Università della Svizzera italiana

Faculty of Informatics

Bachelor Project

January 15, 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

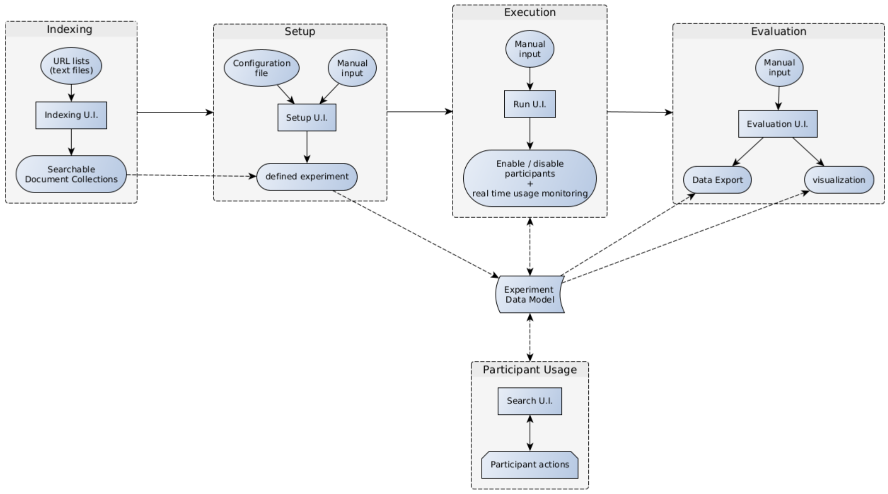
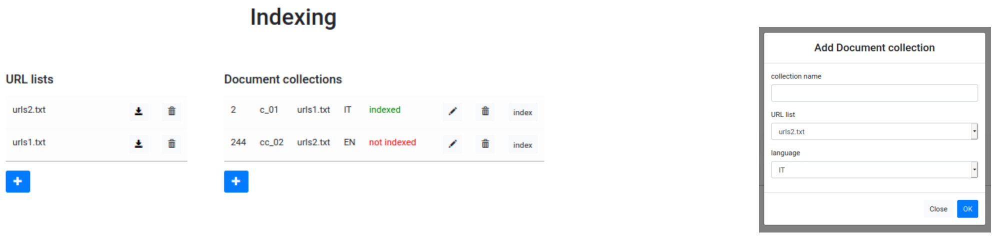


Figure 1. Typical usage workflow



(a) indexing u.i

(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

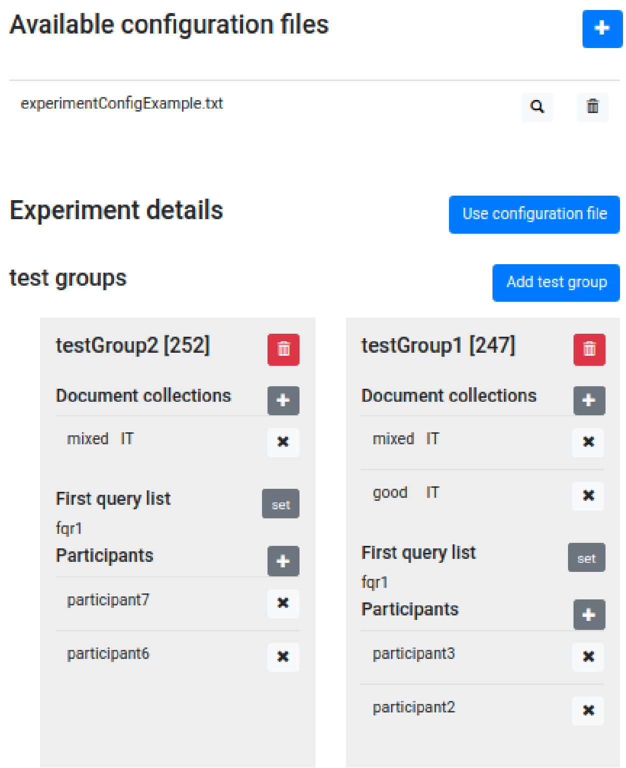


Figure 3. U.I. for experiment setup

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.

