CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of electrical engineering

Department of computer graphics and interaction

Web application for collecting and evaluation of data from remote empiric user studies

Master's thesis

Study program:	Open Informatics
Thesis supervisor:	Ing. Ladislav Čmolík, Ph.D.
Author:	Bc. Jan Oravec, DiS.
Year:	2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Oravec** Jméno: **Jan** Osobní číslo: **457146**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačová grafika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Webová aplikace pro sběr a evaluaci dat ze vzdálených empirických uživatelských testů

Název diplomové práce anglicky:

Web application for collecting end evaluation of data from remote empiric user studies

Pokyny pro vypracování:

Seznamte se s typy empirických uživatelských testů (between subject, within subject) a způsoby provádění vzdálených empirických testů. Dále se seznamte technologiemi pro vytváření webových aplikací a s existující webovou aplikací pro sběr a vyhodnocení dat ze vzdálených empirických testů implementovanou v Ruby on Rails. Identifikujte části existující webové aplikace, které lze zachovat či pozměnit (např. databáze, knihovna pro odesílání dat) a navrhněte a implementujte novou webovou aplikaci v jazyce Java, která umožní:

- Registraci uživatelů na základě pozvánek administrátora.
- Vytváření empirických uživatelských testů se sekcemi a definicí proměnných, které se budou v rámci dané sekce sbírat.
- Sdílení (editace, prohlížení) testů mezi uživateli.
- Pseudonáhodné balancované přiřazovaní participantů do jednotlivých skupin testu.
- Sběr dat pomocí webové služby.
- Vyhodnocení sebraných dat pomocí intervalů confidence a p-hodnot.
- Zobrazení sebraných dat v grafech pomocí D3.js.

Vytvořenou webovou aplikaci otestuje na datech z experimentů provedených v existující webové aplikaci a porovnejte jak nová a existující aplikace data vyhodnotí.

Seznam doporučené literatury:

M. Kuniavsky. Observing the User Experience - A practitioner's guide to user research. Morgan Kaufman, 2003.

J. Sauro and J. R. Lewis. Estimating completion rates from small samples using binomial confidence intervals: comparisons and recommendations. In Proceedings of the human factors and ergonomics society annual meeting, volume 49, pages 2100-2103. SAGE Publications, 2005.

A. Agresti and B. A. Coull. Approximate is better than 'exact' for interval estimation of binomial proportions. The American Statistician, 52(2):119-126, 1998.

J. R. Lewis and J. Sauro. When 100% really isn't 100%: improving the accuracy of small-sample estimates of completion rates. Journal of Usability studies, 1(3):136-150, 2006.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Ladislav Čmolík, Ph.D. Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **02.02.2022** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **30.09.2023**

Ing. Ladislav Čmolík, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

© VUT v Praze, Design: VUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání

Podpis studenta

© VUT v Praze, Design: VUT v Praze, VIC

Declaration:

I declare that I have worked on this thesis independently and that I have stated all of the used information sources in accordance with methodological instructions on ethical principles for writing an academic thesis.

In Prague, ………………………………….   Author signature: …………………………………..

Abstract

This document describes reworking of an existing statistic computing application Sfinx, which gathers data from user interfaces, and returns statistical overview that is useful for gaining insight into chosen topic of empirical research.

Scope of the reworking is to convert its current implementation in programming framework Ruby on Rails to a new application written in language Java. The reason for this conversion is to achieve better maintainability and scalability of the new application while maintaining the functionality of its original.

Described theoretical background includes principles of empirical testing, between/within subject approaches for gathering data, and evaluation through confidence intervals and ANOVA formulas. Structure and function of Sfinx is also described in detail.

Design of the conversion was drafted with respect to each part of the Sfinx application and subsequently implemented. This also involves creation of new features, such as timeout of testers and sharing of experiments. The process of conversion is then described with focus on its integral mechanisms and components, and the resulting graphical interface is presented with its explanation.

Lastly the testing methodology is summarized, including a comparison between confidence interval computations of Sfinx and the converted application. This comparison concluded that the finished conversion provides outputs with no significant differences from the original implementation.

Completion of this thesis allowed easier maintenance of the statistic application and can be used as basis for its expansion. Its increased versatility will then allow more efficient approaches towards statistic science for its users.

Keywords

Confidence intervals – Statistic values which can be used to gain scientific insight from gathered data

ANOVA – Evaluation method, which can be used to gain scientific insight from gathered data

Empiric research – Scientific activity which attempt gain useful insight from data gathered by observing the world

User research – Type of empiric research, which gathers its data from users of applications or services

REST – Standard for creating interfaces, which maps server content and functions to browser URL

URL – Uniform resource locator, a string specifying location of resources on internet

GUI – Graphical user interface, provides visual controls to users

JSON – Type of notation which allows saving and transferring complex data hierarchies

Contents

Abstract 4
1 Introduction 7
2 Analysis 8
2.1 Empirical research 8
2.2 User research 8
2.3 Designing empirical experiments 9
2.3.1 Variables 9
2.3.2 Bias 9
2.3.3 Between subject design 11
2.3.4 Within subject design 12
2.3.5 Null hypothesis testing 13
2.3.6 Confidence intervals 13
2.3.7 Analysis of variance 17
2.4 Original application analysis 20
2.4.1 Intended purpose 20
2.4.2 Components 20
2.4.3 Workflow 22
2.4.4 Used technologies 23
2.4.5 Entity relationship model 25
2.4.6 User interface 28
3 Solution design 31
3.1 Initial task analysis 31
3.1.1 Conversion methodology 31
3.2 Application architecture 32
3.3 Proposed technologies 33
3.4 Backend application 35
3.4.1 Logical data model 36
3.5 Frontend application 40
3.6 Data gathering library 40
3.7 Requirements 41
3.7.1 Functional requirements 41
3.7.2 Non-functional requirements 42
4 Implementation 43
4.1 Initial preparation 43

4.2 Backend conversion 43
4.2.1 Project creation 44
4.2.2 Folder structure 44
4.2.3 Implementing logical model 46
4.2.4 REST implementation 47
4.2.5 Registration 47
4.2.6 Authentication 48
4.2.7 Sharing experiments 50
4.2.8 Data collection 50
4.2.9 Graph computation 52
4.3 Data gathering library conversion 53
4.4 Frontend conversion 53
4.4.1 React project creation 53
4.4.2 Source code structure 54
4.4.3 Routing implementation 55
4.4.4 Login 56
4.4.5 Graphs in D3.js 57
5 Results and verification 59
5.1 Finished application 59
5.2 Testing functionality 64
5.3 Verifying CI computation 65
5.3.1 Import of Sfinx data 65
5.3.2 Verification results 67
6 Conclusion 73
7 References 74
8 Attachments 76
Attachment A: Distribution tables 76
Attachment B: Interfaces of Sfinx application 77

1 Introduction

Empirical testing has its uses whenever there is a need to gain knowledge by means of observation and experience. With growing scope of testing however, it may become difficult to gather and evaluate data in a consistent and effective manner. This is why analysis tools based on user interfaces are often employed to help with the task.

The goal of this thesis is to rework source code of an existing analysis application Sfinx [1] into the Java language. The original project was previously written in Ruby on Rails framework, which was difficult to maintain, so the new project will be using Java language with framework Spring Boot to improve scalability and comprehensibility of the code.

Analysis chapter introduces the problematic of testing user interfaces and acquiring accurate awareness of its comprehensibility and usability. It will touch upon methods of data gathering used by the developers to acquire sufficiently unbiased data about the tested user interfaces, which are called between and within subject. This data can be then evaluated to create statistical overview by using methods of confidence intervals and ANOVA, which will be also introduced here.

Second part of the chapter will then take a look at the application that needs to be reworked, providing description of its general architecture, entity relationship model and intended workflow.

Solution design chapter puts forward proposed implementation of the new application written in Java language, based on information summarized in the previous chapters. While giving explanation of the planned project structure, it also describes new improvements that should be added.

Implementation chapter gives a report about the process of creating each part of the new application, divided into backend, frontend and gathering library sections. It will provide detailed insight into the functionality of integral parts in the converted application and describes the folder structure of the finished project.

Results and verification chapter showcases the graphical interface of the finished application and explains its control elements and purposes. Testing methodology is described next, providing information about how the correct application functionality was ensured.

Following section of the chapter then describes the process of computation output comparison between the original application and its conversion. Import of data from Sfinx database is explained, and the testing results are reported alongside with the drawn observations.

Lastly, the conclusion chapter summarizes steps taken and results achieved during the course of this thesis.

2 Analysis

In this section, empirical research concept will be introduced to understand the context in which the original application operates.

Then the architecture of the original project will be explained in enough detail to design the new reworked version of the project while maintaining functionality of its original.

2.1 Empirical research

Research procedure in general is a process, where a researcher gradually implements predetermined scientific acts, with the purpose of realizing a scientific goal [2]. This goal is typically verifying a scientific hypothesis or acquiring useful information about subject of the study.

Scientific acts that should be used during the research are determined by the chosen scientific methodology. In case of research using empirical methodology (known as empirical research), these acts are observation, measurement, and experimentation.

Empirical methodology is characteristic by its work with concrete data, which is based on existing observed subjects in the world, such as people or objects. It is also known as research that works with data acquired by experience. This methodology uses exact methods to reach specific information as its output.

2.2 User research

Since the topic of this thesis works with an application that gathers and evaluates data based on user input, thus conducting user research, it is important to introduce the concept of user research as well.

User research is the process of acquiring knowledge about preferences and behavior of users, that use a given product or service [3]. This knowledge is used to either gain scientific insight into the characteristics of the chosen user group, or to improve the service in question by adjusting it to better suit the given user group. In the latter case, this improves the quality and effectivity of use when working with the service or product.

This process is implemented by first designing experiments where groups of users produce empiric data by interacting with testing user interface (UI). There are several possible experiment design methodologies that affect circumstances under which the data is collected. Among these, **Within subject design** and **Between subject design** will be introduced in following chapters.

After the data is collected, it is then evaluated by using statistical processing methods. These methods define both specific statistical values that need to be computed from the gathered data, and the way to interpret them in a meaningful way for the sake of research. In the original application, **Confidence intervals** and **Analysis of variance (ANOVA)** have been implemented and will be introduced in later chapters.

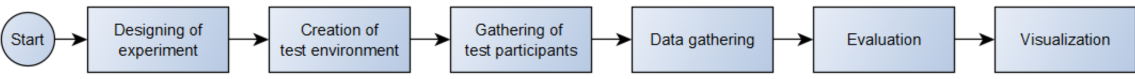


Image 1 – Workflow of empiric user research

Finally, the term "empiric user studies" in the title of this thesis comes from the fact that Sfinx application, whose conversion is the topic of this thesis, provides means to gather data from real people (application users) by means of observation, and uses exact statistical methodology to process the data, which are traits characteristic for empirical research.

2.3 Designing empirical experiments

During the course of empiric user studies, data is gathered in the form of categorized variable values that are collected for each experiment part. Such variables can represent anything that can be measured using the user interface used by the test takers.

Design of the experiment defines how these variables should be collected during testing, which has significant effect on accuracy and relevancy of the gathered data. In the worst case, poor choice of experiment design can lead to incorrect conclusions in the evaluation phase of the experiment.

Following sections will introduce types of test variables used in experimentation, bias that affects them, and the experiment designs used to gather variable data.

2.3.1 Variables

Variable types are mainly determined by the type of their content, and by the dependency between each other.

Continuous and discrete

When the variable contains values anywhere in a specified range, or if the possible values are spaced, it can be said that the variables are continuous or discrete, respectively. Continuous values can be for example time, temperature and speed, while discrete values include indivisible data such as number of people, or categorical data like gender of the tester.

Considering the type of variable content can be important when choosing the way to process the data, since some of the statistical methods won't be always applicable. One such case is mean value which is an often-used statistical value but is inapplicable directly on categorical data values such as "gender".

Dependent and independent

It is also possible for values of one variable to be dependent on values of another variable. In other words, this is the relationship between dependent and independent variables. To name an example, independent variable "age" can influence the dependent variable "test-completion-time".

Evaluating such relationships between variables is often the goal of the scientific studies, and it is one of the ways to obtain scientific insight from gathered data.

Confidence intervals evaluation and ANOVA are both methods that verify whether these relations exist.

2.3.2 Bias

When attempting to conduct scientific study using empiric experiments with user testers, it is important to be mindful of possible bias that can be introduced into the gathered data. This data will almost always contain some form of skew or a misleading value that does not reflect the real characteristic of the tested user group, which may affect accuracy of statistical calculations later on.

To mitigate this bias, researcher needs to be aware of factors that can produce skewed data during the course of their study and design the experiment in a way that limits their influence on gathered data.

Bias comes mainly from used measuring equipment, the testing environment, or the users that are subjects of the study.

Measuring equipment bias

Users that interact with the testing UI may use different controller hardware, such as keyboard and mouse, that varies in its ergonomic standards. This would result in different efficacy of use. Defining standard controller hardware for the experiment will remove this type of bias.

Testing environment bias

The testing environment can influence the gathered data through its effect on the participants of the test. Depending on the way the environment has been organized, test users will behave differently and produce different data. In this case, the important points of the organization are whether the environment is moderated by a researcher, and whether it takes place in a controlled laboratory environment.

When a moderating researcher is present, it is possible to monitor the data gathering process in detail, whether in person or remotely, providing additional guidance to the testing users and verifying the correctness of the process. While this is desirable, it carries with it the cost of employing qualified researchers to monitor the test groups, and the time requirements associated with it.

Controlled laboratory environment provides the benefit of being able monitor all aspects of the experiment in great detail but has the downside of not being easily scalable due to need of securing space in which the test is meant to take place, along with necessary test equipment and associated financial costs. Remote testing on the other hand depends on a public testing interface available through the internet, and can be also moderated remotely by a researcher, but its accuracy cannot be always guaranteed.

It is advised to choose the testing environment suited to the needs of the research study.

User bias

Among the bias types introduced into the experiment, user induced bias is one of the most difficult to control. The lead cause of this are the individual differences among test participants, and the human factor that can introduce mistakes into data.

Individual differences of the users are important to consider because it means that the data produced by taking the test will be different for each user. There will be differences in technical proficiency, personalities, and level of logical reasoning, to name a few of the traits. Depending on how the users are distributed into testing groups, this can produce skewing of the data in groups where some of the personal characteristics are prevalent.

These same differences can be occasionally taken to the extreme, when a small number of the test takers produce data far different from the other users. These users are known as statistical outliers and should not be considered in the evaluation of the gathered data, since they don't actually represent their group.

Another kind of bias is created by the effect of the testing process on the users. Sometimes, an extensive amount of data is needed from each tested user, which can produce some degree of

fatigue. The longer the testing goes on for each individual user, the higher is the chance of making mistakes that otherwise wouldn't be made in the gathered data.

By the same coin, fatigue can also reduce motivation of the users, although motivation itself is also a factor to take note of. Not all experiments require or are able to enforce completion of the entire test. This means that in some cases, users may leave the tests unfinished, especially when using remote testing methods.

Lastly, some of the experiments are designed as a series of similar tests, with specific differences among them. When the users take the first test, they will be able to gain experience and will be able to progress in the following test with progressively greater efficiency. This is called learning effect, and it is another kind of statistical bias, which may be sometimes unwanted in the experiments.

All of these factors can be reduced by using appropriate methodology when gathering and evaluating data. Following chapters will introduce the ones used in this project.

2.3.3 Between subject design

This is one of the data gathering experiment designs [4]. As mentioned in the previous chapter (see chapter "2.2 User research"), participants of the study are distributed into several groups, where each group is assigned to interact with a single user interface.

In case of between subject design, each participant can be part of only one group, which means they will interact with only a single interface, before ending their participation in the experiment.

Interface A	Interface B	Interface C	Interface D
Tester 1	Tester 2	Tester 3	Tester 4
Tester 5	Tester 6	Tester 7	Tester 8
Tester 9	Tester 10	Tester 11	Tester 12

Table 1 – Between subject tester distribution

Advantages:

Since each user only tests a single interface, the learning effect bias is essentially eliminated because the users won't be able to use their experience from previous testing to skew the data anymore.

For the same reason, there is also much less need to worry about bias caused by fatigue, and missing data due to unfinished tests.

Lastly, this design can be easily extended to work with more test participants, since the testers can be picked randomly to keep the tester groups mostly equal in their characteristics when compared to each other. This is especially useful when the scientific study requires input from a large number of people.

Disadvantages:

Many participants are needed to implement this kind of design, which increases financial costs associated with hiring them, and securing a sufficient testing environment. Having a large sample of gathered data is usually desirable in testing, but its requirement to secure a large number of testers may be detrimental when such scope is not required.

There is also a chance that using a large amount of testers will produce statistical noise, which may obfuscate some of the otherwise observable relationships in the gathered data.

2.3.4 Within subject design

Data gathering process in this experiment design [4] differs from between subject design in several aspects. The most notable difference being that every participant is required to test every interface section in the scope of experiment.

Therefore, we need only a single group of testers, where each participant is assigned the order in which they should interact with the testing interfaces.

Interface A	Interface B	Interface C	Interface D
Tester 1	Tester 1	Tester 1	Tester 1
Tester 2	Tester 2	Tester 2	Tester 2
Tester 3	Tester 3	Tester 3	Tester 3
Tester 4	Tester 4	Tester 4	Tester 4

Table 2 – Within subject tester distribution

Advantages:

Advantage of this approach is the small number participants required to conduct the experiment, since it effectively only needs to fill a single group with interchangeable ordering.

Even more important is suppression of noise in gathered data, otherwise caused by individual differences among testers across the experiment.

Disadvantages:

Learning effect remains a problem, as every tester gathers proficiency in using the interface, and carries over such influence into the gathered data. And since every tester must participate in every part of the experiment, fatigue may also become a factor, reducing tester motivation and data consistency towards the end of the experiment.

To fight this problem, an ordering technique called **counterbalancing** can be employed to balance such errors. Idea of this technique lies in balancing the number of participants that tested the given interface early, and the number of those that tested the same section later in the experiment. This means that each participant will be assigned an order in which they should test the interfaces to reduce the bias.

One such testing order distribution is for example latin square distribution shown in Table 3 below, where no interface appears on the same row and column more than once.

Testing order	Tester 1	Tester 2	Tester 3	Tester 4
1	Interface A	Interface B	Interface D	Interface C
2	Interface B	Interface A	Interface C	Interface D
3	Interface C	Interface D	Interface A	Interface B
4	Interface D	Interface C	Interface B	Interface A

Table 3 – Latin square distribution

** It should be noted that within subject design will not be used in the design or implementation of the reworked statistic application. This section has only informative character for the sake of understanding concepts that the original application worked with.*

2.3.5 Null hypothesis testing

In statistic science, null hypothesis H_0 is a claim suggesting that no statistical relationship exists between certain characteristics in a set of given observations and is used to confirm credibility of a hypothesis by using sampled data from the full population [34]. Its opposite is the alternative hypothesis H_A, which is an inversion of the null hypothesis.

Null hypothesis testing is used to ascertain validity of the null hypothesis within a certain probability, or confidence level. Hypothesis testing is usually conducted with the value 0.95. Another value used in the testing is significance level, computed as 1 - confidence level.

Since null hypothesis testing only uses a subset of the complete population data, null hypothesis only assumes that the claim is very likely. If the null hypothesis is disproven, the alternative hypothesis is confirmed instead.

Testing is conducted by first assuming that the null hypothesis is true and sampling the population for data to be compared. Then the data characteristic is compared with the null hypothesis and the difference is measured. Lastly the p-value is computed, which denotes the probability that the observed difference is due to chance, and its value is compared to significance level. If p-value is less or equal, the null hypothesis is considered disproven.

2.3.6 Confidence intervals

Confidence intervals evaluation is a data processing method that can be used to verify relationships between independent and dependent variables defined in the design of the experiment. In other words, the researcher makes a null hypothesis which states that the values of a certain variable have a statistically significant effect on values of other variable and tries to prove or disprove such statement. The goal of this process is to extend validity of such a conclusion to the whole population, that the chosen subset of testers was taken from.

In theory, the relationship between variables could be verified by simply observing if the mean of all values for dependent variable changes once the independent one does. But since the experiment can only gather data from a subset of testers, since testing the whole population would be usually logistically implausible, there will be accuracy errors caused by not having entire data from the population. However, it is possible to get sufficiently accurate estimation by calculating **confidence intervals** from the gathered data and using those values for verifying the hypothesis instead.

Confidence interval refers to the range of values where the mean of values for a given variable would likely fall into, if data from the entire population was used [3]. The narrower this range is, the more stable is the measurement accuracy.

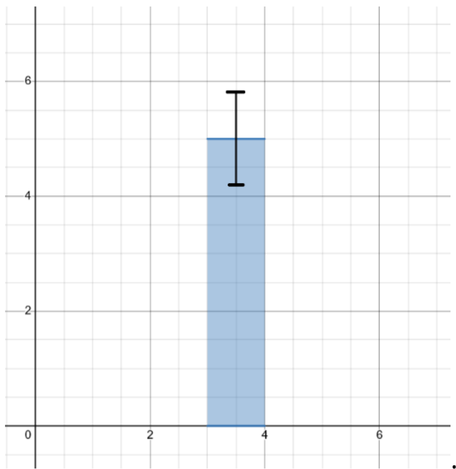


Image 2 – Bar chart with confidence interval whiskers

There are several input factors influencing the width of calculated confidence interval.

- **Level of confidence** is the likelihood that data collected from someone in the whole population will be included in the range of the confidence interval. Typical probability chosen for the calculation is 95%, but values of 90% or 99% are also commonly used. The higher the level of confidence, the wider confidence interval becomes.
- **Sample size** is the number of testers that became part of the experiment. As their number increases, there will be less error caused by not accounting for the entire populace, and the confidence interval will become narrower.
- **Variability** refers to the divergence of individual data samples from each other, and from the mean of all values. It is quantified by calculating standard deviation value, which is one of the values used in statistics for evaluation of data. With higher variability, the width of confidence interval increases.

Verification of hypothesis

To determine whether a change in an independent variable causes statistically significant change in the observed dependent variable, it is possible to compare confidence interval ranges of the dependent variable before and after the change of the independent variable [5].

In general sense, the change can be considered statistically significant if the range of both confidence intervals don't overlap, which by extension means the null hypothesis is considered valid. This method can be inexact however and should be only used for rough estimation.

More optimal way to verify dependency relationship between variables is to compute p-value from the found confidence intervals, which then determines validity of the null hypothesis.

