

Technical documentation

Genomizer

Version 2.2

Publication date: 6/2/2014

Contents

1	Introduction	10
2	Target group and needs	11
2.1	Target group	11
2.2	Client needs	12
2.2.1	Storage	12
2.2.2	Processing	14
2.2.3	Conversion	15
2.2.4	Analysis	16
2.2.5	Visualization	16
3	Service description	17
3.1	Usage	17
3.2	Storage	18
3.3	Annotations	18
3.4	Processing	18
3.5	Genome releases	18
3.6	Mobile	19
4	User manual	20
4.1	Desktop application	20
4.1.1	Login and startup	20

4.1.2	Search	21
4.1.3	Upload	23
4.1.4	Process	27
4.1.5	Workspace	29
4.1.6	Administration	31
4.2	Web application	33
4.2.1	Using the interface	34
4.2.2	Setting up the application	49
4.3	Android application	50
4.3.1	Start the Application and Login	50
4.3.2	Settings	50
4.3.3	Searching for files	51
4.3.4	Pubmed Search	52
4.3.5	Search Results	53
4.3.6	Search Settings View	54
4.3.7	Experiment File View	55
4.3.8	Selected Files	55
4.3.9	Converting Files	56
4.3.10	Process View	57
4.4	iOS application	58
5	Deployment and maintenance	64
5.1	Configure server	64
5.2	Manuals	64
5.3	Configuration	64
5.4	Administer the database	65
5.4.1	Set up postgresql account	65
5.4.2	Upload SQL Script to server	67

5.4.3	Create the <i>Genomizer</i> Tables	67
5.5	Set up processing	67
5.6	Install the server	68
5.6.1	Downloading the source code	68
5.6.2	Creating a runnable JAR file	68
5.6.3	Starting the server	69
6	Interaction design	70
6.1	Desktop clients	70
6.1.1	<i>Windows/OS X/Linux</i> application	70
6.1.2	Web application	71
6.2	Android	73
6.2.1	Login View	73
6.2.2	Search View	74
6.2.3	Search Results View	75
6.2.4	Experiment View	75
6.2.5	Search Settings View	76
6.2.6	Selected Files View	77
6.2.7	Convert View	77
6.3	iOS	78
6.3.1	Navigation bar	78
6.3.2	Login Screen	78
6.3.3	Search View	78
6.3.4	Search Result View	79
7	Architecture design	83
7.1	System overview	83
8	System design	85

8.1	Desktop application	85
8.1.1	View	85
8.1.2	Model	86
8.1.3	Model	86
8.1.4	Requests	86
8.1.5	Response	87
8.1.6	Controller	87
8.1.7	Utilites	87
8.1.8	System Administration	87
8.1.9	Flow of the system	90
8.2	Web application	93
8.2.1	How our web application works	93
8.2.2	System overview	94
8.2.3	Search	94
8.2.4	Process	95
8.2.5	Upload	95
8.2.6	System administration - Web	96
8.3	Android application	97
8.3.1	Class Descriptions	98
8.3.2	Android activities	98
8.4	iOS application	102
8.4.1	Overall system design	103
8.4.2	Segue controll	105
8.5	Server	105
8.5.1	Communication	105
8.5.2	Data Conversion	107
8.5.3	File-transfer	116

8.5.4	Data Storage	118
8.5.5	Database Design	120
8.5.6	The Data Storage Subsystem	120
8.5.7	Interaction	121
8.5.8	Apache	123
9	Implementation	124
9.1	Desktop application	124
9.1.1	Testing	124
9.2	Web application	125
9.2.1	Frameworks	125
9.2.2	Technologies used	126
9.2.3	Testing frameworks	127
9.2.4	Our Tests	127
9.3	Android application	127
9.3.1	Login request	128
9.3.2	Search request	129
9.3.3	Request for Genome releases from the server	132
9.3.4	Request for conversion of RAW files to profile-data	134
9.3.5	Request for status on conversions on the server	136
9.3.6	Testing	138
9.4	iOS application	138
9.4.1	Login	138
9.4.2	Search	138
9.4.3	Experiment Selection	139
9.4.4	File Selection	139
9.4.5	Convert Request	139
9.4.6	Testing	145

9.5	Server	145
9.5.1	Communication	145
9.5.2	Conversion	148
9.5.3	File-transfer	149
9.5.4	Data Storage	150
9.6	Limitations	152
Bibliography		153
A <i>User Stories</i>		155
A.1	Implemented user stories	155
A.2	Product backlog	157
B Android application: <i>UML</i>-diagrams		162
C Desktop application: <i>UML</i>-diagrams		165
D Data Storage: <i>UML</i>-diagrams		166
E Server API		169
F <i>Server commands</i>		187
G <i>Ubuntu 14.04 Installation and configuration manual</i>		189
G.1	Introduction	189
G.2	Installation and Configuration	189
G.2.1	Java	189
G.2.2	OpenSSH	190
G.2.3	Apache2	190
G.2.4	Git	193
G.2.5	Ant	194
G.2.6	PHP5	194

G.2.7	SRA Toolkit	195
G.2.8	PostgreSQL	195
G.2.9	PgAdmin	197
G.2.10	PhpPgAdmin	197
H	<i>Debian 7.5 Installation and configuration maunal</i>	200
H.1	Introduction	200
H.2	Installation and Configuration	200
H.2.1	Installation of Debian	200
H.2.2	Configure Debian repositories	201
H.2.3	Create a super user	201
H.2.4	Locales	202
H.2.5	Java	202
H.2.6	OpenSSH	203
H.2.7	Apache2	203
H.2.8	Git	207
H.2.9	Ant	207
H.2.10	PHP5	207
H.2.11	SRA Toolkit	208
H.2.12	PostgreSQL	208
H.2.13	Inject database copy	210
H.2.14	PgAdmin	211
H.2.15	PhpPgAdmin	211
H.2.16	Genomizer configuration	213
I	<i>Migration of the Genomizer system</i>	215
I.1	Introduction	215
I.2	Steps of migration	215

J	<i>Backup</i>	217
J.1	Introduction	217
J.2	File backup	218
J.3	Database backup	219
J.4	Chrontab	220
K	Known problems	221
K.1	Web application	221
K.1.1	Moving backwards in the browser does not hide modal windows	221
K.1.2	Error handling when uploading experiments	221
K.1.3	Old authorization token causes page redirect	221
K.1.4	Code duplication in SearchResults and Experiments	222
K.1.5	No warning when closing tab during upload.	222
K.1.6	Uploading genome release - does not update list automatically	222
K.1.7	The annotation list can't be sorted	222
K.1.8	Sidebar on adminpage dosen't stay vertical	222
K.1.9	Missing error check on annotation values	222
K.1.10	No warning when closing tab during uploading genome releases	223
K.2	<i>iOS</i> application	223
K.2.1	Unspecified behaviour on loss of internet connection	223
K.2.2	Lack of security	223
K.2.3	No administrative features	223
K.3	Server	223
K.3.1	Communication and control	223
K.3.2	Upload and download	225
K.3.3	Process limitations	225

Chapter 1

Introduction

Genomizer is a system for storing and analyzing *DNA*-sequences. It was designed for researchers in the field of epigenetics, who are interested in where on a *DNA* string a certain protein binds. In order to get this information, experiments are conducted and *raw* data files collected. These data files are then converted, in a series of steps, to files suitable for analysis. *Genomizer* allows the researchers to upload *raw* files to a server and automate the generation of analysis data.

Genomizer was developed by students at *Umeå University* as part of a project for the course *Software Engineering*. This documentation is directed towards three main groups: end users, system administrators and developers.

The first part of this documentation describes the target group for the software. This is followed by instructions on how to use the software. The next section is aimed at administrators and developers and includes instructions on how to deploy the server and software. The final chapters take an in depth look at how the software has been designed and implemented and is aimed at the more inquisitive reader.

Chapter 2

Target group and needs

The *Genomizer* system was designed with a specific target group in mind: Epigenetic researchers. This chapter will explain the needs of these users, the problems they faced before this system was provided and the requirements that were collected and taken into account during the project.

2.1 Target group

The target group for the *Genomizer* system is the *Epigenetic Cooperation Norrland (EpiCoN)*, a diverse group of researchers at *Umeå University* made up of many different nationalities. Their main communication language is English.

EpiCoN are involved in the research of how proteins bind to *DNA* strings and its effects. Experiments are carried out which yield large amounts of raw data. This information, combined with knowledge about the location of genes within a given genome, enable the researchers to gain valuable information about which proteins are active in enabling and disabling genes. These results are important in the study of how cells "remember" which genes should be enabled after cell division.

Previous to the *Genomizer* project the raw data files retrieved from experiments were manually processed by the researchers using inefficient *Perl* scripts. This process also involved using *Bowtie*[1], a program used to unscramble the *DNA* data, and *LiftOver*[2] which is used to adjust results to conform to different *genome releases*.

The researchers at *EpiCoN* have varying computer skills. While they all have basic computer knowledge, not all are familiar with more advanced computing tasks such as running scripts at command line level. As such, some researchers have become dependent on others to process the raw data. At *EpiCoN* the researcher that has the knowledge to use all the scripts and software performs many of these time consuming tasks for other researchers.

From time to time students are interested in working with the data, however their access is limited to viewing and analyzing the data.

2.2 Client needs

The researchers at *EpiCoN* need a system to structure the large amount of genetic data they use daily. The requirements, as described below, were collected and handled as a number of *user stories*, each of which describe a desired function from the end users perspective. A complete list of the *user stories* are presented in Appendix A. When discussed below the title of the relevant *user story* will be used.

There are three main data types used in the research and that the system should handle: *raw*, *profile* and *region* data. *Raw* data is the raw output from an experiment and cannot be analyzed directly. It is first processed to so called *profile* data. *Profile* data describes the amount of reads found for every base-pair in an organism's genome. *Region* data is further processed *profile* data consisting of the regions where every base-pair's read strength is above a given threshold and fault tolerance. The region gets a value based on the average of the base-pair reads for the given region.

2.2.1 Storage

When conducting experiments the researchers treat DNA with a given protein that binds to certain positions. The DNA-strings are then broken up and the bound areas separated. *Raw* data can then be collected on what these pieces look like on a base pair level. This data is the foundation for further research and so must be stored securely and in a structured manner. A system that automates the archiving of *raw* data to a shared location is therefore desired. This requirement was specified in the *user stories* "Single Upload" (see Figure 2.1) and "Single Download" (see Figure 2.2).

Single upload
To store a single data file the researchers want to be able to upload a specific file.

Figure 2.1: User story for uploading a file to the central database

Another way researchers obtain *raw* data is from official publications. When results are published in scientific articles the *raw* data from the experiments are often also provided. One location where these *raw* data files can be published is the *GEO* (*Gene Expression Omnibus*) database. A desire to be able to initialize

Single download
To scrutinize a single data file the researchers want to be able to download a specific file.

Figure 2.2: User story for downloading a file from the central database

a download of *raw* data to *Genomizer* from this source was also expressed and written up in the *user story* “Download from GEO Database”.

It is not possible/highly impractical to gain knowledge of an experiment by viewing the characters that make up the *raw* data files. In order to save information about an experiment and the resulting files, the researchers must be able to annotate them upon entry to the database. There are several *user stories* to do with the annotation of experiments/files and the manipulation of the annotation fields. An example of such a *user story* is “Annotation” (see Figure 2.3).

Annotation
To structure the data files the researchers want to be able to annotate the data files.

Figure 2.3: User story for the annotation of files stored in the database

As the database is filled with data, the researchers want to ensure that it is kept safe. They therefore want to have an authorization system to protect the data from unauthorized access. Another risk that must be mitigated is the risk of hardware failure. These concerns were captured in the *user stories* “Password Protected” and “Backup” respectively (see Figure 2.5 and Figure 2.4).

Backup
To prevent loss of data the researchers want the data to be backed up.

Figure 2.4: User Story for the protection from data loss due to hardware failures

Password protected
To protect the database from unauthorized use the researchers want the application to be password protected.

Figure 2.5: User story for the protection of the database from unauthorized access

2.2.2 Processing

The unordered *raw* data gained from an experiment requires processing in order to be analyzed. The researchers have written a number of scripts and, when combined with the *BowTie* algorithm, generate *profile* data. In this format the *DNA* pieces are ordered and mapped to the *DNA* string. It is important that the system automates this process so that all researchers can easily process the large *raw* files, see Figure 2.6.

Raw to profile
To be able to analyze the researchers want to process raw data to profile data.

Figure 2.6: User story for converting *raw* data to *profile* data

As new discoveries are made in the area, new standards for the order of the base pairs in a *DNA* string are set. This results in a new *Genome Release* for a specific species. These are obtained as a set of files specifying this order and are used in the processing of *raw* data. *Genomizer* must support the uploading of new sets of *genome release* files to be used in processing otherwise the system will very quickly become outdated. This is specified in the user story “Add Genome Release” shown in Figure 2.7.

Add genome release / reference genome
To be able to annotate the data properly and extract genome reference the researchers want to be able to add genome releases and reference genome.

Figure 2.7: User story for the addition of *Genome Releases*

It would also be an advantage if the system could carry out further processing from *profile* to *region* files and is captured in the user story “Profile to Region”

shown in Figure 2.8.

Profile to region
To be able to find regions of interest the researchers want to process profile data to region data (Per's code).

Figure 2.8: User story for the processing of *profile* to *region* files

After processing, the resulting data files should be annotated and saved in the database alongside their parent files. It is important that the parent files remain traceable and that the parameters used in processing are saved so that the process can be repeated and confirmed (see Figure 2.9).

File traceability
To be able to access the underlying raw data or profile data the researchers want the raw data files to be traceable from profile files and the profile files to be traceable from the region data (if available) when the files have been generated on the server.

Figure 2.9: User story for the traceability of processed data

2.2.3 Conversion

Genomizer should also provide a way to convert *profile* data files between different genome releases as specified in Figure 2.10. This involves the ability to upload new *Chain Files* which enable conversion using *LiftOver* and the embedding of this program, see Figure 2.11.

It is not uncommon for errors in a new release to be discovered after publication. It is therefore also important to store files generated using older genome releases for some time after a new release is published.

Convert genome release
To easier handle files the researchers want to convert files between genome releases (<i>LiftOver</i>).

Figure 2.10: User story for the conversion of processed data between different genome releases

Add chain file
To be able to convert between genome releases the researchers want to upload chain files (<i>LiftOver</i>).

Figure 2.11: User story for the addition of new *Chain Files*

2.2.4 Analysis

The researchers also want to be able analyze data using the server. This involves combining regions using logical arithmetic and in such a way construct new regions (“Combine regions”). It should also be possible to create new regions from a reference point (“Create region subset”). Another interesting analysis is overlap analysis which shows how much a number of genome experiments overlap (“Overlap analysis”).

2.2.5 Visualization

In order to view analysis results, the researchers would like a graphical presentation of the results (“Plot overlap analysis”, “Plot average regions”). They also want to be able to use the *Integrated Genome Browser* (IGB)[3] software to view results (“IGB Session”). They therefore want to be able to download a session-file to immediately start the browser.

For a full list of the user stories used in the project see Appendix A. They are divided into two sections, Implemented User Stories and Product Backlog. The *Product Backlog* is the set of user stories that were not completed during the project and will be the target for further development projects.

Chapter 3

Service description

This chapter will present an overview of the services that the *Genomizer* system currently provides.

3.1 Usage

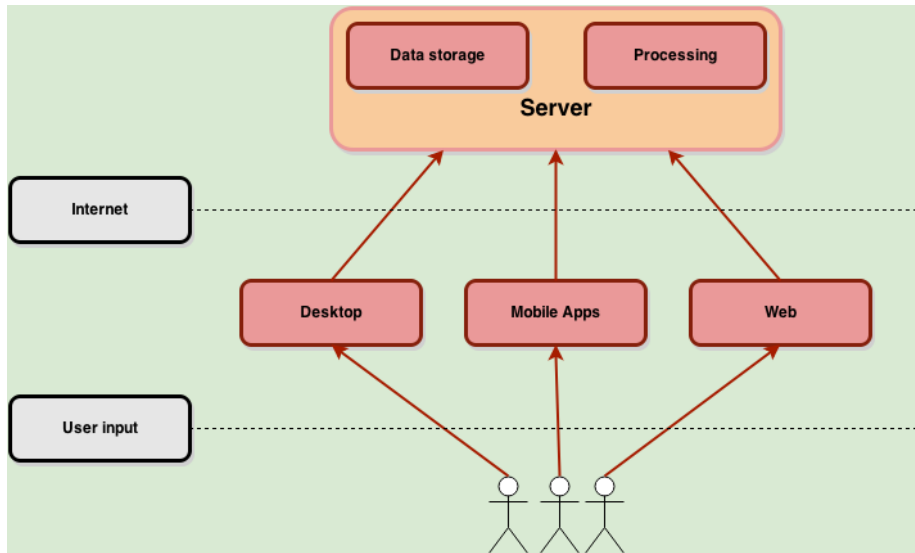


Figure 3.1: Communication diagram of the product

In order to give the users flexibility when using the service there are clients for many different platforms (Windows, Linux, OSX, Web, Android and iPhone). When a user chooses a given task, for example start *raw* to *profile* processing, that task is sent by Internet to the server as shown in Figure 3.1 which will handle the request and send a response back to the user.

3.2 Storage

The main purpose of the *Genomizer* system is to centralize all data. To enable this a user can annotate and upload data to the server using both desktop and web based clients. Advanced database searches can be performed on the annotations to find previously uploaded data. When the required data is found the user can choose to download the files or request that they be processed on the server.

3.3 Annotations

Genomizer not only allows the annotation of files and experiments, but also enables the addition of new annotation fields. For example, if the user has an experiment that was conducted in zero gravity and the database does not have the annotation field “Zero Gravity” the user can add this as a new annotation. In this case a *Drop Down* annotation type may be appropriate, with the simple choices “yes” or “no”. Of course it is also possible to leave the annotation type as *Free Text* which enables users to write freely the value of the annotation.

Dynamic annotations must also be managed in order to keep the system clean and up to date. *Genomizer* therefore provides full editing options for existing annotations. This includes the editing of *Drop Down* annotation choices and the removal of unused annotations.

3.4 Processing

Users can request that a *raw* file set be processed to *profile* files. This procedure is carried out on the server to avoid heavy workload on the clients. The processing carried out between *raw* data and *profile* data involves a number of different steps. The user can choose which steps are carried out and the various parameters used.

3.5 Genome releases

Users can upload new genome release files and use them in the processing of *raw* to *profile* data.

3.6 Mobile

Due to the limited storage available on mobile devices it is not appropriate to enable uploading and downloading of files, however the mobile applications enable the searching of files in the database and the scheduling of processing procedures for the conversion of *raw* to *profile* data.

Chapter 4

User manual

This chapter explains how you use each of the *Genomizer* clients. First instructions on how to use the desktop and the web clients are presented. These are the clients which provide the most functionality. The mobile clients are more lightweight and offer a subset of the functionality presented by the desktop client. Instructions on using the smartphone applications for *Android* and *iOS* are presented in their own sections at the end of the chapter.

4.1 Desktop application

This is a user manual for the desktop client. It will provide guides on how to use the client and the different functionalities it holds. The screen shots shown in this document are made from a Linux machine, but the application also runs on Windows or Mac, and will follow the design principles thereafter. Because of this, some details of the look of the client may vary, but the functionality is the same.

4.1.1 Login and startup

When you start this application the first thing that's displayed is a login screen, as illustrated in Figure 4.1. In this screen you enter your username, password and the IP-Address for the server and then press enter or login to enter the *Genomizer* Desktop.

The application is built with tabs, as illustrated below in Figure 4.2. Each tab contains separate features of the application. There are five tabs: Search, Upload, Process, Workspace and Administration.

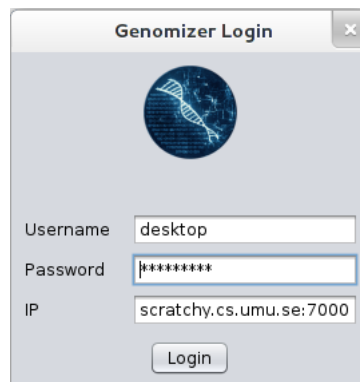


Figure 4.1: Screenshot of the login screen.

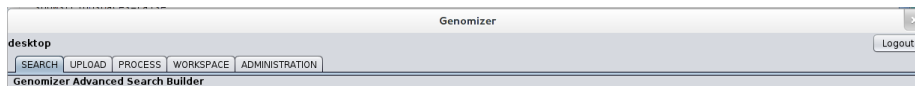


Figure 4.2: Illustration of the different tabs of *GenomizerDesktop* and displaying the Search tab.

4.1.2 Search

The first tab that meets the user after logging in is the Search tab, illustrated in Figure 4.3. The Search tab uses the same query building technique as the “Pubmed Advanced Search Builder”[6]. It has one text field where you either can type in the query yourself or you can use the query builder below it. To switch between manually editing the query and using the query builder there are two radio buttons to the left of the text field. Each row in the query builder has at most five components. These are a logical expression, an annotation name field, a free text field or a drop down menu to insert search words, a minus button and a plus button. The minus button removes a row and the plus button adds a row. These buttons are however not available in each row. The plus button is only available in the last row. The minus button is available in every row except if there is only one row in the query builder. The logical expressions combines the annotations, so they are available in every row but the first. By writing in the annotation text field or selecting a value in the drop down menu you can specify the query the row will produce. Together each row builds a full query. As illustrated in Figure 4.3 below.

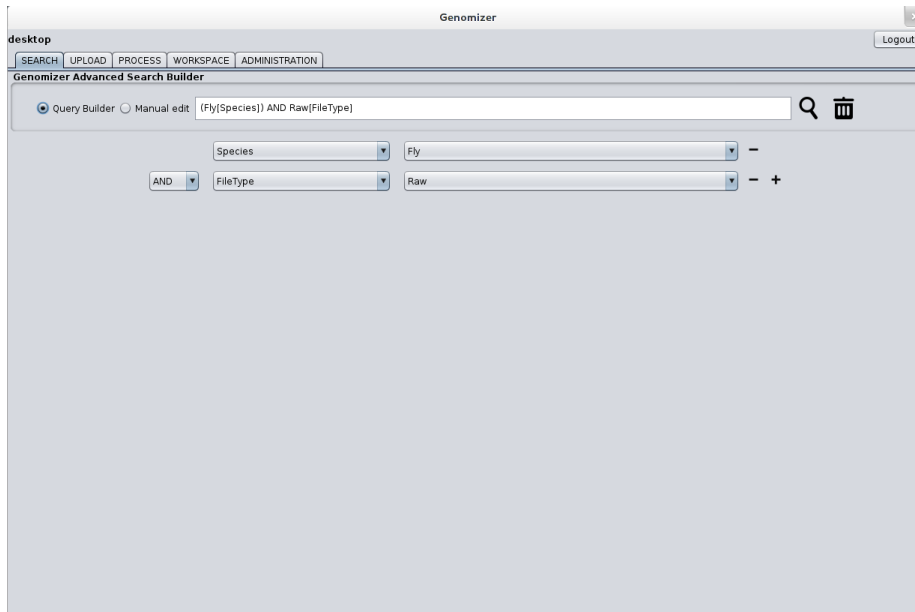


Figure 4.3: Illustration of a query, made by the query builder.

4.1.2.1 Search results

When the search button is clicked the search tab will change it's view to display the search results as illustrated in Figure 4.4. The results are displayed as experiments in a tree table. The experiment nodes in the table can be expanded to view the files associated with the experiment. The tree table can be sorted both vertically by clicking the headings and horizontally by dragging and dropping the columns. The user can choose which columns to display by using the menu in the upper right corner of the table. In the same menu there are also buttons for expanding and collapsing all experiments in the search results. To go back to the previous view, the user can click the *Back* button. There is also a button called *Add to workspace* for adding the selected files or experiments to the workspace.