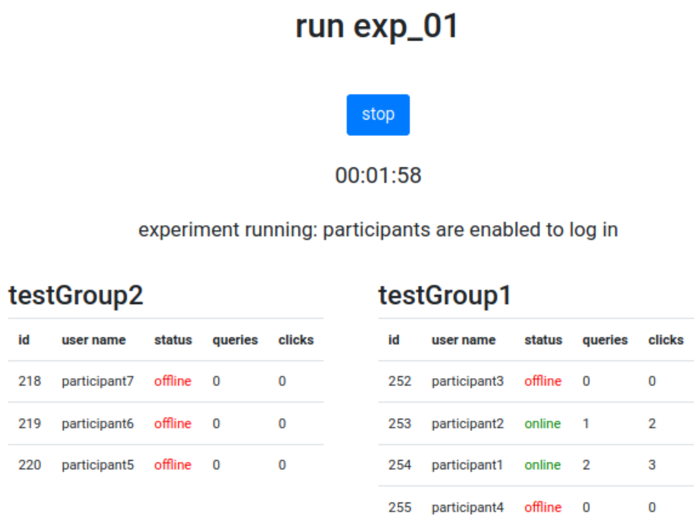


Figure 4. UI for experiment execution

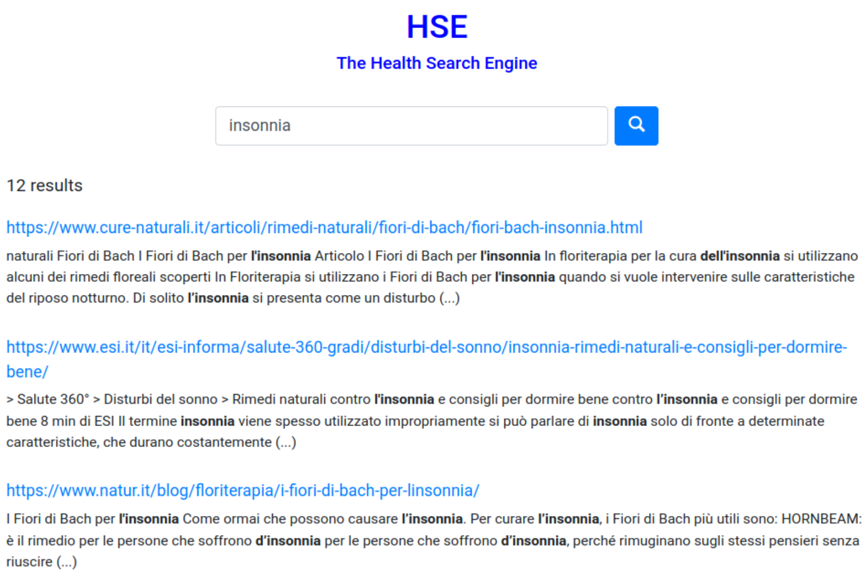


Figure 5. User interface available to participants

- 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.