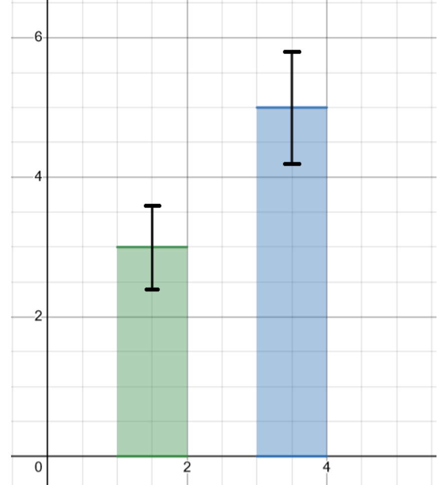


Image 3 – Non-overlapping confidence intervals

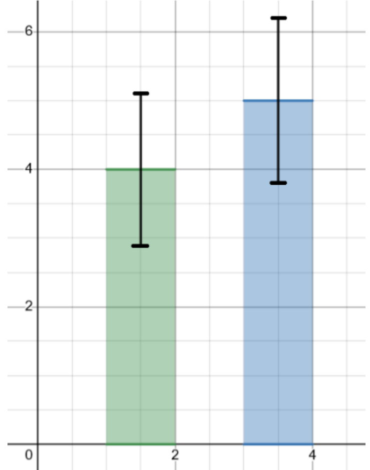


Image 4 – Overlapping confidence intervals

Calculation

Calculation process of confidence interval differs, depending on type of gathered data we use as input. As noted in previous chapter (see chapter "2.3.1 Variables"), the variable values can be either continuous or discrete, both of which require slightly different formula to calculate.

Values that are needed for both formulas are the following:

- **Level of Confidence,** which been already covered in text above, is chosen by the researcher to determine stability and accuracy of the estimated confidence interval. For the probability 95%, value 0.95 would be used in the calculation formulas.

- **Mean** (continuous) or **Proportion** (discrete) are both calculated from the gathered data. In case of mean, all values are summed and then divided by number of samples. Proportion on the other hand first counts the values considered as positive (a value that is monitored), and then also divides them by number of samples.

$$\underline{x} = \frac{\sum \quad x}{n} \tag{1}$$

$$\underline{p} = \frac{p}{n} \tag{2}$$

<u>Legend:</u>
$\underline{x}$ – mean
x – continuous data values
n – number of samples
$\underline{p}$ – proportion
p – positive values

- **Standard deviation** is the average distance of the gathered data values from their mean or proportion.

$$SD = \sqrt{\frac{\sum \quad (x - \underline{x})^2}{n-1}} \tag{3}$$

Legend:
SD – Standard deviation
$\underline{x}$ – mean
x – data values
n – number of samples

- **Critical values** are the values that divide the distribution graph of gathered data into sections that are considered and those that are not [6]. Values are considered when the chance of them appearing in the data is high enough, which is determined by the data distribution that applies to the gathered data.
  In order to obtain critical value for our calculation, we need to know the type of distribution used, degrees of freedom and level of significance. After determining these values, the correct critical value can be found in reference tables for the given distribution (see **Attachment A** of this document).

$$df = n - 1 \tag{4}$$

$$a = 1 - c \tag{5}$$

Legend:
df – degree of freedom
a – level of significance
c – level of confidence
n – number of samples

After calculating these values, it is finally time to compute the confidence interval.

For **continuous values**, t-distribution [7] must be used to obtain the critical value for calculation (see above). The upper and lower bound of confidence interval can be computed with the following formula [8]:

$$CI = \underline{x} \pm T \times \frac{SD}{\sqrt{n}} \tag{6}$$

Legend:
CI – Confidence interval bounds
$\underline{x}$ – mean
T – critical value of the t-distribution
SD – Standard deviation
n – number of samples

For error/completion rates expressed as **discrete values**, adjusted Wald binomial confidence interval calculation [9] can be utilized. This type of calculation uses normal distribution to obtain the critical value. The upper and lower bound of confidence interval can be computed with the following formulas:

$$p_{adj} = \frac{x + \frac{Z^2}{2}}{n + Z^2} \tag{7}$$

$$CI = p_{adj} \pm Z \times \sqrt{\frac{p_{adj}(1 - p_{adj})}{n + Z^2}} \tag{8}$$

Legend:
CI – Confidence interval bounds
p_{adj} – Wald adjusted proportion
x – number of participants that successfully completed the experiment
Z – critical value of the normal distribution

Lastly, there will be cases when the gathered data will be subject to **skewing** [10]. This means that the distribution of gathered data will become distorted, with higher probability in either right or left side of the graph. Variables such as "completion time" have especially high tendency to produce such skewing.

In order to counteract the effect of skewing on confidence interval calculation, there is the option to employ logarithmic transformation on every value of the affected variable. Once the calculation is completed on the transformed values, exponential operation is applied on the results to obtain non-transformed confidence interval.

2.3.7 Analysis of variance

Another method of data evaluation is the **Analysis of Variance (ANOVA)**. Much like the confidence interval evaluation, the goal of this method is to verify a null hypothesis about relationship between independent and dependent variables. And similar as before, it tries to extend the validity of this hypothesis to the entire population that the subset of testers in the experiment were chosen from.

** It should be noted that ANOVA evaluation will not be used in the solution design or implementation of the reworked statistic application. This section has only informative character for the sake of understanding concepts that the original application worked with.*

The difference lies in the way of establishing validity of such relationship. The null hypothesis of ANOVA is that the means of data collected from 3 or more independent groups are equal [11]. The null hypothesis will always have this meaning, and if it is disproven, it indicates that there is a significant statistical difference among the means calculated from data of the independent groups.

Verification of hypothesis

To determine whether the null hypothesis holds or not, the most important value that needs to be computed is the p-value. Once it is obtained, the null hypothesis can be simply verified by comparing the p-value with the level of significance (see Formula 5).

If the p-value is strictly greater than level of significance, then the null hypothesis holds, and there isn't any significant difference between the groups.

Otherwise, if it is less or equal there is a significant difference, and the hypothesis doesn't hold.

Calculation

Unlike the confidence intervals, ANOVA needs to adhere to several conditions to acquire accurate data.

- Firstly, ANOVA needs the dependent variables to be **continuous** only. The calculations of ANOVA do not account for discrete values and cannot be used in such way.
- The evaluation is sensitive to **statistical outliers** and might produce inaccurate results if these are present.
- The gathered data should have **normal distribution**.

If these conditions are met, next step is to choose a suitable set of ANOVA formulas to determine p-value. This document convers two types of ANOVA calculations: One-way ANOVA and Repeated measures ANOVA.

One-way ANOVA is used for data collected from experiments that employ between subject design (see chapter "2.3.3 Between subject design"). Following is the set of formulas [12] used for calculation:

	SS	**DF**	**MS**	**F**
Between	$\sum_{i=1}^{k} n_i(\underline{x_i} - \underline{x})^2$	k - 1	$\frac{SS_B}{k-1}$	$\frac{MS_B}{MS_W}$
Within	$\sum_{i=1}^{k} \sum_{j=1}^{n_i} (x_{ji} - \underline{x_i})^2$	N - k	$\frac{SS_W}{k-1}$	-
Total	$SS_B + SS_W$	N - 1	-	-

Table 4 – One-way ANOVA calculation formulas

<u>Legend:</u>
SS_B – Sum of squares Between groups
SS_W – Sum of squares Within groups
MS – Mean square

DF – Degrees of freedom
F – F-statistic
N – total number of observations
k – number of groups
n_i – number of observations in group i
$\underline{x_i}$ – mean of observations values in group i
$\underline{x}$ – mean of all values
x_{ji} – observation value on position j of group i

After these values are calculated, next step is to acquire F-distribution value from the distribution reference table (see **Attachment A** of this document). For this part, degree of freedom of Within subject and Between subject rows of the Table 4 are required, as well as the chosen level of confidence.

For the last step, use the found F-distribution value and both degrees of freedom to determine the p-value. It is possible to use an online statistical tool to calculate this value [13].

Repeated measures ANOVA uses almost entirely the same principles and formulas as the One-way ANOVA. The main difference is that this method is used for data obtained from experiments that employ Within subject design (see chapter "2.3.4 Within subject design"). This means that the calculation is made on data from the same set of subjects, where each subject was tested in each group of the experiment.

Following is the set of formulas [14] required to calculate this type of ANOVA:

	SS	DF	MS	F
Between	$\sum_{i=1}^{k} S(\underline{x_i} - \underline{x})^2$	k - 1	$\frac{SS_B}{k-1}$	$\frac{MS_B}{MS_{Err}}$
Within	$\sum_{i=1}^{k} \sum_{j=1}^{S} (x_{ji} - \underline{x_i})^2$	N - k	-	-
Subjects	$k \sum_{i=1}^{S} (\underline{s_i} - \underline{x})^2$	S - 1	-	-
Error	$SS_W - SS_{Subj}$	$DF_B * DF_{Subj}$	$\frac{SS_{Err}}{DF_{Err}}$	-

Table 5 – Repeated measures ANOVA calculation formulas

Legend:
SS_B – Sum of squares Between groups

SS_W – Sum of squares Within groups
MS – Mean square
DF – Degrees of freedom
F – F-statistic
N – total number of observations
k – number of groups
S – number of subjects (participants), also number of observation values in every group
$\underline{s_i}$ – mean of observations values from subject i
$\underline{x_i}$ – mean of observations values in group i
$\underline{x}$ – mean of all values
x_{ji} – observation value on position j of group i

The same way as with One-way ANOVA, next step is to find the correct F-distribution in the reference table. Do note that this time, one of the degree of freedom values became DF_{Err} when referencing the table. Otherwise, the process remains the same, including finding of the p-value.

2.4 Original application analysis

The following chapters provide in-depth description of application Sfinx, while paying extra attention to the structure and functionality that should be preserved during the conversion of Sfinx to Java language.

This understanding then becomes the basis for design of the new reworked application.

Project leader of this thesis has provided access to the public GUI of the currently deployed Sfinx application [15], and source code of the Sfinx project [1] for the sake of analysis and future reference in this document.

2.4.1 Intended purpose

Application Sfinx is a web application written in the language Ruby on Rails, which employs server-client architecture and includes a publicly available library to allow gathering of data from other application interfaces.

Primary users of this system are researchers and developers, who wish to conduct statistical study for the sake of empirical research or to improve their implemented service/application, respectively.

Main functions of Sfinx are provided by the server interfaces for gathering data from participants, and an interface for researchers which shows result of processing the collected data into statistical values and graphs.

While there are other such statistical tools already available on the internet, these tools tend to specialize on a specific type of computation or collected data. Contrary to this, Sfinx aims to consolidate all of these computation functionalities into a single application, which can be freely extended as required.

2.4.2 Components

General structure of Sfinx application can be seen in the component diagram below. Subsequent paragraphs then describe each component of Sfinx and its role in the system.

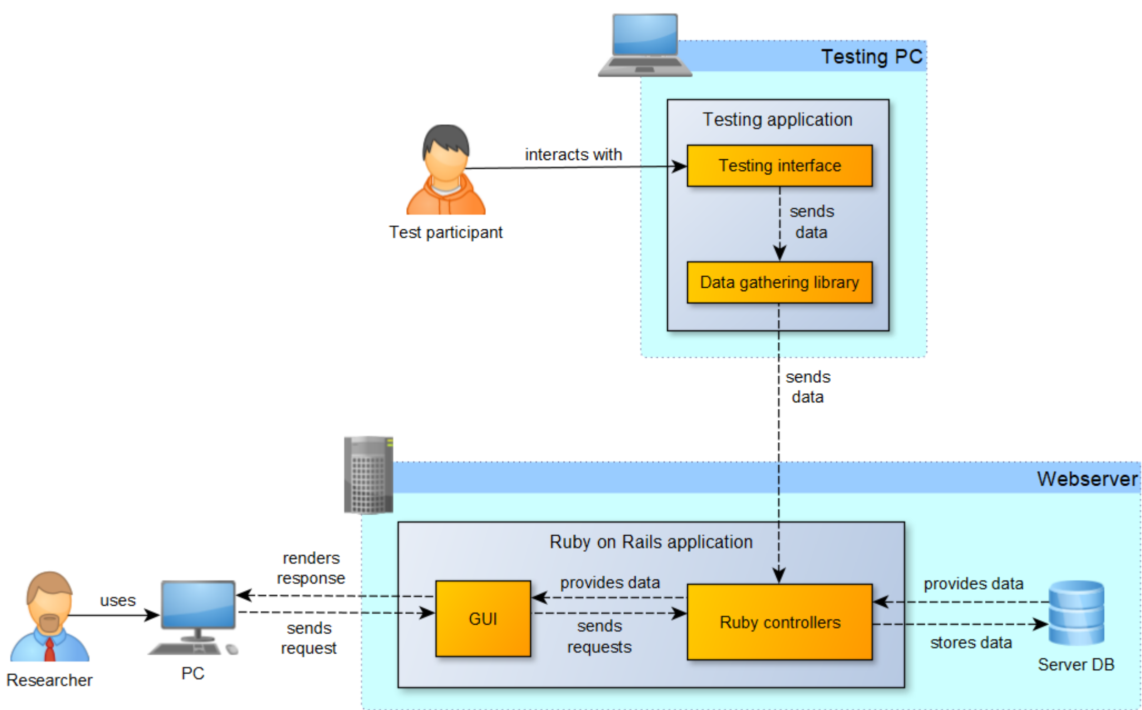


Image 5 – Component diagram of Sfinx

Webserver:
The device which runs the core of the Sfinx web application. Sfinx contains Ruby controller classes which contain all service-level logic for handling and persisting data, and also maps its methods to REST API interfaces available for calling.

Some of the interfaces are used in their raw form to gather data from experiment participants, while the remainder of them are connected to web GUI within the Ruby on Rails framework, which allows researchers to manage their experiments.

The database system is hosted on the same server as the Sfinx application, although its placement there is optional as it can be hosted on a different dedicated server.

Researcher:
Researchers are the primary users of Sfinx web application. These users have rights to create and manage experiments, make them available to the test participants, and browse the collected statistical data by using the GUI of Sfinx webserver.

Testing application:
Application or service with a user interface, meant to be interacted with by the test participants. Developers of this service should transmit results of these user interactions to the public data gathering library provided by Sfinx project, which handles submitting of data to the server.

So long as the data processing library is properly installed, the researcher is free to use any application or service to implement their experiment design.

This component must be available to every participant of the test, which means that multiple testing devices may need to be prepared to run the testing interfaces.

Test participant:
This type of user provides data for Sfinx by participating in experiments designed by the researchers. They are hired or otherwise persuaded by the researchers to interact with a provided testing application, which in turn sends their input as data to the main server.

Aside from sending data through the testing application, they do not possess any other access rights to Sfinx.

2.4.3 Workflow

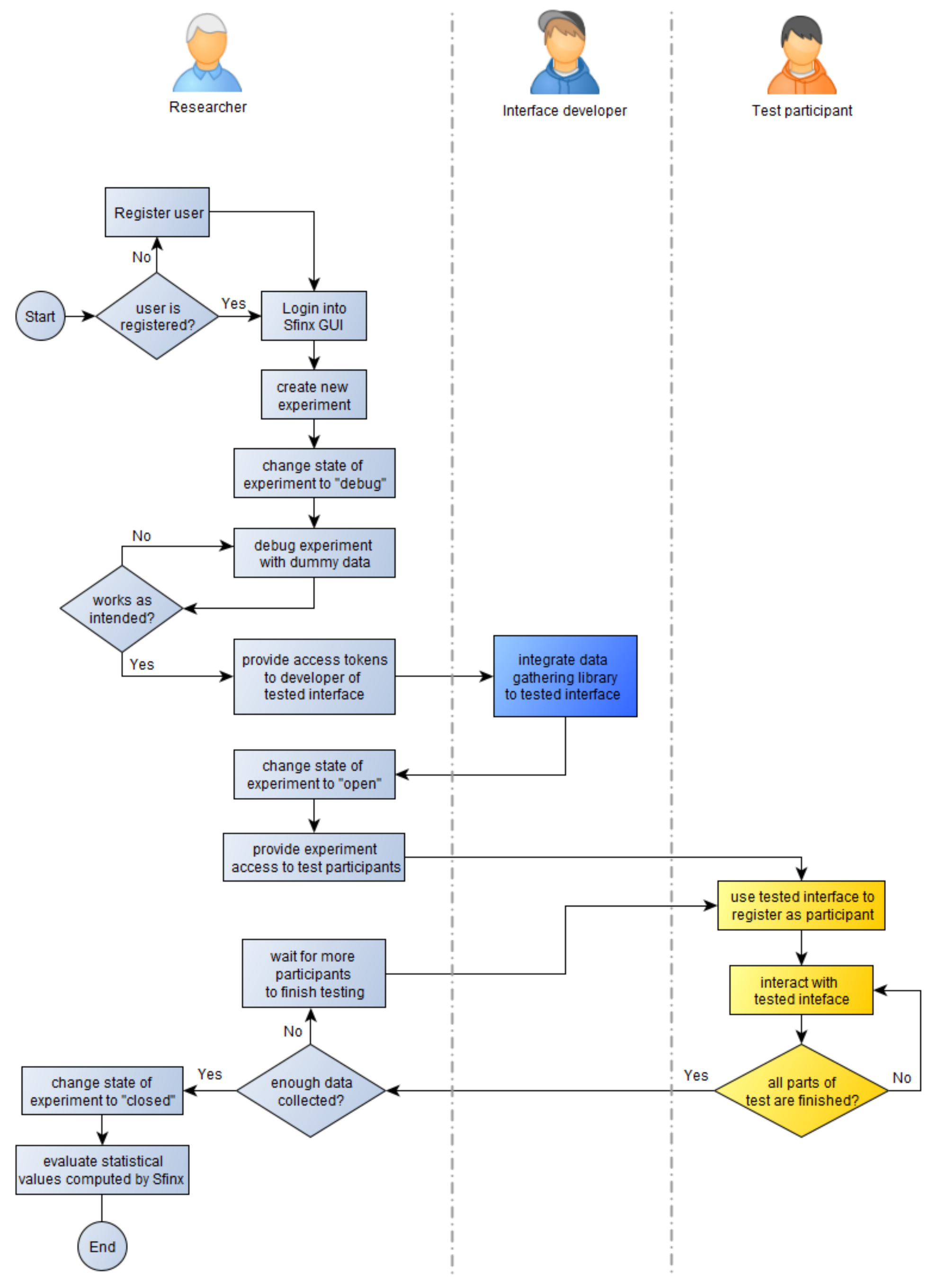


Image 6 – Process diagram of Sfinx lifecycle

After consulting with the supervisor of this thesis and analyzing the deployed application [15], along with available project files [1], the lifecycle process of the Sfinx system has been summarized in this

chapter. Further paragraphs also explain possible state transitions and of experiments in the system and express then in a state diagram.

The above diagram shows the process of implementing an experiment from its creation to its conclusion.

To use the application, developer must first create an account through a web interface hosted by the server.

After successful registration, developer then creates an experiment in the system that will keep track of gathered data in logical structure explained by chapter "2.4.5. Entity relationship model".

While designing the experiment, researchers are encouraged to test the experiment interface with dummy data, to verify the correct structure of collected data. Switching the state of experiment to "debug" state allows the interface to collect data, and if the state is switched back to state "edit", all collected data of the experiment is erased to remove its influence on future computations. Switching to "open" does not have this effect.

Once the researcher is satisfied with the experiment structure, they shall provide the access_tokens of experiment and its parts to the interface developer. When developer integrates the data gathering library into the tested interface, they can use these tokens to authenticate the participant registration and data uploading requests made to the Sfinx application interface. Sending of the requests should be later triggered by the test participant when they interact with the tested interface.

If there are no problems with the integration, researcher can then switch the state of the experiment to "open". At this point, experiment is meant to store actual data of the experiment and cannot be switched back to its previous state. Next step is then to provide the tested interface to test participants, as per the experiments design.

Once all necessary data is gathered, developer can switch the experiment to a "closed" state, which will prevent any other data from being collected. The researcher can then review final statistical results computed by Sfinx, which are based on the collected data.

The following diagram summarizes all possible state transitions used in the experiment:

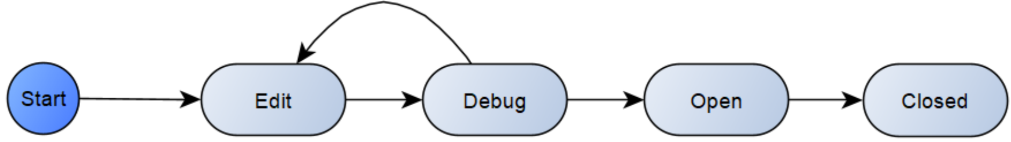


Image 7 – Possible state transitions of Sfinx experiment

2.4.4 Used technologies

JavaScript:
Scripting and multiplatform programming language which is most commonly used for defining behavior of user interfaces in web browsers [28].

JavaScript is utilized by the data gathering library used by Sfinx, and it is also the basis for many tools made in the Node.js environment.

Node.js:
This open-source, multiplatform environment [16] is designed for creating and running web applications in language JavaScript.

Sfinx application employs Node.js to support its packaging system Yarn, and the tools downloaded with it.

Version used in Sfinx: 8.3.0

Yarn:
Yarn is a package management tool [17] running in Node.js environment, which is used to download tools and dependencies for the Sfinx project.

File named “package.json” in the root of the project folder is used by Yarn to keep track of information about dependencies that are required by the project. Dependencies declared in this file are then downloaded or updated every time the build command is called on the Sfinx project.

Version used in Sfinx: 1.6.0

Ruby:
General purpose, open-source and multiplatform programming language [18], with interpreted source code and great focus on performance.

It has been made widely popular due to the success of its framework named Ruby on Rails, which was used to create the Sfinx project.

Ruby on Rails:
Ruby on Rails is an open-source framework [19] based on Ruby language, intended for faster development of web applications. Projects that are created in this framework follow the model-view-controller (MVC) architecture and provide much of its functionality by convention rather than implementation.

This means that the programmer can create an application quickly, while relying on the default settings of the framework and only changing what they need. By the same coin however, these default settings are often not part of the written code, which can make understanding the code much more difficult unless the developer is familiar with the conventions in the framework.

The steeper learning curve produced by this requirement is the main reason why Sfinx needs to be converted to a different programming language.

Version used in Sfinx: 5.1.0

Puma:
Web server for deploying web applications written in Ruby. Allows the application to be made available for request from web browsers.

Its equivalent in Java applications would be Apache Tomcat [25].

Version used in Sfinx: 3.11.4

Slim templates:
Template language [20] which is used in Sfinx as an alternative to HTML language. It has been designed as an attempt to remove most of the redundant syntax which can be seen in HTML, while still being human-readable.

Sfinx utilizes this language to define its server frontend.

REST:
Architectural standard, which allows to map browser URL addresses to functions of the server. Generated automatically by Ruby on Rails for any defined controller functions.

