Università della Svizzera italiana

Faculty of Informatics

Bachelor Project

January 12, 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 [17]. 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

Approved by the advisor on Date: Prof.

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.

1.2 Outline

TODO: describe the sections

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.

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 **TODO: link actual section**. **Figure 2** shows the relevant parts of the interface.

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

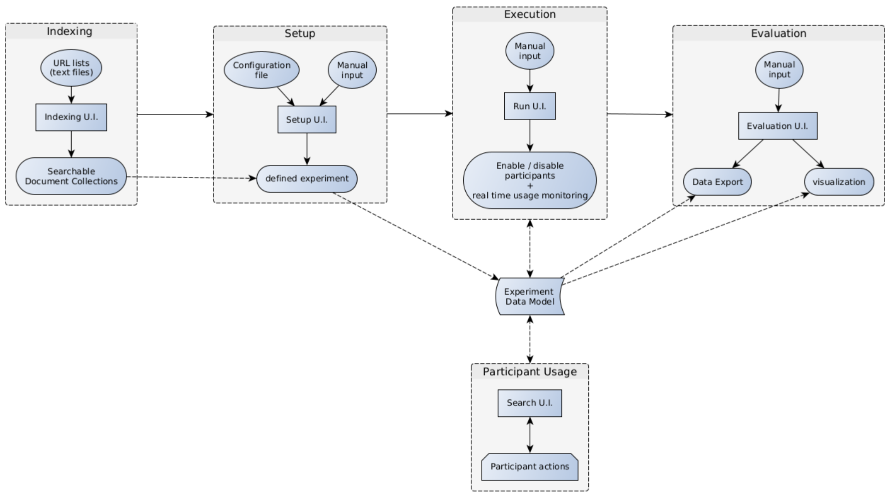
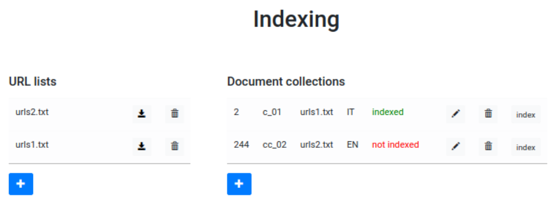


Figure 1. Typical usage workflow



Indexing

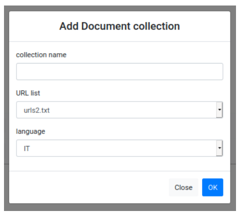
URL lists

urls2.txt		
urls1.txt		

Document collections

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

(a) indexing u.i



Add Document collection

collection name

URL list: urls2.txt

language: IT

(b) popup for defining document collections

Figure 2. Indexing UI page

be set for each test group as predefined result list to be returned after the 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.

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 user interface.

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.

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

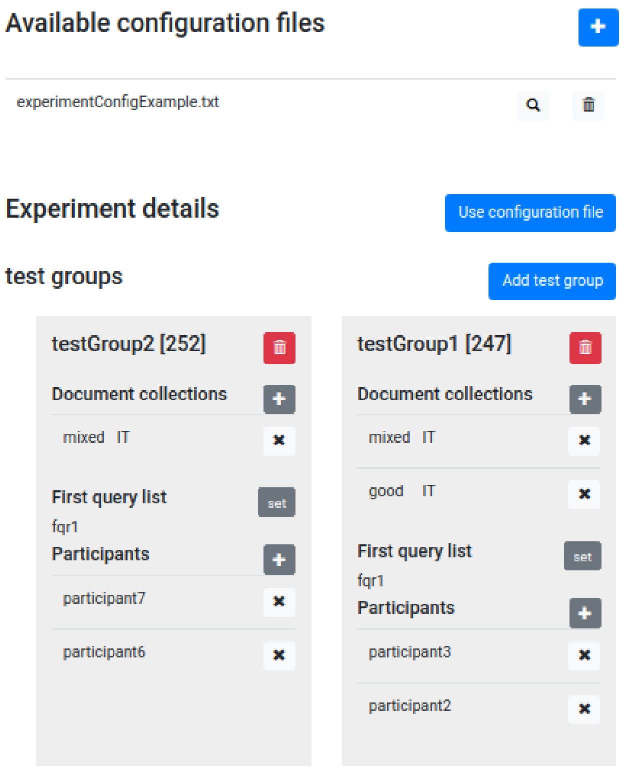


Figure 3. U.I. for experiment setup

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 to which the given document belongs.