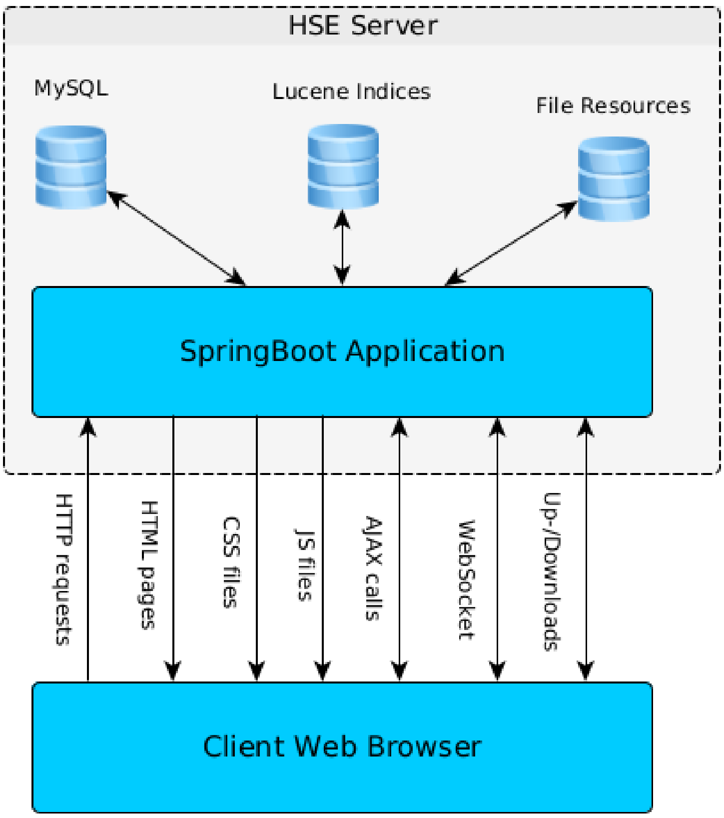


Figure 6. General System Architecture

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 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 I describe each layer 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, which also determins the services they depend on. The `endpoints` package contains the following classes: `AdminController`, `AuthController`, `ExperimentsController`, `FromSurveyController`, `IndexingController`, `SearchController`.