PostgreSQL:
Database system [29] responsible for storing all persistent data of the Sfinx application. Its tables are queried by Ruby on Rails through its ActiveRecord class, which maps itself to tables in the database.

Chart JS:
JavaScript library [21] which allows visualization of processed data as statistical graphs by the Sfinx server frontend. Computed results can be then presented in a more comprehensible way to the researchers.

Devise:
Tool for Ruby on Rails which provides pre-made solutions for authentication in the application [22]. Provides modular implementations to solve login handling, registering users, logging access information, and generating of user entities which map to database table.

2.4.5 Entity relationship model

Analysis of the Sfinx source code [1] has revealed an automatically generated file named “schema.rb”, which contains information about the entire database schema structure at the time of the application's last start. Ruby on Rails also employs class ActiveRecord to map application objects to tables in the database.

The diagram below describes data entities indicated in source code of the application, which are used to handle and store data in the Sfinx application:

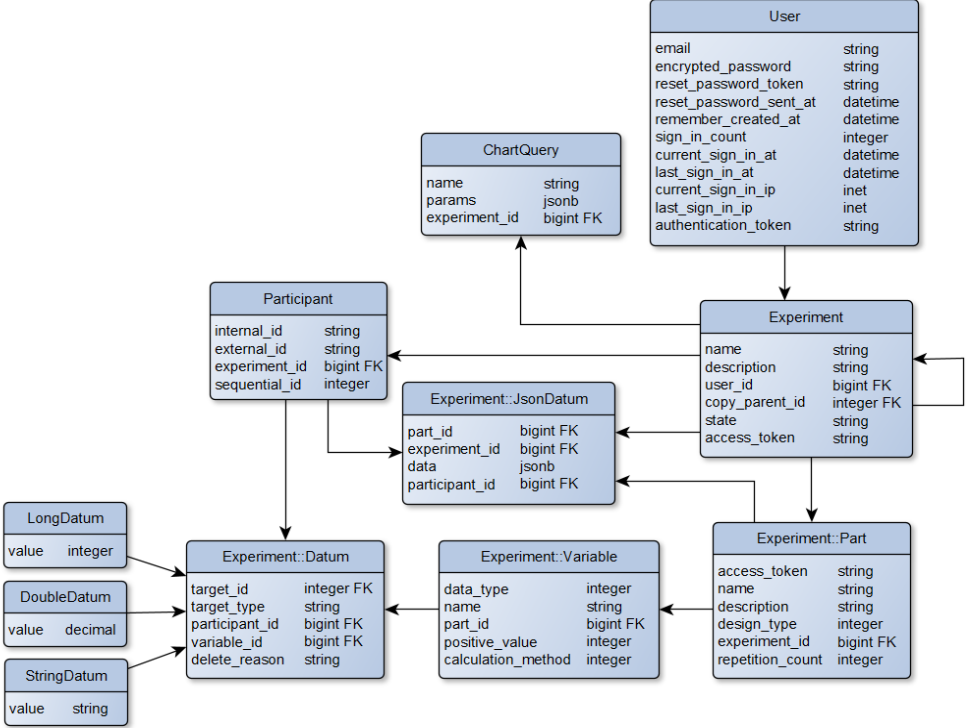


Image 8 – Entity relationship diagram of Sfinx

For the sake of brevity, some columns have been omitted from the above diagram. It holds that every entity described in the diagram includes column “id” of type “bigint”, which is sequentially generated for each new database record. These are meant to uniquely identify any kind of record in the database.

Most entities also contain columns “created_at” and “updated_at”, which track information about time of creation and update of the records, and both columns are of type “datetime”. The exceptions to this rule are listed below:

- ChartQuery
- Experiment::JsonDatum
- LongDatum
- DoubleDatum
- StringDatum

In the next section, all entities of the diagram are described with regard to their use and the contained fields.

User

Entity representing researchers and developers that use the Sfinx application. Its primary use is to authenticate users which are registered in the system. To that end, attributes *“email”* and *“encrypted_password”* are used as credentials to login into the Sfinx application GUI. The password is encrypted by the BCrypt tool in order to heighten the security of the system.

All of the remaining fields are automatically generated by the tool Devise included in the Ruby on Rails project, and their purpose is to log information about access attempts into the application. More information about specific fields can be found in the documentation of the Device tool [22].

Experiment

High-level information about the experiments created by researchers is recorded in these entities. Attributes *“name”* and *“description”* are meant for easier identification of the experiment by the researcher, while *“user_id”* identifies the user owning the experiment.

Functionalities of Sfinx include creating experiment as a copy of an existing experiment, without the gathered data. Experiments which have been created in this way reference their parent in the field *“copy_parent_id”*.

Experiments in the system are always in one of the states *edit*, *debug*, *open* and *closed*, which are stored in field *“state”*. State transitions of this field have been described in chapter “2.4.3. Workflow”.

Lastly, field *“access_token”* is automatically generated from the name of the experiment as a unique value, and serves as an identifier to the data gathering library when a new participant attempts to register to the experiment.

Participant

This is the participant which has been registered through the data gathering library to take part in an experiment defined by the researchers. Field *“experiment_id”* holds reference to the experiment which the participant has been registered to.

Attributes *“sequential_id”* and *“external_id”* serve the same purpose in the sense that they identify the registered participant to the data gathering library. The difference between the two is that the

former is automatically generated by Sfinx, while the latter is defined by the programmer of the testing application. This allows programmers and researchers to define their own format of identification number for participants, if they require to do so.

Lastly, *“sequential_id”* defines the order in which the participant has been registered to the experiment. This is used by the testing application to implement counter balancing described in chapter “2.3.4. Within subject design”.

Experiment::Part

Every experiment consists of several parts, which are represented by this entity. Field “*experiment_id*” denotes the connection between the experiment and its parts.

Much like in entity Experiment, fields “name” and “description” serve to describe part to the researcher, while attribute *“access_token”* is used by the data gathering library as a target identifier of the part, where the testing application should send its collected data. The token is automatically generated as a unique combination of the experiment access token and its own transformed name.

Attribute *“design_type”* is an enumeration that holds information about the design used to implement this part of the experiment. The allowed values are *within_subject* and *between_subject.*

Last attribute *“repetition_count”* defines how many times should every participant send data to this part, before Sfinx considers this part as completed for that participant. Once all parts are completed in this manner, the participant have sent all required data for the experiment.

Experiment::Variable

Each part of the experiment comprises of several variables, determining the type of data values that should be gathered from participant after each data submit. The variable entity holds reference to its part in field *“part_id”*.

Field “name” is used to identify the variable to the researcher and is also used by the Chart.js tool to identify variables that should be rendered in the Sfinx frontend.

Previously mentioned data type, meant to be stored in the variable, can be found in the enumerated field *“data_type”*. Sfinx supports collection of data types *long*, *double* and *string*.

Enumeration field *“calculation_method”* is utilized by Sfinx to store information about the intended method for processing of data from this variable. Allowed values are *log_transformation*, *normal_distribution* and *binomial_distribution*. It should be noted that this value is only for reference, and the actual method is stored in parameters of entity ChartQuery (see below).

The *“positive_value”* is only used when *“calculation_method”* field of this variable is set to *binomial_distribution*. Since binomial distribution calculation needs to know which values are considered positive from the set of collected data points (chapter “2.3.5. Confidence intervals”, section *Calculation*), the value that is regarded as positive in the calculation is defined in this field.

Experiment::Datum

This entity represents data collected during the course of the experiment. However, since the type of collected data can differ, this entity is further sub typed into entities **LongDatum**, **DoubleDatum** and **StringDatum**, which are the real holders of the collected data. This subtyping is implemented by Ruby on Rails in the form of so-called polymorphic association, which means a single entity can reference multiple other entity types by a single identifier attribute. In this case, *“target_id”* may

identify any of the 3 subtypes entities, while *"target_type"* denotes the table that should be referenced by the identifier.

Fields *"participant_id"* and *"variable_id"* denote the associated participant and variable, respectively.

Sfinx implements soft deletion of data every time the experiment is transitioned to state *edit*, to exclude data that has been sent to the experiment during its debugging phase. Once this transition happens, all data entites associated with the experiment have their field *"delete_reason"* set to value "test", which prevents that data from being used in any processing and visualization.

Experiment::JsonDatum

Boxplot graphs need to know which variables have been sent at the same time as a different variable in the same part. Since the Experiment::Datum needs to archive its data into separate entities without any relation for easier querying, this entity has been added to offer an alternative way to view the stored data.

Experiment::JsonDatum gathers information about related experiment, part and authoring participant in fields *"experiment_id"*, *"part_id"* and *"participant_id"* at the same time the data is saved in instances of Experiment::Datum. The main difference is that all values saved in these instances are congregated into a single JSON string and saved into the field *"data"* of this entity. Because PostgreSQL supports querying of jsonb columns, this data can be then filtered and used for construction of boxplot graph as needed.

ChartQuery

Entity for storing user preferences about boxplot graphs shown in the detail page of experiment in Sfinx frontend. These entities are optionally created for entities Experiment, referenced by field "experiment_id".

Property *"name"* denotes name of the graph shown in the frontend of Sfinx, while *"params"* field contains a query to the database to get relevant data for the boxplot graph and parameters of the calculation to be used.

2.4.6 User interface

Interfaces of Sfinx are largely divided into 2 groups: Graphical user interface available to researchers, which allows them to manage their experiments, and pure REST API which receives data from the gathering libraries.

Ruby on Rails generates its GUI from REST interfaces defined in its controllers, so they can be used to adequately describe Sfinx functionality. These interfaces have been summarized in **Attachment B** of this document for the sake of brevity of this document section.

The following paragraphs will showcase some of the essential interfaces provided by the deployed application.

Sfinx GUI interface

First page visible after login can be seen in the picture below. This page lists experiments accessible by the authenticated user, and offers actions which can be executed on them. It should be noted that the available actions are limited by the current state of the given experiment

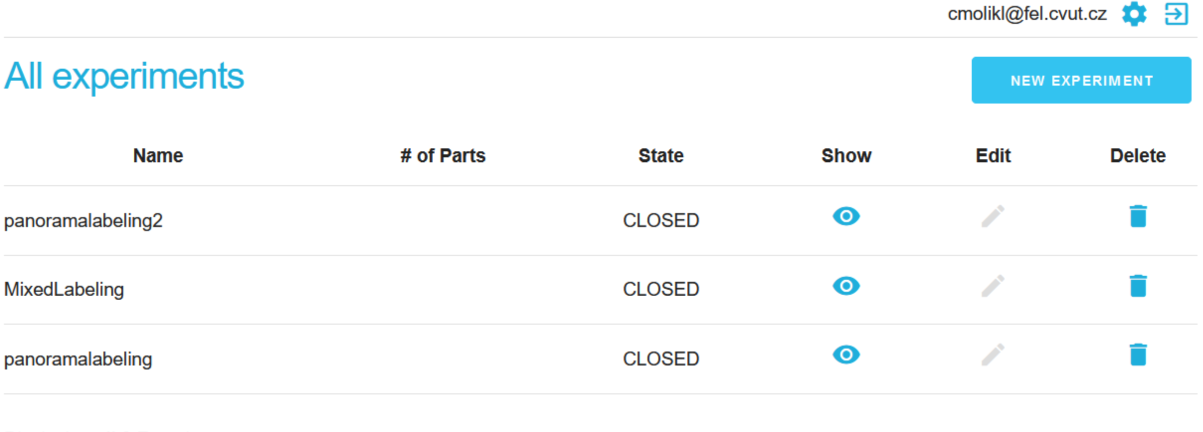


Image 9 – Homepage of Sfinx

Clicking the icon below “Show” label will redirect the user to details page of the experiment. This page lists detailed information about the experiment and its parts and variables.

It also offers option to change current state of experiment (upper left of the screen), and additional functions such as creating copy of the experiment structure in the system, and its export as a JSON file (upper right of the screen).

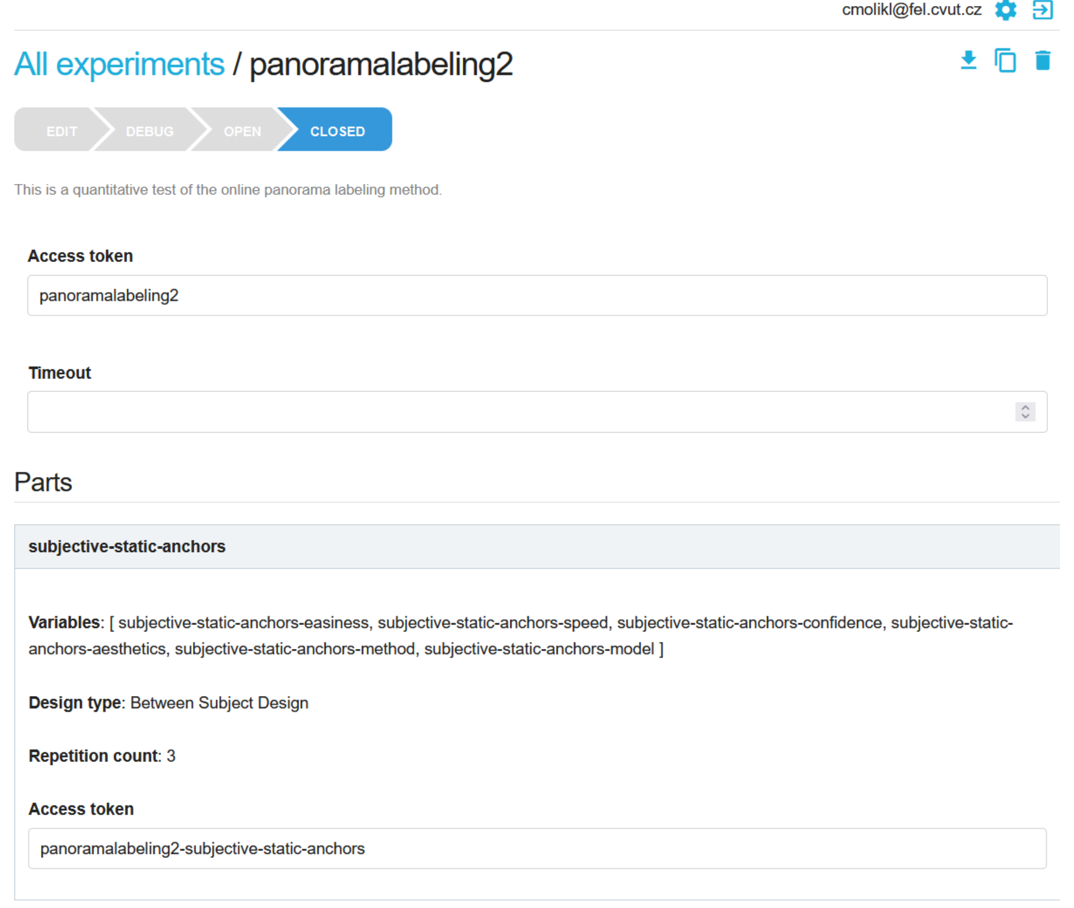


Image 10 – Details page of Sfinx experiment

This page also includes single histogram defined for the experiment, and list of boxplots specified by the ChartQuery records in database.

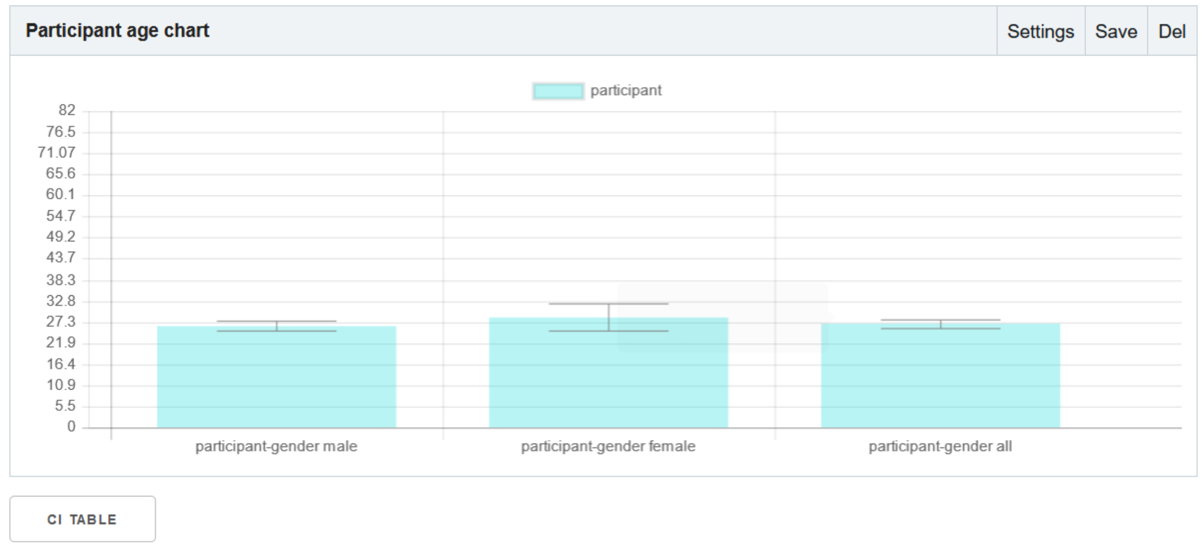


Image 11 – Boxplot graph of Sfinx experiment

Data gathering library

Source code of Sfinx [1] also contains implementation of an example testing application meant for testing by the test participants. JavaScript library "Transfer" integrated in the application is configured to send its data to an experiment named "testapp2", visible in the deployed Sfinx GUI [15], while the testing application implements the workflow of the testing process.

The data gathering library provides 2 functions necessary for gathering of data, both of which connect to REST interfaces of the Sfinx webserver where the data is stored and processed. First function is responsible for registering the participant to the experiment, while the other one handles saving data of registered participant to a specific experiment part.

Workflow in the example application also includes counterbalancing of test assignments to participants, which is driven by the "sequential_id" attribute returned by server upon registration of participant. This is needed since Sfinx server itself does not provide the counterbalancing functionality.

3 Solution design

This chapter describes the solution design of this thesis, which concerns the conversion of Sfinx application into Java environment.

First section goes through the problematic of Sfinx conversion process, and the tasks required of this thesis. The next segments then focus on individual parts of design for the new Java application.

3.1 Initial task analysis

First requirements of the thesis already have been completed by conducting analysis on the theoretical background and design of Sfinx system while summarizing the obtained information in the Analysis chapter.

Next step is to think about the remaining requirements of the thesis and come up with an efficient way to fulfill them.

In essence, the thesis asks for the creation of a completely new application from the very beginning. Its requirements imply that the converted application should consist of the Java equivalents matching the original applications structure and functionalities, while including a fitting modified implementation where direct conversion is implausible. It also specifies a list of functionalities which should be included in the new application which should be handled in a similar manner.

Upon consulting with the project leader some other optional improvements have been identified as well as the parts of general structure which can be omitted from the project. These will be covered by the following chapters in each part of application design.

Finally, computed statistical results obtained from converted application must be compared with those from Sfinx to verify their correctness. Prerequisite for this task is to extract data used for computations from Sfinx and import it to the converted application so that both tools base their results on the same input. This task is made more difficult since export files generated by Sfinx do not include all necessary data for computation, and the only other viable source of data is Sfinx database.

3.1.1 Conversion methodology

After considering all above stated conditions, it has been decided that the best course of action would be to start implementation of a generic Java backend application, outfitted with basic code structure and functions such as REST endpoints, access authorization and the connection to database.

Since the new application requires a web frontend which isn’t included by default in Java, a frontend project should be established as a complement to the previously mentioned backend, which must include login form for researchers and a basic home page. This interface can be then connected to backend REST endpoints to draw data from the system.

Once the base form of the application has been made, implementation should continue by adding core backend and frontend functions equivalent to those in Sfinx. Examples of these functions are CRUD operations of experiments, collection data and displaying of computed statistical results. This also means verifying the connection between the data gathering library and backend and making changes where necessary.

At this point, most of the thesis requirements would be met, and the implementation can be completed by gradually adding new features to the application as dictated by the thesis tasks description.

As the last step, an experiment should be created in the finalized application with the same structure as the data collecting experiment in Sfinx, so that the statistical computation can be verified. Database data of the Sfinx experiment can be extracted in the form of SQL insert scripts, which needs to be then manually loaded into database of the converted application. After that, both experiments contain the same input data so the computation verification can take place.

3.2 Application architecture

Structure of the converted application matches for the most part structure of the original since the same functionality needs to be provided. The only changes that the new design contains are introduced by the difference in used technologies and added features of the existing components.

New structure of the application is shown in the diagram below, and the following paragraphs then explain differences between the new design and original architecture (see chapter "2.4.2 Components").

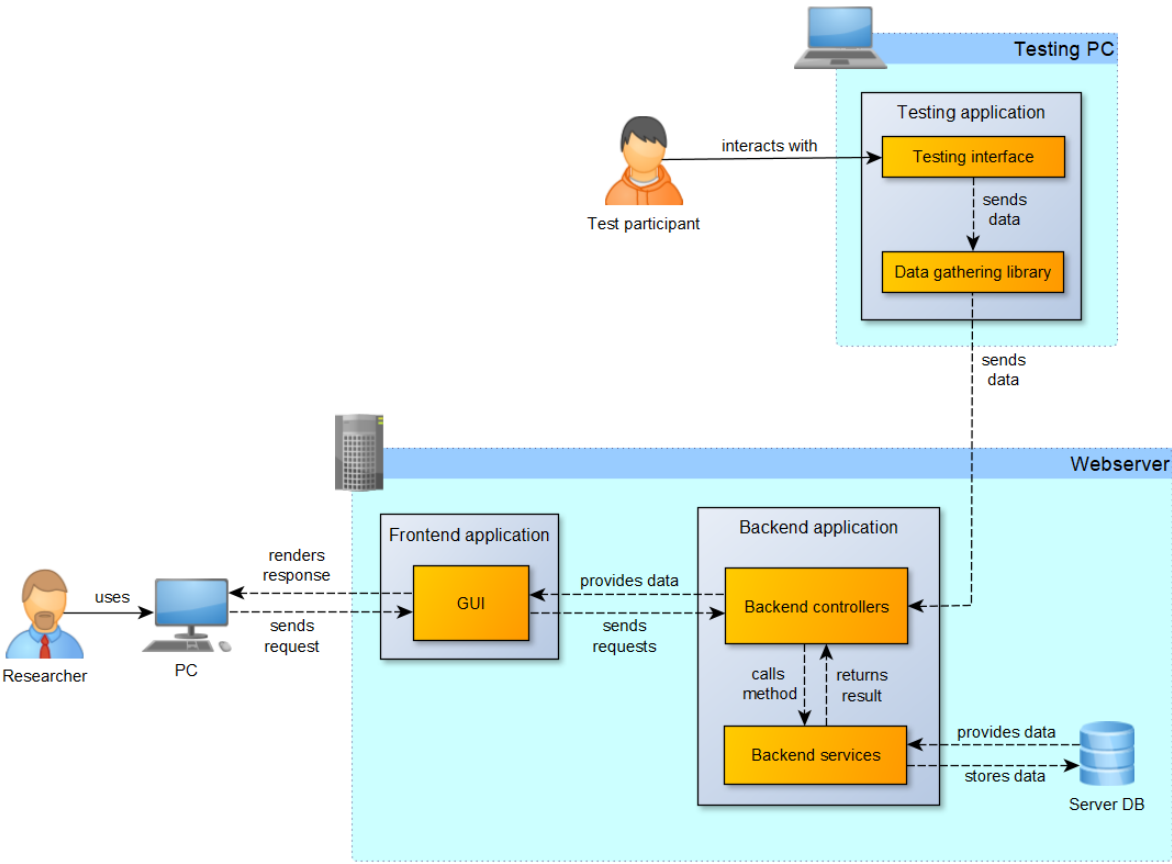


Image 12 – Component diagram of converted application

Webserver:
Implements the same function as the original application but divides its frontend and backend into two different applications to do so.