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

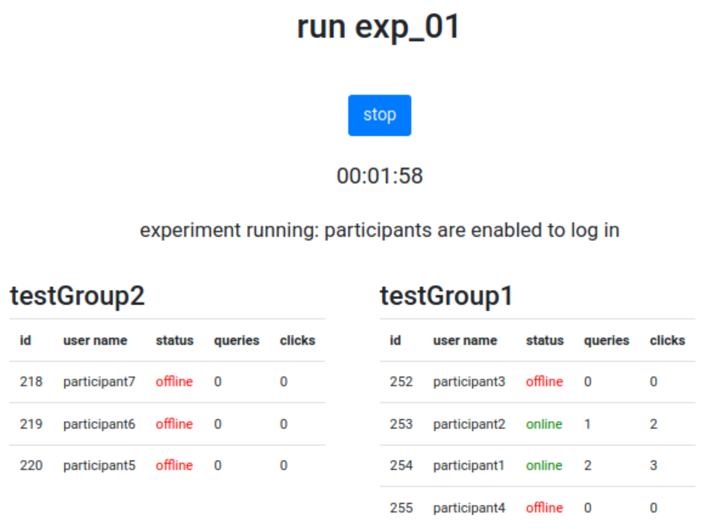
4.1 Raw Data Records

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.


run exp_01

stop

00:01:58

experiment running: participants are enabled to log in

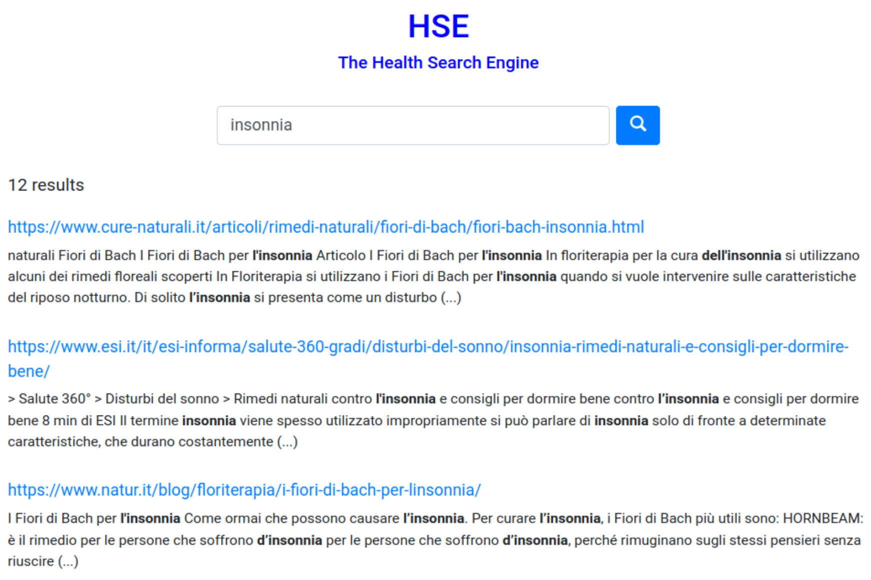
testGroup2

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


HSE

The Health Search Engine

insonnia

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

- 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. Section 6.4 describes the details of how these measures are computed.

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.