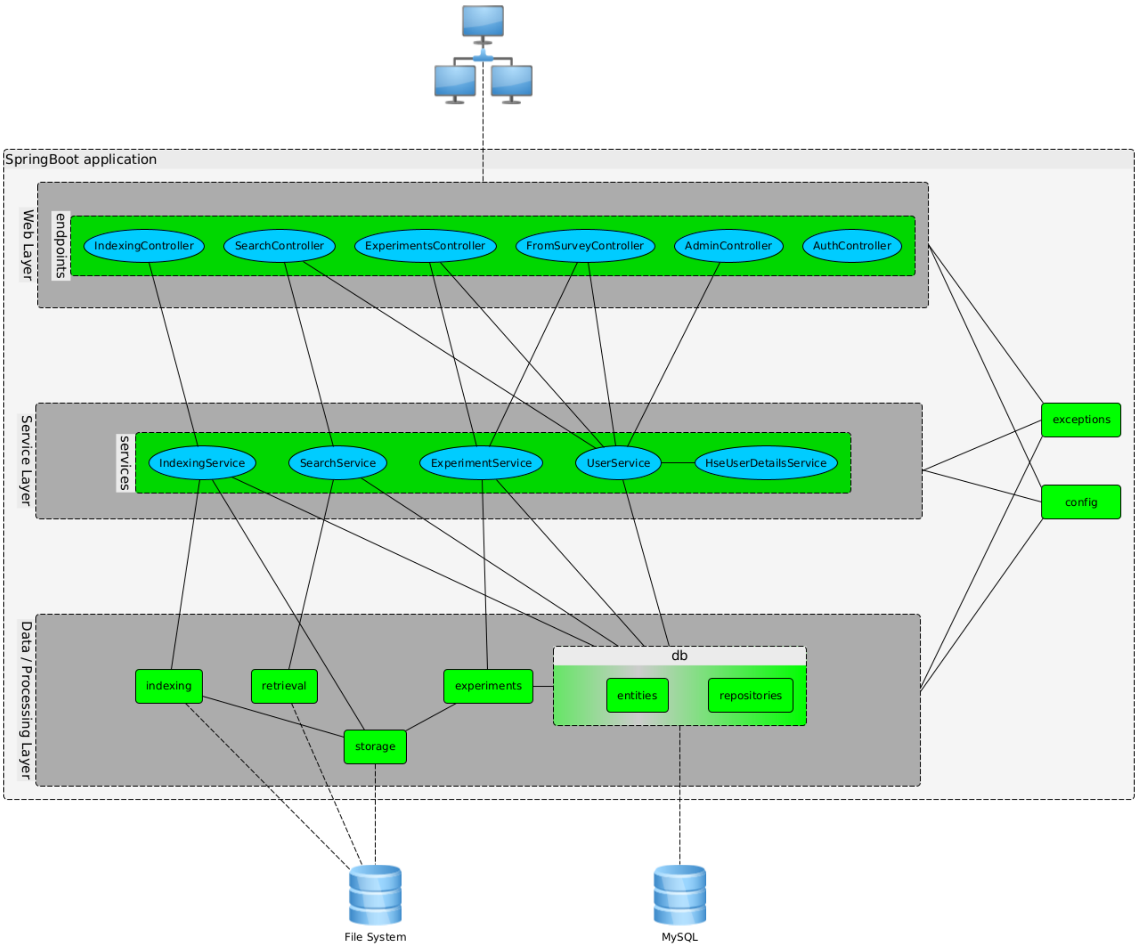


Figure 7. Layered architecture of the back-end application

5.4 The Service Layer: Service Classes Providing the Core Functionalities

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

6 Implementation details

6.1 User Management

6.2 Indexing

6.3 Retrieval

6.4 Data Extraction and Summary Generation

7 Future Work

8 Conclusions

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/applicatio`
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 8**). 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.

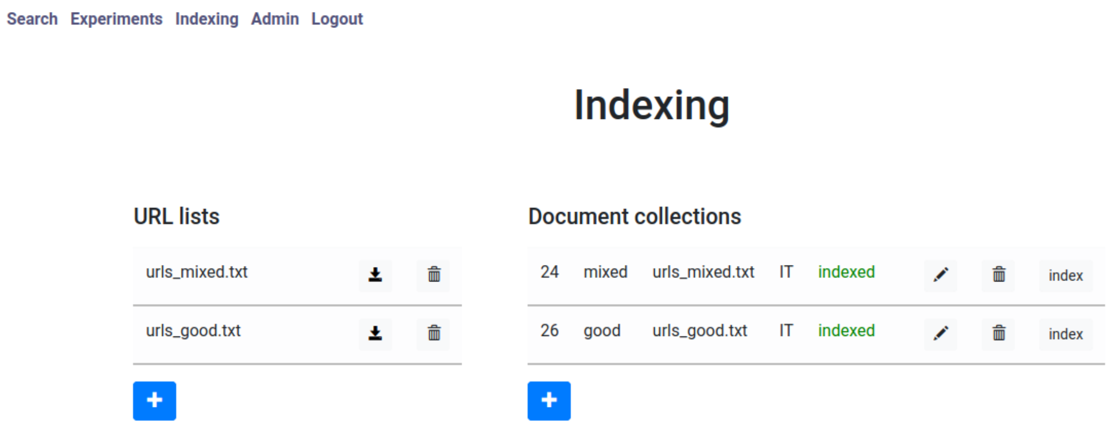
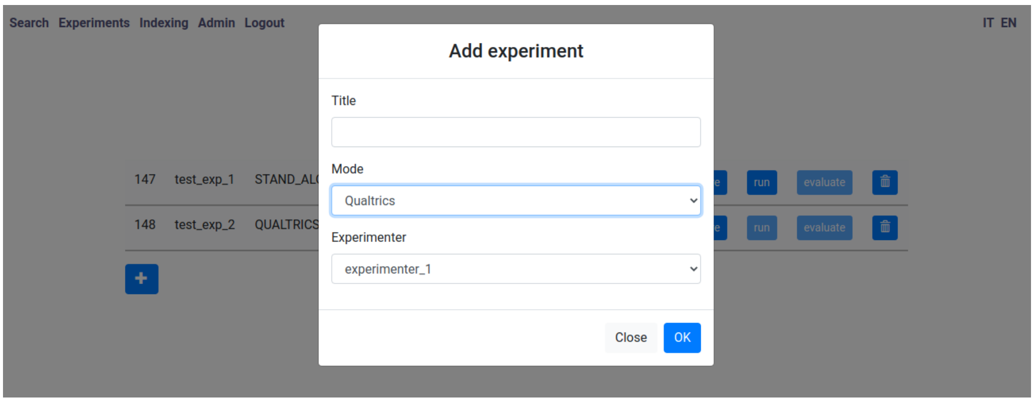


Figure 8. User interface for creating document collections

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 9** shows the experiments interface.