ExpID	Cell line 2	Is forced	magnus	species8472	Testar	Species	Gender2	testhest
▶ batchtest0	No	yes	Yes	Yes	Typ	Fly	daniel	-
▶ c11v1gtest	maybe	-	-	Yes	Yes	Fly	-	-
▶ c11v1gtest2	maybe	-	-	Yes	Yes	Fly	-	-
▶ desktop	maybe	-	-	Yes	Yes	Human	-	-
▶ Dota1.3	maybe	yes	-	Yes	Typ	Superhero	-	-
▶ Dota2	-	yes	-	Unknown	No	-	-	-
▶ fczx	maybe	yes	-	Yes	Typ	Fly	-	-
▶ grs	maybe	yes	-	Yes	Typ	Fly	-	-
▶ Hagrid	maybe	-	-	Yes	Yes	Fly	-	-
▶ hattfnatt	maybe	-	-	Yes	Yes	Fly	-	-
▶ huggasExp	maybe	-	-	Yes	Yes	Fly	-	-
▶ hv	maybe	-	-	Yes	Yes	Fly	-	-
▶ ioshax	maybe	-	-	Yes	Yes	Human	-	-
▶ ioshax3	maybe	-	-	Yes	Yes	Fly	-	-
▶ ioshax4	maybe	-	-	Yes	Yes	Human	-	-
▶ Lytkran	maybe	-	-	No	Yes	Human	-	-
▶ nuFunkasDet	maybe	-	-	Yes	Typ	Fly	-	-
▶ nuFunkasDet0	maybe	-	-	Yes	Typ	Fly	-	-
▶ oasntest	No	-	-	Yes	Yes	Fly	-	-
▶ plutten	maybe	-	-	Yes	Yes	Fly	-	-
▶ real_test_experiment	-	-	-	-	-	Fly	-	-
▶ rua	maybe	-	-	Yes	Typ	Fly	-	-
▶ rua3	maybe	yes	-	Yes	Typ	Superhero	-	-
▶ rua4	maybe	yes	-	Yes	Typ	Superhero	-	-
▶ Ruardhs	maybe	-	-	Yes	Yes	Fly	-	-
▶ Ruardhs Test Exp	maybe	-	-	Unknown	Typ	Fly	-	-
▶ Slemm	Unknown	-	-	No	Unknown	Human	-	-
▶ small_test_files	maybe	-	-	Yes	Yes	Fly	-	-
▶ test1	No	yes	Yes	Yes	Typ	Fly	daniel	Yes
▶ test_desktop7	maybe	-	-	Yes	Yes	Fly	-	-
▶ test_interrupt	maybe	-	-	Yes	Yes	Fly	-	-
▶ test_interrupt2	maybe	-	-	Yes	Yes	Fly	-	-
▶ TestTest	maybe	-	-	Yes	Yes	Fly	-	-
▶ uploadtest232	maybe	-	-	Yes	Yes	Fly	-	-

Figure 4.4: Illustration of search results.

4.1.3 Upload

If the user needs to upload files to the database it can be done through the upload tab. When the tab is pressed the user gets presented with a text field and two buttons. The textfield and the first button is used for searching for existing experiments. The second button is used for creating new experiments. This is illustrated in 4.5.

4.1.3.1 Existing experiment

In order to upload files to an existing experiments the users needs to write the experiment name in the textfield and press the "Search for existing experiment"-button shown in figure Figure 4.6. When this is done the experiment information get retrieved from the server and presented for the user. For an existing experiment no editing of the annotations can be done. So after retrieving the wanted experiment the user presses the "Browse files"-button. Then a file browser window pops up, it is illustrated in ???. Here the user selects the files that are supposed to be added to the experiments and then presses "open". The files will be added to the upload tab and there will be some new choices available for the user. Each file will be associated with one file row, this is also shown in ???. The new choices are whether the new files are either raw, region or profile files. And if it is region or profile there is another choice for which genome release. There is also the possiblity to delete the file row, by clicking the "X"-button, in case this file is not suppose to be added to the experiment. After all is decided and the files are correct the user simply clicks the "Upload files"-button. Then

the progress bar starts to progress and if all goes well it will reach 100% and the files is added to the existing experiment.

4.1.3.2 New experiment

The first thing a user needs to do when creating a new experiment is pressing the "Create new experiment"-button in the upload tab. After pressing this button all the different annotations get retrieved from the server. If the annotation is of the type that should be filled with text there is a textfield to be filled out, and if it's a multiple choice annotation there is a dropdown menu of the different choices. The boldtexted annotations are forced and needs to be filled out in order to create the experiment. There are also three buttons added to the view. This is illustrated in 4.7. In order to add files to this experiment the user needs to press the "Browse files"-button and choose in the file browser window which files are to be added. When the files are added they each get displayed in a file row. The file row consists of the file name and a progress bar. And apart from this there are also three button and a checkbox. The checkbox will be explained in section 4.1.3.3 below. The other three button are used in the same manner as in section 4.1.3.1 above. When all the annotations that are needed is filled and the associated files are added the user presses the "Create with all files"-button. The "Create experiment with selected files"-button is discussed in section 4.1.3.3 below.

4.1.3.3 Batch upload experiments

In order to batch upload experiment the workflow of this application is suggested as follows: First the user start of as usual when uploading one experiment, as explained in 4.1.3.2. But instead of choosing the wanted files for that experiment the user chooses all the files that are supposed to be uploaded to all the different experiments that are supposed to be batched. When the annotations for the first experiment to be uploaded are chosen the user selects the files to be associated to this experiment by click the "select"-checkbox. Then presses "Create experiment with selected files"-button. This creates the first experiment and starts to upload the files to it. And then the user changes the annotations that needs to be changed for the second experiment and then selects the files for that experiment in the same manner. The user then clicks the "Create experiment with selected files"-button again and then changes the annotations to match the third experiment and the selects the files for it and starts the upload. Every file that is finished will disappear from the view and for each finished experiment a popup window will be shown. So when all files are gone from the view they are all added to the different experiment that the user filled out.

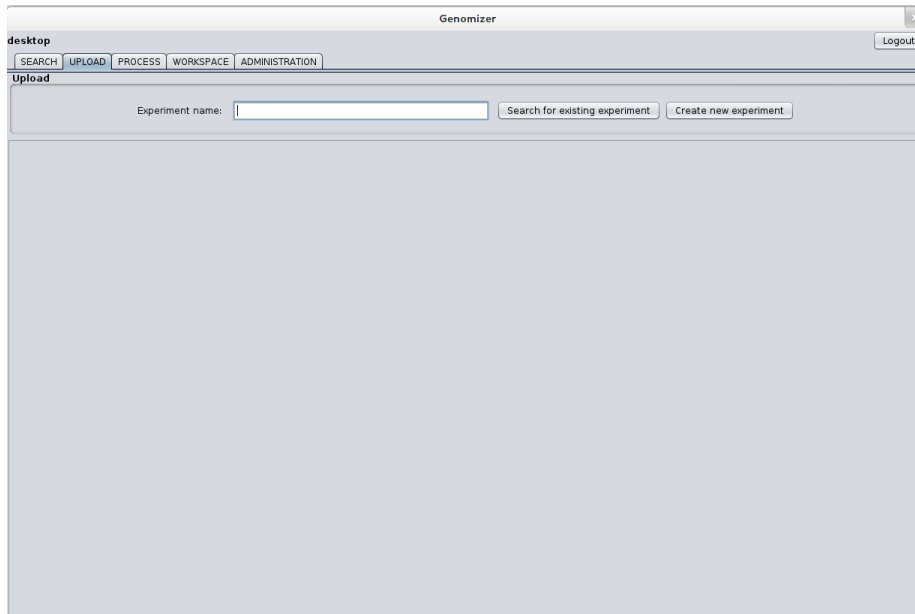


Figure 4.5: Illustration of the starting view of the upload tab.

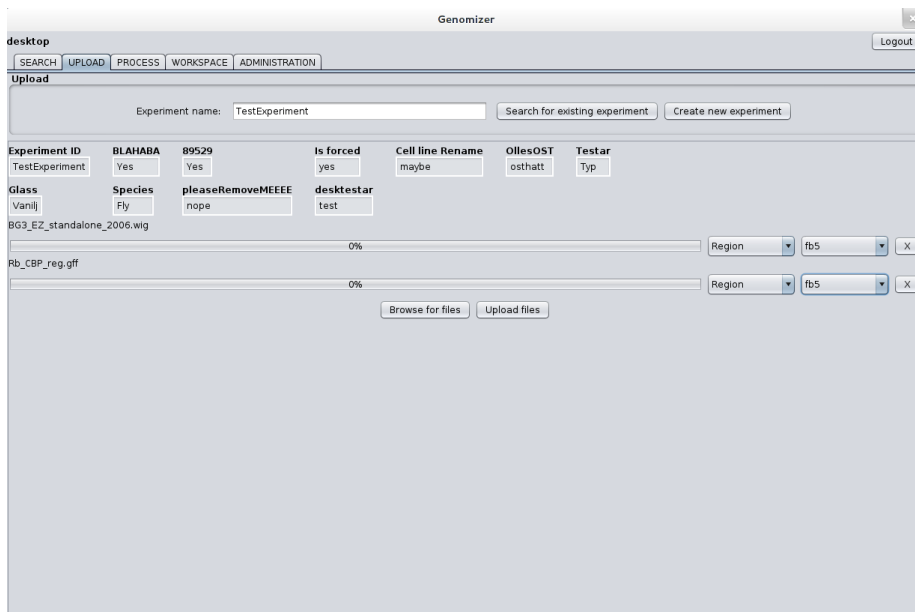


Figure 4.6: Illustration of the add to existing experiment part of the upload tab.

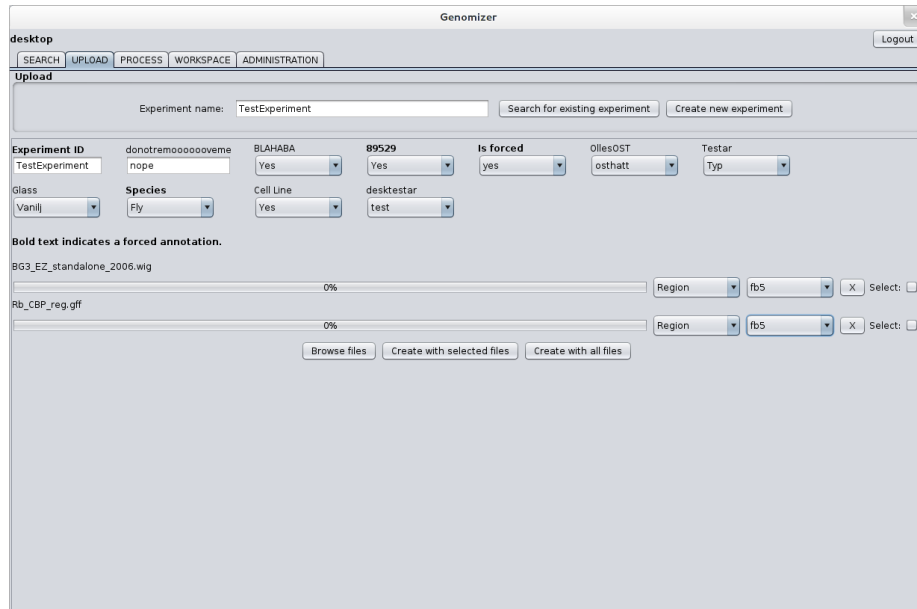


Figure 4.7: Illustration of the create new experiment part of the upload tab.

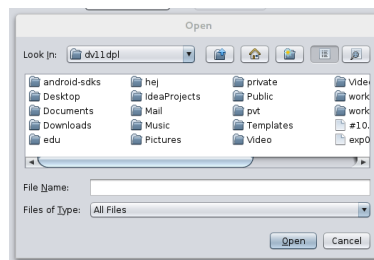


Figure 4.8: Illustration of the file browsing window.

4.1.4 Process

In the process tab there is a list of files to the left. These files are chosen from the Workspace tab for process, see 4.1.5. From this list the user can mark RAW-files and choose to create profile data. By left clicking on the files they will be marked. If the user left clicks once again on the same file it will be unmarked. For each file there exists only one specie, the list shows the user which specie a file has. When a file is marked the *Genome release files* dropdown list will be filled with all genome versions that exists for that specie. If the user then enters the create profile data tab and presses the Start process button which is visible in the middle of the tab see Figure 4.9, all the files that are marked will now be processed to profile data. This list of files will be empty unless the user has chosen to process selected RAW-files from the workspace tab. If that is the case then those selected RAW-files will then be visible in the list of files in the process tab. When the user has selected some RAW files the user has the option to change processing parameters that is above the Start process button as illustrated in Figure 4.9. These parameters has pre-set values and allowed intervals. The conversion parameters are *Flags*, *Genome release files*, *Window size*, *Smooth type*, *Step position*, *Step size*, *Print mean* and *Print zeros*. Information about all the different parameters can be found in a popup windows showed in Figure 4.10. For the user to reach this window he/she needs to press the information button that is on the upper right side in the process tab. To be able to process files some parameters needs to be set in order for the process to start. If the parameters are invalid, empty or wrong parameters then process will not be able to start until that is fixed. Depending on what format the user chooses to process to different parameters will be enabled. For example ratio calculation parameters cant be set unless SGR format is used.

If the user has selected some RAW-files and pressed the Start process button, then if all went well and the server could process the files a message "The server has started process on file: <File> from experiment: <Experiment>" will print in the Console for each file that was converted to profile data. If for some reason the server couldn't create profile data for any RAW-file another message "WARNING - The server couldn't start processing on file: <File> from experiment: <Experiment>" will print in the console that is visible in the middle bottom of the process tab see Figure 4.9. If the user wants to perform a ratio calculation while processing a file the user has the option to press the *Use ratio calculation* button. When pressed a popup window appears and the user gets the option to write in several ratio calculation parameters. These parameters consists of eight parameters *Ratio calculation*, *Input reads cut-off*, *Chromosomes*, *Window size*, *Smooth type*, *Step position*, *print mean* and *print zeros*. If the Console area gets filled with messages then the user has the option to clear the Console area from text. This is possible when pressing the Clear console button which is positioned bottom/center in the process tab. When a user has started a process he/she can choose to check which priority that process currently have. This is done by pressing the Get process feedback button which is located in the bottom/right corner of the process tab se Figure 4.9.

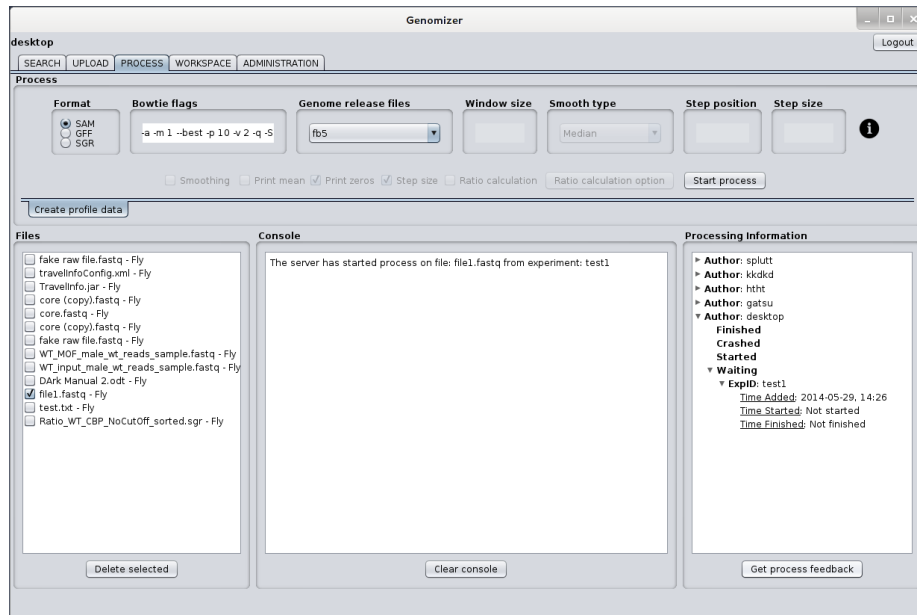


Figure 4.9: Screenshot of the process tab in the program.

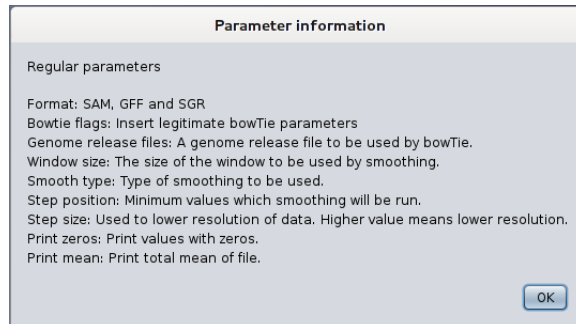


Figure 4.10: The parameter information popup window.

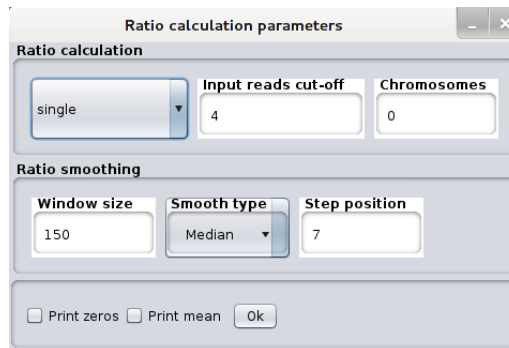


Figure 4.11: The popup window for ratio calculation parameters.

4.1.5 Workspace

The workspace Tab seen in Figure 4.12 is a tab where a user can temporarily store experiments and their files, and choose different options for action. Results from various searches can be stored here, and the contents of the workspace is saved as long as the program is running. Files and/or experiments are chosen by clicking them, multiple files by using either Shift-click, Ctrl-click or simply holding down the mouse button and dragging the cursor over multiple files. By choosing an experiment, all of the containing files are selected. Items can be deleted from the Workspace by pressing *Remove from workspace*.

4.1.5.1 Delete from database

To delete the selected data from the database the *Delete from database* button should be used instead. When pressing the delete button a small popup window with a progress bar will be displayed. By closing this window the deletion of data can be aborted.

4.1.5.2 Upload to

If the user wants to upload files to an experiment they have in the workspace, they can simply click the *Upload to* button to switch to the upload tab and upload to the experiment they have selected. If multiple experiments have been selected, only the first one will be uploaded to.

4.1.5.3 Process

If the user wants to add files to the process tab there is a *Process* button which transfers the selected files to the process tab file list.

4.1.5.4 Download

The user can make the choice to download files to their local computer. If the user presses the *Download* button seen in Figure 4.12, then the user gets to choose a directory where the files will be saved. When a directory has been chosen, the files get downloaded and all current and completed download can be seen in the tab *downloads*, see Figure 4.13. The current downloads can be aborted by clicking the X button and completed downloads can be removed in the same way. The down

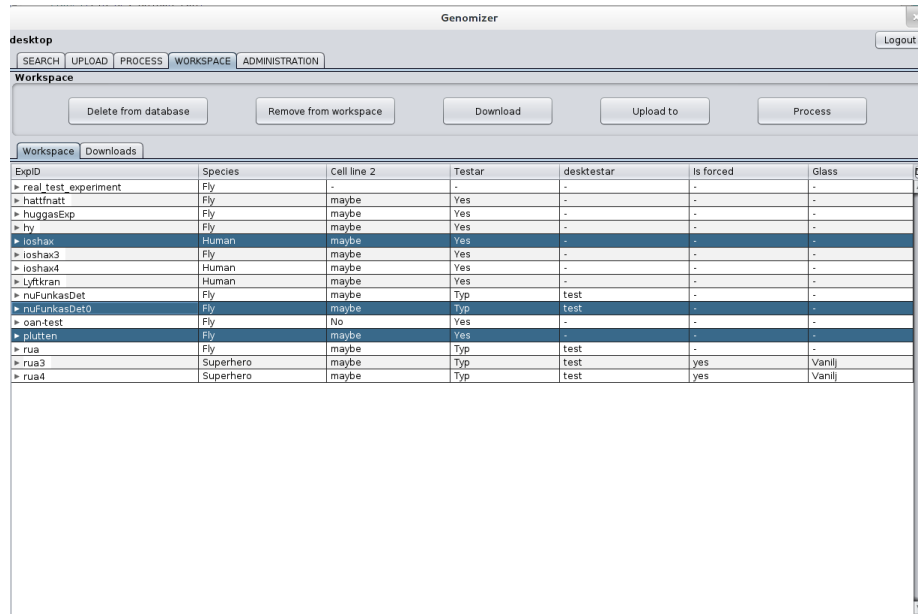


Figure 4.12: Screenshot of the workspace tab in the program.

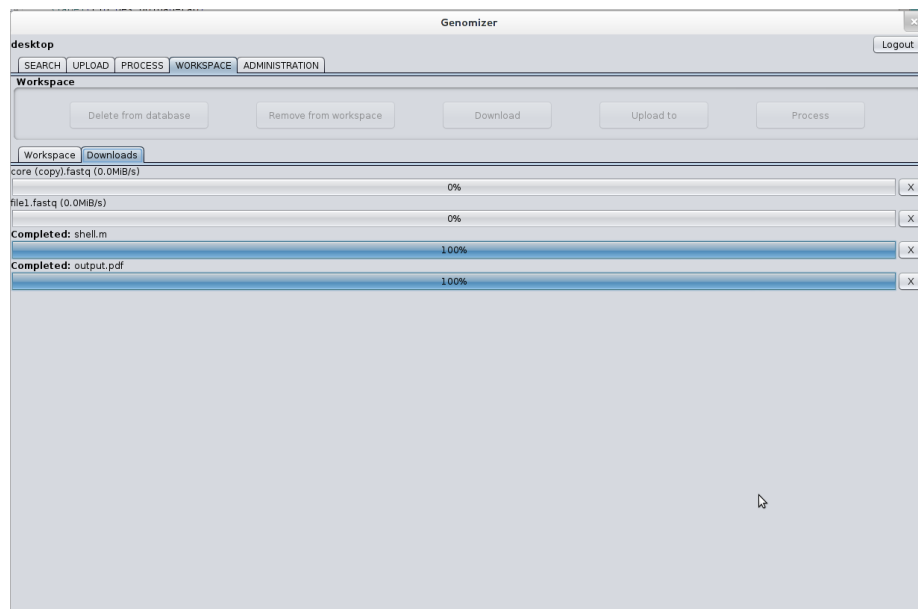


Figure 4.13: The downloads tab of the workspace

4.1.6 Administration

The system administration tools for the desktop client is available under the Administration tab. There are two different tools: Annotation and Genome files. The annotation tab is the first sub tab in the Administration tab. Annotations are used for specifying properties of uploaded data. For example, if new data from an experiment done with rat tissue is uploaded, the data should have an annotation called "species" with the value "rat". The Annotations sub tab in the Administration tab gives the user the tools to create, edit and remove annotations and annotation values.

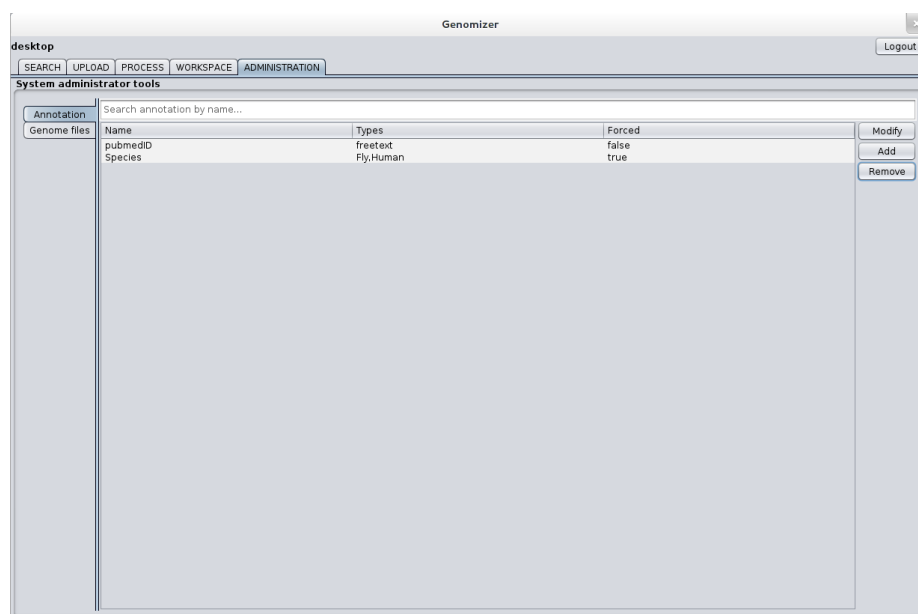


Figure 4.14: The annotation view

In the annotations tab, when a user selects the "Add" button in the sidepanel a new popup window appears. It is possible to write the name of the new annotation and name of new values in this popup, as well as check a "forced" annotation box. The "forced" value determines if the annotation will have to be present in all future file uploads. See Figure 4.15

If the user wants to have free text as a value, for example if the annotation is pubmedID, the value of that annotation will not be able to be chosen from a drop-down menu, since the number available values is enormous. The user might then want to use a freetext annotation, which allows them to type any value they want. To create a freetext annotation the user clicks on the freetext tab on the "add" popup.