Backend and frontend of the server are implemented by respective Spring Boot and React projects, which are covered in following sections of the document.

Backend application:
This Java application implements the backend of webserver. Its functionality is separated into controller classes which are mapped to REST API interfaces without producing any GUI, and the service classes which are called by controller to provide the actual backend logic, such as request processing and computation of statistical data.

Services are further dependent on repository and model classes implementation, which are responsible for connecting to and drawing data from the database. In the diagram above, these components have been omitted for the sake of brevity.

Frontend application:
Application running separately from backend of the server, which provides GUI to the researcher users. Contains methods for rendering web pages in the browser, which can be nested into each other to prevent code repetition.

Another part of the application then defines service methods which query the backend REST interface to populate web pages with the necessary data.

Data gathering library:
Although implementation and function of the library stays largely unchanged from the original system, it previously used hard-coded latin square table array to implement counterbalancing for the example testing application.

The new design adds a new method into the library, which generates odd and even variant of latin square table depending on the inputted size argument.

3.3 Proposed technologies

Next paragraphs introduce technologies used in design of converted application and criteria which they fulfill.

Java:
General-purpose, object-oriented programming language [23] which can be used to create applications for many different platforms. Developer-friendly and very popular, which also means it is well documented.

This technology choice is required by the thesis.

Spring Boot:
This framework is designed to allow creation of standalone applications in Java, with support of web server implementations [24]. It can be used to replace the backend part of the previous Ruby on Rails server implementation.

Criteria fulfilled by this choice is the great extensibility and reliability offered by the framework as well as its well-maintained documentation which makes future maintenance easier.

Gradle:
Gradle is an open-source build automation tool focused on flexibility and performance [31]. It is necessary to download dependencies into the project.

The tool has good documentation and can handle any dependency which could be needed for the project, which makes it an appropriate choice.

Apache Tomcat:
Web server for deploying web applications written in Java. Allows the application to be made available for request from web browsers.

Tomcat [25] is known for being lightweight and reasonably secure, while covering most use cases in development, which is the reason for this technology choice.

React:
Web framework written in JavaScript which is meant for creation of template-based frontend [27]. Since projects written in React are standalone applications, it needs to communicate with a backend API to render its data-driven web pages.

Prerequisite for deploying React project is installation of Node.js.

This choice was made since Spring Boot does not provide its own web frontend which is required by the thesis, and the technology has good documentation.

Node.js:
Open-source, multiplatform environment [16] designed for creating and running web applications in language JavaScript. Used by previous Sfinx application to deploy Ruby on Rails framework. And its tools

This technology is required to deploy the React framework.

Npm:
Package manager for applications created for the Node.js platform.

This technology is required to deploy the React framework.

JavaScript:
Scripting and multiplatform programming language [28] formerly used to create data gathering library of the original application.

Since the data gathering library doesn't require extensive modifications and JavaScript fulfills its role well enough, this technology choice does not need to be changed.

PostgreSQL:
Database system [29] needed for storing the logic model and all persistent data of the converted application. Sfinx uses this system in the original implementation.

This technology can fulfill its purpose in the design of the new application and its change is not required, so its choice remains unchanged.

Hibernate:
Framework [30] written in Java which implements object-relational mapping (ORM) for the application. In other words, it allows mapping of Java application classes to database tables which simplifies management of database data.

Implementing ORM is a fairly standard way for Java applications to handle data in database and Hibernate provides functionality which covers most of the usually developed use cases. That makes Hibernate a good choice for the purpose.

D3.js:
JavaScript library for creation of visualizations from processed data [33]. It offers many different chart and graph types usable in the web frontend.

This library can be deployed either as a local library or through the Npm delivery system powered by Node.js.

This technology choice is required by the thesis.

REST:
Architectural standard, which allows to map browser URL addresses to functions of the server. Used by implementation of the original application.

Using this standard is essential to provide data from backend application to data gathering library and React frontend.

3.4 Backend application

Backend of the new system should use Java framework named Spring Boot, which handles default server configurations and later deploys REST API interface on a local port. The framework implements Inversion of control principle, meaning that the developer only needs to define project components with the appropriate annotations, and Spring Boot delivers instances of these components where they are declared.

For this purpose, it is advisable to separate components of the project into different folders, each of them addressing different concerns of the application. These concerns should include especially the following:

Configuration:
Settings of the port and base URL on which the application is deployed, as well as security settings for authentication and required dependencies for the project.

Most configuration needed by the project can be specified by extending one of the default configuration classes provided by Spring, which handle different aspects of the server functionality. Other parts should be present in the root of the project and “resources” folder.

Controllers:
Backend must provide REST interfaces which return the processed data. Spring Boot solves this concern through annotated controller methods which map to specific URLs and REST methods.

After triggering a controller method, an appropriate service function is called to produce the requested response.

Services:
Most logic of the application should be implemented in the service classes. After being called by controllers, they will validate input arguments passed from REST API and query repository methods to query or manage database data.

Once the necessary processing is done, service method then outputs a structured response back to the controller, so that it can be sent as an answer of the REST API.

Database model:
Backend application needs to know how to connect to the database and manipulate its data. To this end, repository classes provide methods which automatically load connection settings of the database and execute generated SQL queries, while using defined entity classes to identify the correct tables.

Entity classes act as definition of the database structure known to the application, which is mapped to real database tables for easier management of persisted data. Every entity represents one table in the database, while its attributes are equal to the table columns.

Utility:
Any code which supports the main implementation, separated for better readability of the code. These should involve the structured responses of the REST API, enumerations of certain data types and the calculator class which contains mathematical implementation for processing of collected data.

3.4.1 Logical data model

Diagram beneath shows the logical data model for backend of the new application. Several modifications have been made from the original application design, all of which are explained in the paragraphs below.

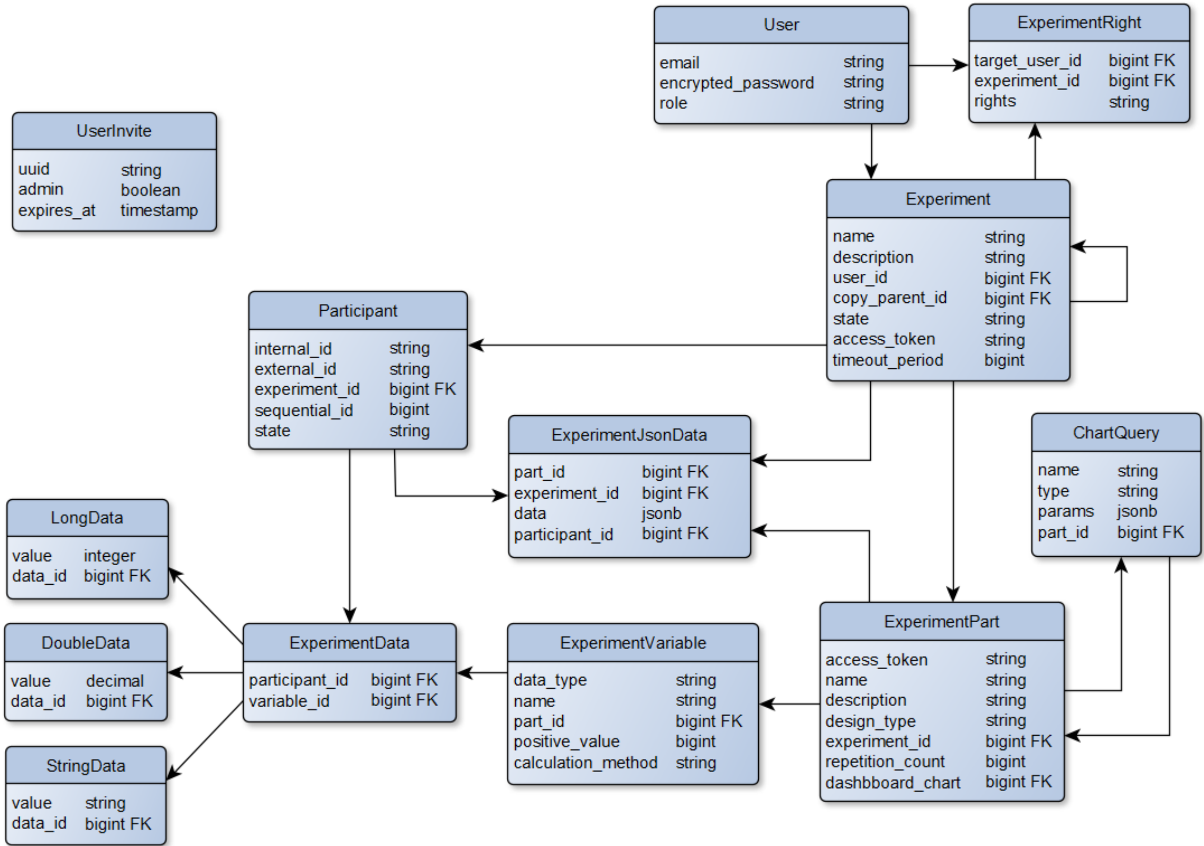


Image 13 – Logic model of converted application

For the sake of brevity, some columns have been omitted from the above diagram. It holds that every entity described in the diagram includes column “id” of type “bigint”, which is sequentially

generated for each new database record. These are meant to uniquely identify any kind of record in the database.

Another omitted field is *"deleted"* which is used for soft removal of records from the database. The only exceptions to this rule are tables LongData, DoubleData and StringData, which do not contain it.

Most entities also contain columns "created_at" and "updated_at", which track information about time of creation and update of the records, and both columns are of type "datetime". The exceptions to this rule are listed below:

- ExperimentData (contains only *"created_at"*)
- UserInvites (contains only *"created_at"*)
- LongData
- DoubleData
- StringData

In the next section, all entities of the diagram are described with regard to the differences from the original design and their meaning in the new application.

User

Entity representing researchers and developers that use the Sfinx application.

Fields *"email"* and *"encrypted_password"* all retain their original function and meaning, which is explained in the analysis of the original application (see "2.4.5 Entity relationship model"). This includes BCrypt encryption of the password.

Field *"role"* stores information about user's role in the system, which can have values "ADMIN" or "USER". Users with the "USER" role are able to use the regular functionality of the new application, while "ADMIN" possesses the same abilities as well as the right to invite new users into the system.

Compared to the original design, this table doesn't contain fields auto-generated by Ruby on Rails which tracked information about access attempts to the system.

Experiment

High-level information about the experiments created by researchers is recorded in these entities.

Fields *"name"*, *"description"*, *"copy_parent_id"*, *"state"* and *"access_token"* all retain their original function and meaning, which is explained in the analysis of the original application (see "2.4.5 Entity relationship model").

Field *"user_id"* identifies the user which created the experiment but does not provide access right to it. Access rights are instead covered by the ExperimentRight table.

Field *"timeout_period"* is a part of a new functionality, which removes participant from the experiment if they are inactive for a specified amount of time in minutes stored in this column. Sequential identifier of the participant can be then assigned to new participants which helps balance their distribution among the tested interfaces.

ExperimentRight

This table contains information about access rights of users to experiments, effectively implementing N:M relationship between them.

Field *"target_user_id"* refers to user which is granted a right to the experiment.

Field *"experiment_id"* refers to the experiment targeted by this right.

Field *"rights"* is a string enumeration specifying which right was granted to the user. Possible values are as follows:

- VIEW – User is able to view the experiment.
- EDIT – User is able to view and edit the experiment.
- SHARE – User is able to view, edit and share the experiment to other users.
- OWNER – User is able to view, edit, share, delete and transfer ownership of experiment to other users. This right cannot be possessed by more than 1 user per each experiment.

Participant

This is the participant which has been registered through the data gathering library to take part in a experiment defined by the researchers

Fields *"internal_id"*, *"external_id"*, *"experiment_id"* and *"sequential_id"* all retain their original function and meaning, which is explained in the analysis of the original application (see "2.4.5 Entity relationship model").

Field *"state"* denotes how far the participant has progressed with their testing. The allowed values of this enumeration are as follows:

- IN_PROGRESS – Participant has registered into experiment and is currently undergoing testing.
- TIMEOUT – Participant has not sent any data for an amount of time specified by the experiment and has been removed from it. Their *"sequential_id"* value can be reused by new participants.
- DONE – Participant has submitted all data of every part in the experiment and is no longer testing. Criterium of sending all data of a part is specified as having reached a certain number of submits to the part, defined in its *"repetiton_count"* column.

ExperimentPart

Every experiment consists of several parts, which are represented by this entity.

Fields *"access_token"*, *"name"*, *"description", "design_type"*, *"experiment_id"* and *"repetition_count"* all retain their original function and meaning, which is explained in the analysis of the original application (see "2.4.5 Entity relationship model").

Field "*dashboard_chart*" references a ChartQuery record which will be used to render a miniature graph in the detail page of the parent experiment. If null, no graph will be rendered.

ExperimentVariable

Each part of the experiment comprises of several variables, determining the type of data values that should be gathered from participants after each data submit. Such variables are represented by this entity.

All fields retain their original function and meaning when compared to original design, which is explained in the analysis of the original application (see "2.4.5 Entity relationship model").

ExperimentData

This entity represents data collected during the course of the experiment.

Fields *“participant_id”* and *“variable_id” all* retain their original function and meaning, which is explained in the analysis of the original application (see “2.4.5 Entity relationship model”).

When compared to the original design, this entity has removed some fields from the model. The most apparent change is removal of fields *“target_id”* and *“target_type”* previously used to implement polymorphism between this representative entity, and values stored in tables **LongDatum**, **DoubleDatum** and **StringDatum.**

This polymorphism was formerly managed by Ruby on Rails framework which could understand this structure, but this is not the case for Spring Boot used in the new design. This is why the target columns were removed, and ExperimentData is instead referenced directly by the renamed tables **LongData**, **DoubleData** and **StringData** in their column “data_id”. Aside from this change, these tables function same as before.

Finally, field *“delete_reason”* has been replaced by *“deleted”* value with boolean type, since the attribute will only change during experiment transition to EDIT state, which is the only case when the collected data can be invalidated.

ExperimentJsonData

Boxplot graphs need to know which variables have been sent at the same time as a different variable in the same part. Since the ExperimentData entity archives its data in separate records without any relation between them, this entity offers an alternative way to view the stored data. It is also used to count the number of commits from Partipant to an ExperimentPart.

All fields retain their original function and meaning when compared to original design, which is explained in the analysis of the original application (see “2.4.5 Entity relationship model”).

ChartQuery

Entity for storing specifications of boxplot and histogram graphs, defining which collected data should be used in the graph computation and parameters used during construction of the graph in frontend.

Field *“name”* is used for identification of the graph to the researcher, same way as in the original design (see “2.4.5 Entity relationship model”).

Original design of this entity contained only specifications of the boxplot graphs, which were connected to Experiment and shown in its main detail page.

The converted design allows assigning both boxplots and histograms to ExperimentPart entities instead, where each part possesses its own detail page where the graph can be viewed.

Field “*type*” denotes enumerated type of graph this record has specification for, and its allowed values are BOXPLOT and HISTOGRAM.

Field “*param*” contains specification of the graph in JSON format, which is interpreted according to the *“type”* value.

Field “*part_id*” identifies the ExperimentPart this ChartQuery is assigned to.

UserInvite

Entity for storing invite tokens issued by an administrator user.

Field *"uuid"* is used during construction of an invite link, which can be used by a new user to register into the system.

Field *"admin"* specifies whether user registered by this token should have administrator role in the system.

Field *"expires_at"* contains timestamp after which the invite token becomes no longer usable. Default token validity is set to 2 hours. Deletion of all expired tokens is also triggered whenever a new invite is created.

3.5 Frontend application

Frontend application of the new design uses React framework written in JavaScript language. React projects are deployed through the Npm package management tool, which provides default scripts for building or deploying the project, and manages dependencies required by the project.

Basic building blocks of the framework are the React components, contained in each JavaScript file of the project. Each component is tasked with rendering individual elements or entire pages of the frontend. Additional Npm dependencies can be included through the import statements to expand its functionality with premade solutions, as long as the package manager has installed them for the project.

React components drive the contents of the rendered web pages through their inner state, which contains a set of variables defined by the developer. Any change in the inner state is immediately reflected in the rendered content, which is useful since there is no requirement to reload the page to update it.

Entry point of React project is the index file, which loads the App component contained in the project root. App component holds general information applicable to the whole project, such as state of user login, and routes URL requests from browser to the components that need to be rendered while also providing them with input arguments.

Part of the new design are the service methods written in pure JavaScript, which are available for import to all components in the project. These methods are provided as means for the web frontend to communicate with the server backend. For that purpose, each of these methods prepare a native fetch request targeting a single backend REST API method, which can be asynchronously executed and its result delivered after the backend response.

3.6 Data gathering library

After analyzing the data gathering library and the example testing application it has been integrated into, it was initially decided that no changes would be necessary. Current form of the library could fulfill its role in the new application in the same way as before, although its connection settings might be later changed to match a different experiment and server host during the debugging process.

Consultation with the thesis supervisor has however revealed that the new system would require better means of counterbalancing participant distribution among tests. Implementing of counterbalancing on the server side has been judged too complicated because of the many varying testing parameters that could be incorporated into the experiment design.

This is why it was decided to leave the implementation of counterbalancing to developers of the testing application, but to provide them with better means to do so. The only addition to the data

gathering library would become a new method written in JavaScript, which generates balanced latin square distribution array [41] from size argument supplied as input. The size also determines whether odd or even variant of the table will be produced.

These tables would be then used by researchers and developers of testing application to decide assignment order of tests to participants.

3.7 Requirements

This section provides full list of functional and non-functional requirements that need to be met by the newly implemented web application and its associated data gathering library.

3.7.1 Functional requirements

FR1: The system allows registered user to login with their username and password.

FR2: The system allows logged-in administrator user to create invitation URL link, which can be used to register a new administrator or regular user into the system.

FR3: The system allows logged-in user to logout from the system.

FR4: The system allows logged-in user to define a new experiment, including its parts and variables, and save it into the system under their account.

FR5: The system allows logged-in user to view a list of all experiments accessible to their account.

FR6: The system allows logged-in user to transition state of an experiment accessible to their account, provided that the transition is valid for the current state and the user has experiment access right to do so.

FR7: The system allows logged-in user to alter the structure of an experiment accessible to their account, provided that the current state of the experiment allows it, and the user has experiment access right to do so.

FR8: The system allows logged-in user to delete an experiment owned by their account.

FR9: The system allows logged-in user to share experiment access right to another user, provided that the sharing user has the experiment access right to do so.

FR10: The system allows logged-in user to alter or revoke access right that they shared to another user.

FR11: The system allows logged-in user to view a list of experiments that have been shared to them.

FR12: The system allows logged-in user to view all access rights of an experiment that the user has granted to other users.

FR13: The system allows logged-in user to view detailed information about an experiment and its parts, provided the experiment is accessible to their account.

FR14: The system allows logged-in user to create histogram or boxplot specification for an experiment part accessible to their account.

FR15: The system allows logged-in user to view collected data of an experiment part accessible to their account in the form of histogram and raw data values, provided they have created specification for it.

FR16: The system allows logged-in user to view collected data of an experiment part accessible to their account in the form of boxplot and computed CI interval values, provided they have created specification for it.

FR17: The system allows logged-in user to delete histogram or boxplot specification for an experiment part accessible to their account.

FR18: The system allows logged-in user to select histogram or boxplot of an experiment part accessible to their account, which replaces the selection made previously and causes a miniature version of the selected graph to be shown in the details page of the parent experiment.

FR19: The system allows logged-in user to deselect histogram or boxplot of an experiment part accessible to their account, which removes miniature of the graph from the details page of the parent experiment.

FR20: The system allows data gathering library to register new participant to the experiment, provided that the library submits correct identifying credentials of the experiment, and experiment state allows new registrations.

FR21: The system allows data gathering library to store data gathered from active participant to an experiment part, provided that the library submits correct identifying credentials of the experiment part and experiment state allows new data to be submitted.

FR22: The system allows logged-in user to export the structure and data of an experiment accessible to their account in the form of a file.

FR23: The system allows logged-in user to import structure and data of an experiment from an export file.

FR24: The system allows logged-in user to create a new experiment as a copy of an existing experiment accessible to their account, which omits all collected data of the original.

FR25: The data gathering library provides a method for creation of odd or even latin square array, which can be used for implementation of counterbalancing in the testing application.

3.7.2 Non-functional requirements

NFR1: Backend application should be written in Java language.

NFR2: Frontend application should be written in React language.

NFR3: Data gathering library should be written in JavaScript language.

NFR4: The system should use a PostgreSQL database to store its persistent data.

NFR5: Backend should provide its functionality to frontend application through REST interfaces.

NFR6: Format of experiment files exported from system should be JSON.

NFR7: All graphs shown by frontend application should be rendered by D3.js tool.

4 Implementation

In the following chapter, implementation process of the solution design is explained, and its outputs are summarized.

First section starts by describing the initial preparations done to facilitate the conversion process. Then the conversion process is explained for the main components of the system, and result of the conversion is presented to the reader.

Lastly, the statistical computations done by converted application are verified, recorded step by step and a conclusion is drawn on its correctness.

4.1 Initial preparation

Initial efforts of the conversion process focused on analysis of the Sfinx application, which is target of the conversion. To this end, access to the deployed web application interface [15] and source code of the application [1] has been provided by the project leader.

While the deployed interface provided insight into the intended functionality of the web application, the greater challenge laid in proper analysis of the source code. In order to more efficiently browse through its structure and to improve its readability, IntelliJ IDEA development environment [35] was used to view a local copy of the source code, which was downloaded to local workstation.