(a) Experiments list



(b) Popup for defining a new experiment

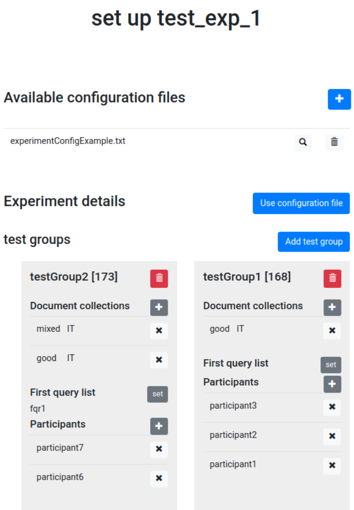
Figure 9. Interface for defining experiments

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 10a** shows the page after two test groups have been added and edited; **Figure 10b** 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 10. 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 11 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.

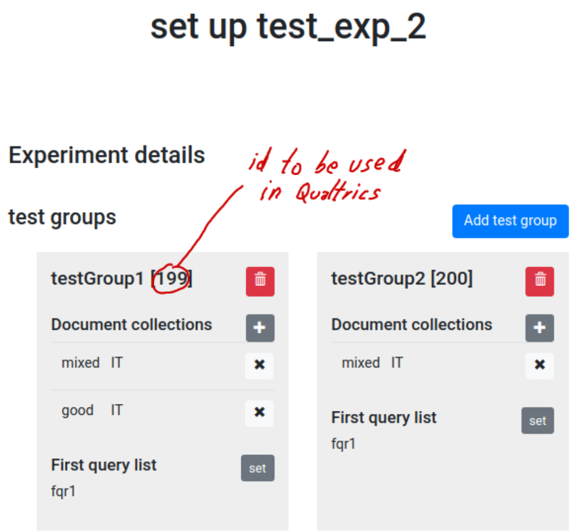


Figure 11. Experiment setup UI for Qualtrics mode

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 12** shows the interface for running an experiment.

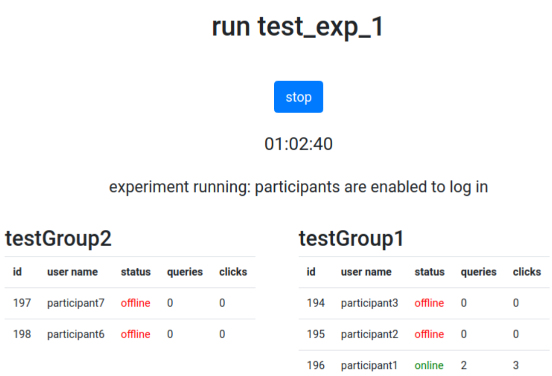


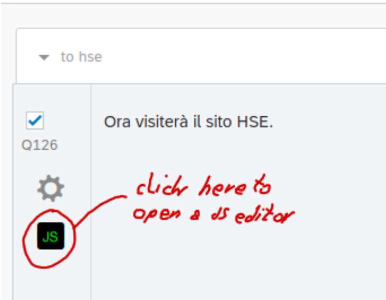
Figure 12. Experiments run UI

B.6 Setting up a linked Qualtrics survey

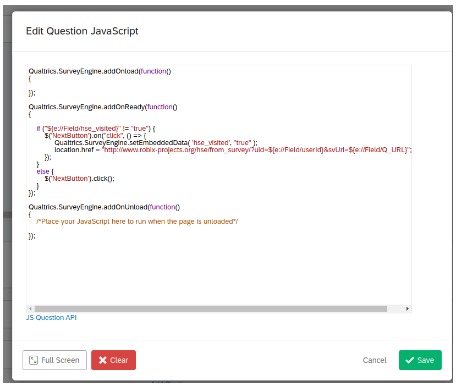
For linking a survey to an experiment, the survey must contain a block whose “next” button redirects to HSE, a failure block to be displayed in case accidentally the experiment is not running, or there is some connection error, and some settings at the beginning of the *Survey flow*.