To remove an annotation, the user selects an annotation from the table in the center of the view, and clicks on the remove button on the right side. The user then has to confirm this deletion. After that the annotation is completely

Annotation name: Sex

Not adding any values will result in a Yes/No/Unknown drop down annotation.

Add Values

Value: Unknown +

Value: Male -

Value: Female -

Forced Annotation: Yes

Figure 4.15: The add annotation popup

removed and cannot be brought back to life, see Figure 4.16. Some annotations cannot be removed for security reasons, 'Species' is such an annotation. Trying to remove it will generate an error message.

Genomizer

Genome researcher 1 (Desktop User) Logout

SEARCH | UPLOAD | PROCESS | WORKSPACE | SYSTEM ADMINISTRATION

System administrator tools

Name	Types	Forced
Development Stage	freetext	true
ExpID	freetext	false
Sex	Female, Male, Unknown, Does not matter	true
Species	Human, Fly, Rat	true
Tissue	freetext	true

Modify Add Remove

Select an Option

? Are you sure you want to delete the Tissue annotation?

Cancel No Yes

Figure 4.16: The remove annotation popup.

The genome files tab shown in Figure 4.17 contains a table with information about which genome release versions are stored on the server. If the user clicks on one of the entries, a smaller frame is displayed at the bottom of the table showing which files are included in the selected genome release. To the right of the genome release tab are the tools for adding new genome releases. The user can name the new genome release in the text field and is then able to upload the files associated with that genome release. When the desired files are selected, progress bars representing the upload of those files appear at the bottom of the "Add Genome Release" frame. When the user presses "Upload", the upload of the selected files will commence and the user can follow the upload progress from the progress bars. After the upload is finished, the user will be notified of its success or failure with a message dialog. Genome releases can also be removed by selecting the release version from the table and pressing the "Remove genome release" button which appears at the bottom of the table when a release version is selected. This will remove the genome release and all associated files.

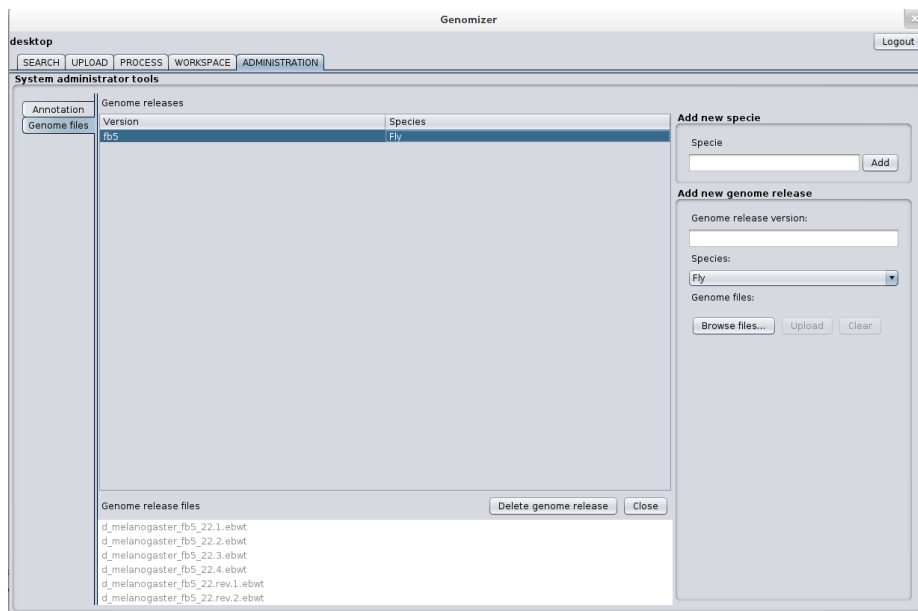


Figure 4.17: The genome release view.

If the user wants to add a new species to add or remove genome releases for, this can be done in the top right corner of the genome release tab. The user simply writes the name of the new species and presses the "add" button and the species will be added to the "Species" annotation.

4.2 Web application

To access the web application, navigate to a domain and directory that publicly serves the web page. An example of this could be: `scratchy.cs.umu.se:8000/app/`. All functionality of the web application is (or rather should be) fairly self-

explanatory and intuitive. A short description and explanation will be given for each component that have been implemented so far.

4.2.1 Using the interface

This section will describe how to use the interface and how to interact with it.

4.2.1.1 Start view

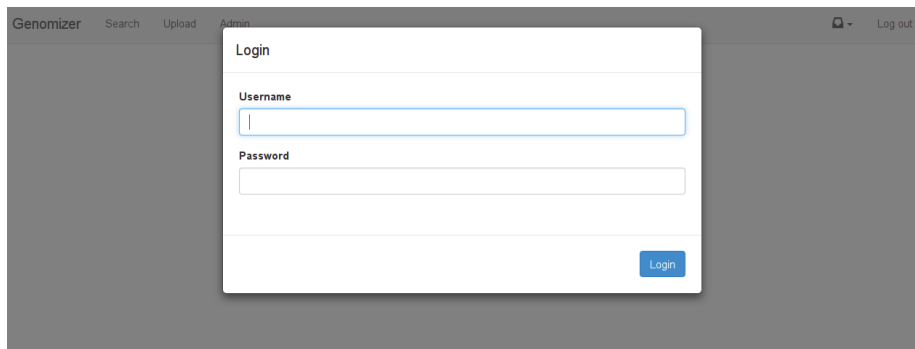


Figure 4.18: The login modal.

When first entering the webpage the login modal in Figure 4.18 is shown and the user will have to enter their username and password to gain access to the application.

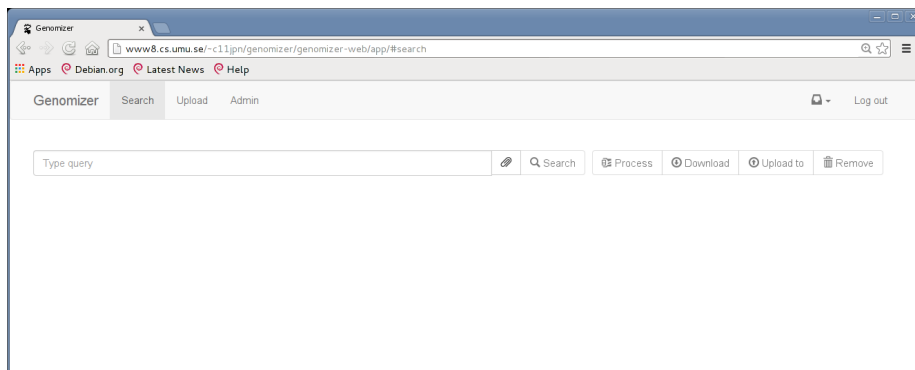


Figure 4.19: The welcome screen of the webpage.

When the user has logged in, the user is taken to the search page as shown in Figure 4.19.

The navigation bar at the top has four buttons to the left and two buttons to the right with the following functionality:

- Clicking the “Genomizer” logo takes the user right back to the start view.
- The “Search” button will bring up the search view where the user can enter search strings to be sent to the server, and view search results.
- The “Upload” button will bring up the upload view where the user can select files to be uploaded and input annotation to a new experiment.
- The “Admin” button will bring up the admin view where the user can handle genome releases and annotations.
- The inbox icon on the left side opens a process status dropdown.
- The "Log out" button will log out the user.

This navigation bar is persistent through all sub pages and can easily be accessed.

4.2.1.2 Search view

Below the navigation bar a “search-and-functionality” bar is visible, there is a search field and there are six buttons, Query-builder, Search, Download, Upload to and Process. However, when first entering the page some buttons will be disabled. When you enter something in the search field the search button will become enabled and clickable. To search the user can either write a pubmed style query (for example: Exp1[ExpID]) or use the query builder, by clicking the paperclip icon.

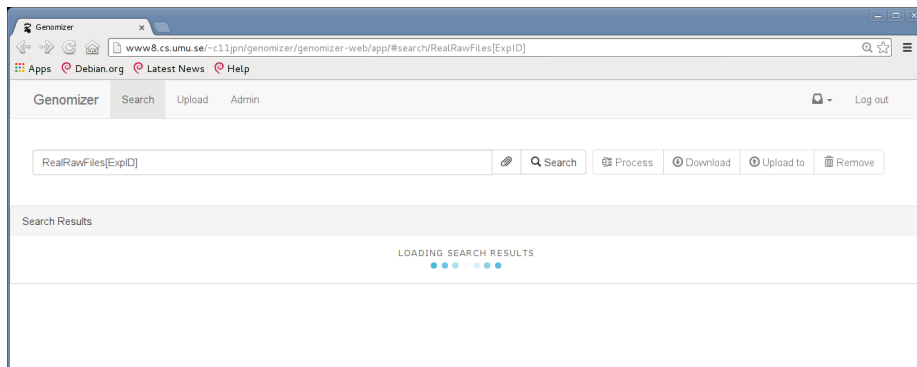


Figure 4.20: Will be shown while searching for data in the database before any results are found.

After having typed a query and pressed search, the search results will load displaying the loading spinner as can be seen in figure Figure 4.20.

The view shown in Figure 4.21 contains two major elements; a “search-and-functionality” bar and a list of search results retrieved after searching for ‘Exp1[ExpID]’. The buttons next to the search bar do the following:

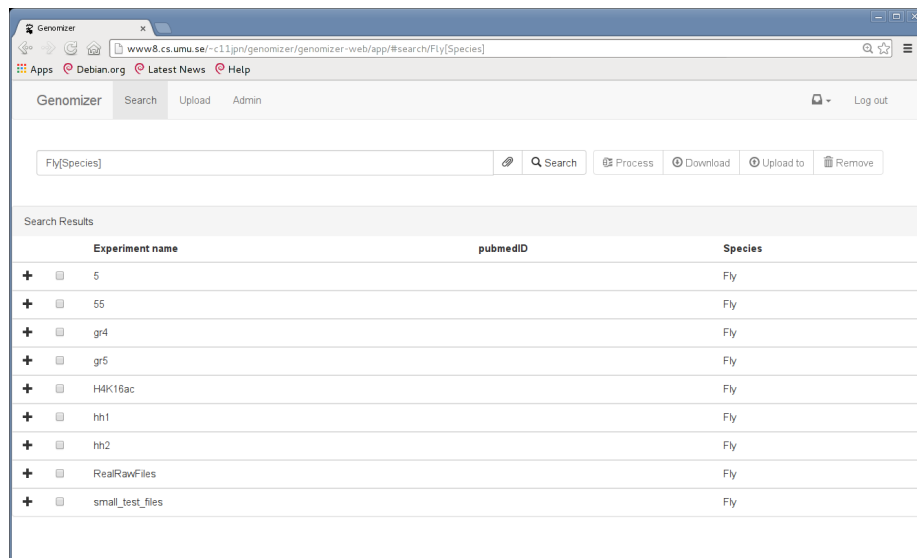


Figure 4.21: The search tab after a search for 'Fly[Species]'.

- The paperclip brings up a Query builder.
- "Search" searches for the query in the search bar.
- "Process" brings up a new window in front of the search view with options for file processing. This feature is demonstrated further in Figure 4.25.
- "Download" downloads the selected files.
- "Upload to" opens the upload view with the selected experiments selected where the user can upload new files to an already existing experiment.
- "Remove" opens a new view where the files which are going to be deleted are presented along with a confirmation dialog that the user really wants to delete those files and experiments.

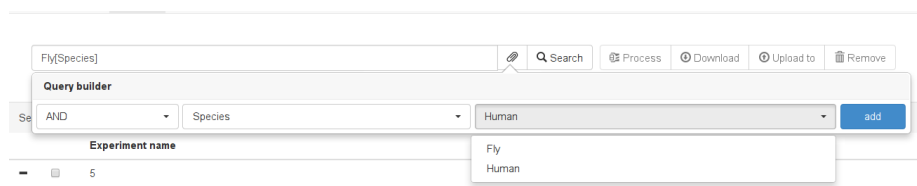


Figure 4.22: The query builder.

The search query builder as shown in Figure 4.22 can be used to easily build pubmed-styled search queries. Just select a value in the three fields and press add. The correct pubmed-styled query will be shown in the search field and the three query fields will be reset so the user can add more things to search for in their query.

Below the search bar in Figure 4.21 is the “search results” list. This list contains all experiments returned from a search. Every experiment can be expanded to show the file types it contains. Each file type can be expanded to show all files of that type in the experiment. All files and experiments has a check box next to it that is used to select what to process, download, remove or upload to.

Search Results					
Experiment name	pubmedID	Species			
+ <input type="checkbox"/> 5		Fly			
+ <input type="checkbox"/> 55		Fly			
+ <input type="checkbox"/> gr4		Fly			
+ <input type="checkbox"/> gr5		Fly			
- <input type="checkbox"/> H4K16ac		Fly			
+ Raw Files					
- Profile Files					
Filename	Genome release	Metadata	Uploaded date	Author	
<input type="checkbox"/> H4K16ac_male_wt_reads_sorted_v1.gff	fb5	-a -m 1 -best -p 10 -v 2 -q -S, fb5, y,	May 28, 2014	Genomiz	
<input checked="" type="checkbox"/> input_male_wt_reads_sorted_v1.gff	fb5	-a -m 1 -best -p 10 -v 2 -q -S, fb5, y,	May 28, 2014	Genomiz	
<input checked="" type="checkbox"/> H4K16ac_male_wt_reads_sorted_v1_v1_median_winSiz-10_minProbe-5.sgr	fb5	-a -m 1 -best -p 10 -v 2 -q -S, fb5, y, y, 10 1 5 0 0, y 10, . .	May 28, 2014	Genomiz	
<input type="checkbox"/> input_male_wt_reads_sorted_v1_v1_median_winSiz-10_minProbe-5.sgr	fb5	-a -m 1 -best -p 10 -v 2 -q -S, fb5, y, y, 10 1 5 0 0, y 10, . .	May 28, 2014	Genomiz	
+ Region Files					
+ <input type="checkbox"/> hh1		Fly			
+ <input type="checkbox"/> hh2		Fly			
- <input type="checkbox"/> RealRawFiles		Fly			
- Raw Files					
Filename	Genome release	Metadata	Uploaded date	Author	
<input checked="" type="checkbox"/> MOF_male_wt_reads.fastq			May 22, 2014	Ruanidh	
<input type="checkbox"/> input_male_wt_reads.fastq			May 26, 2014	Ruanidh	
+ Profile Files					
+ Region Files					

Figure 4.23: The search results table zoomed in, displaying a raw file’s information after having expanded an experiment.

If a search is successful, you will be met with a table of results. This table has a header displaying the annotation types. Below that, all the experiments returned from a search and their corresponding annotation values, as can be seen in Figure 4.23.

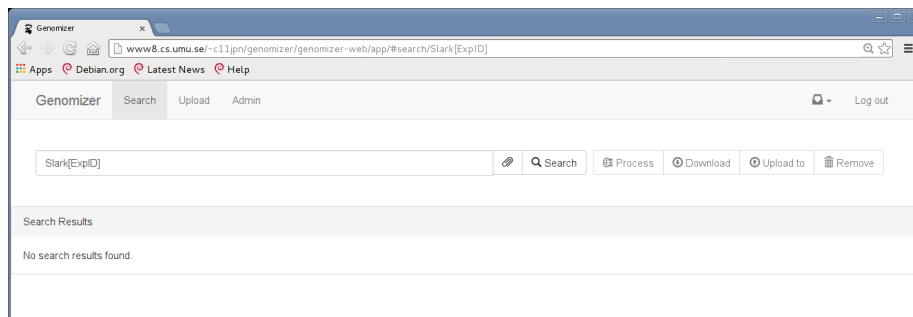


Figure 4.24: The search view if no data was found given the search query entered by the user.

If the search is unsuccessful, the Search Results table will be empty stating “No search results found” as can be seen in Figure 4.24.

4.2.1.3 The processing modal

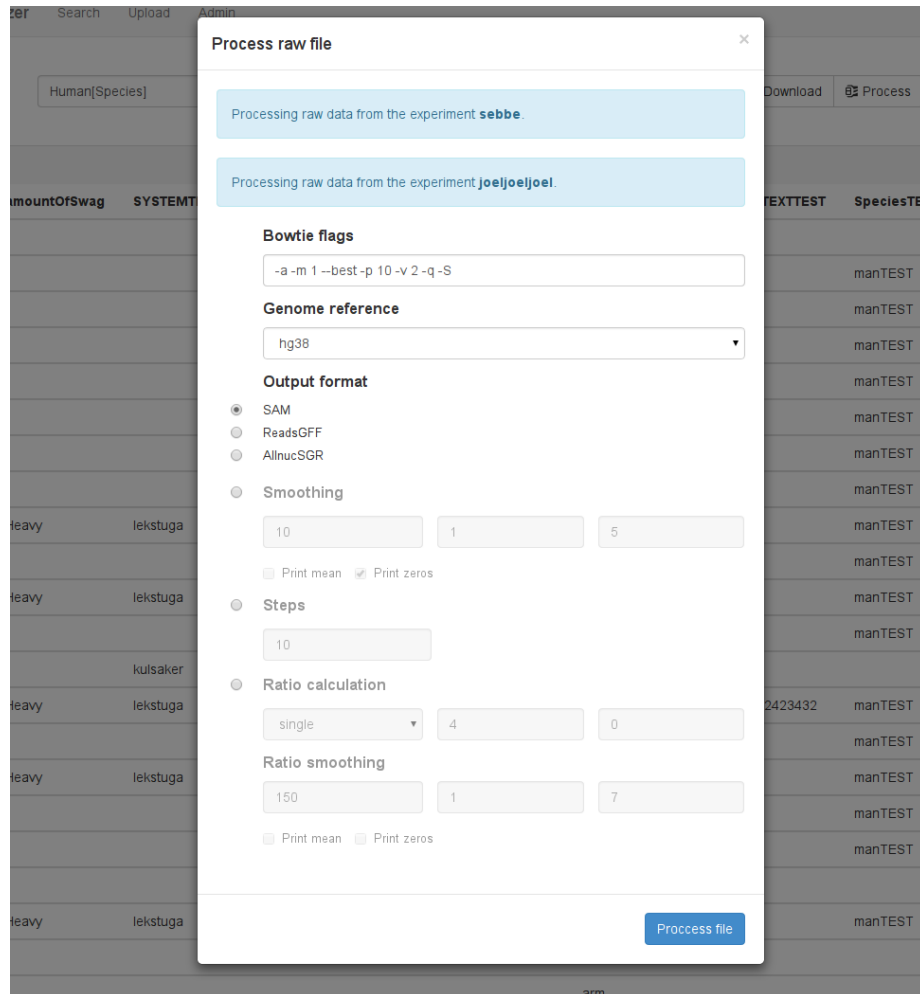


Figure 4.25: The processing modal.

When the user has selected some files that are going to be processed the user will be presented with the view from Figure 4.25. The user can here choose which level of processing should be done on the raw files. By clicking the radio buttons on the left side that much processing will be done on the raw files. All the steps above the selected will also be executed since they are needed to reach that level of processing. At the top of the modal the experiments currently going through processing are presented.

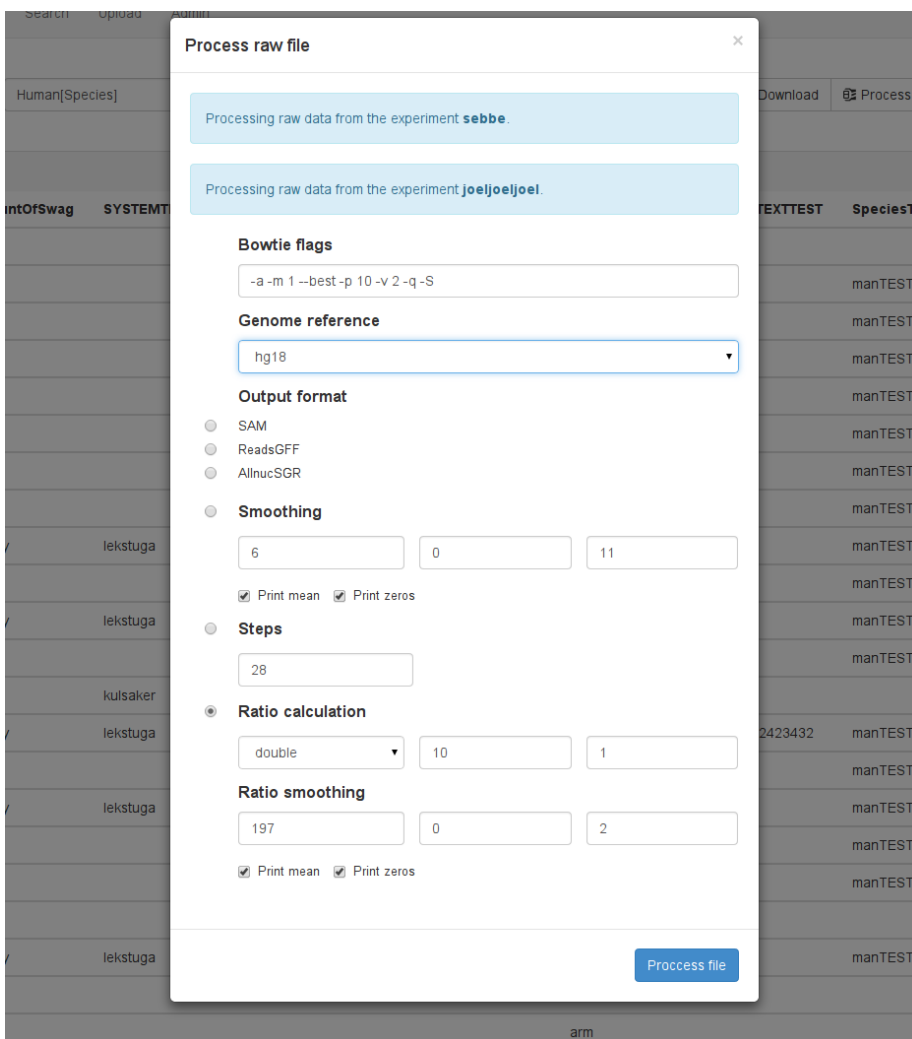


Figure 4.26: The process modal with selected parameters.

When the user has decided the parameters as shown in Figure 4.26 and wants to start the processing the process button in the bottom right should be pressed.

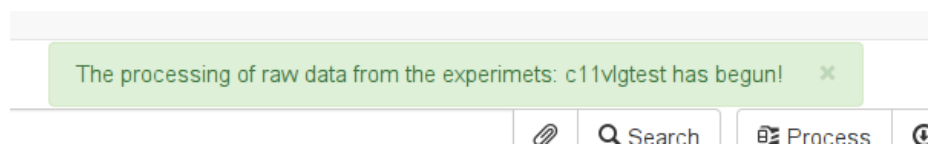


Figure 4.27: Success message.

When results are received from the server and they were all successful the processing modal will disappear and a success message indicating that the processing is starting will be displayed to the user like in Figure 4.27. If some of the

Bowtie flags

-a -m 1 -best -10 -3 -S

The processing of c11vlgtest failed. please try again. X

Genome reference

12345 ▼

Figure 4.28: Fail message.

files that was going to be processed did for some reason fail the user will learn this by a warning message that tells the user which experiment did not start processing and which did as shown in Figure 4.28. The ones which started to process will be removed from the modal and the ones that did not start to process will remain. The user can now choose other parameters or do something else to make it work and try to submit a processing request again.