Since Sfinx is written in Ruby on Rails framework which depends heavily on the “convention before configuration” principle, it was necessary to become thoroughly familiar with the concepts of the framework and its workflow before any analysis could be done. Once this was achieved, the source code yielded information about specific logic and structure of the Sfinx application which was then recorded in chapter “2.4 Original application analysis” of this document. Appropriate conversion design was then drafted in chapter “3 Solution design” to serve as a guideline during implementation of the converted application.

Once the general outline of the new application design was understood, it was time to prepare the development environment to facilitate the conversion process. It was decided that a local workstation with IntelliJ IDEA environment installed would be used to develop the new application, since IDEA explicitly supports Spring Boot framework required by the new design.

PostgreSQL system, Node.js environment and npm package management tool have been installed as well for the sake of data persistence and frontend support during development.

Project leader has also provided a Gitlab repository [36] which is meant to contain source code of the finished application once the development concludes and can be connected to IDEA interface to submit any changes in the project.

The next chapters describe the conversion process of backend, frontend and data gathering library portions of the web application. It should be noted that although these parts are explained separately, their implementation took place in parallel with each other as required.

4.2 Backend conversion

Backend part of the web application was created first since other components of the system depend on its functionality. Steps and methodology taken during its implementation are described by the following chapters.

4.2.1 Project creation

While it would be possible to create a new Spring Boot project completely from scratch, it is more optimal to start creation of new web application from a publicly available skeleton template. By using online tool Spring Initializr [37], a new blank web application project was generated for IDEA environment which contained preconfigured dependencies. This saved a lot of time that could be later used on actual development.

The parameters used for creation of the project are as follows:

- Build tool: Gradle 7.6
- Java SDK: 17.0.5
- Packaging format: JAR
- Spring Boot version: 2.7.7
- Dependencies:
    - Spring Web
    - Spring Security
    - PostgreSQL driver

All of the above dependency versions have remained in the finalized build of the backend application.

Once the project has been loaded into IDEA environment and its ability to run verified, the next step was to create appropriate folder structure.

4.2.2 Folder structure

Structure of the project is partially dictated by the Spring Boot framework, which adheres to a basic standard. It was nonetheless necessary to create structure of the source code within that structure, all of which is described below.

Root of the project previously generated by Spring Initializr contains general configurations applicable on the whole project. This includes the *„build.gradle“* file which contains all dependencies that need to be downloaded by the Gradle build tool and definition of this project as JAR artifact, which is created as a runnable .jar file whenever the project source code is built.

Other root files include default settings and wrapper of Gradle, whose meanings are covered in their official documentation [31].

The *„main“* folder placed in root then provides space for the developed source code and resources used by it, allocated respectively in folders *"java"* and *"resources".*

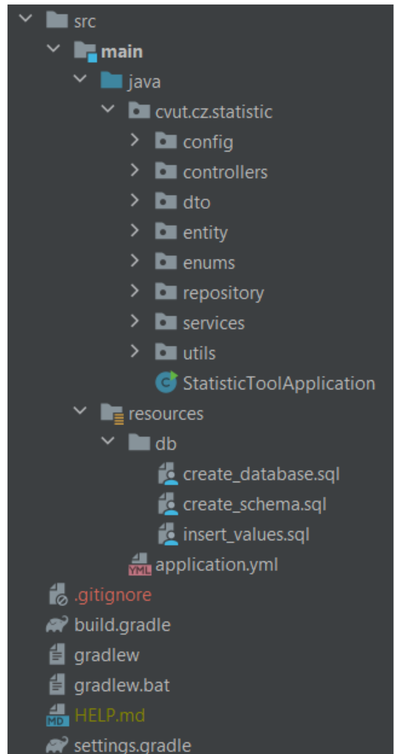


Image 14 – Folder structure of the finished backend project

Resources folder:

This folder is meant to contain SQL scripts which can be used to create required database tables for the backend application. These scripts won't be automatically deployed by the project, so developer needs to manually run them in the chosen database.

The other file in the *"resources"* folder is *"application.yml"* which should contain general configuration for the whole backend project. This includes port and context URL where the backend application is meant to be deployed, and connection settings for the database.

Java folder:

Contains the source code of the backend application, placed in a package with name *"cvut.cz.statistic".* File *"StatisticToolApplication.java"* encapsulates the main() method which is used to start the application, while the folders contain developed source code in structured form.

Contents of these folders are as follows:

/config
Settings and services used for authentication are decided by configuration files in this folder. This includes a token authorization filter whose implementation will be described in the next steps of the conversion process.

/controllers
Classes which generate REST interface provided by the backend. Querying one of the REST interfaces triggers call of a service method, which then prepares the appropriate response.

/dto:
Definitions of Data Transfer Objects (DTO), which encapsulate data sent from the backend interface.

/entity:
Entity classes which are mapped to database tables for easier management of persisted data. Every entity represents one table in the database, while its attributes are equal to the table columns.

/enums:
Some of the persisted values should only have a specific finite set of values, such as information about possible variable types or computation methods. Enumerations of these values can be placed in this folder.

/repository:
These classes accommodate methods for manipulation or querying of database records, while using defined entities information to identify the correct tables.

/services:
Service classes use repository methods to query or manipulate data from database and prepare suitable responses to be returned by controllers. They also supply graph values computed from collected testing data.

/utils:
Helper methods which deliver specific functionality to the rest of the source code. This includes calculation of confidence intervals, handling of access tokens for security chain and getting the currently logged-in user for verification in service methods.

4.2.3 Implementing logical model

After the structure of the project was created, the next step was to implement logical model of the database and connect it to the application, so that it could manage the persisted data.

Database
All of these efforts started by implementing the appropriate logic model inside of the PostgreSQL database, which was previously deployed on the local workstation for debugging purposes. To that end, set of SQL scripts has been written in the "/resources/db" folder, and then manually run in the PostgreSQL database to create the table structure described in chapter "3.4.1 Logical data model" of this document. The tool used for connection to database was chosen to be DBeaver [38].

Once the database has been made capable of holding application data, it was time to connect it to the backend implementation. Connection credentials for the database have been placed in the *"application.yml"* file in the *"resources"* folder, where they can be automatically loaded by the Spring Boot framework when required.

Entity model
Then the logical model contained in the database had to be understood by the application. Entity classes have been created with the appropriate annotations, identifying the table that a given entity represents, and attributes which match with columns of the table. Since this project is written in Java, getter, setter and constructor methods have been also generated through a helper tool of IDEA interface.

Repositories
Finally, the repository classes were defined for every entity to implement manipulation of the database data. Most of their functionality has been solved by extending JpaRepository interface [26], which automatically handles connection to the database with the previously mentioned settings in *"application.yml"*. The extended interfaces also generate callable SQL queries for the specified entity

from names of any declared methods in the repository and provide basic CRUD operations which are available by default.

4.2.4 REST implementation

After the means of data manipulation and persistence were established, focus of the development shifted towards the service methods. The following process was later used as a template for implementing REST interfaces for any other functionalities provided by the system.

DTO
In order to define input and output format which would be later used in REST interfaces, two types of data transfer objects would be defined for every manipulated entity: The creation DTO and information DTO.

Purpose of the creation DTO is to define a valid request input format for creation or update of the database record. They are also accepted as constructor and update method arguments of the representative entity, which simplifies implementation of these operations.

On the other hand, information DTO is used to define format of the response which is returned back to the controllers and subsequently outputted by the REST interface. This type of DTO is later interpreted by frontend and data gathering library as a JavaScript object populated with data from backend.

Service
After these helper objects were defined, a service class would be established to process request arguments passed by the controllers from the REST interface. To do that, the service would first need access to the repository methods.

Spring Boot allows adding of annotated components of its implementation, so-called *"beans"*, to another component in form of dependencies. Since the repositories were properly annotated before, they could be added to the service class at this point to allow manipulation of data in database.

Lastly, the service methods have been implemented to provide necessary operations for the manipulated entity through the repository calls. One point of interest is that since Experiment entity consists of multiple ExperimentPart and ExperimentVariable entities, its service methods are used to handle their creation and updates as well by using the nested DTOs of the original input.

REST
Once the service implementation was done, the final step was to create controller class for the entity and its REST methods defined through Spring Boot annotations. These methods then only needed to call the already prepared service functions, which have been provided by service class dependency added to the controller.

Each completed REST method was then tested by Postman tool [39] while monitoring database data changes through DBeaver tool to verify ability of the interface to manipulate persisted data. Alternatively, frontend interface could be also used if already implemented by that time. More about frontend development can be found in chapter “4.3 Frontend conversion” below.

4.2.5 Registration

Before the authentication mechanism of the application could be implemented, it was first necessary to prepare user information in the database which could be validated against login attempts.

To this end, one of the thesis requirements would be met by creating the invitation system, which allows generation of URL links that can be then used to register new users into the system. General outline of this process is similar to the one described in previous chapter: Implement service methods which provide these operations, and then expose them through a controller class as a REST interfaces.

Invitation interface
First the invite generating method has been created, which simply creates a random UUID number from the native Java library and saves it to the database. This UUID is stored with some additional information: An argument from input of the method, which declares whether the new user will be administrator or not, and expiration timestamp generated during creation of entity, which is used to delete all older invites every time the method is called.

Registration interface
Before the registration function could be added as well, there was a need for password encryption utility that could be used to obfuscate saved user credentials. This is important since by basic safety standards, a database should never store passwords plain text form. To implement this encryption, BCryptPasswordEncoder bean was added to the configuration file *"WebSecurityConfig.java"* located in folder *"/config"* of the source code and added as a Spring bean in the user service.

This utility was then added as a dependency in the user service and used to encrypt password of the user in the newly added registration function. This function also validates the invitation token supplied as input argument, and grants administrative role to the user if the token information found in database indicates to do so.

Finally, both REST methods were called by Postman for sake of testing their functionality, and to create an initial user of the system. This user would be useful during the debugging stage of the following authentication implementation.

4.2.6 Authentication

Functionality of the backend application needs to be secured by an authentication mechanism that allows only authorized users to use it. Spring Boot framework solves this concern through configuration class components that need to be defined within the project and marked with fitting Spring annotations. In the converted application, this configuration has been placed in file *"WebSecurityConfig.java"* located in folder *"/config"* of the source code.

After consulting with the thesis supervisor, it was decided that authentication would be implemented through the creation and validation of JWT tokens [40]. These tokens would be generated and returned after every successful authentication through the exposed login interface, and could provide access to the secured interfaces when included as *"Bearer token"* in the *"Authorization"* header of every request.

Implementation of this mechanism would be done in four stages described below.

AuthenticationManager
First part of the authentication process would be handled by the AuthenticationManager bean, defined in the configuration file of the project. It is an interface provided by Spring Boot whose implementation can be configured to validate combination of username and password against user information in database.

AuthenticationManager requires two input dependencies before it can be used:

- UserDetailsService
    - Spring boot interface meant for finding persisted user information, which should be considered valid during authentication
    - Automatically used every time the manager is asked to authenticate a login attempt
    - In this project, it was implemented to query user data from database
- PasswordEncoder
    - Spring boot interface used for encryption of passwords
    - The manager uses it to encrypt password from login attempt before its validation
    - Encrypted form of password from the login attempt should match the encrypted password in database
    - Already implemented and used in chapter "4.2.5 Registration interface"

JwtTokenUtil
In addition to validation of user credentials, the application also needed a utility class which would handle generation of JWT tokens, reading of information encrypted within and verification of their ownership and expiration time. This class was implemented through the use of *"jjwt"* library, which was added to the Gradle dependencies of the project. A secret key was also added to configuration file *"application.yml"*, which is used by the utility class to encrypt and decrypt information from the JWT token.

Once the application acquired abilities to both verify user credentials and generate access tokens, both of these tools were brought together in user service method, which would handle login attempts from a new REST interface and return generated JWT token when login succeeded.

SecurityFilterChain
As the last step, application would need to start enforcing its security rules on all incoming requests and recognize valid authorization tokens contained within. The first part of this setting was defined by the SecurityFilterChain bean, placed in the configuration file *"WebSecurityConfig.java"* located in folder *"/config"*.

Spring Boot implements its security through the so-called security filter chain. It is essentially a list of rules which is sequentially applied against all incoming requests, and which rejects requests that do not fulfill some of the given criteria. This validation can also end before end of the chain is reached, if there is a rule that explicitly grants access under a specific condition.

As long as there is a definition of this chain in the project, and the Spring security dependency is added by the package manager, these rules will be automatically enforced.

This project utilizes filter chain functionality to grant free access to interfaces which should remain without authentication, such as the login and registration API. Some additional settings include CORS configuration, which tells the frontend browser to allow access to the backend interface since it runs on a different domain.

The important part however is the custom rule which was implemented and added at the start of this list of rules. JwtRequestFilter rule has been defined to first read the *"Authorization"* header of the request and extract its access token. Then it uses JwtTokenUtil to decrypt and extract username and expiration timestamp contained within, former of which is then compared with information in the database.

If there is a match, and the token expiration time hasn't passed yet, the user details are saved to the application context as authenticated. This then trips the subsequent *authenticated()* rule in the chain which allows requests from authenticated users to pass.

With this, the authentication process has been successfully implemented. Postman and the frontend interface were used to test its functionality

Additional implementations
Once the authentication system became operational, it was possible to acquire information about currently logged-in users from the SecurityContextHolder which is available everywhere in the Spring Boot framework. This fact can be utilized by the service methods to better validate the incoming requests which work with user information and access rights. For this purpose, SecurityUtils class has been implemented and made available to most services as an added dependency.

As a secondary step, an interface to change stored user credentials has been also implemented to increase ergonomics of the new application.

4.2.7 Sharing experiments

Unlike the original implementation, the new design defines user's access rights for every experiment available to them. This is implemented through the ExperimentRight records in the database. As a reminder, the implemented permission rights granted by this entity can be seen in chapter "3.4.1 Logical data model".

The rights system defines which experiments are available to the user, and actions they are allowed to execute on them. This is important in the frontend of the application, which should only list accessible experiments, and offer only valid actions. This rule is further enforced by implementing experiment permission validation on all interfaces of backend, so there is no danger of bypassing.

Ownership of the experiment is denoted by the OWNER permission, which is automatically granted to the user every time they create an experiment. This right is unique for each experiment and it denotes full access to the experiment actions, including transfer of ownership and deletion.

Remaining permission levels are meant for sharing of experiments to other users. Interface implemented in the backend allows creation of new ExperimentRight records which grants access rights to users other than the owner. This action can be done either by the owner, or user with the access right SHARED.

Furthermore, REST interface for listing of experiments shared to user has been implemented to allow management of these experiments. In a similar fashion, listing of users with access to given experiment can be also queried by the backend interface.

4.2.8 Data collection

Data collection interface of the backend application copies for the most part implementation of the original Sfinx application. Participants of the experiment first need to register themselves through the data gathering library, and then send data submits to a specific experiment part while including all its variable values.

It should be noted in advance, that neither of these actions are available if the current state of experiment disallows submitting of new data (states EDIT and CLOSED).

Participant registration

Every participant registers to a single experiment which can be identified outside the web application by an access token generated from the experiments name. The response returned from backend then contains sequential id of the new participant, which is used by the testing application to decide which interface should be tested next.

Participant id is also part of the response, and it should be used to validate data submits into the system.

Data collection

Registered participants of the experiments are able to submit data to a specific experiment part, identified by an access token generated from a combination of experiment and part names. They also supply the participant id they are registered with to validate their submit.

Format of the input data is a list of JavaScript objects, each being interpreted on the backend side as a DTO containing the name of a variable in the experiment part, and its submitted value. These values are persisted in two different ways:

- ExperimentData records
  - Each value in the list is saved as a separate record
  - Each record is further sub-typed into LongData, DoubleData and StringData, which contain the actual values
  - Used primarily for histogram computation
- ExperimentJsonData
  - All values in the list are formatted into JSON string and saved as a single record
  - Used primarily for boxplot computation
  - Also used when counting data submits from participant so far

The main reasons for this duplication are the advantages of each approach. ExperimentData provides simple access to individual values but does not provide information about which values were sent with the same submit. Boxplot computation requires this information, which is why this approach alone cannot be used.

ExperimentJsonData on the other hand provides linked data values from each submit but is difficult to query when individual values need to be retrieved, which is needed by histogram computation. This is why both approaches were implemented in the new application.

On a side note, it was considered whether the ExperimentData could be omitted, and only the sub-typed tables LongData, DoubleData and StringData used to hold stored data. This approach was rejected though, since that would mean each table would have to duplicitly define all columns of ExperimentData, which was judged inefficient for future scoping of the application.

Timeout

This is a new feature previously not present in the Sfinx application. Each participant now has a state attribute, which denotes how far they have progressed with their testing. Their full list has been previously described in chapter “3.4.1 Logical data model”.

This state is used to remove participants which have not sent any data for too long and allows to “recycle” the sequential id that was assigned to them during registration. This helps to better balance the distribution of participants assigned to testing.

The mechanism is invoked during calls of either the registration or data submit methods. During registration, all participants of the target experiment are checked, and their timeout state is updated. Then the oldest timed out participant with unclaimed sequential id is selected, and their id given to the new participant.

If the method invoked was for data collection, the mechanism updates state when necessary and forbids access if the participant is timed out.

Lastly, if the participant has sent the required number of submits to all parts, the state of participants is set to DONE, and they are no longer considered during timeout validation.

4.2.9 Graph computation

The original Sfinx design allowed users to specify boxplot specifications as ChartQuery records, which were used to query data of specific variables from the database and contained parameters for graph rendering. When frontend requested graph data from backend, these specifications were loaded, and the queried variable data computed to produce confidence intervals, which were then rendered in experiments detail page.

The new application was implemented to use the same mechanism when computing its graphs, with two notable differences.

First, ChartQuery records are specified for parts of the experiment, instead of the experiment itself. The idea is to allow rendering list of graphs within the context of a specific part.

Second, the ChartQuery records can be specified for both histograms and boxplots, instead of boxplots only. The ChartQuery now includes *"type"* attribute which determines how the specification should be interpreted.

It should be noted that backend does not need to handle any computation for histogram graphs, since only the raw values and auxiliary parameters are necessary input for graphs in frontend.

Boxplot computation
The focus of this conversion was to reproduce functionality and output of the original application as closely as possible.

Computation starts by extracting specification of three values from the ChartQuery entity:

- targetVariable
    - The values which shall be input of the computation
- filterVariable
    - Variable which is contained in the same experiment part as targetVariable
- filterVariableValues
    - List of values in filterVariable, specified by user
    - Each value is used to filter a specific subset of targetVariable values
    - Each of these filtered subsets is used to compute one bar of the boxplot graph

These values are then used to query ExperimentJsonData records in the database. Specifically, the contents of the JSON column has to be parsed by the SQL query in addition to the normal table columns so that records with relevant data could be quickly filtered.

This functionality was unavailable in the JpaRepository interface, which is used in majority of the new application, so a custom repository function had to be also implemented. The set of filtered values was then assembled into a native HashMap object and supplied as input to the CiCalculator class.

CiCalculator is a utility class which was implemented to handle all computations of the confidence intervals. It is added as dependency to service methods to process filtered data from database, and its output is the final response returned by REST interface for each computed boxplot.

The only change in this implementation is the library which supplies values characteristic for T-distribution and Normal distribution. Ruby on Rails had its own libraries for this purpose, which were not usable in Java. After some research however, it was found out that their functionality could be supplemented by NormalDistribution [42] and TDistribution [43] libraries from the Apache Commons package.

4.3 Data gathering library conversion

As indicated in the design chapter of this document, the only change required is to add a new JavaScript method which generates balanced latin square distribution from inputted size argument.

This task was implemented in the file *"main.js"*, located in folder *"/public/labeling/js"* according to the required algorithm [41]. This file contains most of the utility functions used in the testing application, and unlike Transfer object, it is imported into almost all files which are used.

This implementation consists of two methods. The one named *limitNumber()* is a helper method which returns looped value of input number, if it exceeds an upper limit defined by second input argument. In other words, it starts counting back from 0 it is too high.

The other method is named *createLatinSquare()*, and it implements generation of the balanced latin square table array. After validating the input size, it first generates a latin square array with the even latin table algorithm. If the input size is even, the method returns the array as result, otherwise the mirror image of this array is appended to the existing array rows before being returned.

4.4 Frontend conversion

Frontend part of the application started being developed once the backend interface became capable of supplying REST interfaces which could be called.

This chapter describes the initial creation of the React frontend project, and the approach taken during its implementation.

4.4.1 React project creation

After researching React framework documentation [44], it was revealed that the package management tool npm [32] provides a script which can be used to generate an empty React project with default settings. This script was executed in the command line of the local workstation to create the frontend project in the *"/react"* folder of the converted application.

Since all dependencies of the new project are handled through the npm tool as well, they needed to be installed before the new application could make use of them. The following is the full list of dependencies which were installed to the frontend application during the course of its development:

- @emotion/react@11.10.5 // Dependency for the @mui components
- @emotion/styled@11.10.5 // Dependency for the @mui components
- @mui/material@5.10.16 // React Pagination component
- d3@7.6.1 // D3.js graph rendering library
- react-dom@18.2.0 // React DOM handling dependency
- react-router-dom@6.4.3 // React Router component
- react-scripts@5.0.1 // React launcher scripts
- react@18.2.0 // React core library
- web-vitals@3.1.0 // Performance monitoring tool

Once the project was successfully established, its ability to start has been tested by inputting the following command in console:

npm run start

Default React page of the application then became available in web browser under the address:

http://localhost:3000/

4.4.2 Source code structure

Structure of the generated project is much simpler when compared to the backend application. Most of the project source code consists of React components which represent the rendered web pages or their visual elements. Both define their own behavior logic and contain inner state which can be understood as a set of developer defined variables whose change is immediately projected into rendered content.

Therefore, it was only required that both types of these React components have allocated folders in the project structure with connection to the React framework and the implementation which connects the application to backend.

Root of the React application contains file *"package.json"*, which contains specifications of all dependencies which are used in the project (see chapter "4.4.1 React project creation"). Package manager tool npm downloads and manages these dependencies in folder *"node_modules"*, making them available to the project.

Implementation of the project is then contained in Folder *"src".* File *"index.js"* is the entry point of the running application, which loads the React framework as well as React component *"App.js"* responsible for handling every incoming URL request for the frontend domain.

The other notable file in the root folder is *"config,js"* which contains environmental variables usable in the project. Remainder of the files in the root are system files of the React framework or CSS styles.

