: 06.01.2021).

[4] Bootstrap web site. https://getbootstrap.com/. (accessed: 06.01.2021).

[5] Docker web site. https://www.docker.com/. (accessed: 06.01.2021).

[6] jquery web site. https://jquery.com/. (accessed: 06.01.2021).

[7] Mysql web site. https://www.mysql.com/. (accessed: 06.01.2021).

[8] Spring boot web site. https://spring.io/projects/spring-boot. (accessed: 06.01.2021).

[9] Spring data jpa reference documentation. https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference. (accessed: 06.01.2021).

[10] Spring.io: Accessing data with mysql. https://spring.io/guides/gs/accessing-data-mysql/. (accessed: 06.01.2021).

[11] spring.io: Uploading files. https://spring.io/guides/gs/uploading-files/. (accessed: 06.01.2021).

[12] Thymeleaf web site. https://www.thymeleaf.org/. (accessed: 06.01.2021).

[13] Wikipedia: Ajax. https://en.wikipedia.org/wiki/Ajax_(programming). (accessed: 06.01.2021).

[14] Wikipedia: Rest. https://en.wikipedia.org/wiki/Representational_state_transfer. (accessed: 06.01.2021).

[15] Wikipedia: Websocket. https://en.wikipedia.org/wiki/WebSocket. (accessed: 06.01.2021).

[16] Q. E. Management. https://www.qualtrics.com/it/. (accessed: 06.01.2021).

[17] O. G. Michael McCandless, Erik Hatcher. *Lucene In Action*. Manning Publications Co, 2nd edition, 2010.

[18] I. of Communication and Health. https://www.ich.com.usi.ch/en/about-us/institute. (accessed: 06.01.2021).