4.2.1.4 The remove modal

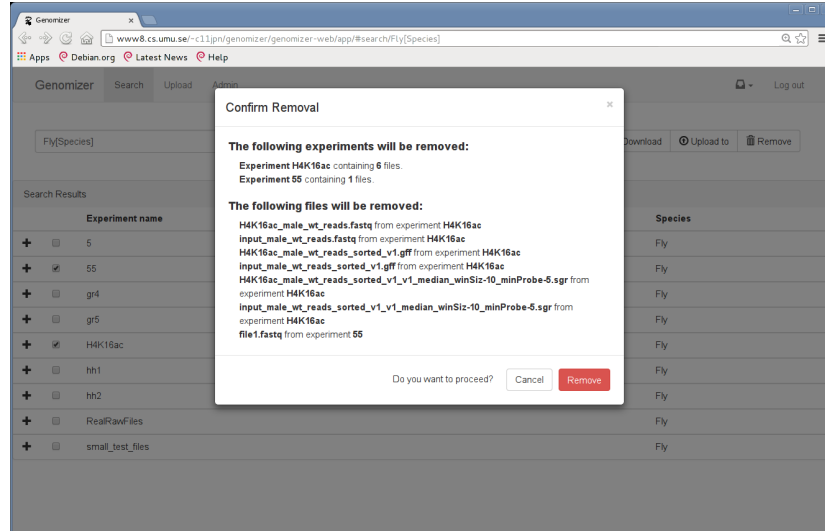


Figure 4.29: The remove modal.

When the remove button is pressed the modal in Figure 4.29 is shown displaying which files and experiments will be removed when the remove button is pressed.

4.2.1.5 The process status dropdown

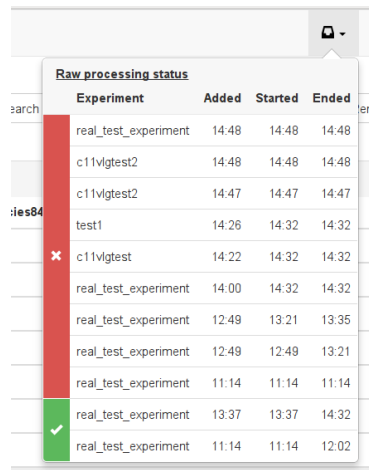


Figure 4.30: The process status dropdown.

When pressing the inbox icon a dropdown is shown as in Figure 4.30 displaying processing status of experiments being processed. There are four different status a processing can have: Waiting, Running, Complete and Failed. These are

grouped together with a yellow color for waiting, blue for running, green for complete and red for failed.

4.2.1.6 The process status dropdown

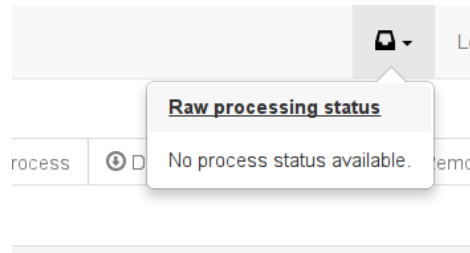


Figure 4.31: The process status dropdown with no status available.

If there are no processes status available the user will see the text as shown in Figure 4.31

4.2.1.7 The upload view

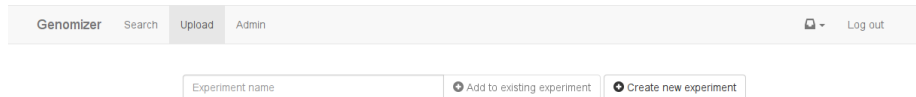


Figure 4.32: The upload view.

When the user clicks the upload tab in the navigation bar, the view in Figure 4.32 will appear. The user has the option to create a new and fresh experiment or to load an existing experiment by entering its experiment name.

After clicking the "Create new experiment" button, the view in Figure 4.33 will appear. Here the user can input the annotations for the experiment through either free text fields or drop-down lists. If a free text field has a red border around it, that annotation is required and the experiment can not be uploaded before all required fields have been filled in and at least one file has been added. The user can create more empty experiments by clicking the "Create new experiment" button and a new, empty, experiment will be placed below the first experiment.

The user can fill in annotations in one experiment that should be the same for several experiments. By clicking the "Clone experiment" button, a copy of the

Figure 4.33: Creating a new experiment.

experiment's annotations will be appended as a new experiment. The user can change the annotations that should be different from the cloned experiment.

Up in the corner of the experiment is a button that can remove the unwanted experiment from the view.

To add files to the experiments the user can browse for local files and upload them by clicking the "Select files to upload" button. The user will only see file types that have to do with experiments but have the ability to search for all file types. There is also a way of adding files to the experiment by dragging them from a file browser and dropping it onto the experiment "drag and drop".

An experiment can only contain two RAW-files and if the user tries to upload more a message with this information will appear and the experiment cannot be uploaded before the extra RAW-file/s is removed.

To add files to a existing experiment the user types the name of the experiment in the field next to the "Upload to existing experiment" and clicks the button. If the experiment exists on the server it will appear in the experiment view the same way that a new experiment is shown. The annotations of an existing experiment cannot be changed from this view and if there are files already in this experiment they cannot be manipulated. Adding new files to existing experiments works the same way as to a new experiment.

When the user selects files, they will appear below the annotations as in Figure Figure 4.34. The file name is displayed in a text field on the left side of the file view. Next to the file name is a box that shows the size of the selected file in a human friendly format (B, KiB, MiB, GiB, TiB). On the right side there is an option to select what type of file is being uploaded and an option to remove the file from the experiment. If the file type is either profile or region, there is an option to select what genome release the file is mapped to. The file type option will automatically be filled in with a guessed value depending on the file ending as follows: fastq files are considered raw and all other formats (sgr, wig, gff) are interpreted as profile.

When the user is done selecting files, filling in annotations and clicks the "Upload

The screenshot shows the 'Experiment 322' upload interface. At the top, there are navigation tabs: 'Genomizer', 'Search', 'Upload', and 'Admin'. A 'Log out' link is in the top right. The main form is titled 'Experiment 322' and contains the following fields:

- experiment name:** Experiment 322
- pubmedID:** 47
- Species:** Fly (dropdown menu)

Below these are six rows of file upload information:

File Name	Size	fb5	Selection	Action
BG3_EZ_standaone_2006.wig	1.30 KiB	fb5	Profile	✕
HP1a_modENCODE_2668-repset.4621227.smoothedM.wig	112.72 MiB	fb5	Region	✕
MOF_male_wt_reads.fastq	1.15 GiB		Raw	✕
MOF_male_wt_reads_sample.fastq	14.11 KiB		Raw	✕
Ratio_WT_CBP_NoCutOff_sorted.sgr	39.72 MiB	fb5	Region	✕
Rb_CBP_reg.gff	273.95 KiB	fb5	Profile	✕

At the bottom of the form, there is a 'Select files to upload' button and the text 'or drop files here.'. Below the form are two buttons: 'Clone Experiment' and 'Upload experiment'. At the very bottom, there is a row of buttons: 'Experiment name', 'Add to existing experiment', 'Create new experiment', and 'Upload all experiments'.

Figure 4.34: Files selected for upload.

experiment” button the experiment view will be minimized showing only the name of the experiment and the progress bar of the files being uploaded. When the progress bar is done it turns green and now the experiment with all the files have been uploaded to the server. The user also has a way of uploading several experiments at the same time by clicking "Upload all experiments".

4.2.1.8 System administration view

This part of the web application is only accessible if the user have administrator-rights. It is integrated with the rest of the web UI and accessible through an admin-tab. The administrator can through this site see all

annotations, add new annotations and edit existing ones.

The start page of this section has a "Create New Annotations" button, a list of existing annotations in the database and an edit button per existing annotation. The view looks like in Figure 4.35.

For each annotation in the annotations list, an Edit button is available. When pressed, it will take you to a page in which you can edit the selected annotation to change its name and what values the drop-down list will have if it's not a free text field (See Figure 4.36).

In the edit page the admin can see the attributes of the chosen annotation and is able to delete the chosen annotation or change it's information. The delete Annotation button will delete the whole annotation and for that reason two

Name	Values	Forced	
Cell line 2	No,Unknown,maybe,known	false	Edit
89029	Yes,No,Unknown,asdf	true	Edit
Is forced	yes	true	Edit
Gender	Yes please	false	Edit
john	Yes,No,Unknown	false	Edit
Teststar	Typ,Yes,Unknown,No	false	Edit
Glass	Vaniij,Choklad,Jordgubb,Hallon,Senap	false	Edit
Species	Fly,Superhero,Human	true	Edit
deskdstar	test	false	Edit
Gender2	daniel,John,wolf	false	Edit

Figure 4.35: The start page for the administrator in the web client

Figure 4.36: The edit annotation view

popup windows will appear to make sure that the administrator is sure of the action.

The administrator can change the list of annotation values and the site will automatically check whether something is added, removed or both and sends a request to change the annotation values to the server when the Update Annotation button is clicked.

If the admin clicks on Create new annotation from the admin start page, another view will open with the following structure:

- Annotation Name
Admin can enter a name for the annotation
- Annotation Types
Yes/No/Unknown - this will create a drop-down list with those three options.

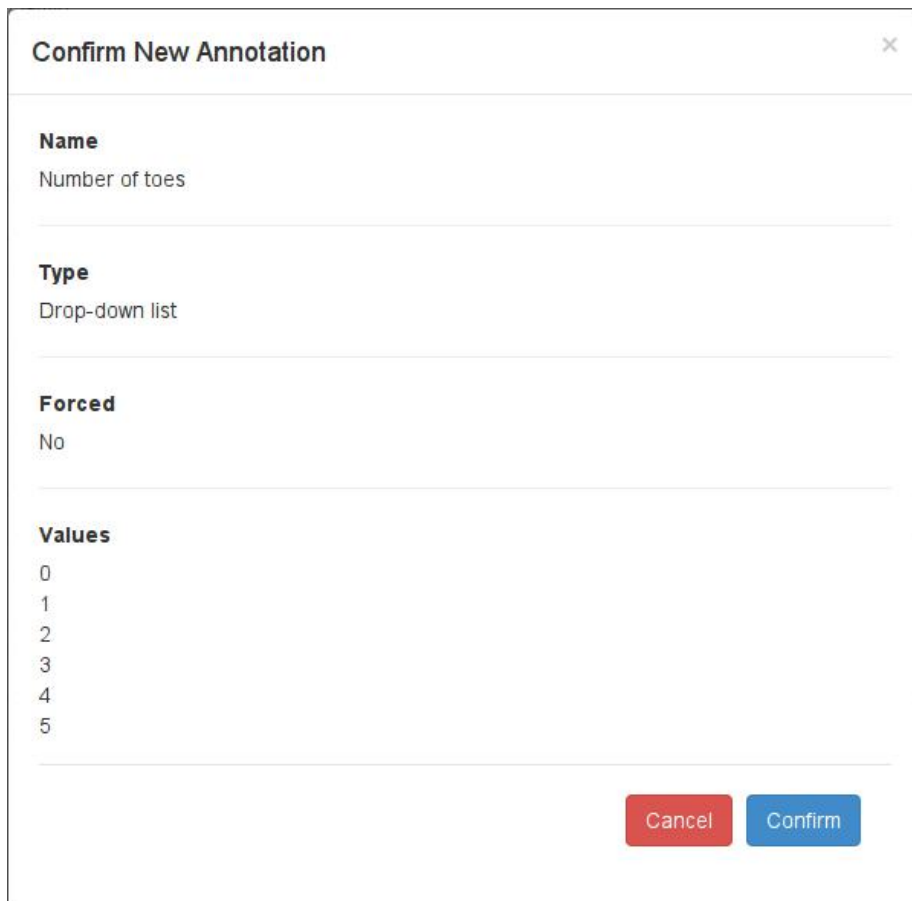
free text - will create an annotation that the users will be able to enter anything.

Drop-down list - will enable a fourth field enabling the admin to enter which items that this list will contain.

- **Forced Annotation**

Admin can choose if the new annotation should be forced for users to enter.

A Create Annotation will, if all necessary information has been entered, result in a popup (see Figure 4.37) showing the resulting annotation and if confirmed, the annotation is added to the database. If canceled the administrator can keep making changes or go back to exit this view. If not all values is entered the admin will be alerted of the mistake and nothing will be created.



Confirm New Annotation [X]

Name
Number of toes

Type
Drop-down list

Forced
No

Values
0
1
2
3
4
5

Cancel Confirm

Figure 4.37: The confirm annotation popup

The example in Figure 4.37 will result in a drop-down annotation with the name Number of toes and possible values: 0, 1, 2, 3, 4, 5 with 0 as default and is not forced.

Genomizer Search Upload Admin

Annotations

Genome-releases

Create new annotation

Annotation Name

Annotation Type

Yes/No

Forced Annotation

Yes

Items in drop-down list: (Example: Male,Female,Unknown)

Create Annotation Back

Figure 4.38: The view for administrators where new annotations can be created

A back button which takes the user back to the annotations start page is also available in this view. In Figure 4.38 the create annotation view can be seen.

The "Genome-releases" link on the sidebar takes the administrator to a page where it's possible to add and remove genome releases to and from the server (see Figure 4.39).

Genomizer Search Upload Admin Log out

Annotations

Genome-releases

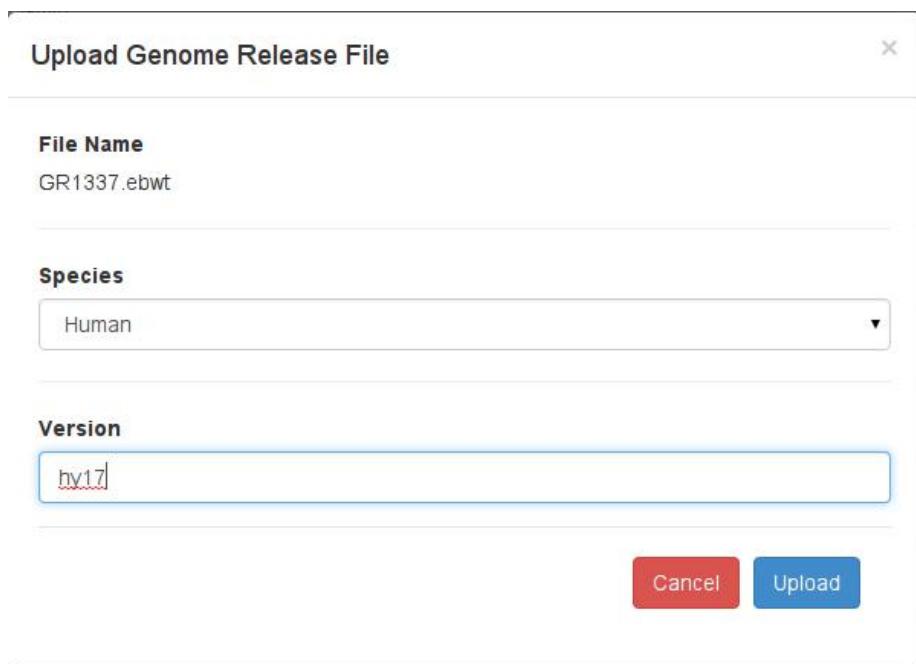
Select files to upload

Species	Version	Filenames	Delete
Fly	test	coolloop.sh, ...	Delete
Fly	fb5	d_melanogaster_fb5_22.1.cbwt, ...	Delete
Fly	BABA	Unicorn5.fasta, ...	Delete
Fly	da	asd.png, ...	Delete
Fly	12345	Ajjevalad3.docx, ...	Delete
Fly	BABA	stad.jpg, ...	Delete
Fly	test	mozilla.pdf, ...	Delete
Human	hg38 sorted	hg38 sorted fa.out, ...	Delete

Figure 4.39: The genome-release view

The button "Select files to upload" opens the native file explorer where the user can select one or multiple files and click on "OK". This will open a popup-window, seen in Figure 4.40, showing what files that were chosen and asks for species and genome version before uploading.

When the upload begins the popup closes and a progress-bar appears showing the progress, showing "Upload completed" when done. The user can at this stage move between pages without disturbing the upload but should not close or refresh the web browser.



The image shows a web browser popup window titled "Upload Genome Release File" with a close button (X) in the top right corner. The form contains three sections: "File Name" with the text "GR1337.ebwt" and a horizontal line below it; "Species" with a dropdown menu showing "Human" and a downward arrow; and "Version" with a text input field containing "hv1.7". At the bottom right of the form are two buttons: a red "Cancel" button and a blue "Upload" button.

Figure 4.40: Popup for uploading genome releases

Every genome release in the table can be deleted by clicking on the "delete" button next to the release. This will prompt a small popup asking for user confirmation and if given a positive response, deletes the genome release from the server and updates the view.

If any genome release is used by an experiment already a error will appear telling the user exactly that.

4.2.2 Setting up the application

To setup the application, move the content of the folder called app in genomizer-web to the desired location from where the application should be run. To run the webpage open a web browser and enter the url to the folder which contains the `index.html` file (where the content of app was placed). Ex. given that the genomizer-web folder is placed in my home folder and i want to put the webpage in a folder called `public_html` which is also in my home folder. In linux i do the following steps.

1. Navigate to the app folder: `"cd /genomizer-web/app/"`
2. Move the contents of app to the folder called `public_html`: `"mv * /public_html/"`
3. Given that the url to the `public_html` folder is: `"www8.cs.umu.se/ c11abc/"`
4. To run the application start a web browser and type `"www8.cs.umu.se/ c11abc/"`

This will open the webpage in the browser.

4.3 Android application

In this section instructions for the usage of the *Genomizer* Android application is presented. In subsection 4.3.1 there is a description on how to start the application and subsection 4.3.3 gives instructions on how to search for experiments.

4.3.1 Start the Application and Login

The user needs to login in order to start working with the *Genomizer* app. The user name and password is inserted in the corresponding boxes and the clicking the *Sign in* button initiates the main application. If you dont have a user name or password, the system administrator should be contacted to help with the creation of an account.

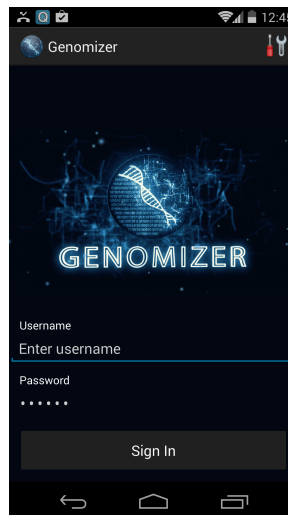


Figure 4.41: Login View

In Figure 4.41, the tool button in the upper right corner leads to the *Settings View*, described in subsection 4.3.2 below.

4.3.2 Settings

The *Settings* view acts to enable the user to choose which server to connect to when using the Genomizer application. As of the current release of the application, in the Settings View, the user is able to:

1. Select one of previously used server URLs

2. Add a server URL
3. Remove a server URL
4. Edit an existing server URL

The left most image in Figure 4.42 show three buttons in the top right corner of the view. These buttons are used to access the functionalities listed above. The button with a green plus sign enables the user to add a new server URL, as illustrated in the image in the middle of Figure 4.42. The button in the middle with a paint brush icon will on selection show the server URL edit view, as illustrated in the right most image in Figure 4.42. And the left most button with a red cross icon will upon selection enable the user to remove the currently selected server URL from the drop down menu containing all saved server URLs.

Any selection, removal, edit or addition of server URLs are stored locally on the device and are loaded upon subsequent application launches.

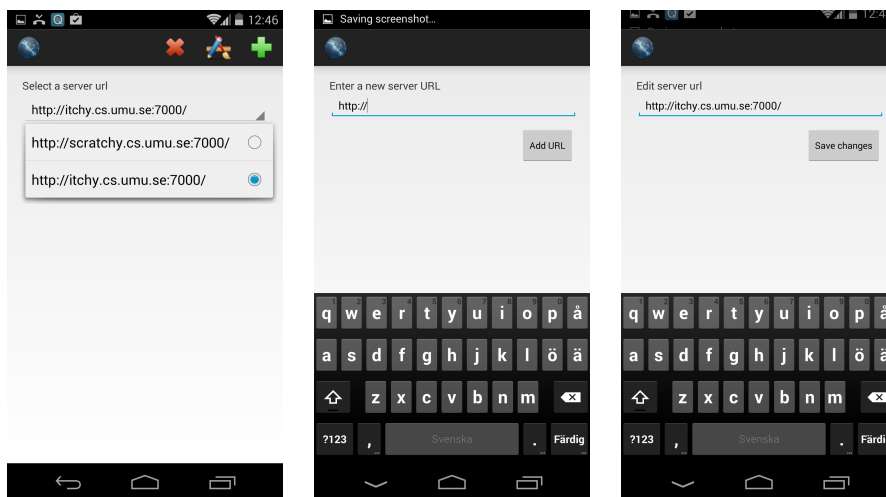


Figure 4.42: Settings View

4.3.3 Searching for files

When entering the Search View, as illustrated in Figure 4.43 all annotations are automatically downloaded from the server and displayed as a list. Each annotation consists of an annotation-identifier, a dropdown table/text-input field where the user may specify desired value, and a checkbox. When putting a check-mark in the checkbox, it means that this particular annotation type should be used when searching for files in the database. The search is initiated by pressing *Search* at the bottom of the view.

Once the user has been logged in to the system, three buttons will always be visible in the top right corner of each view:

1. Search button
2. Selected Files button
3. Process Status button

Clicking these buttons switches the context of the application and allow the user to quickly navigate between different functionalities.

The search view also contain a button visible in the top right corner, used to activate the advanced search mode described in the following subsection 4.3.4.

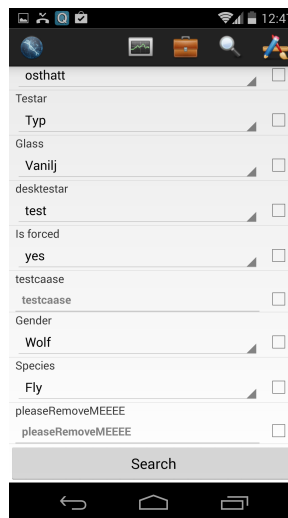


Figure 4.43: The Search View

4.3.4 Pubmed Search

The Pubmed Search view provide the means of free-text search using Pubmed-Style queries as seen in Figure 4.44. This view include a text-input field together with two buttons. The text field is populated with the annotations that the user may have selected within the regular Sarch View. However, if no annotations have been previously selected in the Search View, the user must input all annotations manually. The annotations selected in the Search View are associated with logical connectives. These logical connectives, as well as annotation values, can be manually modified by the user. The supported logical connectives are:

1. AND
2. NOT
3. OR

The user may also choose to provide perentheses to device more specific searches.

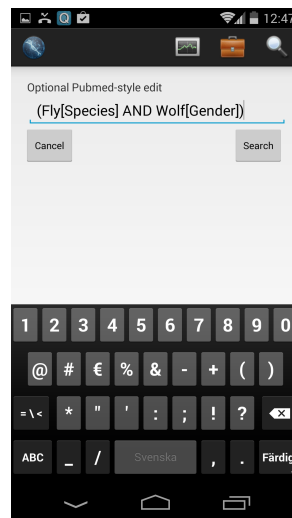


Figure 4.44: The Pubmed Search View

4.3.5 Search Results

When searching the user will be redirected to the search results view that displays a list of available experiments matching the search annotations. Every experiment is listed showing the experiment name. To receive more information about data files that are available for each experiment, click on an experiment in the list. By clicking an entry you will be taken to a new view displaying all available data files for that experiment, presented in the Experiment List View.

Clicking on the cogwheel button in the top right corner of the view enables the user to modify which annotations are presented within the Search Results View, and is described further in the following subsection 4.3.6.

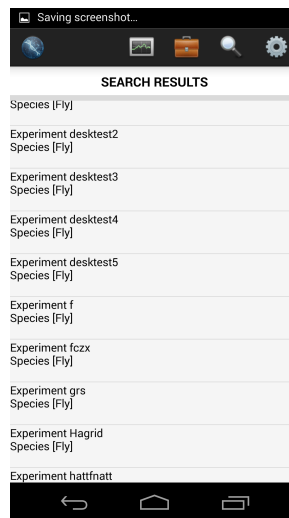


Figure 4.45: The Search Results View

4.3.6 Search Settings View

The Search Settings View display settings for the files presented to the user after a search is done, as illustrated in Figure 4.46 below. The Search Settings View contains all different annotations the user will be able to display about the experiments presented in the Search Results View. The user are able to select annotations by marking the checkbox next to the annotation name and then clicking the Save settings button to save changes. If the user has no special requests it is also possible to use default settings, which will display (experiment-Id, created by, pubmed and type) annotations for the files displayed.