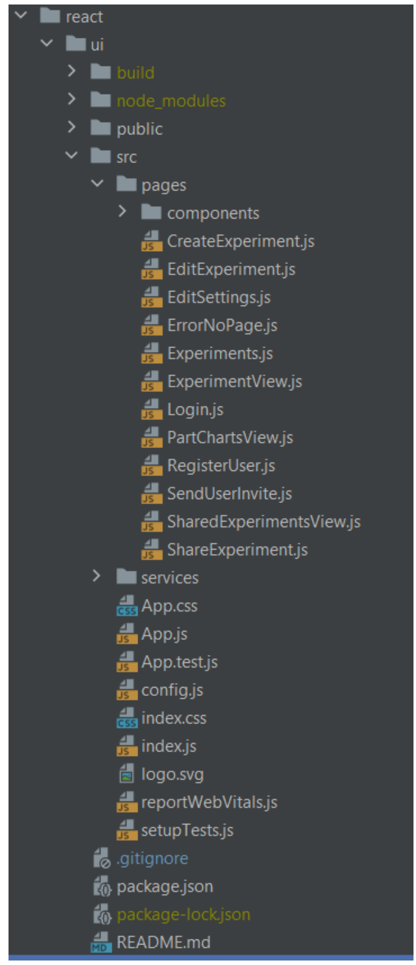


Image 15 – Folder structure of the finished frontend project

Lastly, folders "pages" and "service" are meant to contain React components for rendering web pages and JavaScript methods for connecting to the REST interface provided by backend, respectively. Sub-folder *"components"* then holds visual components of the webpages which represent the different parts of rendered content.

4.4.3 Routing implementation

As mentioned in the previous chapter, React component contained in file *"App.js"* handles all incoming URL requests to the frontend domain. Right after the project creation however, the application does not implement any routing and the only content shown is a default React page referenced by *"App.js"*.

The latter problem was solved by creating a new React component which would render a homepage for testing the routing functionality. Several temporary dummy pages were also created for this purpose.

Then the <BrowserRouter> component was added to *"App.js"*, which is a project dependency for reading the current URL of the browser and selecting a React component which will be rendered as a reaction to it.

Every React component selected by the Router is defined by the *element* attribute and accepts input properties which can be used to drive its content rendering and behavior.

```
return(
    <BrowserRouter>
        <Routes>
          <Route path="/" element={<Navbar isAdmin={this.state.isAdmin} loggedIn={true} handleLogout={this.
            <Route index element={<Experiments jwtToken={this.state.jwtToken}/>} />
            <Route path="/settings" element={<EditSettings jwtToken={this.state.jwtToken} mail={this.state.
            <Route path="/new" element={<CreateExperiment jwtToken={this.state.jwtToken}/>} />
            <Route path="/shared" element={<SharedExperimentsView jwtToken={this.state.jwtToken}/>} />
            <Route path="/edit/:id" element={<EditExperiment jwtToken={this.state.jwtToken}/>} />
            <Route path="/share/:id" element={<ShareExperiment jwtToken={this.state.jwtToken}/>} />
            <Route path="/detail/:id" element={<ExperimentView jwtToken={this.state.jwtToken}/>} />
            <Route path="/detail/part/:id" element={<PartChartsView jwtToken={this.state.jwtToken}/>} />
            <Route path="/admin/invite" element={<SendUserInvite jwtToken={this.state.jwtToken}/>} />}/>
            <Route path="*" element={<ErrorNoPage />} />
          </Route>
        </Routes>
    </BrowserRouter>
);
```

Image 16 – Routing table definition of the frontend

Lastly, the functionality would be tested by <Link> components rendered in the navigation bar of the testing homepage, which could be clicked to navigate through different rendered dummy pages.

4.4.4 Login

Next the login mechanism needed to be solved before content could be developed for the frontend application. This would mean transmitting user credentials to the backend login interface and using the returned access token to gain access to the backend REST functionality.

First, the *jwtToken* variable would be added to the inner state of component *"App.js"*, which contains access token obtained from the backend. If there is no token set, it is understood that there is no logged user and the regular routing will not be invoked.

Instead, there is an alternative routing table meant for unauthenticated access. These pages include especially the login and registration pages.

```
render() {
    if (this.state.jwtToken === null) {
        return (
            <BrowserRouter>
                <Routes>
                    <Route index element={<Login handleLogin={this.handleUpdate}/>} />
                    <Route path="/user/invite/:uuid" element={<RegisterUser />} />}/>
                    <Route path="*" element={<Login handleLogin={this.handleUpdate}/>} />
                </Routes>
            </BrowserRouter>
        );
    }
```

Image 17 – Routing table for unauthenticated access to frontend

The login page has been reworked from the testing homepage created in the previous chapter and renamed to *"Login.js"*. This page is responsible for querying the backend login form and saving the obtained token to *jwtToken* attribute of the *"App.js"* component. Session storage of all user information has been also implemented in case of web page refresh.

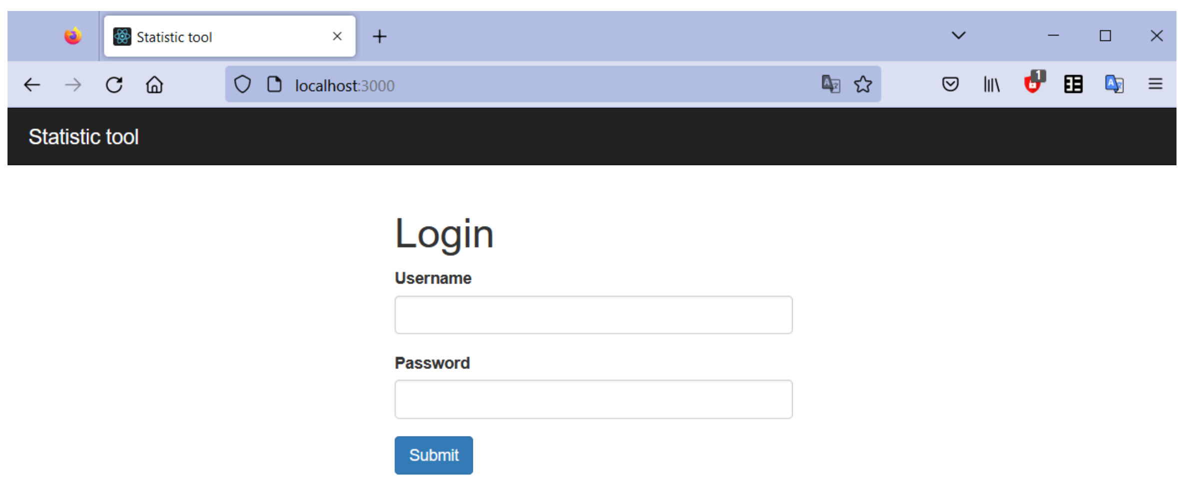


Image 18 – Login form of frontend

Connection of the login form and all subsequently developed web pages to backend is facilitated by a set of JavaScript fetch methods, which have been defined in the *"services"* folder of the project and imported in every page as needed. These service methods need access to a valid *jwtToken* to function properly, which is why the obtained token is propagated to every rendered component through its input property defined in the routing table (see Image 16).

Majority of the pages developed afterwards use this pattern to query the backend REST interface and populate their content with the returned data.

Lastly, a logout button has been also added to the navbar which is present in every page, to remove the access token from *"App.js"* component and session storage.

4.4.5 Graphs in D3.js

D3.js is a JavaScript library containing premade methods for defining and rendering of graphs. This library can be understood as a drawing tool which renders individual primitive shapes and visual components into the drawing area of a <svg> html tag, which acts as a container.

For the sake of rendering histograms and boxplots in the detail pages of experiment and its parts, two different classes were defined in the *"components"* folder of the project, to act as factories to produce the visual components: *"CustomHistogram.js"* and *"CustomBoxplot.js"*.

The general workflow of both factory methods is the same, with differences in handling of the inputted data. Each factory is implemented with default constructor values which determine how the graph needs to be rendered. These values can be then overridden according to the developer needs inside the React component responsible for web page rendering, which should also supply input data from backend that needs to be shown in the graph.

An example of the graph usage by the React component can be seen below. This sample uses native React method *componentDidMount()* to render the graph into a selected html element with a specified identificator after page finishes loading.

```
componentDidMount() {
    var histogram = CustomHistogram(this.state.rawData, {
        width: 800,
        height: 250,
        barLegend: this.state.histogramData.variable,
        xAxisSteps: this.state.histogramData.step,
        xAxisBars: this.state.histogramData.bars,
        xAxisMin: this.state.histogramData.min,
        xAxisMax: this.state.histogramData.max,
        isStringData: this.state.histogramData.variableType === "STRING"
    });

    d3.select("#histogram" + this.state.histogramIndex).html(histogram.outerHTML);
}
```

Image 19 – Usage of CustomHistogram

Usage of the constructor values in the implemented code has been then divided into two sections: The definition and rendering. The definition section is tasked with processing the input data into usable format and configuring instances of objects from the D3 library.

The rendering section then calls rendering methods of these D3 objects to draw individual visual elements of the graph into a prepared <svg> container according to the prepared input data. This container is then returned to the React component which can render its content in the web page.

Different characteristics of both factory methods have been summarized in the following paragraphs.

CustomHistogram
There are two types of data which can be supplied to this factory method as input: Numeric or string.

String input data is understood by the graph as categorical values. Occurrences of each unique value in the dataset is counted and plotted as individual bars in the histogram graph in this case.

Numerical data type on the other hand needs to be distributed into histogram bins. This distribution is handled mostly by the *histogram()* object from D3 library, which needs to know the interval of values that needs to be divided, and the input values.

If the graph specification from backend contains information about maximum or minimum of allowed values, input data is trimmed as necessary before being processed.

Likewise when bin spacing or count of bars is specified, thresholds are configured in the *histogram()* object and applied in the implementation, otherwise D3 library will populate these thresholds with appropriate default values.

Once the configured *histogram()* function is invoked, a set of bin objects is produced with the information necessary for rendering of the histogram bars.

CustomBoxplot
Input of this factory method is almost entirely preprocessed by the backend implementation, which returns a map containing computed values for each filter variable value specified for the boxplot. This map is interpreted in frontend as a JavaScript object with attributes containing the data that should be plotted.

This means that the data can be used directly with almost no processing on the frontend side. The only exception is when the computed means or lower bounds of a filter variable value reach negative values. After consultation with the thesis supervisor, it was decided that these values would be adjusted to zero before rendering. Then the boxplot bars and error margins can be rendered for each filter variable value present in the inputted data.

5 Results and verification

This chapter describes the results which were achieved through the implementation of the converted application. Its source code can be seen in the Gitlab repository provided by thesis supervisor [36].

First section showcases the user interface implemented in the application and explains its functionality. The next section then describes the testing methodology used to ascertain functionality of the application.

Verification of confidence interval computation and its comparison to the values originally outputted by Sfinx is then given special attention in the paragraphs following after.

5.1 Finished application

Conversion of the application tried to preserve the GUI used in the original, although some changes were made after consulting with the thesis supervisor. The following page can be seen after authentication described in chapter “4.4.4 Login”.

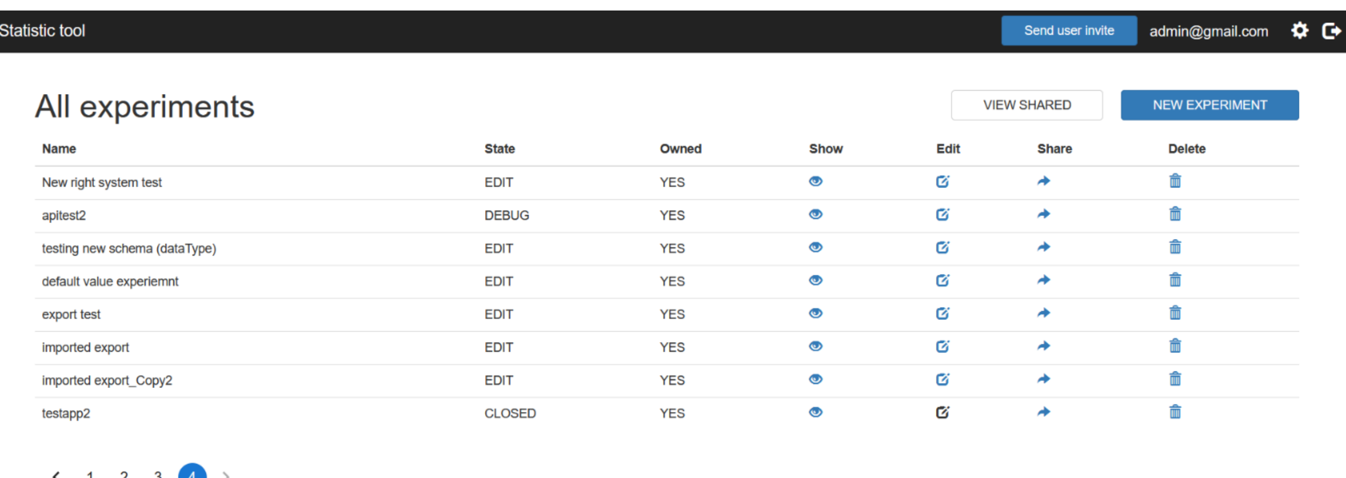


Image 20 – Homepage of converted application

Homepage contains paginated list of all experiments accessible to the user and serves as an entry point for most other functionalities of the interface. Each row the table offers buttons which allow the users to manage their experiments. These buttons include:

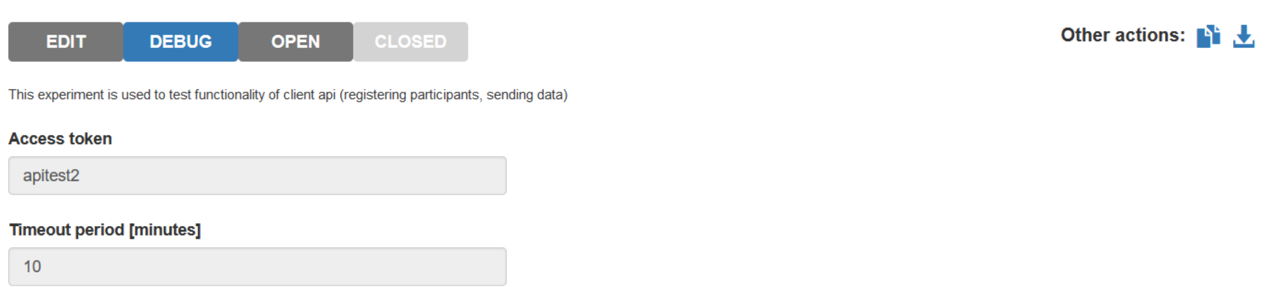
- **Show:** Redirects user to detail page of experiment
- **Edit:** Allows user to change structure of experiment
- **Share:** Allows user to share experiment to other users, and manage their access rights
- **Delete:** Removes the experiment from system

Aside from the actions available in the table rows, other buttons allow users to create a new experiment and view experiments shared to them in a listing similar to the homepage.

If the user has been granted administrative role, the navigation bar will also contain button which redirects them to page for generating registration invites.

Experiment details

The page shown in the following image shows details of the experiment, such as its name, description and the data structure arranged in dashboard fashion, with information about individual parts and variables.



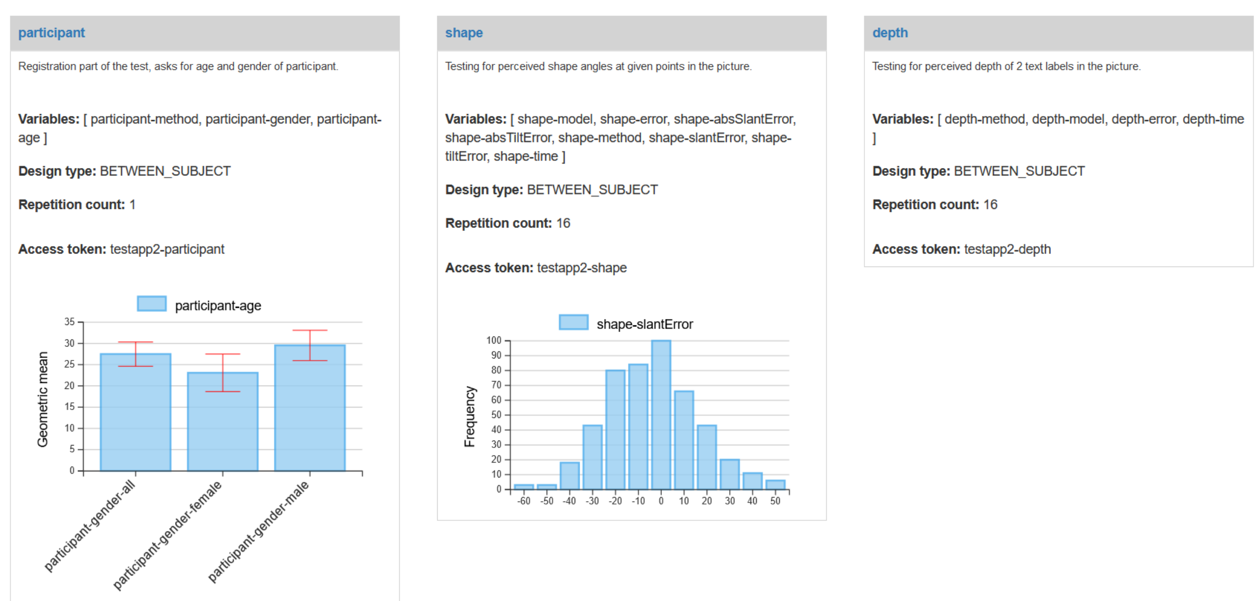


Image 21 – Experiment details page of converted application

This page allows users to transition between states of the experiment with the buttons in upper left, and the set of buttons on the upper right allows them to create a copy of the experiment structure or export the experiment along with all its collected data.

Title of each experiment part can be also clicked to view the part charts page, which renders full-sized graphs specified for each part. Its usage is explained below.

Part charts

This page contains two sections: The list of histograms and the list of boxplots. Each of these sections contains forms for adding new ChartQuery specification which will be rendered here, and each graph can be deleted or bookmarked with one of the buttons in the upper right corner of their card.

Only one graph of any kind can be bookmarked for each part and doing so will cause its miniature to be added to the part information shown in the previous experiment details page.

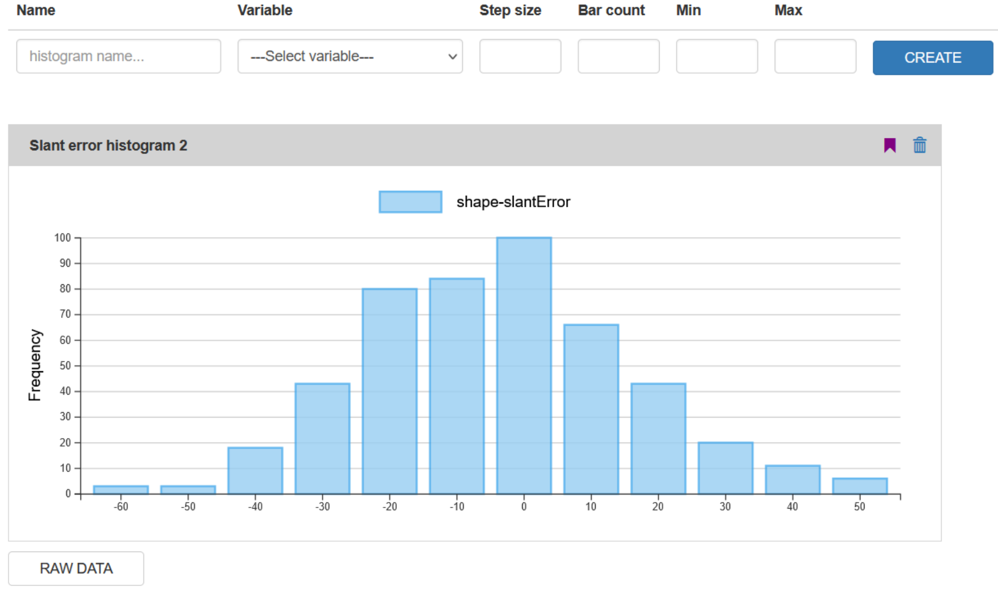


Image 22 – Histogram and its creation form in converted application

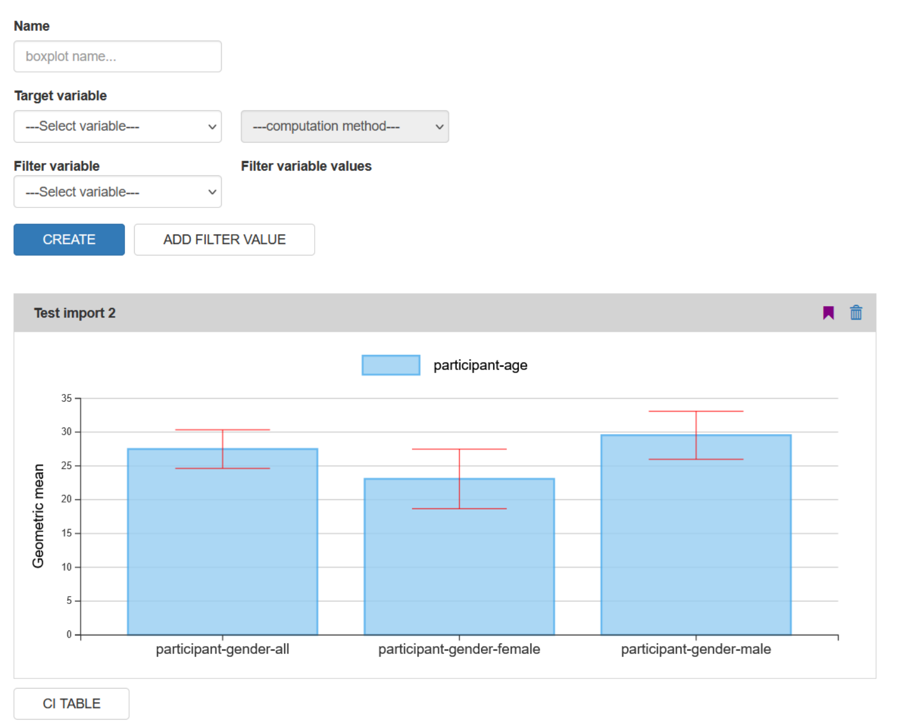


Image 23 – Boxplot and its creation form in converted application

Create experiment

This form can be accessed from the homepage, and it is used to create a new experiment structure, which can be then used to store data gathered from participants.

Structure of this page is almost entirely copied in the form which allows editing of the experiment, where the already existing parts and values are filled in with the information from backend. One notable difference however is the import field, which allows creation of an experiment from a JSON file previously generated in the experiment details page.