B.6.1 Survey Questions

The redirection is set via *JavaScript* in the survey question (see **Figure 13**). The failure block needs no specific configuration.



(a) Opening the JS editor



(b) Edited JS

Figure 13. 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:

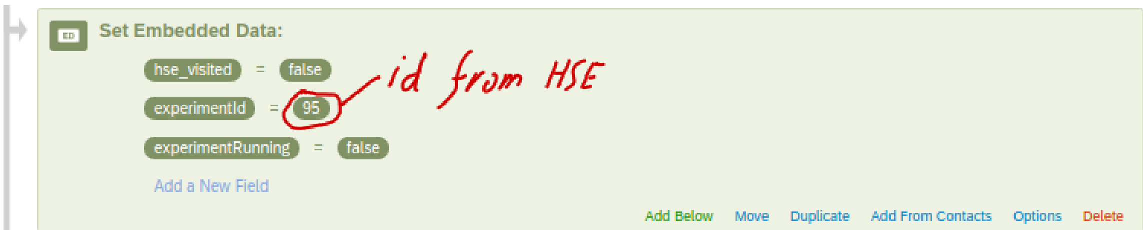


Figure 14. Embedded Data Element

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

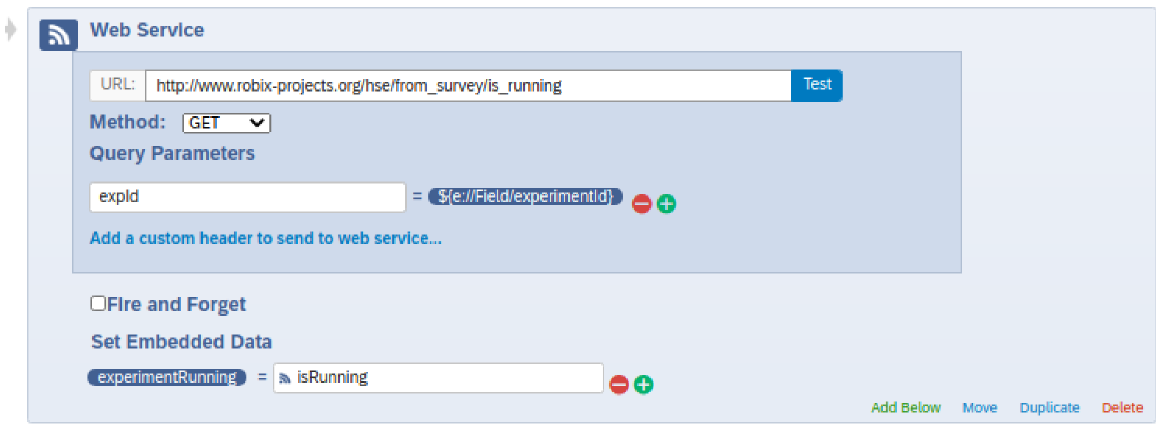