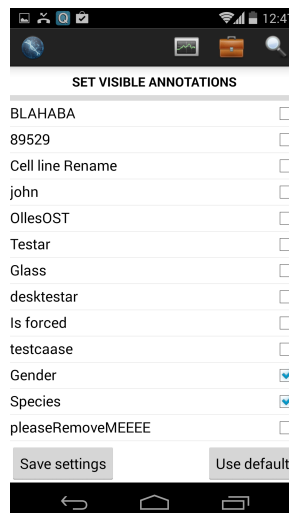


Figure 4.46: Search Settings View

4.3.7 Experiment File View

The Experiment File View is used to present the user with all files associated with an experiment. This includes all raw, profile and region files derived from the experiment. A user may select and add an arbitrary number of files to the Selected Files view, which is described in subsection 4.3.8, by marking the checkbox of the desired files, as done in Figure 4.47, and pressing *Add to selection*.

Clicking on a file presented within this view creates a popup containing all different annotations for the selected file, as illustrated in the right most image in Figure 4.47.

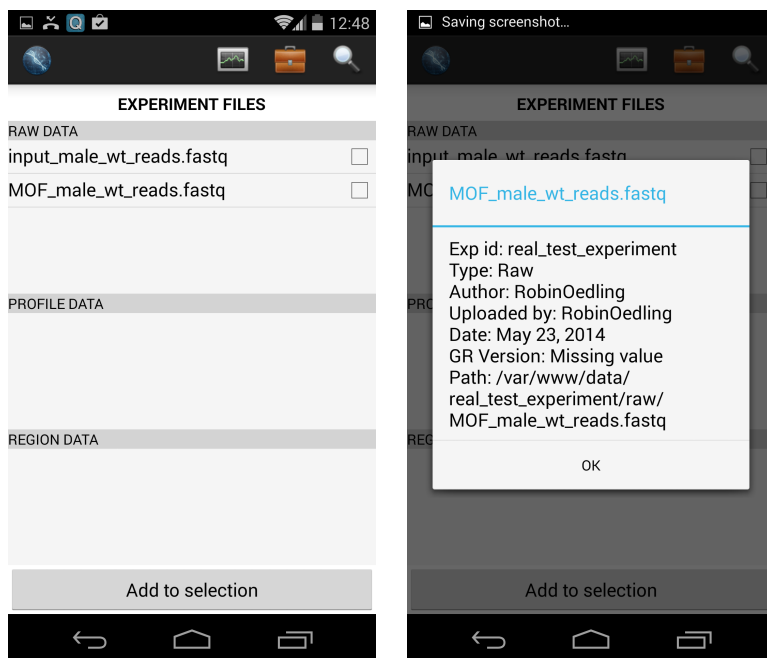


Figure 4.47: The Experiment File View

4.3.8 Selected Files

Once the user has signed in to the server, the user is presented with a *Selected Files* view, as illustrated in Figure 4.48. This is the main part of the application where all work and conversions are done to files, when the user has searched and found files that are interesting for further use, it can be moved to the selected files area. The page contains three different tabs that the user may use to show different type of files saved in the selected files workarea. All files stored in this page are only saved during the current session and is meant to be used as a temporary grouping area for files.

- *Raw*, will display all the Raw files that the user has chosen to save to the temporary work area. The files here can be marked and used for converting to profile data.
- *Profile*, will display the Profile files that the user has chosen to move to the selected files area. No conversions or other work can be done at this stage to profile files.
- *Region*, this page will display all the region files the user has selected to move to the selected files area for further work. No conversions or other work can in this stage be done to region files.

Similar to the Experiment View, clicking on a file will present the user with a popup containing the annotations for that file.

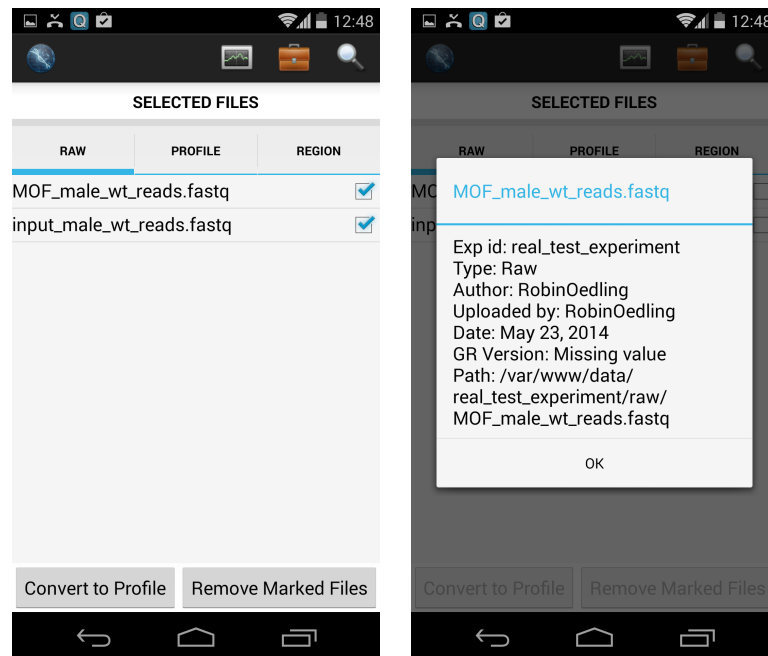


Figure 4.48: Selected Files View

4.3.9 Converting Files

When the user has chosen a file (or several files) for conversion, the user will be presented with the Conversion View as seen in Figure 4.49. In this view the user may enter the parameters needed to perform a Raw-to-Profile-file conversion. There are 9 different parameters to be specified in this page for the conversion to be done in a proper way. All parameters do not have to be filled, but they have to be specified in the order that is presented to the user. In order to fill out parameter number 3, both parameter 1 and 2 have to be filled out first.

1. *Bowtie*, is a freetext field where the different parameters for the bowtie program are to be inserted.
2. *Genome Version*, is a dropdown menu where the user is presented with all the different genome versions that can be used for the conversion.
3. *Sam to GFF*, is an on/off option.
4. *GFF to SGR*, is an on/off option.
5. *Smoothing*, free text field for the parameters for smoothing if it is to be used.
6. *Stepsize*, free text field for which stepsize is to be used for the conversion.
7. *Ratio calculation*, on/off field which determines if the ratio calculation is to be used. If checked it will require both next two fields to be filled out.
8. *Ratio*, free text field with the parameters for the ratio, if ratio calculation is wanted.
9. *Smoothing*, free text field for parameters regarding the smoothing for the ratio calculation.

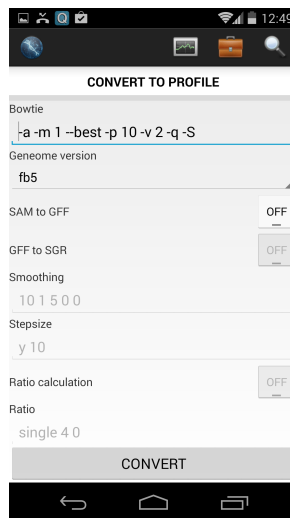


Figure 4.49: The Conversion View

4.3.10 Process View

The process view, as illustrated in Figure 4.50 below, is used to visualize the current workload on the server. The view contains a list of tasks that has been assigned to the server. Each task contains the name of the experiment in which the process is currently operating in, the time when the process was added, the time when the process was started and the time when the process was finished. Each item also contains information about the process current state.

Each process may have one of these four states:

1. Waiting - The task is awaiting processing by the server
2. Started - The task is currently being processed by the server
3. Finished - The task has been completed
4. Crashed - The task was not successfully completed

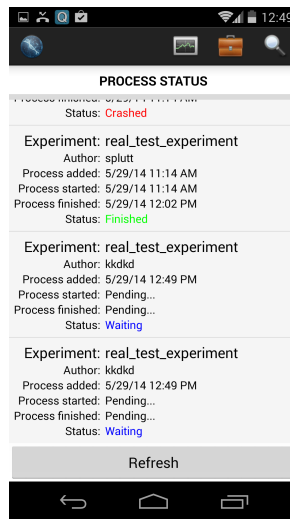


Figure 4.50: The Process View

4.4 iOS application

In order to use the program, import the project from github into Xcode from the following repository: <https://github.com/genomizer/genomizer-iOS.git>

To compile and run the program, press **cmd+R**. A simulator will start and the login screen will be shown as seen in Figure 4.51. A user gets logged in when accepted credentials are entered in the 'username' and 'password' fields and the 'Sign in' button is pressed. If incorrect credentials are entered, a popup message is shown, informing the user that the username or password is incorrect. The user can also change the server to connect to by pressing the symbol in the top rightmost corner. The user will then be presented with a popup window as seen in the rightmost view in Figure 4.51.

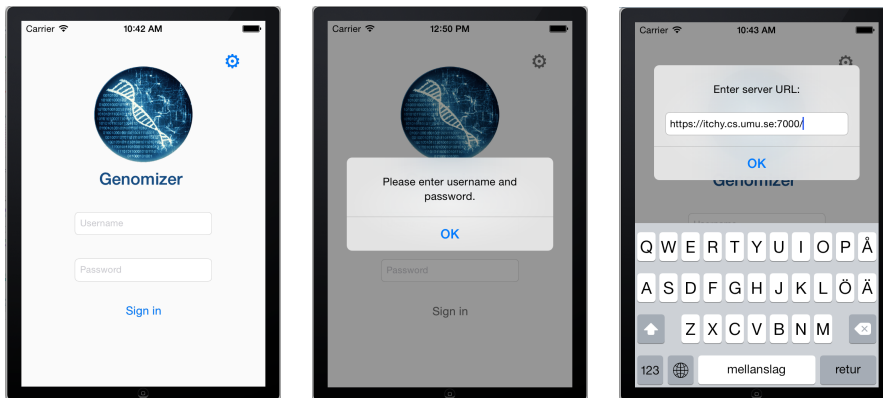


Figure 4.51: The login screen.

After logging in, the user is presented with a search view as seen in the leftmost view in Figure 4.52. The bottom menu bar is used to navigate between the Search-, Selected files-, Processes- and More-view in accordance with Apple's current GUI standards for iOS 7. In the current state, the More-menu only contains a logout button which is used to log out. The Selected files-menu contains a list of files selected by the user, sorted by file type.

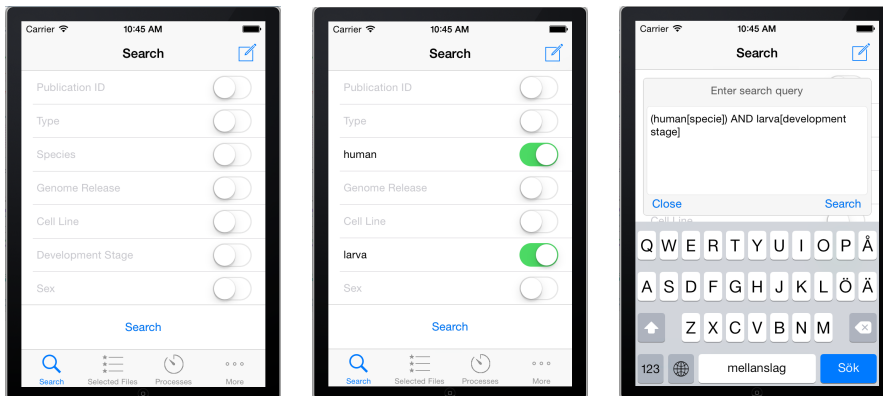


Figure 4.52: The search screen.

In the search view, the user can search the database for results matching any number of search criteria. To be able to modify the search quickly, a toggle button is available in the rightmost edge of each search field which enables or disables each search field. For example, if the user wants to search for experiments where species is 'human', the user clicks on 'Species', chooses human, clicks done and the toggle button will automatically switch to active as seen in the view in the middle in Figure 4.52. In top rightmost corner, there is a button for opening an advanced search view as seen in the rightmost view in Figure 4.52. Here user is supposed to enter a search query in 'pubmed-style-format'. If a user fills in fields in the regular search view and then opens the advanced search view, the fields that currently have values set at the regular

search view will appear as a query in the advanced search view.

When the search button has been pressed, the user is presented with all matching experiments in the Search Results view shown to the left in Figure 4.53. To manage which annotations should be displayed for every experiment, the user can press the edit button in the top rightmost corner and will then be presented with the middle view in Figure 4.53. Here the user can set the visibility for each annotation. When the user then presses the 'back'-button, the annotations chosen will be shown for each experiment as seen in the rightmost picture in Figure 4.53. To see which files are associated with each experiment, the user can click on the experiment in order to get to the Files view shown in the leftmost view in Figure 4.54.

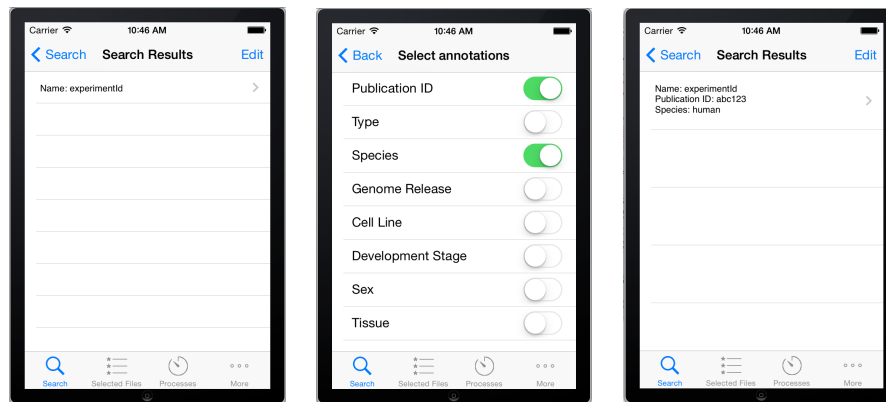


Figure 4.53: The search result screen.

In the Files view, the user can see all files connected to the selected experiment, sorted by type. The user can also get additional information about a file by simply clicking the blue information sign close to the filename. The file information is shown in a popup window as shown in rightmost view in Figure 4.54. If the user selects files and presses the 'Convert files'-button, the user is shown the Select task view as seen in Figure 4.57. More about that later.

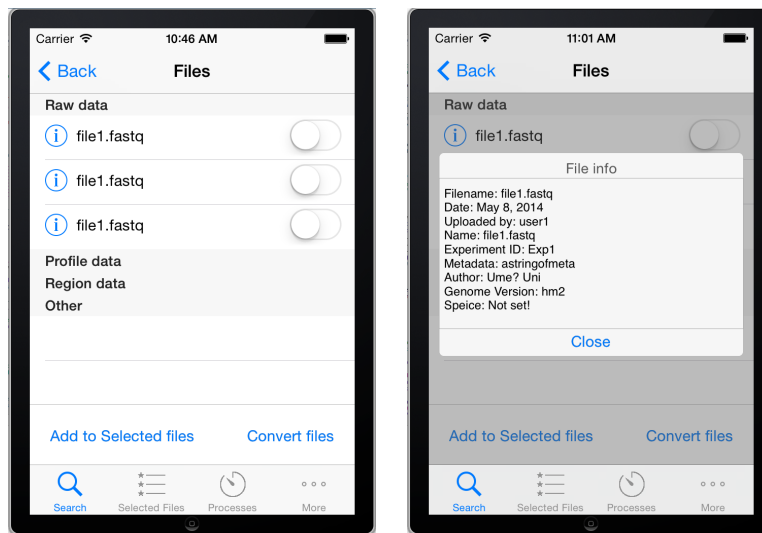


Figure 4.54: The files screen.

The user can also move files to the selected files view. This is done by simply turning the switch to the right of the filename as seen in the left view in Figure 4.55 and then pressing the 'Add to Selected files'-button. The user will then be presented with a popup window as shown in the rightmost view in Figure 4.55.

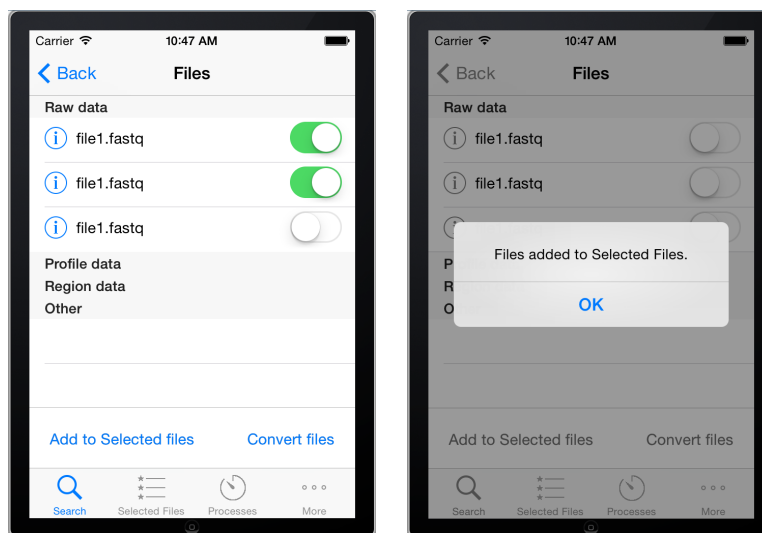


Figure 4.55: The files screen.

If the user presses the 'Selected Files'-button in the menu, the Selected Files screen is presented, containing all files the user has added to selected files, as seen in the leftmost view in Figure 4.56. If the user wishes to see more information about a file, it is possible to simply click the blue information sign close to the

filename. This shows file information in a similar way as in the files view seen in Figure 4.54. In the selected files screen the user can select files and then press the trashcan icon in the top rightmost corner to delete the currently selected files, as seen in the two rightmost views in Figure 4.56. The user can also select a task to perform on the currently selected files by pressing the ‘Select task to perform’-button. The user will then be presented with the Select task screen as seen in the leftmost view in Figure 4.57.

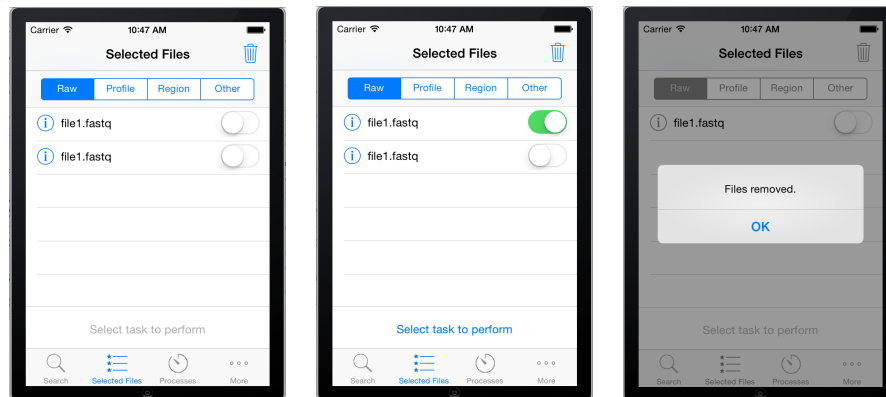


Figure 4.56: The selected files screen.

In the Select Task view, the user is presented with a list of possible tasks that can be performed on the currently selected files. To perform a task, the user can simply click on that task.

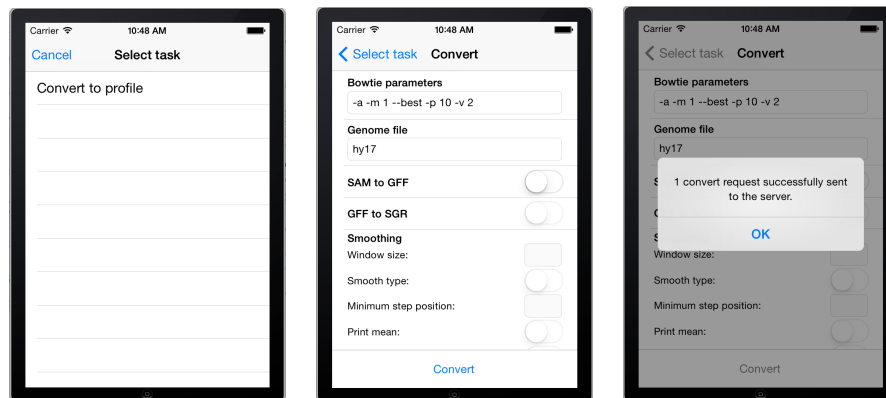


Figure 4.57: The select task screen.

If the user chooses ‘Convert to profile’, the Convert Raw to Profile screen is shown as seen in the middle view in Figure 4.57. In this view the user can enter parameters used in the converting process. Every step of converting (i.e. SAM to GFF or GFF to SGR) requires that all previous fields are filled in, since every convert step uses results from the previous steps in the process. The minimum number of parameters are the first two, Bowtie and Genome file. When the user has entered the desired parameters, a convert request is sent by clicking

the 'Convert' button. The user will then be presented with a popup window, showing the number of convert requests that were sent as seen in the rightmost view in Figure 4.57. If the user wants to see the status for the processes on the server, the user can click on the 'Processes'-button in the menu and the Processes view will be shown as seen in Figure 4.58.

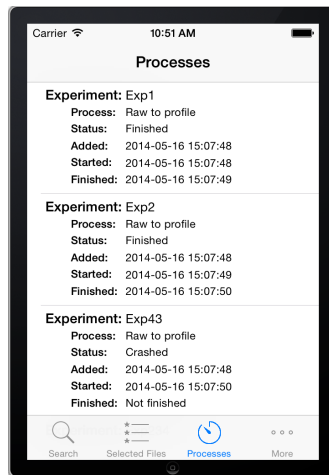


Figure 4.58: The select task screen.

In the Processes view, the user can see all processes that the server have completed in the last two days and also all processes that users have added that are either currently running or waiting to be executed. If the 'More'-button in the menu is clicked the More screen is shown as in Figure 4.59.

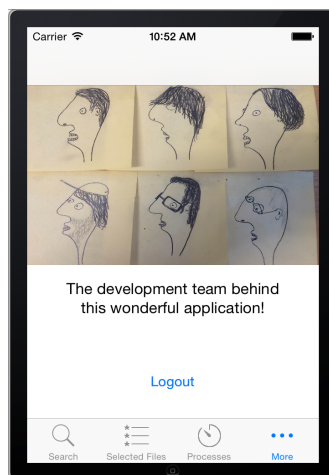


Figure 4.59: The more screen.

In the more view the user has the possibilities to log out from the server and to see pictures of the development team.

Chapter 5

Deployment and maintenance

This chapter is directed towards administrators and developers that wants to set up a server and install the software needed to get a fully functional system. It also gives instructions on how to maintain the system in case of problems that can arise.

5.1 Configure server

This chapter is directed towards administrators and developers who want to set up a server and install the software needed to get a fully functional system. It also contains instructions on how to maintain the system in case problems arise.

5.2 Manuals

To set up the server with the necessary software and configurations two guides are available. These two manuals are created to help a system administrator in the installation process of the server machine needed for the project. The manuals are written for configuration of the server on two different operating systems, Ubuntu 14.04 and Debian 7.5.

The manuals can be found in Appendix G and H.

5.3 Configuration

The *Genomizer* system needs special configuration to work properly. See the manual for the running operating system to get the correct settings for the *Genomizer* server machine. All settings can be changed, but when changed the

system may not work properly anymore. Please only make changes that are documented in the corresponding manual.

5.4 Administer the database

The following guide assumes access to a server with postgresql installed. If you do not yet have a database, username and password for *Genomizer* to use proceed to *Set up postgresql account*.

This guide was written using Ubuntu 14.04 LTS (GNU/Linux 3.13.0-24-generic i686) and postgres-9.3.4.

5.4.1 Set up postgresql account

This step is only required if you do not already have a `psql` username and password. If you have been assigned this from a sysadmin proceed to *Upload SQL Script to server*.