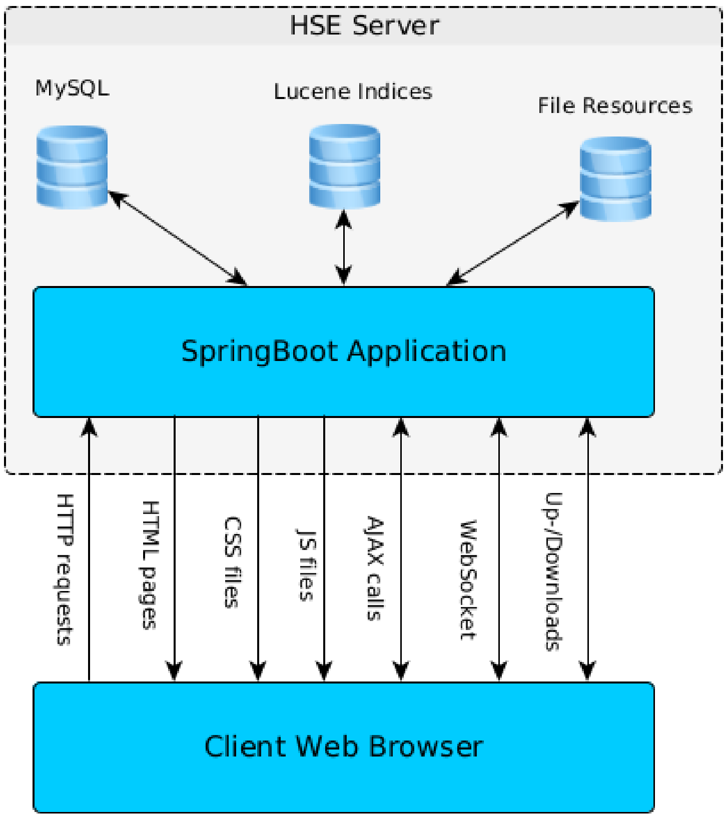


Figure 6. General System Architecture

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 templates, or simply a string representing the name of a template file.

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 independently 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 my 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 my 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 *Spring* by setting parameter of type *MultipartFile*. 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 a *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 data 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 both 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 sections 6.2 and 6.3.

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 tightly coupled classes are placed in the same package. My 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”. This layer 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 implementation layer below.

Implementation 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, edges represent connections through instantiation or method calls.

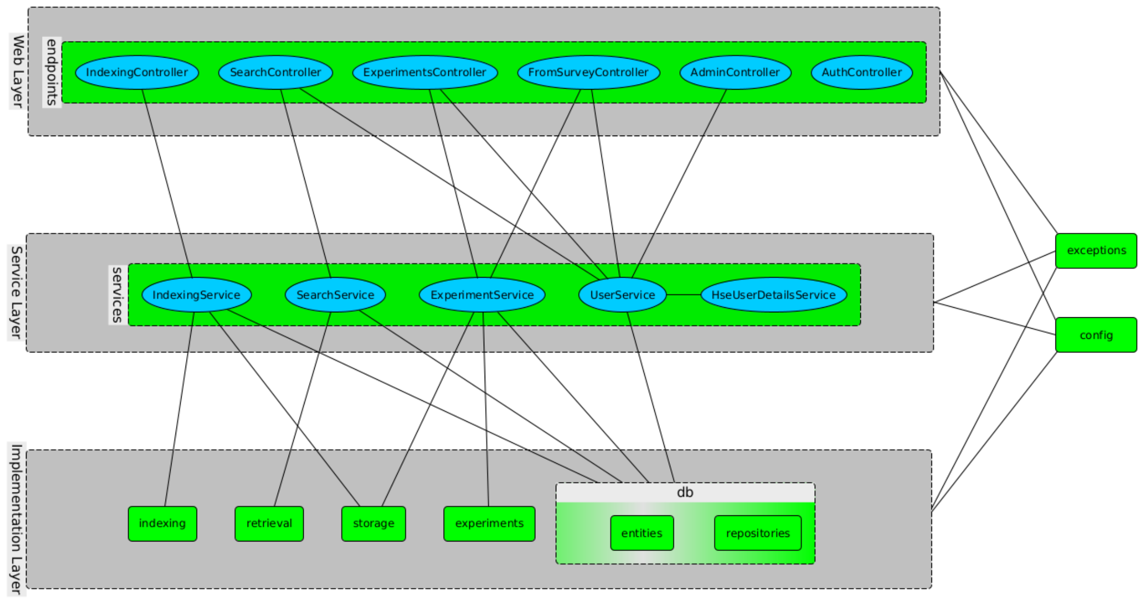


Figure 7. Layered architecture of the application

6 Implementation details

6.1 User Management

6.2 Indexing

6.3 Retrieal

6.4 Data Extraction and Summary Generation

Appendices

A The Lucene Information Retrieval Library

B HSE Setup and usage

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