Figure 15. Web Service Element

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

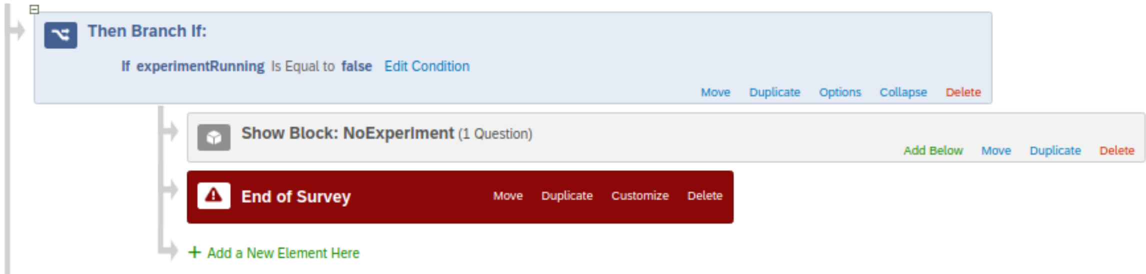


Figure 16. Branch Element

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

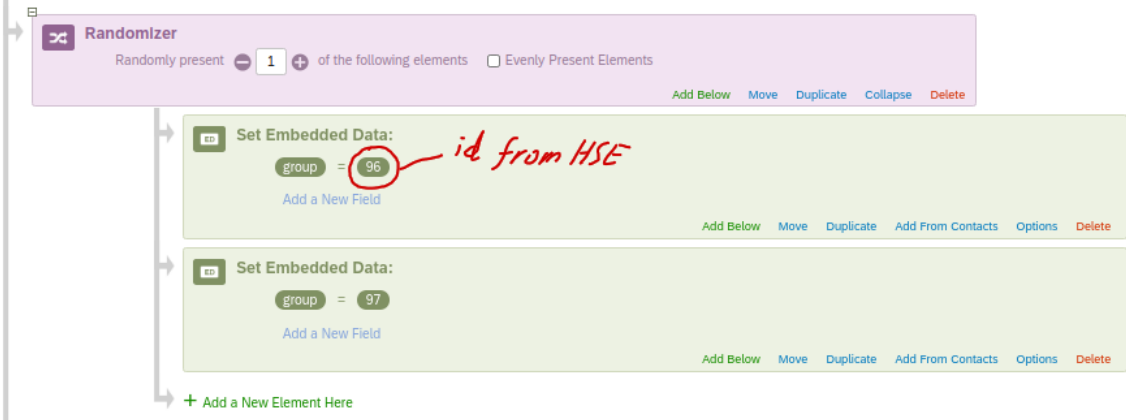


Figure 17. Randomizer Data element

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

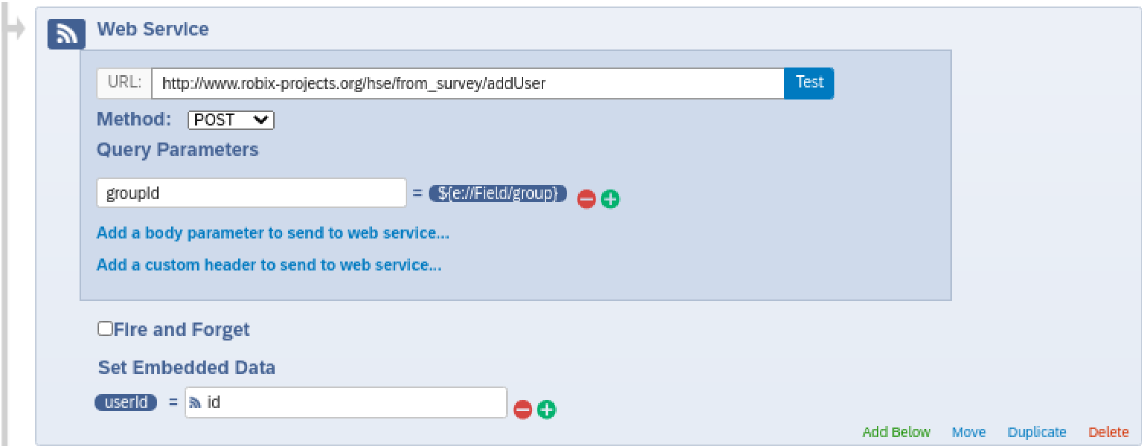


Figure 18. Web Service element

Figure 19 shows the entire edited survey flow.