1. Log in to the server:

```
> ssh <username>@<host>
```

2. Become sudo-user "postgres":

```
> sudo su postgres
```

3. Add yourself as a postgresql user:

```
> createuser <username>
```

4. Log into postgresql as root:

```
> psql
```

5. Set your password:

```
> \password <username>
```

6. Create database:

```
> create database genomizer;
```

7. Grant yourself all permissions on the *Genomizer* database:

```
> grant all on database genomizer to <username>;  
> \q
```

8. Navigate to postgresql configuration folder:

```
> cd /  
> cd etc/postgresql/9.3/main
```

9. Navigate to postgresql configuration folder:

```
> sudo nano postgresql.conf
```

10. Change connection settings:

Locate line:

```
#listen_addresses = '<settings>'    # what IP address(es) to listen on;
```

Change to:

```
listen_addresses = '*'    # what IP address(es) to listen on;
```

11. Write changes and exit:

Hold down ctrl and press o

Hold down ctrl and press x

12. Open configuration file:

```
> sudo nano pg_hba.conf
```

13. Change Client Authentication Configuration:

Locate the heading:

```
# IPv4 local connections:
```

Under the heading, add the line:

```
host    all    all    127.0.0.1/32    md5
```

14. Write changes and exit:

Hold down ctrl and press o

Hold down ctrl and press x

15. Restart postgresql:

```
> cd /  
> sudo /etc/init.d/postgresql restart
```

5.4.2 Upload SQL Script to server

1. In a terminal window navigate to the folder where the `genomizer_database_tables.sql` script resides.

2. Establish secure ftp connection to the server:

```
> sftp <username>@<host>
```

3. Create a new folder on the server:

```
> mkdir SqlScripts
```

4. Upload `genomizer_database_tables.sql`:

```
> put genomizer_database_tables.sql SqlScripts/
```

5. Exit `sftp`:

```
> exit
```

5.4.3 Create the *Genomizer* Tables

1. Log in to the server:

```
> ssh <username>@<host>
```

2. Log in to the database:

```
> psql genomizer
```

3. Run `genomizer_database_tables.sql`

```
> \i SqlScripts/genomizer_database_tables.sql
```

The *Genomizer* database is now ready to use.

5.5 Set up processing

To be able to run the processes such as raw to profile conversion the right scripts and programs need to be in the folder resources. The scripts needed for converting will be there but bowtie need to be downloaded and extracted to resources, which need to be a folder in the servers root directory.

5.6 Install the server

To start the server, java needs to be installed on the computer and a runnable JAR file needs to be created. This requires the following things to be installed on the computer: Git, Ant and Java JDK. Refer to Appendix H on how to install these. If these are already installed refer to 5.6.1 on how to download the source files.

5.6.1 Downloading the source code

The source code for the Genomizer server is hosted at Github and is completely open source. It can be downloaded in two ways. Either manually from <http://www.github.com/genomizer/genomizer-server> where there is a button to download the entire project as a zip file or using Git from command line in the following way:

```
git clone https://github.com/genomizer/genomizer-server.git
```

This will create a directory named genomizer-server in the current directory.

5.6.2 Creating a runnable JAR file

5.6.2.1 Command line

When the source code is downloaded (and unzipped if downloaded manually), use the terminal to navigate into the genomizer-server directory.

```
ant jar
```

A file called server.jar should be created in the same directory.

5.6.2.2 Eclipse

To create the runnable JAR file with Eclipse, follow these steps:

1. Open eclipse and import all the code into a project.
2. Right-click on the project and choose export.
3. Expand the folder "java" and then choose "runnable JAR file".
4. Make or choose an already existing launch configuration where ServerMain is the class containing the main-method.
5. Choose an export location for the runnable JAR file.

5.6.3 Starting the server

Here the actual startup of the server will be explained in a step by step manner. In order for this to work, the runnable JAR file must have been created.

1. Choose a computer that should host the server.
2. Make a runnable JAR file of all the code and place it inside a folder on the computer.
3. Start the terminal and navigate to the folder containing the runnable JAR file.
4. In the terminal, type: `java -jar filename.jar`. Server configuration is explained in more detail in Appendix H.2.16.

Chapter 6

Interaction design

This chapter goes into detail on how the graphical and interactive parts of the clients are designed. It starts with a general view of the interaction design and then divideds into chapters based on the different clients.

6.1 Desktop clients

Screen clients use a tab based navigation between views, these tabs are shown at the top of the user interface. The common views in the current system are search, upload and process.

Search results are displayed in a table, experiments can be expanded to reveal the files contained in the experiment. The files in an experiment are grouped by types where each type consists of a row in the table that may be expanded to reveal the files of that type.

The upload view consists of experiment groups. Each experiment group contains a set of input fields for annotation and a list of files added to this experiment. The user may create new experiments in this view or add files to an existing experiment, multiple files may be added to multiple experiments simultaneously.

The base for the process view contains a set of input fields for the parameters that are to be used when processing a file.

6.1.1 *Windows/OS X/Linux* application

The desktop application is constructed in a topdown approach that separates all the different functionalities into groups. Similar functions will be grouped together to utilize space. The application is built with tabs that simplifies work by letting the user easily switch between different views. Each tab is described

by appropriate name and contains related functionality.

The workspace tab lets the user easily manage files and experiments. It has easy access to the download and process functions.

The administration tab design is centered around to have different views that can be reached from the buttons on the left side of the screen. The the Annotations view is where you can add new annotations to the database. This view has a table of all current annotations in the middle of the screen and a toolbar on the right side. Additional functions can be reached from pop up windows when a user clicks on the buttons in the tool bar. A principle in the design is when the user types in something wrong, an alert (popup) will be shown telling what went wrong and why, for example if the user did not type in a name of the annotation a popup telling that a annotation needs a name will be shown.

6.1.2 Web application

Generally the design of the user interface for the web application is an integration of the principles previously described with core design elements of web and the twitter bootstrap element library.

6.1.2.1 Layout and Structure

The structure of the application is in most cases shallow, the navigational depth is usually two steps but sub views with modal views may result in a depth of 3. There are three types of views which are hierarchical in some way, main views contain sub views and modal views, sub views may contain modal views.

- **Main views:** A main view covers the entire page. The structure among main views is shallow and the user may freely navigate between all main views using the navigation bar. Typically a main view contains a toolbar and a set of panels.
- **Sub views:** A sub view is a part of a main view. In this case the main view has a vertical navigation bar on the left side used to navigate between sub views, sub views may not be directly navigated outside of its main view. The user may navigate to other main views from a sub view. Except for the sub navigation bar the sub view covers the entire main view, replacing its content.
- **Modal views:** Modal views “rolls over” the current main view and are used for specialized operations. Modal views can be navigated to using buttons inside main views and sub views. Usually the user will be taken back to the previous view when the modal is closed but navigation in a sequence of modal views could be implemented in the future.
- **Panels:** Content that belong together is grouped using so bootstrap panels. Main views and sub views should contain one or more panels.

- **Toolbars:** In main views and sub views we use a toolbar at the top of the view where operation controls available to the user are presented.
- **Popovers:** For elements that belong to a view but have no need to be visible at all times are shown in bootstrap popovers. Popovers that do not belong to a specific view may be placed in the nav bar.

6.1.2.2 Colors

Grayscale colors are mostly used, black or dark gray is used for text, icons and borders while white or light gray is used for backgrounds. Colors of different hues are used to distinguishing elements from each other and to highlight important elements. Colors with high saturation are reserved for smaller elements while colors with lower saturation can be used regardless of elements size. Light gray of varying brightness may also be used to highlight or distinguish elements.

6.1.2.3 Icons

Buttons that perform actions should always contain an icon as well as text so that the experienced user may more quickly desired actions by identifying buttons at a glance instead of having to read the button text.

6.1.2.4 Batching

For operations performed on objects that there are multiples of e.g. experiments or files, let the user perform these operations on multiple objects at the same time in cases where it makes sense.

6.1.2.5 Systemadministration - Web

The admin page is built up by four views: the sidebar, the main view, the create annotation view, the edit annotation view and the genome-release view. The first one is main view which consists of a sidebar and a empty div-tag. The empty div-tag is then replaced with the annotation list view which has a Create new annotation button and a list of the available annotations on the database with an option to edit.

When the user clicks on for example Create New Annotation, the div tag in the main view is replaced with the create annotation view. The same goes for the Edit buttons on each annotation. This way we only have to render that specific div-tags current information and the sidebar is unaffected.

The design is made so that the user should be able to avoid mistakes. For example in the create annotation page the user is not able to create an annotation

without filling in all the fields. Further more the field for Items in drop-down list is disabled if the user don't choose Drop-down list as the annotation type.

In the Edit annotation view the same principles apply, but also there is a Delete Annotation button on this page which will delete the entire annotation on the database. For that reason we decided to ask if the user is sure of this action and ofcourse made the button red.

The back buttons on the different views work as one would expect and the sidebar option Annotations takes the user back to the main adminview.

The sidebar item "Genome-releases" takes the administrator to the page for adding and editing genome-releases. This page have the same look and feel as the previous. The delete buttons are red and will prompt a confirmation-popup.

The "Select files to upload" will as expected open the file explorer and the user chooses files according to normal operativesystem standards, then the "Upload" button will prompt the user for information about the files such as species and genomeversion before uploading.

6.2 Android

The *Genomizer* Android application was designed to allow for a quick search of the database while on the move. It also makes it possible to start file conversions in advance so that the data is ready when further work and analysis is to be done. The app will also provide a way to continuously view the status of the users file conversions.

The application was designed in close collaboration with the *iOS* application in order to provide a consistent experience on both platforms. We did, however, find it necessary to take into consideration some of the Android specific design paradigms which distinguish Android applications from other smart phone platforms. One of theses paradigms is the actionbar at the top of the screen that provides navigational functions. In this section the layout and design decisions will be described.

6.2.1 Login View

There are two textfields available for the user to type user name and password and a button to click when user is ready to log in. This is a popular layout for many login screens and thus a design many users are familiar with.

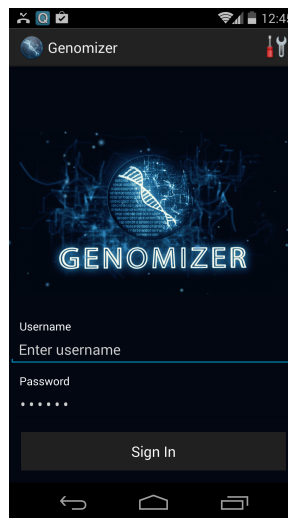


Figure 6.1: Login View

6.2.2 Search View

The design illustrated in Figure 6.2 show the search view, which is also the view the user is presented with upon successful login. The search annotations are displayed in a list and it is easy to learn how to search. Scroll bars are used for multiple options and textfields are used for free text. At the bottom of the view there is a button to press in order to start the search.

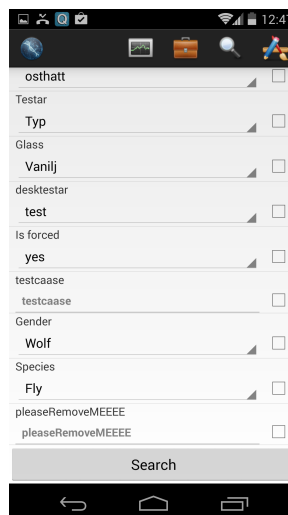


Figure 6.2: Search View

6.2.3 Search Results View

The design illustrated in Figure 6.3 below show the search result view. The result is shown in a list, sorted by experiments. The list displaying search results is large to facilitate usage for user and to take advantage of the screen space. It's easy to learn how to navigate the list. Scrolling is available if the list is long and if the user clicks on an experiment they are redirected to the experiment view displaying more information about that experiment.

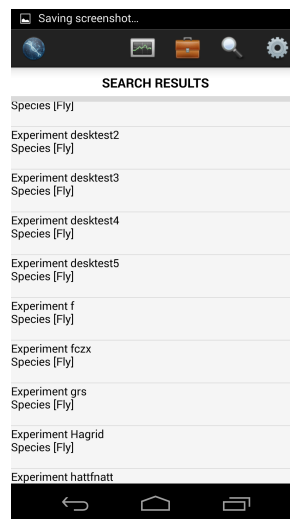


Figure 6.3: Search Result View

6.2.4 Experiment View

The design illustrated in Figure 6.4 shows more information about a specific experiment. All files for the experiment selected in the search result view is displayed here organised by data type. Checkboxes are commonly used and most users are familiar with how to handle them when making choices and selecting items. The button *Add to selection* will be used to send selected files to the conversion view.

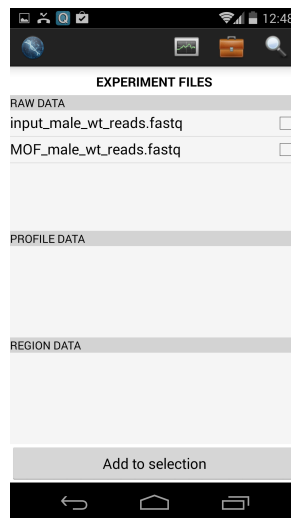


Figure 6.4: Experiment View

6.2.5 Search Settings View

The design illustrated in Figure 6.5 is showing the view for search settings. This is a way for the user to select annotations to be displayed in the search result view. The user can select annotations by checking the checkbox next to the annotation name and then click the button to save changes. The changes are stored on internal storage and saved between runs of the application. If the user has no special requests it is also possible to use default settings. This functionality gives the users the possibility to design the search result view the way they want to have it which often is appreciated.

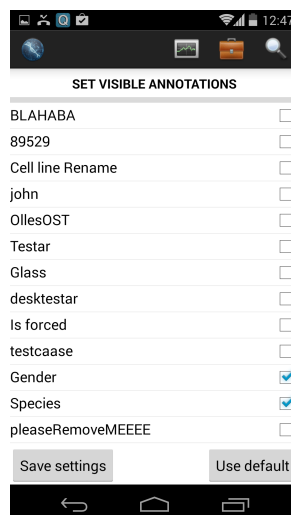


Figure 6.5: Search Settings View

6.2.6 Selected Files View

The design illustrated in Figure 6.6 shows the view for selecting files. The view has four tabs, one for each data type and one for results. To make it easy to navigate the user can switch tab by sliding your finger horizontally. There is also the option of clicking the tabs.

The files are displayed in a list which gives a clear view to the user. At the bottom of the view there is a button with option to what to do with the data files stored. For example in the view for raw data there is a button to click to convert the files into profile data.

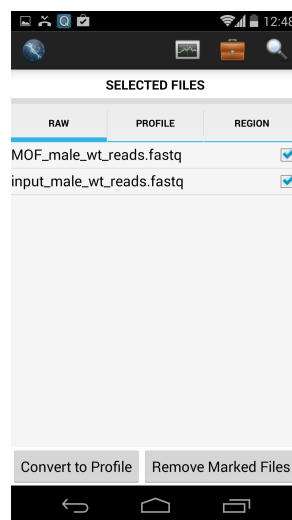


Figure 6.6: Selecting Files View

6.2.7 Convert View

The design illustrated in Figure 6.7 shows the conversion view. This view displays different parameters that needs to be set before starting to convert data files. The view is clear with headlines and hints guiding the user to what parameters that needs to be set that facilitates the work for the user and also prevents errors from occurring. At the bottom of the view there is a button to press to start the conversion when all the parameters are set.



Figure 6.7: The Convert View

6.3 iOS

Focus has been on making a nice looking application with an intuitive workflow and to follow the iOS design principles. Some of the design decisions are motivated in the text below.

6.3.1 Navigation bar

A navigation bar is used to make access to different main functionalities available at all times. In the navigation bar iOS approved icons are used combined with a short describing text directly under it.

6.3.2 Login Screen

The login screen has two responsibilities; to make a nice first impression and to make it easy for the user to login. The design is kept simple and clean to avoid distractions.

6.3.3 Search View

The search view is designed to be usable for both advanced and new users. A list with available annotations is displayed to make it easy to do basic searches fast. Some annotations can only be selected with a picker view, while others are edited by typing free text. The reason for the occurrence of the picker views is to simplify searches and help the user to make correct search requests. For

example, the sex of an individual can only be male, female or unknown. Other values for the sex annotation would be nonsense!

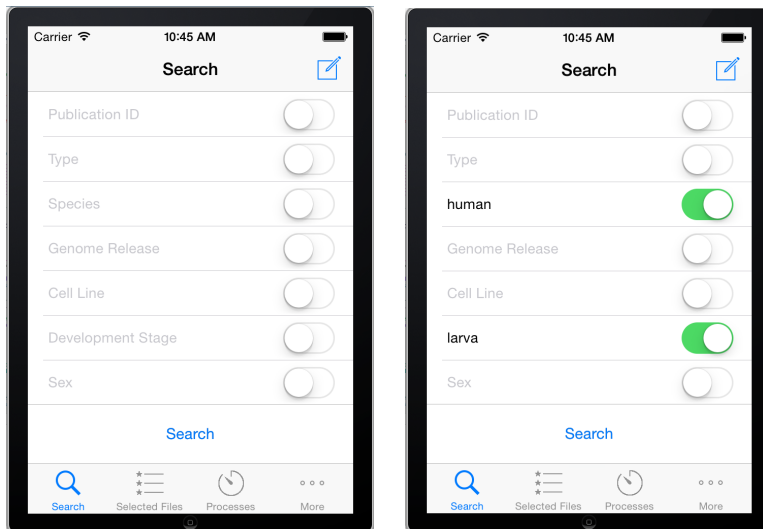


Figure 6.8: The search screen.

Each annotation has a corresponding switch button as seen in Figure 6.8. The button determines if the annotation should be included in the search request. This makes it easy to make small changes to the search, while not clearing the annotation values.

The advanced user can customize the search query sent to the server. This gives the user the possibility to make more complex search queries and possibly make use of already acquired PubMed-search skills.

6.3.4 Search Result View

The main purpose of the search result view is to give an overview of the search results. The challenge with this view was to summarize a large amount of information in a small area. The small screen of the iPhone made it impossible to have columns for each annotation. Instead, a decision was made to group the files by experiment as seen in Figure 6.9. The table with the experiments will only expand vertically, both when the number of shown annotations and the number of experiments grows. Thus, the user never has to scroll sideways, which would be awkward.

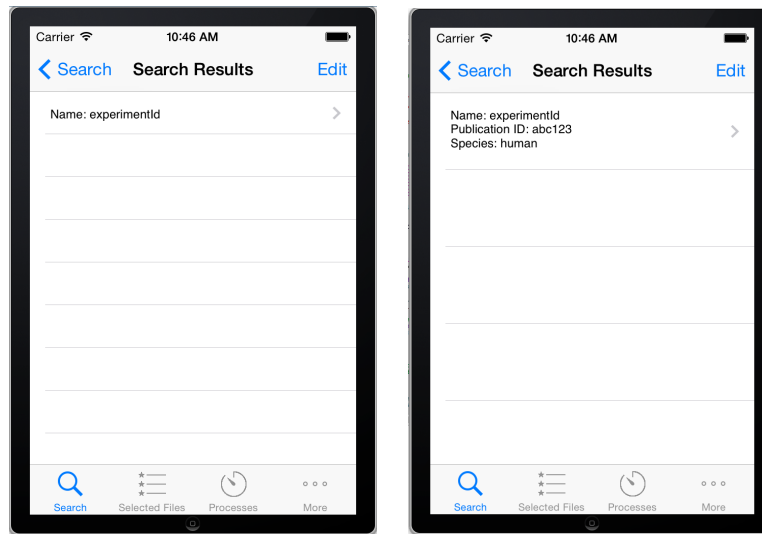


Figure 6.9: The search result view.

The user can choose which annotations to display in the result view. This gives the user the possibility to only show the annotations which are interesting at the moment.

The file view (see Figure 6.10), which is shown when the user selects an experiment, only contains the filename of the files in the specific experiment. The annotations is not shown in this view to avoid information overload and to give the user a good overview of the files.

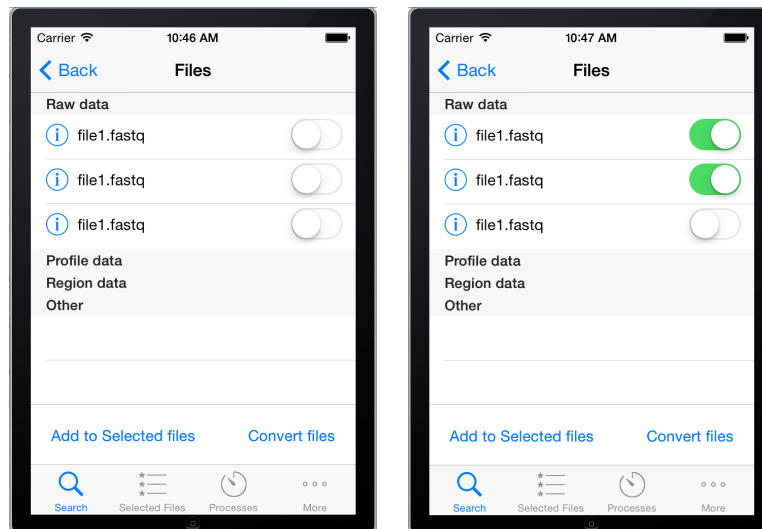


Figure 6.10: The file view.

The functionality of the Convert files button can be reached from other views, but was added to this view as well to improve the workflow. Instead of first

selecting the files, then going to the Selected files view and initiate the conversion from there, the user can quickly convert files directly from the search results.

6.3.4.1 Selected Files

The selected files view can be seen in Figure 6.11. The files are grouped into four categories: raw, profile, region and other. This is done by showing each type of files in its own tab. The reason for this is to avoid the possibility to select files of different types since the tasks to perform are file type specific. It also gives a better overview of the files when only one type is shown instead of showing all files at the same time. Additionally, the top tab bar menu is following the iOS design guidelines.

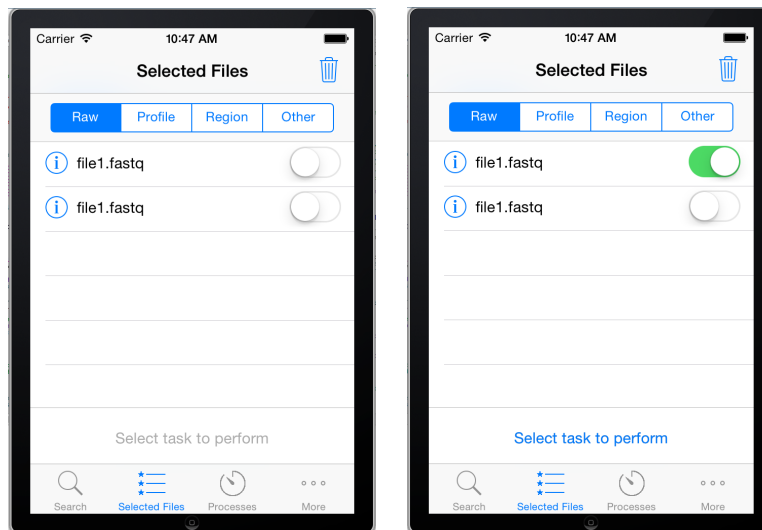


Figure 6.11: The selected files view.

6.3.4.2 Select task and Convert

The select task menu can easily be expanded when new functionality is added to the application. The simplest way we could think of to select a task was to simply press it and hence it's implemented that way.

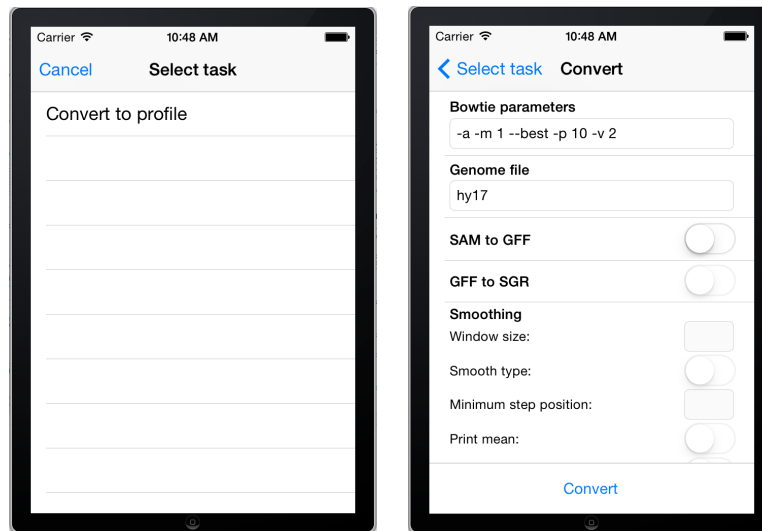


Figure 6.12: The convert view.

As seen to the right in Figure 6.12, the convert view, shows a number of parameters to be filled. Only the first two fields has to be filled before a convert request can be sent. Our thought was that a user should be able to skip all non-optional settings, and let the server have the standard parameter values.