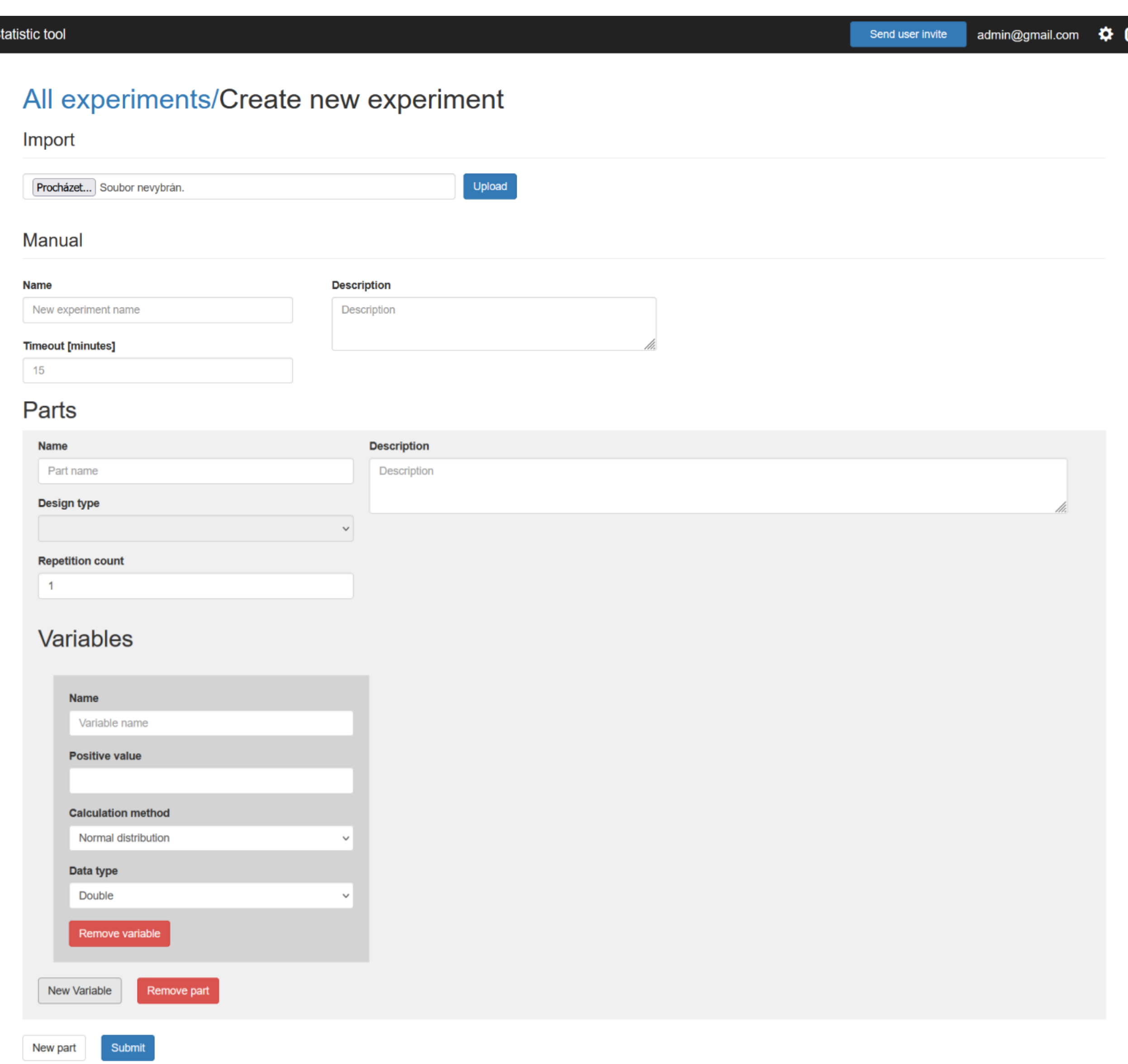


Image 24 – Experiment creation page of converted application

Share experiment

One of the action buttons in the experiments listed on the homepage will allow the user to manage sharing of a given experiment to other users. The page shown after clicking the button will allow user to add new access rights for the experiment or manage the existing ones.

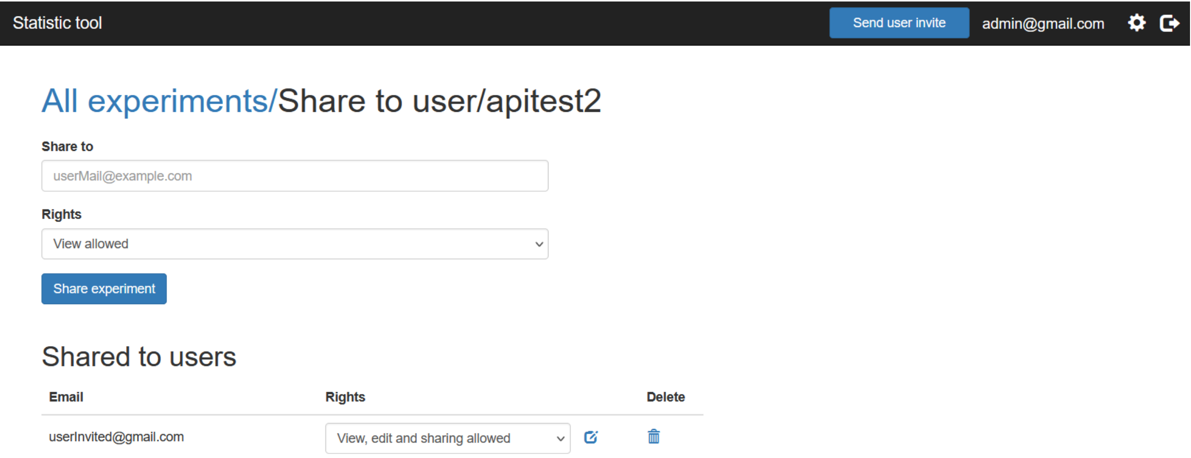


Image 25 – Experiment sharing page of converted application

User invites

This form is accessible from the navigation bar at the top of every page, but only if the logged-in user has administrative role assigned to them. If that is the case, the page shown below will allow them to generate an invitation link, which can be used in a web browser to access the registration page.

Registration page will ask for login credentials the user wishes to use for access to the application. After submitting, the backend logic will then add them to the system with an user role specified by the information associated with the invite identifier.

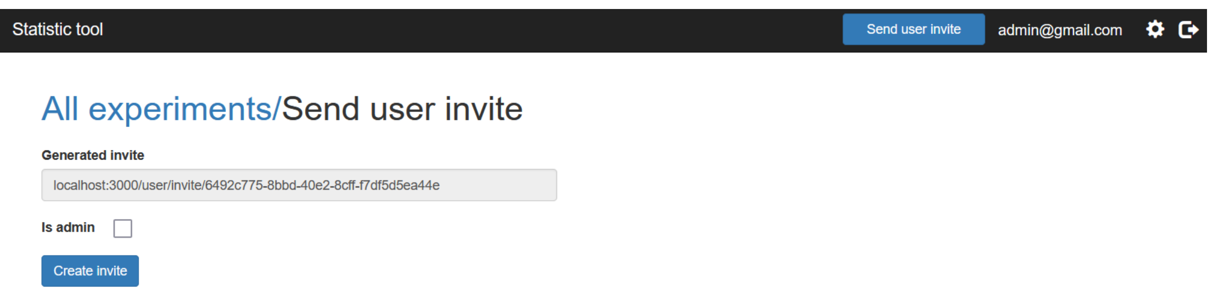


Image 26 – User invite page of converted application

Image 27 – Registration page of converted application

5.2 Testing functionality

The application has been continuously tested during the course of the development, as evidenced by the data visible in the screenshots above. This data has been created on the local workstation to test new features of the application for the sake of verifying its functionality.

The following paragraphs cover testing methodologies used for each part of the application.

Backend testing
Testing done for the backend application was focused on the REST interfaces, which are offered to the frontend application and data gathering library.

Every time a new feature was added, it would be called through the Postman tool or frontend GUI of the frontend, and behavior of the system would be observed under all possible initial conditions. Special attention would be further given to the user access rights and validation implemented for each interface.

If the user was allowed access, this observation would be used to affirm the expected database data changes and the output of the interface. To this end, DBeaver tool was used to monitor the database, and runtime state of the application was occasionally analyzed through the debugging and breakpoint utility of the IDEA environment.

Frontend testing
Focus of frontend application testing was centered around navigation mechanism of the web pages, its local behavior logic, and verifying the connection to backend.

Navigation itself did not need much testing, since it was handled by the premade React component <BrowserRouter>, described in chapter “4.4.3 Routing implementation”. User credentials would be submitted by the login form to verify its connection to backend, and the correct handling of the returned access token. Behavior of the system would be then observed through the browser and console logs to debug its functionality, while also making sure the access is granted only when a token is present.

Once the access mechanism was verified, the application was continuously tested for each new feature. First the methods in the *“service”* folder would be invoked to call backend methods and their output would be either observed through console logs on the frontend side, or their changes would be tracked in backend to confirm the connection.

Then the expected behavior of the web pages would be verified, depending on the response from backend. Rendering of the data would be apparent in the browser window, and inner state of the application would be debugged through the console logs.

Data gathering library testing
This part of the testing has been implemented by first creating an experiment design which matches the one used by testing application used in Sfinx. This new experiment has been designated as “apitest2” in the local database.

Second step involved browsing local copy of the code to find all references to the Transfer object, which contains implementation of the data gathering library, and changing the connection settings in all locations to match the locally deployed backend application. This has been made easier through the lookup functionality of the IDEA development environment.

Connection to the experiment has been then tested and debugged by opening the *"index.html"* file of the testing application located in folder *"/public/labeling"* of the project and interacting with the testing interface in browser.

Lastly, database data changes were observed through the DBeaver tool to verify that the data has been submitted while taking into account validation of the interface, and that the timeout mechanism works as intended.

5.3 Verifying CI computation

Final step of the testing was to verify the capability of the converted application to compute confidence intervals and compare its output against the original Sfinx functionality.

This chapter first describes the method of importing data from Sfinx, which will be input of the computation in both applications, and subsequent verification process which reports reached results and conclusions.

5.3.1 Import of Sfinx data

The application outputs would be verified for both the rendered histograms and boxplots, while using the same underlying database data in both applications. To that end, two types of data had to be imported from Sfinx into the new application: Individial values of Experiment::Datum entity and the same values submitted as a set in Experiment::JsonDatum records.

Preparing storing structure
After consultation with the thesis supervisor, it was decided that data from the experiment named *"testapp2"* in Sfinx application would be imported and tested for correctness in the new application.

Its structure would be recreated by the finished frontend interface, and a dummy participant was manually prepared which would be considered to have submitted all of the imported data.

Importing Experiment::JsonDatum
When initially considering options for exporting data from Sfinx, its already inbuilt export feature was considered as a possible candidate. Once the exported data was analyzed however, it was found out that the data is exported only as individual values, which could not be used to populate the ExperimentJsonData table for boxplot computations.

After consulting with the thesis supervisor, access to the Sfinx production database was granted for the express purpose of exporting gathered value sets for the verification test.

The next issue was how to import the data into the storage experiment structure in the new application. The foreign key identifiers in the Sfinx export are meant for data in the original database and would have to be adjusted to match the converted structure.

It was decided that the export process would be implemented by exporting the Experiment::JsonDatum data as INSERT scripts in SQL format, and this format would be adjusted in a text editor. The required changes were not very varied and could be easily identified by basic editor searching features, which made this the suitable choice.

The tool for accessing and exporting the data from database was chosen to be DBeaver. After connection was established with the credentials from thesis supervisor, a query was invoked to select all Experiment::JsonDatum records belonging to the required experiment.

Then all of records were selected in the tool interface and copied as SQL scripts.

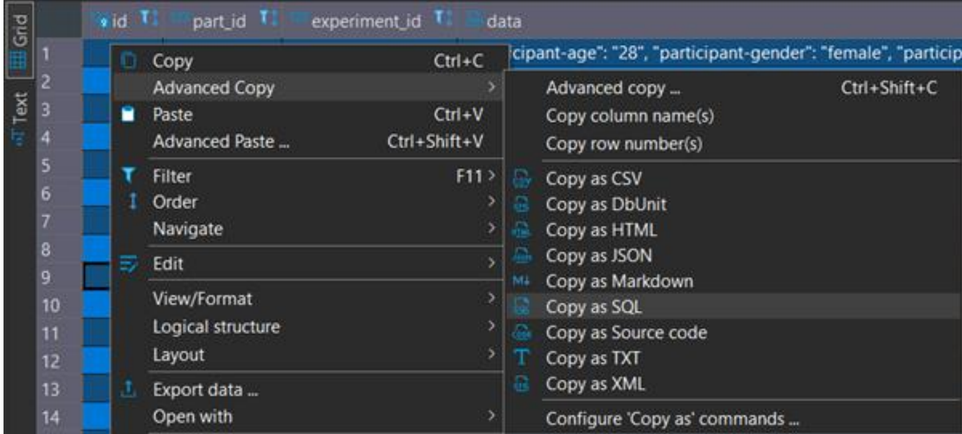


Image 28 – Exporting Sfinx JSON data

The exported INSERT scripts were then placed in text editor of the IDEA environment, and a combination of substring replacements and regex searches were applied on the whole data set to make the experiment identificators match the structure in the converted application.

Once the modifications were finished, the INSERT scripts were then executed in the application database to import the data.

Importing Experiment::Datum

Although the Sfinx export file could not be used to import JSON data, it was the perfect choice for importing the individual values. Even more so because of the import functionality already implemented in the converted application, which uses different export format than Sfinx, but could be reworked with minimal changes to offer the import functionality that is required.

To this end, a new interface would be created for the express purpose of importing data from Sfinx to a specified experiment and participant. Then the Sfinx export format was defined in the application so that the object mapper would be able to interpret the inputted JSON file.


```
{
  "experiment":{
    "name":"Testapp2",
    "description":"",
    "state":"closed",
    "access_token":"testapp2",
    "timeout":null,
    "number_of_parts":null,
    "parts":[
      {
        "access_token":"testapp2-participant",
        "name":"participant",
        "description":"",
        "design":"between_subject",
        "variables":[
          {
            "name":"participant-age",
            "type":"long",
            "data_to_json":[
              {
                "value":60,
                "participant":121,
                "delete_reason":null
              },
              {
                "value":25,
                "participant":120,
                "delete_reason":null
              },
```


Image 29 – Formatted export file structure of Sfinx

Lastly, the existing import service logic was copied and adjusted to map data from the Sfinx export file, which was then imported through the REST interface.

5.3.2 Verification results

As indicated in the previous chapter, the verification process compares both histogram and boxplot computations of both Sfinx and the converted application.

Each of the computations were tested in the GUI interface of both applications, under specific initial conditions and while having equal database input data used for computation (see chapter “5.3.1 Import of Sfinx data”). The obtained results were then summarized below.

Histograms

Histograms accept array of either string or numeric data as their input. The input domains used to test both applications involve following cases:

- String input data
- Numeric input data

String input:
For this test, string variable *participant-gender* was chosen to be rendered in the histogram. The following results were observed in Sfinx and the converted application:

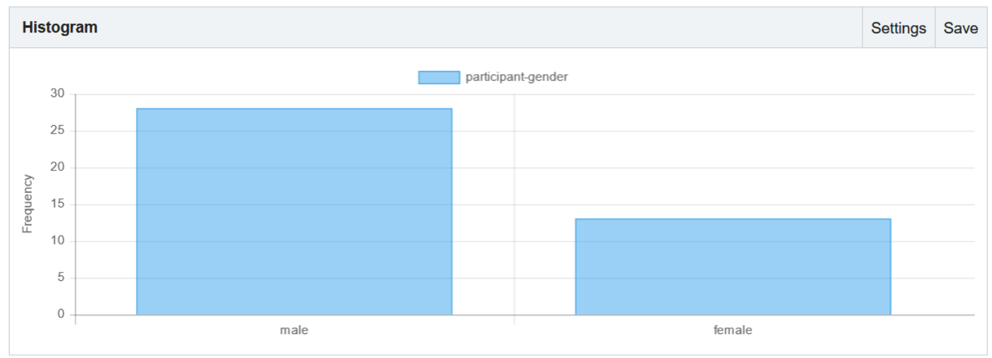


Image 30 – Sfinx histogram with string data

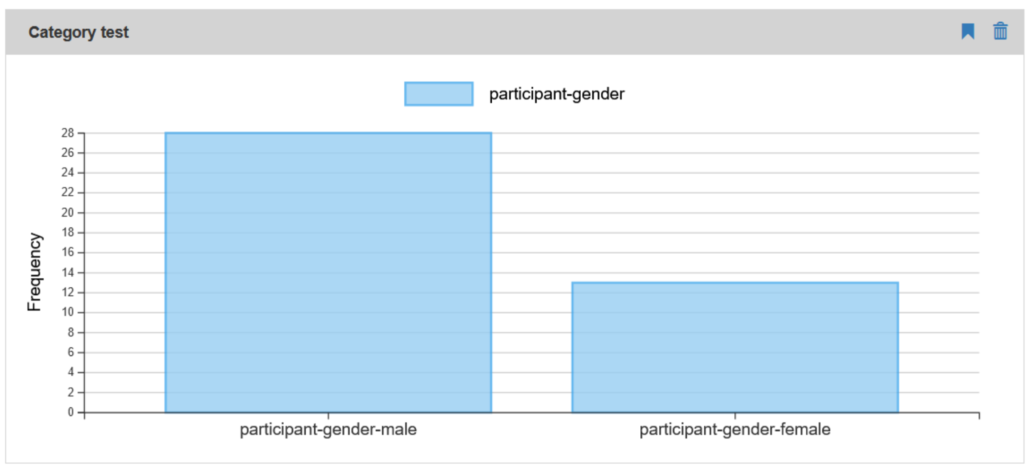


Image 31 – Converted application histogram with string data

Sfinx	**X axis values**	participant-gender male	participant-gender female
	Frequency	28	13
Converted application	**X axis values**	participant-gender male	participant-gender female
	Frequency	28	13

Table 6 – Histogram output with string data

Both graphs have been rendered with 2 bars, which had equal values in both applications. As a result of these observations, it was concluded that both applications rendered these graphs with equal outputs.

Numeric input:
For this test, numeric variable *participant-age* was chosen to be rendered in the histogram. Furthermore, the bar count of the histogram was specified to value 8. The following results were observed in Sfinx and the converted application:

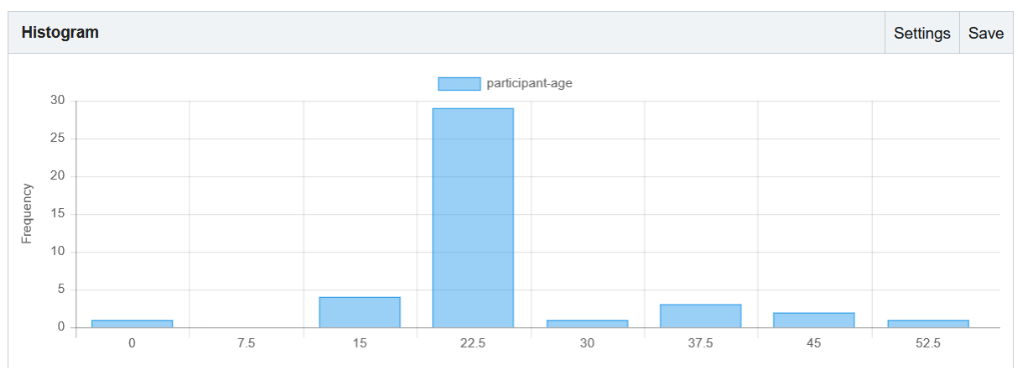


Image 32 – Sfinx histogram with numeric data

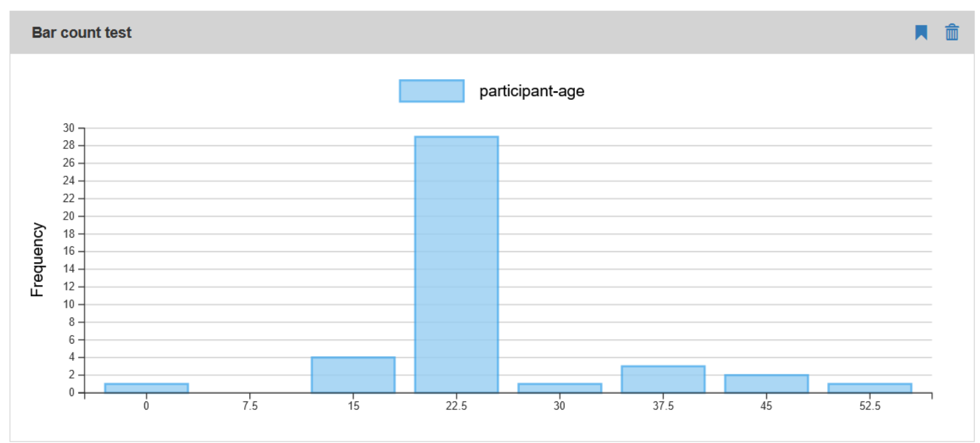


Image 33 – Converted application histogram with numeric data

Sfinx	X axis values	*0*	*7.5*	*15*	*22.5*	*30*	*37.5*	*45*	*52.5*
	Frequency	1	0	4	29	1	3	2	1
Converted application	**X axis values**	*0*	*7.5*	*15*	*22.5*	*30*	*37.5*	*45*	*52.5*
	Frequency	1	0	4	29	1	3	2	1

Table 7 – Histogram output with numeric data

Both graphs have been rendered with 8 bars, which had equal values in both applications and were specified in same positions on the X axis. As a result of these observations, it was concluded that both applications rendered these graphs with equal outputs.