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

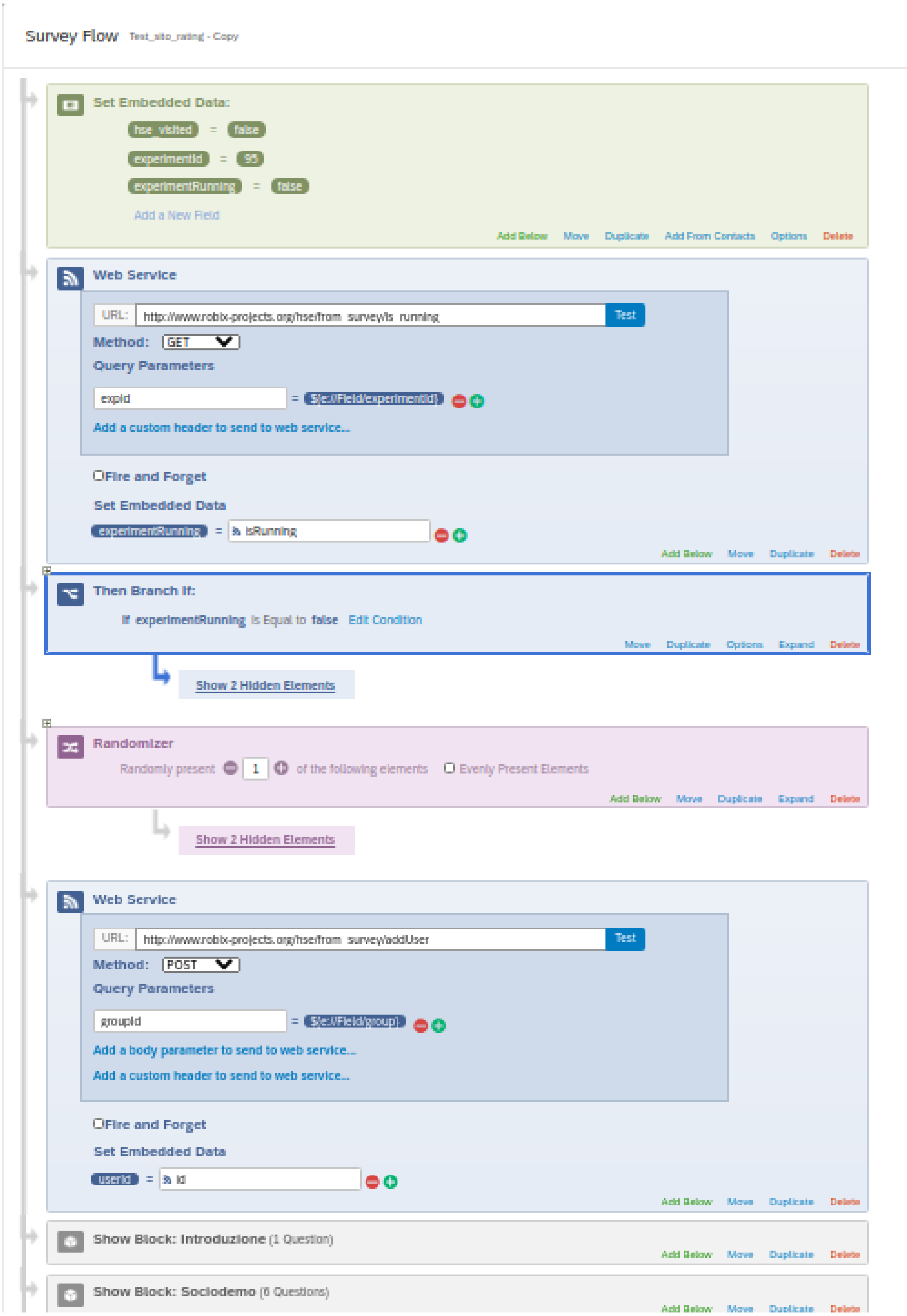


Figure 19. Edited Survey Flow