6.3.4.3 Processes

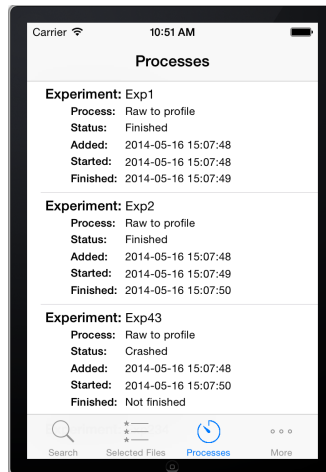


Figure 6.13: The process status view.

As visible in Figure 6.13, we chose to build the processing status view with a simple tableview, to make it dynamic and easy. We also think this gives the user the best possible overview of current processes.

Chapter 7

Architecture design

To get an understanding on how the system is designed as a whole, this chapter will try to explain the architecture of the system on a more broad level.

7.1 System overview

The *Genomizer* is a server-client system, which involves four different clients, a Java server and one *postgresql* database. The different kinds of clients are:

- iOS
- Android
- Web
- Desktop

All of these clients use the *RESTful* protocol together with *Json* to communicate with the server, sent over a non persistent *HTTP-socket*. How the different requests sent over this socket is specified in the API which can be found in the appendix or at docs.genomizer.apiary.io.

The server is also divided in different parts, each with a specific responsibility. These are:

- Communication - Handles requests and responses
- Data storage - Handles the database
- Data transfer - Handles URL and file paths aswell as routing
- Process - Handles processing of files

Figure 7.1 shows a simple flow diagram which describes how the client and server communicates. The particular example shows the data flow when the client process a file.

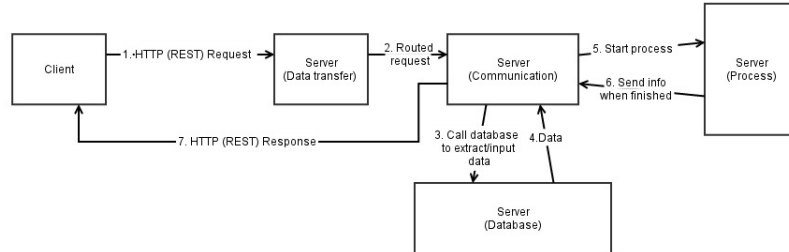


Figure 7.1: A simple flow diagram for the system

Every request the client does creates a non persistent connection to the server. When the server receives a request it checks which kind of request it is and routes it to either the communication part of the server or handles it directly. This is done by the data transfer.

If the request is routed to communication a specific command is created. The command is an object which consists of information from the *RESTful*-header and *Json* body sent from the client. The command is then parsed and sent to different parts of the server, usually the database first, which returns information from a *SQL query*. Depending on the requests this information can later be used to, for example processe a file or be sent back to the clients directly.

The clients are always going to receive a response code after each request, but in some cases the respond also contains a *Json* body with information which can be shown to the user. This is the case for requests like *getAnnotations*. The response can also contain error messages, describing what went wrong when executing the command.

After a client receives the response the connection with the server is lost until the next request.

There is a special kind of user called system admin. A user with these priveleges has the rights to add and delete annotations.

Chapter 8

System design

A more indepth look at how the system is designed with UML- and class-diagrams. It is divided into two main sections for the server and clients. The client section contains the different clients. After that follows the server section that is divided into different parts that makes up the whole server.

8.1 Desktop application

The desktop client is constructed around the model-view-controller pattern. It relies heavily on action events being performed in the graphical interface which is then handled by the controller. The model is the part handling the communication and the storing of important information such as ongoing downloads and the user token (used for communication authorization). In Appendix C a UML-diagram of the desktop client is presented (??).

8.1.1 View

The view of the *Genomizer* Desktop client is constructed with tabs. There are 5 different tabs. These are Search, Process, Upload, Workspace and Administration.

Each tab in the view is represented by its own java class. The `QuerySearchTab` class which represents the search tab can display both a search view and a results view. It uses the `QueryBuilderRow` class to construct the rows in the query builder which is used to construct search queries. The `QueryBuilderRow` class represents a row in the query builder and each row is dynamic and can change accordingly to user interaction. The search results are also implemented in the `QuerySearchTab` and the results are displayed with the `TreeTable` class which is further described in the utilities section below.

The UploadTab Class represents the upload view of the GUI. It has functionality to both upload a file to an existing experiment (which is separately handled in the UploadExistingExpPanel) and to create and upload a new experiment.

The ProcessTab class represents the process view in the GUI. It contains a list where files to be processed can be stored and a large number of processing parameters which can be changed by the user. There process tab also contains a console for displaying direct feedback on processes and an area which contains the status of all current processes which are being handled on the server. The later can be updated manually with a refresh button.

The major part of the WorkspaceTab class consists of a TreeTable which holds all the experiments and the corresponding data which the user has added to the workspace. Then there is also five buttons implemented which allows the user handle the data in the TreeTable. These buttons are Remove from workspace, Delete from database, Upload to, Download and Process. The TreeTable view can be changed to a view which displays all current and completed downloads. This is made using a tabbed pane containing the TreeTable view and the Downloads view.

8.1.2 Model

The model part of the system contains methods for doing most of the logic in the system. For example there are methods for sending login requests and for downloading files. There are separate classes for downloading and uploading files as well as a class for regular communication with the server called Connection. New connections are created with the ConnectionFactory class. The model also acts a storage for importating information such as the user token and list of ongoing downloads and uploads.

The AnalyzeTab Class is not yet implemented.

8.1.3 Model

The model part of the system contains method for doing most of the logic in the system. For example there are methods for sending login requests and for downloading files. There are separate classes for downloading and uploading files as well as a class for regular communication with the server called Connection. New connections are created with the ConnectionFactory class.

8.1.4 Requests

The Request package contains the Request class , the RequestFactory and all the classes that extends the Request class. Request is the super class and can make a JSON package that all the other Request classes can use. All requests must have a name, type and an URL, but can consist of more information. For

example `LoginRequest` also has `username` and `password`. `RequestFactory` is a class that can create all objects from all types of requests. It is a way to easily create all requests from the same place.

8.1.5 Response

This package consists of all types of responses that the server can send to the client-program. There is a class named `Response` that all the other response classes extends from. For example there is a response class for the login request called `LoginResponse`. All types of responses have different properties. There is also a class `ResponseParser` that can parse the responses so that the important information can be taken out of a JSON-package. This information can then be used to tell the client program what should happen next in the user interface.

8.1.6 Controller

The controller part of the system consists of `ActionListeners` for the different buttons and functionalities in the view. For example there are `Listeners` for searching, downloading and processing. The `Controller` class has access to both the view and the model and acts as a middle hand between those two parts of the system. Usually a `Listener` in the controller reacts upon user input and then modifies the model and gives information about the change to the view.

8.1.7 Utilites

There are several classes which represents different data in the system. There are classes for experiment data, file data and annotation data. For example when a search response is received from the server it is parsed into experiment data and the experiment data contains file data and annotation data. There is also a class representing `Process feedback data`.

The `TreeTable` class represents the table which displays experiment data, annotation data and file data in the `Search` and `Workspace` tabs. It is specially constructed to handle the data classes and it allows vertical sorting.

8.1.8 System Administration

The system administration is developed separately from the rest of the GUI, and therefore has a slightly different way of communicating.

8.1.8.1 Communication with the Server

All communication between the server and the system administration tab follows a line of steps. See Figure 8.1 below.

1. An event is triggered by the user clicking something.
2. The listener for the active tab receives the event and sorts out which type it is, and calls the appropriate method in the *SysadminController* class.
3. The *SysadminController* has the connection to the *Model*, and calls the associated method there.
4. The *Model* creates the corresponding request for the server, and then creates a new connection.
5. The *Connection* receives the request from the *Model* and sends the request to the server.

If the event triggers a request for data, the *Model* will use a parser to parse the data before sending it back to the GUI to present it to the user.

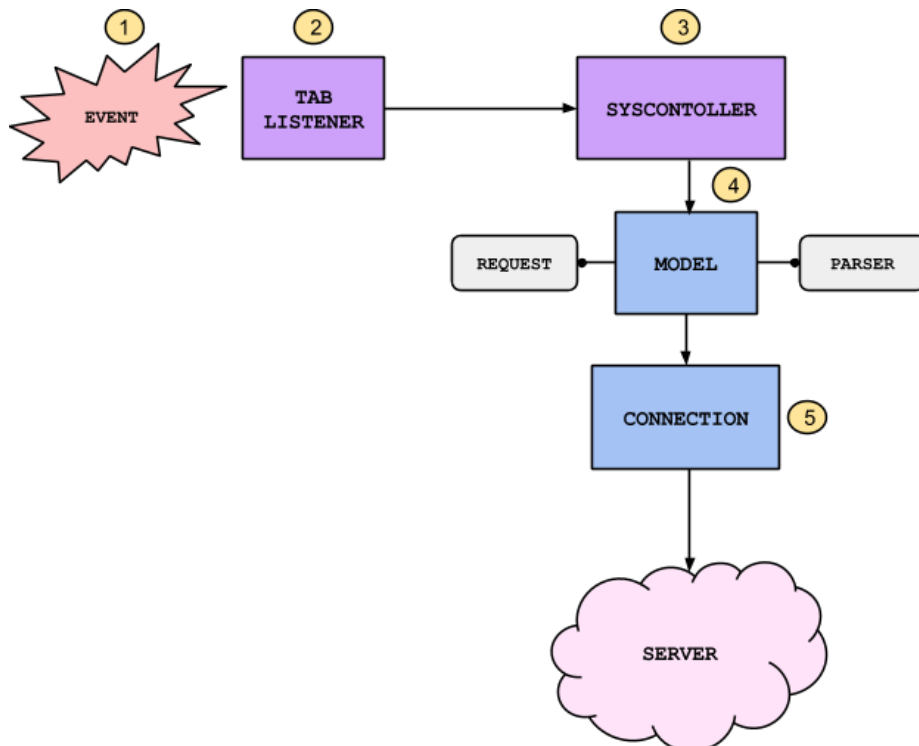


Figure 8.1: Communication Overview

8.1.8.2 A communication example

As an example, assume that the user clicks the 'Genome Files' tab. This triggers the *SysadminTabChangeListener* to receive an event. The desired behavior of the tab is to directly show the available genome releases, so now they have to be fetched from the server. The *SysadminTabChangeListener* therefore calls the *SysadminController*. This class then retrieves the *GenomeReleaseTableModel* to be able to use it when sending the data to the user view. After that it calls the *getGenomeReleases()* method in the *Model*. This method creates a *GetGenomeReleaseRequest* to be sent to the server by using the *RequestFactory* class. The *Model* then creates a new *Connection* by using the *ConnectionFactory*. The request is then sent to the server. The *Connection* receives the result and the *Model* can read from it. In this case the response will be a JSON string containing all the the genome releases on the server. This string needs to be parsed into something more useful and thats when the *ResponseParser* is used. It uses the Google Java library Gson, which is used to convert a JSON string into a Java object. In this case the *ResponseParser* will convert the response JSON string into an array of *GenomeReleaseData* objects. This array is then returned back by to the *SysadminController* which updates the *GenomeReleaseTableModel* with the *GenomeReleaseData* objects. And the user can now see all available genome releases on the server.

8.1.8.3 Building the Tabs

8.1.8.3.1 Building the Administration Tabs

All tabs under the Administration tab are built in a similar fashion and then added to a *JTabbedPane* in the *SysadminTab* class. Each tab has it's own package containing all classes associated to the particular tab. All tabs are also built step by step by using smaller methods creating panels and components. Each tab has at least one main listener that is added to all components that require listeners. Once an event is triggered in a tab the corresponding listener simply use a switch case based on button/tab names to decide which action to take. The main listeners have an instance of the *SysadminController* to be able to further handle requests from the user and send them forward to the *Model* if neccessary.

8.1.8.3.2 Important classes

The system administration part of the desktop application depends on quite a few classes and is based loosely on the model-view-controller design pattern. Here follows a list of the most important classes and a short description of their function and responsibilities.

- *SysadminController* - Handles the communication between the *SysadminTab* and the *GenomizerModel*. The *SysadminController* creates all

ActionListeners for the buttons in the different views. Some minor commands are handled within the sysadmin package, but user commands requiring input or output from the server are received from the different components of the SysadminTab and sent to the GenomizerModel which converts them to Request objects and sends them on to the server.

- SysadminTab - Builds all of the different views that are displayed within the system administration tab. When creating the views it also adds the ActionListeners to the buttons and fields. It also holds a reference to all of the view components it has created so that information can be sent to and from the controller when needed.
- The listener classes - These are added to all of the components of the view that the user can interact with. When an action is performed, the listener performs the action that is assigned to the command string associated with the action. All of the command strings are stored in the SysStrings class for easy access.

8.1.8.3.3 Button and Tab names

To simplify the naming of buttons and tabs a class called SysStrings is used. All buttons or tabs are named here and then this class is used when setting the actual names. This is to avoid hard code as well as making names easy to change and hence more dynamic.

8.1.9 Flow of the system

The sequence diagram in Figure 8.2 describes the flow of the system when the user presses the download file button and the diagram in Figure 8.3 describes how the desktop clients reacts to a login.

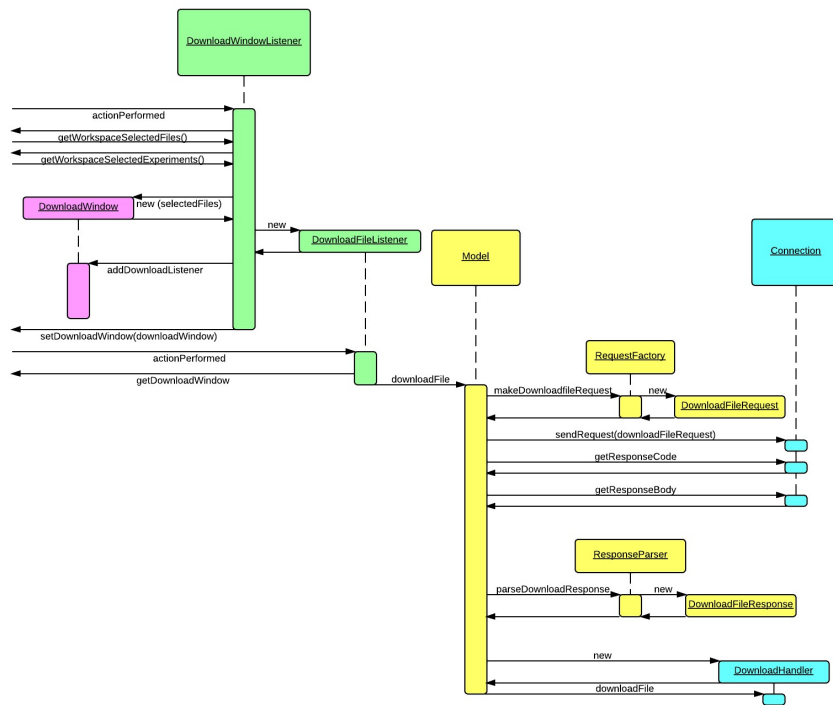


Figure 8.2: UML sequence diagram of downloading a file

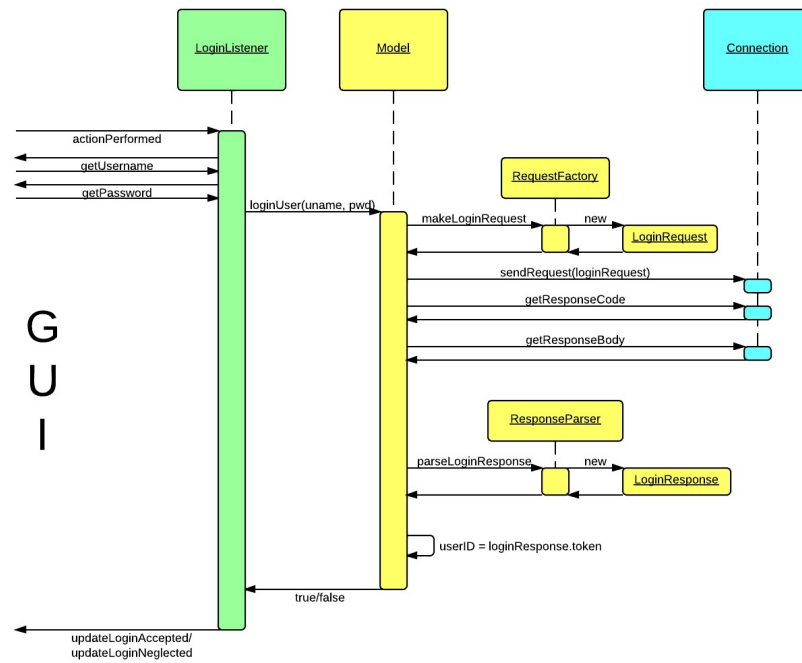


Figure 8.3: UML sequence diagram of login

8.2 Web application

This section describes the overall design of our system, first with a system overview and then with more in depth information about our tabs.

8.2.1 How our web application works

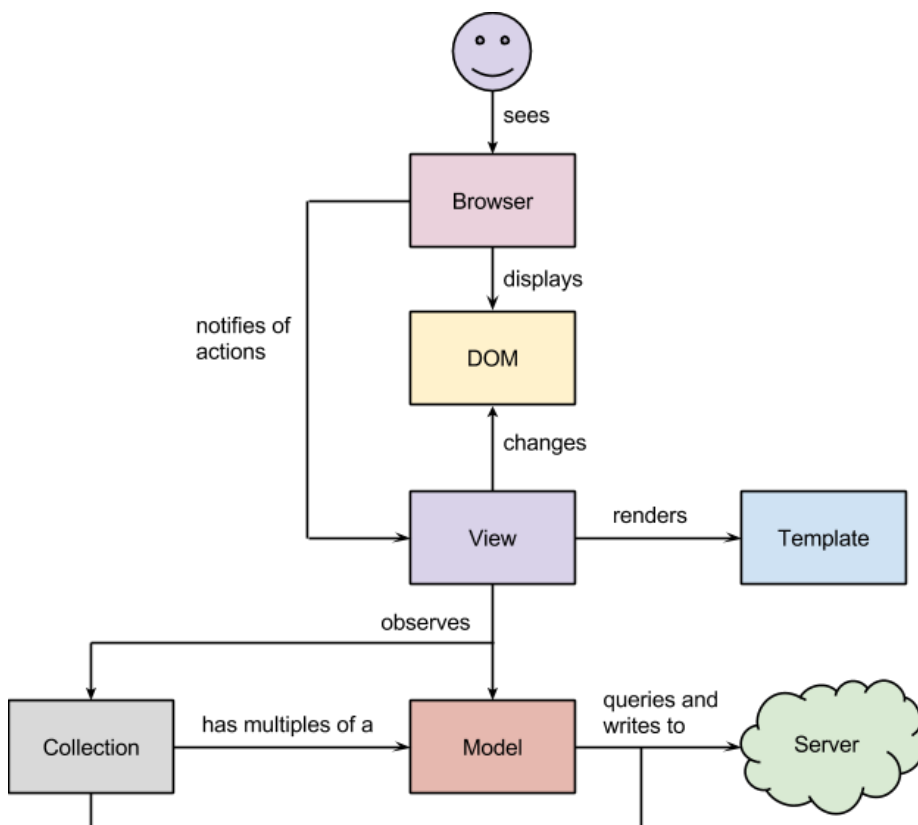


Figure 8.4: A general build of a backbone web app.

Figure 8.4 shows how a backbone[11] web application works in general. We have a user that interacts with a browser. A browser renders the DOM¹ of our web application. How it does this is up to the browser. Different browsers might display it differently. Models and Collections will talk to the server to update themselves. For example, our *Experiments* collection will retrieve experiments from the server and update itself with a call to its `fetch()` method. Out of the components that go into this figure, we are in charge of and only capable of changing a few of these; **View**, **Template**, **Collection** and **Model**. See *Backbone* in section 9.2.1 more information.

¹*Document Object Model*, a convention for representing and interacting with objects in HTML.

8.2.2 System overview

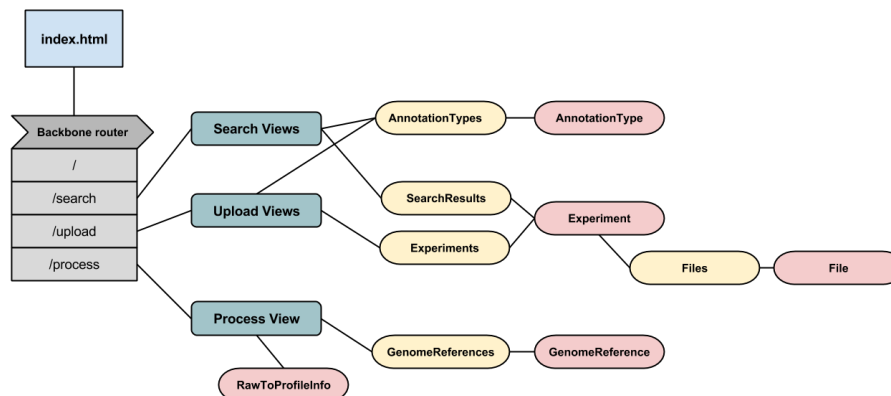


Figure 8.5: Overview of the relations between the different Javascript prototypes in the system.

Since our app is built using Backbone[11], our app is divided into the parts **Misc**, **Views**, **Collections** and **Models**. In Figure 8.5, we can see the system overview. The **views** are the parts in green, the **collections** the parts in yellow and the **model** the parts in red. The parts in grey represent the router which belongs in our Misc category. It is responsible for rerouting links. For example, when a user clicks the search tab, the router navigates to `/search`, but instead of loading the whole `/search` over the page we are currently on, our router will open our search tab below our navigation bar. The **Misc** category also holds our `Main.js`, which is in charge of setting up and starting the app. The views are responsible for the user interface, displaying information and handling events. The collections and models are responsible for holding the data.

8.2.3 Search

The search tab has three views, that together make up the "Search Views" as we have denoted them in Figure 8.5. When searching for data, the models and collections will update themselves to contain the new annotations, experiments and files pertaining to that particular search, so the *Search Views* can display them. Once new data has been retrieved, the user can perform a number of actions on the displayed data. For example, when a user chooses to remove a file, the *Search Views* will receive the event, and tell the file's model to destroy itself. The model will then send a delete request to the server, and disappear.

In Figure 8.6 is a simple sequence diagram for the search tab. If a user enters a query in the search field and then presses the search button, the *Search* view will update the *SearchResults* collection to have a new query. Once *SearchResults* has a new query, it will try to fetch search results corresponding to the query from the server. If successful, new experiment models for every experiment retrieved will be created and set in the *SearchResults* collection. *SearchResults*

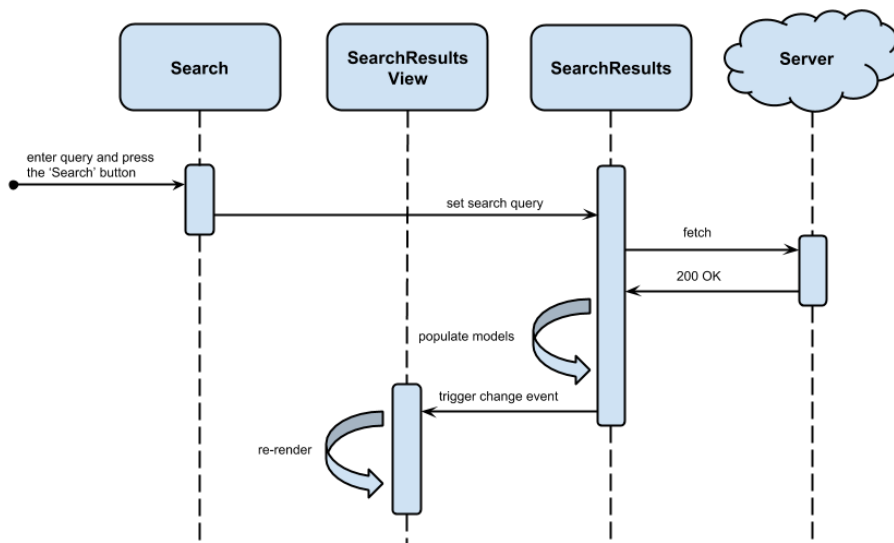


Figure 8.6: a sequence diagram showing what happens when a user enters a valid search query and results are fetched.

then triggers a ‘change’ event that *SearchResultsView* listens to. When that event occurs, *SearchResultsView* knows that *SearchResults* has been changed, and re-renders itself.