Boxplots

Boxplot graphs accept map objects, where string filter variable values are mapped to lists of numeric values from a target variable. Input domains tested are as follows:

- Using normal distribution computation
- Using binomial distribution computation
- Using log transformation computation

Normal distribution:
For this test, target variable *participant-age* was chosen to be rendered in the boxplot, with all possible values of filter variable *participant-gender* on the X axis. The following results were observed in Sfinx and the converted application:

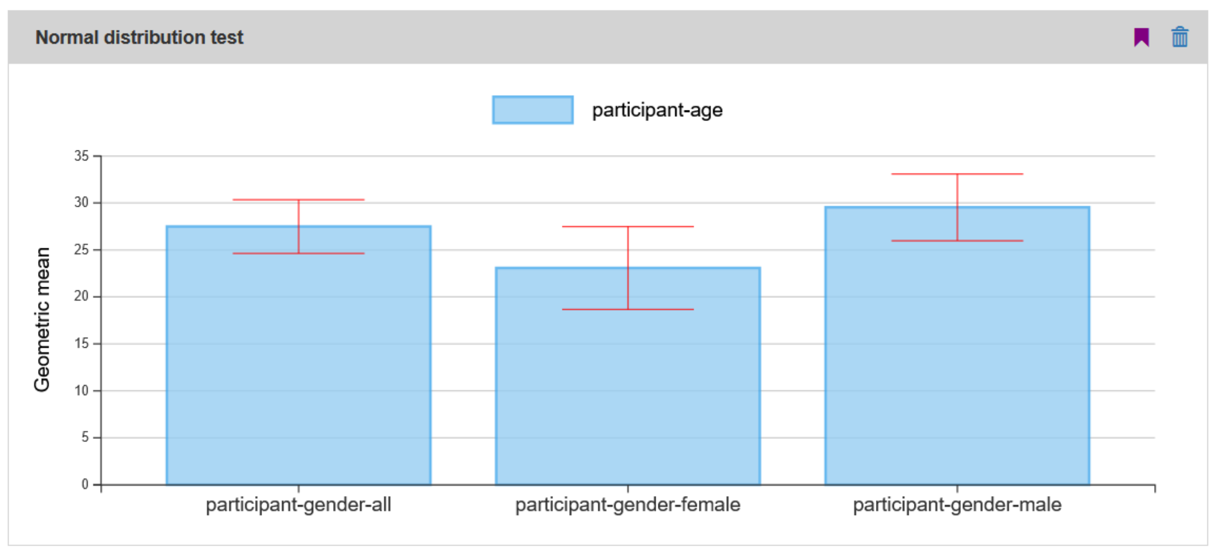


Image 34 – Converted application boxplot with normal distribution

		participant-gender female	participant-gender male	participant-gender all
Sfinx	Geometric mean	23.077	29.536	27.488
	Arithmetic mean	23.077	29.536	27.488
	Lower	18.661	25.984	24.631
	Upper	27.493	33.087	30.344
	Median	25	25	25
Converted application		participant-gender female	participant-gender male	participant-gender all
	Geometric mean	23.077	29.536	27.488
	Arithmetic mean	23.077	29.536	27.488
	Lower	18.661	25.984	24.631
	Upper	27.493	33.087	30.344
	Median	25	25.5	25

Table 8 – Computed results of normal distribution

A difference was detected in the computed median value for the "male" filter variable value. After referring to the underlying input data, it was found out that the median was computed for even number of samples, and the two center values of the dataset were 25 and 26.

Further analysis revealed that Sfinx implementation treated these center values as integers, and the division operation required to find the median has discarded their remainder because of this. Therefore, it was judged that there is no problem with the converted implementation, and the resulting outputs are otherwise equal.

Binomial distribution:
For this test, target variable *text-error* was chosen to be rendered in the boxplot, with all possible values of filter variable *text-model* on the X axis. The following results were observed in Sfinx and the converted application:

Image 35 – Converted application boxplot with binomial distribution

		text-model 5xbt	text-model 5oax	text-model 6exm	text-model 6ej7	text-model all
Sfinx	Geometric mean	19.513	28.223	21.929	19.727	21.456
	Arithmetic mean	0.182	0.273	0.207	0.184	0.211
	Lower	11.408	19.018	13.42	11.544	17.179
	Upper	27.618	37.428	30.438	27.911	25.734
	Median	0	0	0	0	0
Converted application		text-model 5xbt	text-model 5oax	text-model 6exm	text-model 6ej7	text-model all
	Geometric mean	19.513	28.223	21.929	19.727	21.456
	Arithmetic mean	0.182	0.273	0.207	0.184	0.211
	Lower	11.408	19.018	13.42	11.544	17.179
	Upper	27.618	37.428	30.438	27.911	25.734
	Median	0	0	0	0	0

Table 9 – Computed results of binomial distribution

Both applications have computed identical results for all observed filter variable values. As a result of these observations, it was concluded that both applications have produced equal outputs for this type of computation.

Log transformation:
For this test, target variable *shape-time* was chosen to be rendered in the boxplot, with all possible values of filter variable *shape-method* on the X axis. The following results were observed in Sfinx and the converted application:

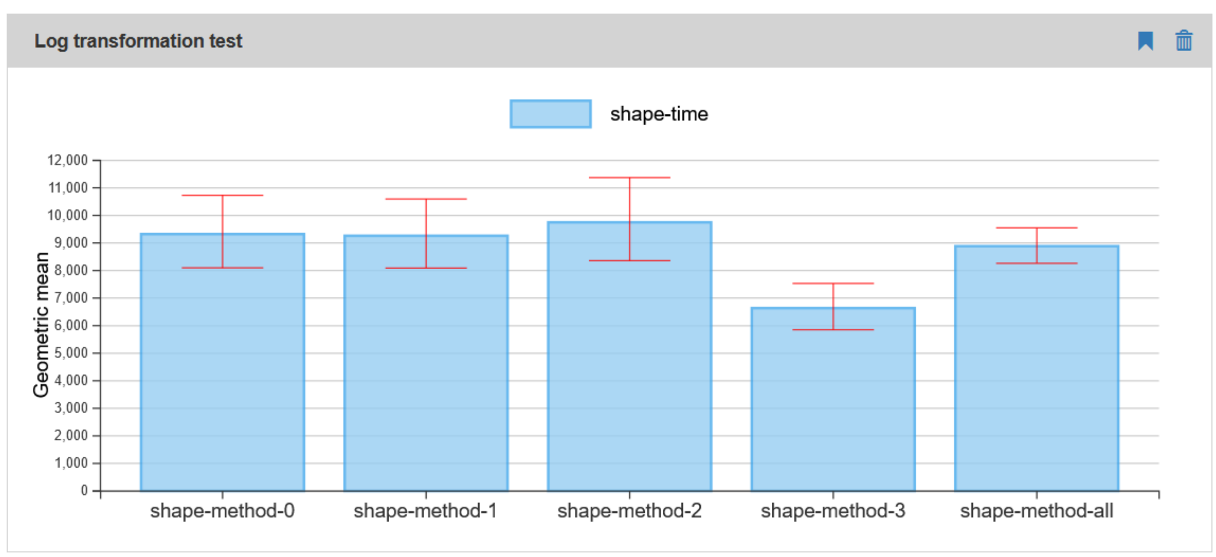


Image 36 – Converted application boxplot with log transformation

		shape-method 0	shape-method 1	shape-method 2	shape-method 3	shape-method all
Sfinx	Geometric mean	9320.7	9259.747	9747.245	6635.677	8882.844
	Arithmetic mean	17376.158	12957.963	14903.277	8208.455	13958.016
	Lower	8096.385	8090.728	8353.693	5849.667	8264.892
	Upper	10730.152	10597.676	11373.268	7527.302	9546.998
	Median	7695.06	8567.83	7261.2	5883.3	7411.973
Converted application		**shape-method 0**	**shape-method 1**	**shape-method 2**	**shape-method 3**	**shape-method all**
	Geometric mean	9320.7	9259.747	9747.245	6635.677	8882.844
	Arithmetic mean	17376.158	12957.963	14903.277	8208.455	13958.016
	Lower	8096.337	8090.688	8353.651	5849.655	8264.884
	Upper	10730.216	10597.729	11373.325	7527.317	9547.007
	Median	7695.06	8567.83	7261.2	5883.3	7411.973

Table 10 – Computed results of log transformation

Slight differences in the values of lower and upper bounds were observed this time. The converted application has computed smaller lower bounds with differences ranging approximately from 0.01 to 0.04, and higher upper bounds with differences in range around 0.015 to 0.06.

This implies that there is a difference in the delta computation, which is applied to the geometric mean to get the bounds. It should be also noted that the testing of normal distribution computation didn't exhibit this difference.

After analyzing source code of both applications, it has been concluded the difference was caused by a rounding error, after input data with high precision had been amplified by a power of two operation, only present in the delta computation. This also explains why the difference wasn't observed in the previous tests, which used input data with much lower precision.

The likely cause for this error has been determined to be the different platforms of both implementations. Regardless, it has been judged that the observed difference is insignificant enough, and that the resulting outputs of both applications are otherwise equal.

6 Conclusion

This thesis has been conducted to convert an existing application Sfinx, written in framework Ruby on Rails, to Java language. The purpose of this conversion was to simplify its maintenance and provide a basis for its future expansion.

Analysis on the theory of empiric testing was summarized in this report to better understand concepts implemented by the original application. Structure and functionality of each part of the Sfinx application was also identified to grasp the scope of necessary conversion. This step required acquisition of familiar knowledge with Ruby on Rails framework.

Design of the new application was then drafted with respect to each part of the original application, which was investigated before. The new design divided server implementation into backend and frontend projects, meant to be respectively written in frameworks Spring Boot and React. New logic model was also designed to fit the needs of the new application and described in this report, to guide the following development efforts.

Next the implementation process has been described with focus on the solutions used for its major mechanisms and component architecture. The conversion started with implementation of the backend application, which would be connected to a local database for debugging purposes. Once the backend was able to provide its REST interface, frontend application would start its development in tandem until both applications were complete.

Data gathering library didn't change except for a new method addition, which allowed it to generate balanced latin square distribution used for assigning participants to testing interfaces.

To compare functionality of both applications, data of the experiment "testapp2" was imported from Sfinx to be used in testing of the confidence interval computation. The testing then concluded that there was no significant difference between the outputted values of both applications.

In the future, the application created by this thesis can be expanded with support of the *within design* gathering method and the *ANOVA* evaluation functionality. This will expand the versatility of this tool for the purpose of statistical research.

7 References

[1] Lebedeva Antonina, 2018, Sfinx, [Bitbucket repository], Accessed: 6.1.2022, Available: https://bitbucket.org/cmolikl/sfinx/src/master/

[2] Metody výzkumu, Masarykova univerzita, [online], Accessed: 13.5.2022, Available: https://www.fsps.muni.cz/emuni/data/reader/book-8/04.html

[3] M. Kuniavsky, Observing the User Experience - A practitioner's guide to user research, Published by: Morgan Kaufman 2003, [e-book], Accessed: 13.5.2022, Available: https://www.sciencedirect.com/science/article/pii/B9780123848697000012

[4] Raluca Budiu, Between-Subjects vs. Within-Subjects Study Design, [online], Accessed: 14.1.2022, Available: https://www.nngroup.com/articles/between-within-subjects/

[5] Jim Frost, Using Confidence Intervals to Compare Means, [online], Accessed: 16.5.2022, Available: https://statisticsbyjim.com/hypothesis-testing/confidence-intervals-compare-means/

[6] Critical Values: Find a Critical Value in Any Tail - Statistics How To, [online], Accessed: 16.5.2022, Available: https://www.statisticshowto.com/probability-and-statistics/find-critical-values/

[7] Rebecca Bevans, 2021, T-Distribution | What It Is and How To Use It, [online], Accessed: 16.5.2022, Available: https://www.scribbr.com/author/beccabevans/

[8] Rebecca Bevans, 2021, Understanding Confidence Intervals | Easy Examples & Formulas, [online], Accessed: 16.5.2022, Available: https://www.scribbr.com/statistics/confidence-interval/

[9] Confidence Interval Calculator for a Completion Rate – MeasuringU, [online], Accessed: 16.5.2022, Available: https://measuringu.com/calculators/wald/#wilson

[10] Rebecca Bevans, 2021, Skewness | Definition, Examples & Formula, [online], Accessed: 16.5.2022, Available: https://www.scribbr.com/statistics/skewness/

[11] Zach Bobbitt, How to Interpret the F-Value and P-Value in ANOVA, [online], Accessed: 16.5.2022, Available: https://www.statology.org/anova-f-value-p-value/

[12] Zach Bobbitt, How to Perform a One-Way ANOVA by Hand, [online], Accessed: 16.5.2022, Available: https://www.statology.org/one-way-anova-by-hand/

[13] Bogna Szyk & Anna Szczepanek, p-value Calculator, [online], Accessed: 16.5.2022, Available: https://www.omnicalculator.com/statistics/p-value

[14] Zach Bobbitt, How to Perform a Repeated Measures ANOVA by Hand, [online], Accessed: 16.5.2022, Available: https://www.statology.org/repeated-measures-anova-by-hand/

[15] Lebedeva Antonina, 2017, Diplom, [web application], Accessed: 17.12.2022, Available: http://sfinx.felk.cvut.cz/users/sign_in

[16] OpenJS Foundation, About | Node.js, [online], Accessed: 17.12.2022, Available: https://nodejs.org/en/about/

[17] Facebook, Home |Yarn – Package manager, [online], Accessed: 17.12.2022, Available: https://yarnpkg.com/

[18] Yukihiro Matsumoto, About Ruby, [online], Accessed: 17.12.2022, Available: https://www.ruby-lang.org/en/about/

[19] David Heinemeier Hansson, Ruby on Rails, [online], Accessed: 17.12.2022, Available: https://rubyonrails.org/

[20] Daniel Mendler, Github – slim-template, [Github repository], Accessed: 17.12.2022, Available: https://github.com/slim-template/slim

[21] Chart.js | Opensource HTML5 charts for your website, [online], Accessed: 14.1.2022, Available: https://www.chartjs.org/

[22] Community project, Github – heartcombo/device, [Github repository], Accessed: 17.12.2022, Available: https://github.com/heartcombo/devise

[23] Oracle, 2022, What is Java and why do I need it?, [online], Accessed: 17.12.2022, Available: https://www.java.com/en/download/help/whatis_java.html
[24] WMware, 2022, Spring Boot, [online], Accessed: 17.12.2022, Available: https://spring.io/projects/spring-boot
[25] The Apache Software Foundation, Apache Tomcat® - Welcome!, [online], Accessed: 17.12.2022, Available: https://tomcat.apache.org/
[26] Oliver Gierke and Thomas Darimont, Spring Data JPA - Reference Documentation, [online], Accessed: 31.12.2022, Available: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.definition
[27] Facebook, React – A JavaScript library for building user interfaces, [online], Accessed: 27.12.2022, Available: https://reactjs.org/
[28] Refsnes Data, JavaScript tutorial, [online], Accessed: 27.12.2022, Available: https://www.w3schools.com/js/
[29] The PostgreSQL Global Development Group, PostgreSQL: The world's most advanced open source database, [online], Accessed: 27.12.2022, Available: https://www.postgresql.org/
[30] Hibernate. Everything data., [online], Accessed: 27.12.2022, Available: https://hibernate.org/
[31] Gradle User Manual, [online], Accessed: 27.12.2022, Available: https://docs.gradle.org/current/userguide/userguide.html
[32] Isaac Z. Schlueter, npm | npm Docs, [online], Accessed: 27.12.2022, Available: https://docs.npmjs.com/cli/v6/commands/npm
[33] Mike Bostock, D3.js - Data-Driven Documents, Accessed: 27.12.2022, Available: https://d3js.org/
[34] Adam Hayes, Null Hypothesis: What Is It and How Is It Used in Investing?, [online], Accessed: 27.12.2022, Available: https://www.investopedia.com/terms/n/null_hypothesis.asp
[35] JetBrains s.r.o, 2022, IntelliJ IDEA – the Leading Java and Kotlin IDE, [online], Accessed: 29.12.2022, Available: https://www.jetbrains.com/idea/
[36] Jan Oravec, Statistic tool, 2022, [Github repository], Accessed: 29.12.2022, Available: https://gitlab.fel.cvut.cz/oraveja1/statistic-tool
[37] VMware, Spring Initializr, [online], Accessed: 6.1.2022, Available: https://start.spring.io/
[38] DBeaver Community | Free Universal Database Tool, [online], Accessed: 31.12.2022, Available: https://dbeaver.io/
[39] Postman API Platform, [online], Accessed: 31.12.2022, Available: https://www.postman.com/
[40] JSON Web Tokens - jwt.io, [online], Accessed: 1.1.2023, Available: https://jwt.io/
[41] Martyn Shuttleworth, 2009, Counterbalanced Measures Design - Counterbalancing Test Groups, [online], Accessed: 2.1.2023, Available: https://explorable.com/counterbalanced-measures-design
[42] NormalDistribution (Apache Commons Math 3.6.1 API), [online], Accessed: 3.1.2023, Available: https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/distribution/NormalDistribution.html
[43] TDistribution (Apache Commons Math 3.6.1 API), [online], Accessed: 3.1.2023, Available: https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/distribution/TDistribution.html
[44] Create a New React App – React, [online], Accessed: 3.1.2023, Available: https://reactjs.org/docs/create-a-new-react-app.html
[45] F Table for alpha levels from .01 to .10 StatisticsHowTo, [online], Accessed: 7.1.2023, Available: https://www.statisticshowto.com/tables/f-table/
[46] T-Distribution Table (One Tail and Two-Tails) - Statistics How To, [online], Accessed: 7.1.2023, Available: https://www.statisticshowto.com/tables/t-distribution-table/

8 Attachments

Attachment A: Distribution tables

F-distribution table

This distribution table is an abbreviated sample taken from online sources [45].

DF2 / DF1	1	2	3	4	5	6	7	8	9	10
1	39.86346	49.50000	53.59324	55.83296	57.24008	58.20442	58.90595	59.43898	59.85759	60.19498
2	8.52632	9.00000	9.16179	9.24342	9.29263	9.32553	9.34908	9.36677	9.38054	9.39157
3	5.53832	5.46238	5.39077	5.34264	5.30916	5.28473	5.26619	5.25167	5.24000	5.23041
4	4.54477	4.32456	4.19086	4.10725	4.05058	4.00975	3.97897	3.95494	3.93567	3.91988
5	4.06042	3.77972	3.61948	3.52020	3.45298	3.40451	3.36790	3.33928	3.31628	3.29740
6	3.77595	3.46330	3.28876	3.18076	3.10751	3.05455	3.01446	2.98304	2.95774	2.93693
7	3.58943	3.25744	3.07407	2.96053	2.88334	2.82739	2.78493	2.75158	2.72468	2.70251
8	3.45792	3.11312	2.92380	2.80643	2.72645	2.66833	2.62413	2.58935	2.56124	2.53804
9	3.36030	3.00645	2.81286	2.69268	2.61061	2.55086	2.50531	2.46941	2.44034	2.41632
10	3.28502	2.92447	2.72767	2.60534	2.52164	2.46058	2.41397	2.37715	2.34731	2.32260

Table 11 – F-distribution table sample

T-distribution table (one tail)

This distribution table is an abbreviated sample taken from online sources [46].

df	a = 0.1	0.05	0.025	0.01	0.005	0.001	0.0005
∞	ta = 1.282	1.645	1.960	2.326	2.576	3.091	3.291
1	3.078	6.314	12.706	31.821	63.656	318.289	636.578
2	1.886	2.920	4.303	6.965	9.925	22.328	31.600
3	1.638	2.353	3.182	4.541	5.841	10.214	12.924
4	1.533	2.132	2.776	3.747	4.604	7.173	8.610
5	1.476	2.015	2.571	3.365	4.032	5.894	6.869
6	1.440	1.943	2.447	3.143	3.707	5.208	5.959
7	1.415	1.895	2.365	2.998	3.499	4.785	5.408
8	1.397	1.860	2.306	2.896	3.355	4.501	5.041
9	1.383	1.833	2.262	2.821	3.250	4.297	4.781
10	1.372	1.812	2.228	2.764	3.169	4.144	4.587

Table 12 – T-distribution table sample

Attachment B: Interfaces of Sfinx application

ExperimentDataController

Interfaces in this controller are meant for interacting with the data gathering library and serving its requests. The controller does not generate any GUI.

Available interfaces are as follows:

POST /participants
Handles registration of new participants into experiment.

Expected request parameters:

- *experiment_id:* Access token used to identify experiment to register into
- *external_id:* Optional parameter for defining alternative identifier of participant

POST /experiments/parts/{partId}/data
Handles receiving of data from registered participants.

Expected path parameters:

- *partId:* Access token used to identify experiment part to send data into

Expected JSON request body:

- *variable_values:* JSON string containing data for every variable in the current part
- *internal_id:* Identifier of participant, used for authentication
- *external_id:* Interchangeable with internal_id, if specified for participant

ChartQueriesController

One of the interface controllers available to the researchers. Handles only saving and deleting of user preferences for shown boxplot graphs.

Available interfaces are as follows:

POST /chartqueries
Handles creation of new boxplot graph to be shown in experiment detail.

Expected JSON request body:

- *name:* Name for boxplot graph to be shown in its title
- *experiment_id:* Identifier of experiment
- *target_variable:* Name of a numeric variable whose values will be used on Y axis of the boxplot
- *filter_variable:* Name of string variable acting as a frame for possible values placed on X axis of the boxplot
- *filter_variable_values:* List of values which will be used on X axis of the boxplot
- *calculate_method:* Enumerated value denoting type of computation used to compute the boxplot

DELETE /chartqueries/{chartId}
Handles registration of new participants into experiment.

Expected path parameters:

- *chartId:* Identifier of boxplot graph to be deleted

ExperimentsController

Controller containing all interfaces commonly used to manage experiments by the researchers. Also used in project as the main source of generated GUI for the Sfinx server.

Available interfaces are as follows:

POST /experiments
Handles creation of new experiment.

Expected JSON request body:

- *name:* Name of the new experiment
- *description:* Description of the new experiment
- *part_attributes:* List of attributes defined by the entity Experiment::Part, which contains nested attribute *variable_attributes* with attributes defined by entity Experiment::Variable

GET /experiments
Returns paginated list of experiments for the currently logged-in user.

Expected request parameters:

- *page:* Page of the experiment result set to be returned

GET /experiments/{experimentId}
Returns detail page of the specified experiment. Detail page also includes graphs and computed values based on currently available data.

Expected path parameters:

- *experimentId:* Identifier of experiment

PUT /experiments/{experimentId}
Handles update of the specified experiment.

Expected path parameters:

- *experimentId:* Identifier of experiment

Expected JSON request body:

- *name:* Name of the new experiment
- *description:* Description of the new experiment
- *part_attributes:* List of attributes defined by the entity Experiment::Part, which contains nested attribute *variable_attributes* with attributes defined by entity Experiment::Variable

DELETE /experiments/{experimentId}
Handles deleting of specified experiment.

Expected path parameters:

- *experimentId:* Identifier of experiment

POST /experiments/{experimentId}/copy
Handles creating of new experiment as a copy of an existing one. Only copies structure of the experiments, without the collected data.

Expected path parameters:

- *experimentId:* Identifier of experiment

GET /experiments/{experimentId}/to_debug
Handles transition of a given experiment to “debug” state. Fails if current state doesn’t support the transition.

Expected path parameters:
- *experimentId:* Identifier of experiment

GET /experiments/{experimentId}/to_edit
Handles transition of a given experiment to “edit” state. Fails if current state doesn’t support the transition.

Expected path parameters:
- *experimentId:* Identifier of experiment

GET /experiments/{experimentId}/to_open
Handles transition of a given experiment to “open” state. Fails if current state doesn’t support the transition.

Expected path parameters:
- *experimentId:* Identifier of experiment

GET /experiments/{experimentId}/to_closed
Handles transition of a given experiment to “closed” state. Fails if current state doesn’t support the transition.

Expected path parameters:
- *experimentId:* Identifier of experiment