8.2.4 Process

Process has a single view that is a *modal*, meaning that it is not a full page like the other tabs but a pop-up that appears over the search view when a user chooses an experiment to process. Process has collections and models to store and send data necessary for a process, like genome releases available for the chosen experiment’s specie.

8.2.5 Upload

The upload tab has three views, that together make up the "Upload Views" as we have denoted them in Figure 8.5. Unlike search (see section 9.2.1) that uses experiment and file models to retrieve information about experiments and files, upload uses the same models to create new experiments and files. To do this, it needs to be aware of what annotations are available, so it uses an annotation type collection to retrieve the current annotations offered when a user wants to create a new experiment.

8.2.6 System administration - Web

The system administration part of the web client is developed using the same tools and frameworks as the rest of the web client. This admin part of the system is made up of view classes, model classes and collection classes. The classes are described below:

Classes used by all views

Gateway - this is a model class used solely for communication with the server. It is a static class in the sense that it doesn't have to be created. It only needs to be included and then its functions can be called immediately without having to be instantiated. The gateway class retrieves the URL from the main JavaScript file this way the URL only needs to be declared once. The URL can then be fetched by any class that includes the Gateway class.

SysadminMainView - the main view for the admin tab, this view is used together with every other admin view. It contains a sidebar menu used to navigate between different admin views.

Classes used to handle annotations

Annotation - this is a backbone model that represents an annotation. An annotation consists of three fields. A name, a list of values and a forced field. The name simply specifies the name of the annotation. The list determines whether this annotation is a drop-down list, or a free-text field. If the list contains one element called free-text, the annotation is a free-text field. Otherwise it is a drop-down list with the values in the list. The forced field determines if the annotation has to be filled in by the user when a file is uploaded.

Annotations - this is a backbone collections that consists of several Annotation models. It also has a URL that it uses to fetch annotations from the server, the URL is retrieved from the Gateway class.

AnnotationsView - this view is the basic view for displaying annotations. It has a search field and a button for creating new annotations. Pressing the button renders the newAnnotationView.

The AnnotationsView has a child view called AnnotationListView. This way the list view can be rendered separately from the search field when the user types in searches.

AnnotationListView - this view uses the Annotations collection to fetch all the annotations from the server and renders them dynamically in a list. In the list is an Edit button for every annotation, the edit button will retrieve the name of the desired annotation and navigate through the router to the EditAnnotationView with the name as a parameter. The view also has a button

that will take the user to the `NewAnnotationView`.

EditAnnotationView - this view uses the name parameter received from the `AnnotationListView` to retrieve a specific annotation from the collection of annotations. It then renders the fields with the values from the annotation. This view has a button to delete an annotation. It will send a delete message to the server using the Gateway model to delete the annotation. An annotation can also be modified in different ways.

NewAnnotationView - this view is used to create a new annotation. It consists of a couple of fields and a create button. Pressing the create button renders a `ConfirmAnnotationModal` which displays the values for the annotation.

ConfirmAnnotationModal - this class extends the `ModalAC` class. It is simply used to display information that the user has to confirm. Pressing confirm creates a message using the Gateway class and sends it to the server.

Classes used to handle genome releases

GenomeReleaseView - this view is used for viewing, adding and deleting genome releases. It contains a button "Select files to upload" which opens up file explorer and lets the user select one or multiple files for uploading. When the user then presses upload the `UploadGenomeReleaseModal` will open. Below the button the view has a table showing the current genome releases available on the server. The user can hold the mouse over files too see all files included in that genome release. A "Delete" button is shown next to every genome release and if pushed sends a delete request to the server through the Gateway class.

UploadGenomeReleaseModal - this modal shows the user which files has been selected for upload and asks for information about which species and genome version they are for. Then at the press of "Upload" the files starts to upload and the user will see a progress bar over the complete upload progress.

GenomeReleaseFiles - this is a collection with `GenomeReleaseFile` as models. It handles the ordering and filtering of its models.

GenomeReleaseFile - this model represent a genome release and can contain multiple files in itself since one genome release is almost never just one file. This class takes care of uploading itself to the server and thereby also updates the progress bar through events that propagate up to the `GenomeReleaseFiles` collection.

8.3 Android application

The following sections describe the architectural design of the Android application. All functionality of the system components are described in this section. Worth noting is that the figures referred to in this section can be found further

down in the document.

8.3.1 Class Descriptions

The focus of the Class Descriptions section is on describing the functionality of each class. The connection between the classes labeled model can be seen in Figure B.1 while the other classes can be seen in Figure B.2. Both are located in Appendix B.

8.3.2 Android activities

Activities are in Android used to contain visual components, i.e. fragments. For anything to be displayed it must be a part of a fragment that is a part of an activity. Each activity used in this application extends the class `SingleFragmentActivity` which main purpose is to create a new fragment of an arbitrary type. This means that in its current implementation the application contains one fragment per activity and each fragment described in this section is coupled with an activity. By extending `SingleFragmentActivity` one can easily implement and visualize new Fragments.

Fragment Classes

<i>LoginFragment</i>	LoginFragment is the first view to be visualized when the application is started. It allows the user to connect to a server by specifying username and password. The static class ComHandler is used to send and validate the login request. If the username and password are incorrect the user will be informed through an android toast explaining what happened. Likewise if the server cannot be reached. A successful login will start the SearchListFragment.
<i>SettingsFragment</i>	SewttingsFragment is used for selecting which server to use. There is also choices for adding, deleting and editing server locations.
<i>SearchListFragment</i>	SearchListFragment is the search-view for the <i>Genomizer</i> app, the annotations that can be chosen are downloaded from the server and they are, as such, dynamic.
<i>SearchPubmedFragment</i>	SearchPubmedFragment is a fragment that handles the search if the user want to manipulate it with the PubMed style of input. When started the current search from the searchFragment is converted and displayed for the user, who can continue using the choosen search values in a PubMed style.
<i>SearchSettingsFragment</i>	SearchSettingsFragment handles settings for which annotations are displayed in the search result. Available annotations is shown in a ListView and there is option to select annotations and save into internal storage. There is an option to set default settings which will show first two available annotations together with experiment name and who the experiment is created by. Settings for using available settings or using default settings is stored in a file in internal storage.
<i>ExperimentListFragment</i>	ExperimentListFragment handles the displaying of search results to the user. The fragment includes a ListView with an ArrayAdapter set to it. An OnItemClickListener is used to detect when the user is selecting an item in the list and is currently starting FileListActivity when a list item is selected. This fragment receives a HashMap with search values from SearchListFragment and when activity is starting an AsyncTask is started to send and receive search results from the server through the ComHandler class. When an experiment is selected from the list the file names belonging to that experiment is sent to FileListFragment that will display the file information.

Fragment Classes

<i>FileListFragment</i>	FileListFragment displays all files associated with a chosen experiment. The fragment is using three ListViews, one for each data type. The data types that are available are: raw, region and profile. Each list element will show the name of the data file and have a checkbox connected to it. A custom ArrayAdapter is used to handle checkbox interaction and displaying information to the user. There is option to select multiple files in the view by checking several checkboxes. The file names displayed are the ones currently available from the server for each available experiment.
<i>SelectedFilesFragment</i>	SelectedFilesFragment gives the user an overview of all files added to the selected files. The view contains a TabHost which in turn consists of a number of fragments. The selectedFilesFragment lets the user explore the tabs by either swipe or by simply pressing the tab the user wishes to see. The tabs consists of the following fragments; RawFragment, ProfileFragment, RegionFragment.
<i>RawFragment</i>	RawFragment keeps track of all raw-files added to the selected files. When this fragment is first created it collects all saved raw files of the type raw from the DataStorage and initializes a listview, visualizing the objects.
<i>ProfileFragment</i>	ProfileFragment keeps track of all profile-files added to the selected files. When this fragment is first created it collects all saved files of the type profile from the DataStorage and initializes a listview, visualizing the objects.
<i>RegionFragment</i>	RegionFragment keeps track of all region-files added to the selected files. When this fragment is first created it collects all saved files of the type region from the DataStorage and initializes a listview, visualizing the objects.

Fragment Classes

<i>ConverterFragment</i>	<p>ConverterFragment displays to the user all the different parameters the conversion can have and what is expected of them. The different inputfields are connected together, so that the user has to fill them out in the right order to be able to get access to the next field. There are some exceptions to that manner, the first two parameters Bowtie and Genome-release has to be used to start a conversion. After that the following fields has to be filled out to gain access to the next inputfield, the last two fields are also an exception to that as they are linked together to form the parameters for the ratio-calculation. When created the fragment calls the server in an async-task and retrieves information about the different genome-releases that is to be found on the server, a spinner with the choices are setup when downloaded. The inputfields are collected when the user press the convert button, by checking which fields/toggleButtons that are enabled. When collected the parameters are sent to the server in another async-task, when the conversions are started and confirmed by the server the ProcessFragment is started up.</p>
<i>ProcessFragment</i>	<p>ProcessFragment is a fragment that presents the user with information about the status of the different conversions that is currently under progress on the server. When created, the fragment will start an async-task and retrieve current conversion status from the server. The information is visualized in a listview.</p>

Model Classes

<i>ComHandler</i>	ComHandler is a static object that is called by the fragments in the view to gain access to the models different functions. At this stage the ComHandler can be used to login, search for files and to request raw to profile conversions, although the latter is not yet integrated. The url that ComHandler tries to communicate with can be changed with a public method which makes it possible to implement a way for the user to change server.
<i>Communicator</i>	Communicator is used to manage the sending and receiving of messages between the server and the application using a http connection.
<i>MsgFactory</i>	MsgFactory creates the JSON messages that can be sent to the server.
<i>MessageDeconstructor</i>	MessageDeconstructor interprets JSON messages and returns appropriate information.
<i>GenomizerHttpPackage</i>	GenomizerHttpPackage stores the body and status code of an http-response.
<i>GeneFile</i>	GeneFile is used to store and transfer the information of a genome file.
<i>Annotation</i>	Annotation is used to store and transfer one or several annotations and their value.
<i>Experiment</i>	Experiment is used to store and transfer information about an experiment.
<i>ProcessingParameters</i>	ProcessingParameters is used to simplify the transfer of parameters in a processing request.
<i>DataStorage</i>	DataStorage is used to save lists of GeneFile objects on the device locally. DataStorage is a static class, which makes it possible to access the saved data anywhere in the application. This simplify the transfer of files between activities.
<i>GenomeRelease</i>	The GenomeRelease class is used to store information about a genomerelease.
<i>ProcessStatus</i>	ProcessStatus is used to store information about the status of a process.
<i>Genomizer</i>	The purpose of Genomizer is to make it possible to create and visualize toasts anywhere in the application. Genomizer is a static class that extends Application.

8.4 iOS application

The following sections describes the system design of the iOS application. The overall system design is discussed followed by a more detailed description of how the segues are controlled.

8.4.1 Overall system design

The system is designed using the model-view-controller principle. Each view is controlled by its own controller class which reacts to user input and triggers changes in the model and updates the view accordingly.

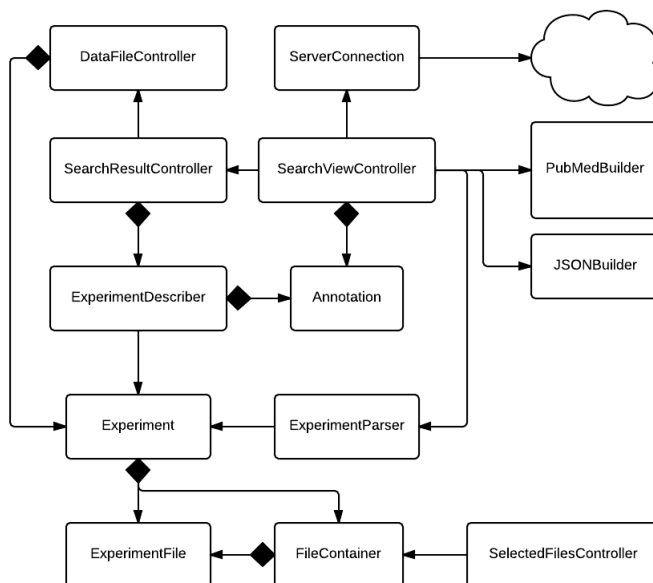


Figure 8.7: UML diagram.

Figure 8.7 gives an overall image of the system design. Some classes are excluded from the figure to make it easier to get an overall idea of the system. The controller classes of the table cells and some other controller classes are not illustrated in the diagram. The non-excluded classes are described in Table 8.1.

Class	Description
<i>Annotation</i>	Contains information about an annotation and can format the annotation name to an aesthetically more pleasing representation.
<i>DataFileViewController</i>	Controls the File view presented in Figure 4.54. It contains a reference to an experiment and lists all its files in a table.
<i>Experiment</i>	A class that contains information related to an experiment, as well as its files.
<i>ExperimentDescriber</i>	Generates a description of an experiment using annotations chosen by the user.
<i>ExperimentFile</i>	Contains information about a file from an experiment.
<i>ExperimentParser</i>	Parses experiment information from a NSDictionary to an Experiment object.
<i>FileContainer</i>	Contains files and sorts them by file type.
<i>JSONBuilder</i>	Creates different JSON requests.
<i>PubMedBuilder</i>	Creates a pubmed search query.
<i>SearchResultController</i>	A controller class for the Search Results view presented in Figure 4.53. It configures the table which holds the information about the experiments a search resulted in. An ExperimentDescriber is used to generate a description of the experiments.
<i>SearchViewController</i>	A controller class for the Search view, see Figure 4.52. It checks which annotation-fields are used and tells the JSONBuilder to generate a corresponding search query when the user presses the search button. The class also contains a advanced search to allow the user to manually enter search queries.
<i>SelectedFilesController</i>	A controller class for the The selected files view shown in ???. The selected files controller contains information about files saved by the user.
<i>ServerConnection</i>	Sends and receives JSON objects to and from the server.

Table 8.1: Description of some classes of the system.

A more detailed description of these classes, and the ones not mentioned here, can be found in comments in the source code.

8.4.2 Segue controll

To avoid several segues to be executed at the same time, a segue controll package has been implemented. Instead of extending `UIViewController`, `UITableViewController`, `UITabBarController` and `UINavigationController` the corresponding XYZ class should be used instead. An overview of this design can be seen in Figure 8.8. This figure also includes the classes `XYZDataFileController` and `XYZSearchResultController` as two examples of such implementations.

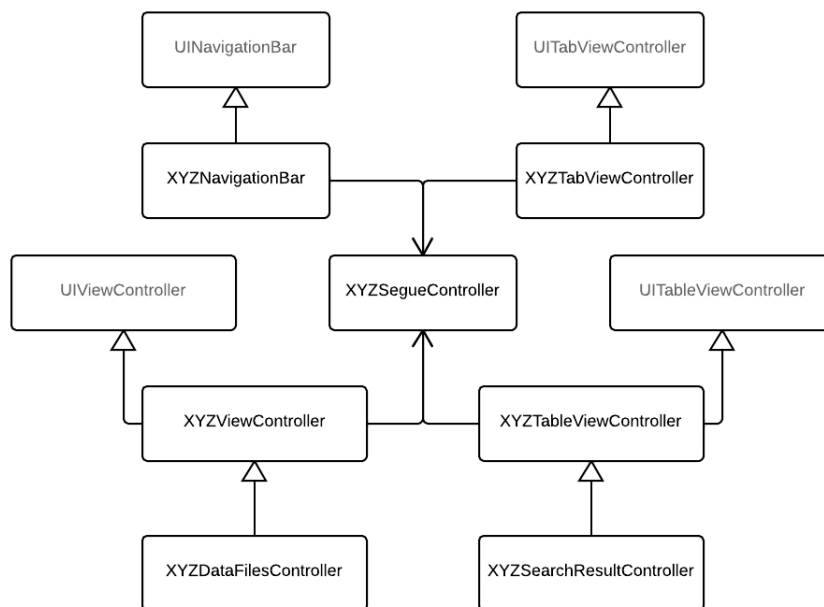


Figure 8.8: UML diagram describing the segue controll.

8.5 Server

The system design of the different parts of the server.

8.5.1 Communication

The server is based around HTTP, where clients send requests on a non-persistent connection and the server responds to these requests. All communication is ini-

tiated by the client and the server has no way of contacting clients except when responding to a request.

Clients send requests to the communication part of the server first. When a request is recognized by the server, the request is parsed and a command is created depending on the request. The command then communicates with the other parts of the server in order to extract or input relevant data.

To identify clients a unique token is used, which is generated when a user logs in by comparing the sent password with the password on the server. The password is stored in a sha-256 hashed and salted string on the server. The password is sent from the client in plain text.

The token is sent back to the client, and the client must include this token with all following requests. Since there is no persistent connection between the client and the server this token is the only way for the server to identify the sender for any given request. The token is also used to prevent unauthorized requests from being executed on the server.

Most commands are executed immediately when the server gets a request, and the result is sent back to the client when the command is finished. This happens when for example searching the database. To take away some effort from the rest of the system, a queue with all the heavy ProcessCommands runs on another thread. These commands are executed one at a time in the order first in first out.

The commands implemented for the server are:

- *login*
- *search*
- *annotation*
- *experiment*
- *file*
- *process*
- *genomeRelease*

The *login* command can take either a '*POST*' method or '*DELETE*' method, depending on if the user wants to log in or log out.

search is used for searching for experiments in the database. Results will display all experiments which match the search query, and the user can chose o expand these experiments in order to view the containing files.

The *annotation* command can be used to modify and view annotations associated with experiments. The server can respond to a '*GET*' request with an array of all possible annotations currently in use in the database. There is also

possible to add new annotations, update the values for an annotation or delete a complete annotation field.

Clients can get information about a specific *experiment* by using a '*GET*' together with this command. The server will respond with information about the experiment as well as information about all the files associated with the experiment. Clients can also add, modify and delete experiments.

An experiment contains *files* which can be uploaded with this command. A '*POST*' will let the client upload a file to a specific experiment. Clients can also download, modify and delete files. When a client downloads a file, the communication part of the server is never contacted. This is because a download URL is already present in the file on the client side. Therefore no contact is needed with the communication part of the server, but instead the file system server takes care of the request.

In order to convert files, the client can send the command *process* together with a '*PUT*'. This will convert specific raw files into profile files. If the client instead sends a '*GET*' it will receive a list of all processes started, and their remaining time until completion.

genomeRelease can be used to edit genome releases, these files are then used when converting files. The client can specify to convert a file from one genome release to another, if it exists in the database.

A more detailed specification of the API can be found in Appendix E.

8.5.2 Data Conversion

The *Genomizer* service needs to be able to convert, process and visualize data. This chapter explains how this is done in the system.

As can be seen in Figure 8.9 the *RawToProfileConverter* extends the *Executor* class. When a call comes to the *ProcessHandler* it then starts the correct conversion which right now only can be a raw to profile conversion.

8.5.2.1 Executor

The executor class, as seen in figure 5.2.1, is an abstract superclass that is an entity that is able to execute various commands. The executor class is able to run programs as well as scripts and shell commands. In order to run scripts and programs the executor has a *parse-function* that parses a string into separate arguments.

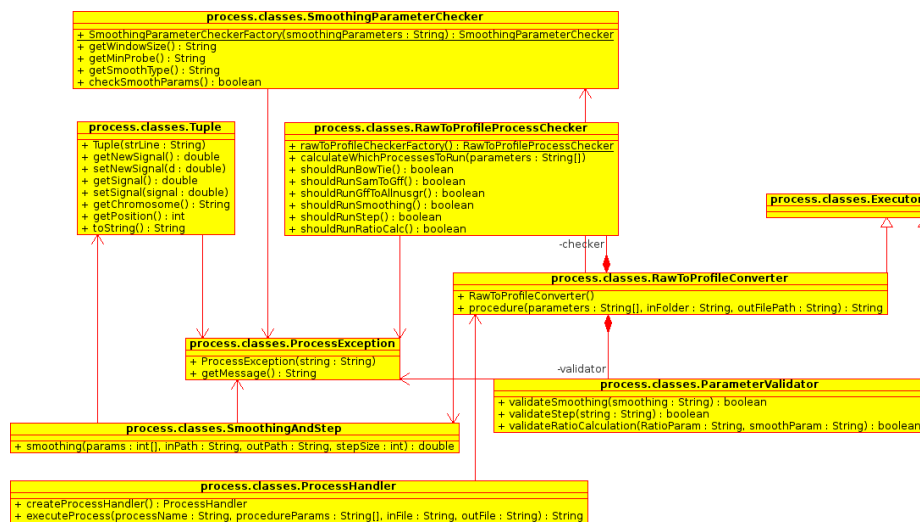


Figure 8.9: Class-diagram for Process

<i>executeCommand</i>	<i>executeCommand</i> is a private method that is being used by the <i>executeScript</i> , <i>executeProgram</i> and <i>executeShellCommand</i> methods. Firstly a <i>processBuilder</i> is used to ensure a safe way to execute commands, after that the working directory is set and the error output stream is merged with the standard output. After a command has been started the output stream is then recorded with the help of a scanner object and a <i>stringBuilder</i> object. When the command has been executed the recorded string is sent back to the caller.
<i>executeScript/executeProgram</i>	Both methods are very similar. The difference is that <i>executeScript</i> has a static file-path added to the second argument. This is because the first argument when calling a script is the script language instead of the actual script file. E.g. <code>shell resources/script.sh</code> .
<i>parse</i>	In order to receive a command string and to be able to run it a parse method had to be implemented. This is because the <i>process-builder</i> takes a <i>String array</i> as argument. With the help of a tool called <i>stringTokenizer</i> the string is parsed into a <i>String array</i> separated on spaces.
<i>cleanUp</i>	Receives a <i>stack</i> with folder names as strings and removes the folders files and then the folder itself. Used to clean up after a process have been executed and generated files during the procedure.

8.5.2.2 RawToProfileConverter

The purpose of the *RawToProfileConverter* class is that it will be used by *ProcessHandler* and do all the different steps needed to make a *raw file*. These steps are done by using the program *BowTie* and by running two different scripts which are executed with methods that is extended from *Executor* class. When ratio calculation is supposed to be done, there are 2 more steps that will be done.

8.5.2.3 Description of Procedure

A description of the steps the procedure method does to create profile data from raw data, all steps are run in order and the user can choose at which step to stop the procedure and get a file from the last executed step.

1. *BowTie*: Creates unsorted *.sam* files. Puts the files in a created temp folder with the name `result_X`, where X is the number of the current thread. All other folders created is placed inside the folder from where the files used where placed.
2. *sortSam*: Sorts the *.sam* files and creates new *.sam* files. Puts the files in a folder called `sorted`.
3. *Run Gff*: Processes the sorted *sam* file and creates a *gff3* file. Puts the files in a folder called `reads_gff`.
4. *Allnucsgr*: Processes the *gff3* file and creates a *sgr* file. Puts the files in a folder called `allnucs_sgr`.
5. *Smooth*: smooths the file and creates a large *.sgr* file, converted the customers *Perl script* by following the algorithm they sent us. This makes it more efficient. Puts the files in a folder called `smoothing`.
6. *Step*: Takes the smoothed *.sgr* file and takes samples from it with a specified interval and creates a smaller *.sgr* file. If stepping is done the files will be placed in the same folder as the previous step.
7. *Ratio Calculation*: Creates four *.sgr* files with the *Perl script* provided by the customer. Puts the files in a folder called `ratios`.
8. *Smooth*: After the ratio calculation, smoothing needs to be done again with different parameters. Puts the files in a folder called `smoothing`

<i>procedure</i>	Executes all the steps to make a profile .sgr file from a raw file, it checks the directory it gets as file-path so that it contains the raw files and that there are not more than two files, but at least one file to process. Does the procedure to create a profile data and move it to the folder thats specified as a parameter.
<i>runBowtie</i>	Constructs a long string with the full execution line for BowTie. It then uses this string as a parameter when calling the method parse. The resulting array is then used when calling executeProgram and the result of the execution is returned.
<i>sortSamFile</i>	Constructs a string with the full execution line to sort a sam file. It then calls parse to create a string array from the full string and sends it as parameter to executeShellCommand which runs a shell command to sort the file and creates a new .sam file that is sorted with the specified parameters. <ul style="list-style-type: none"> • <i>makeConversionDirectories</i> <ul style="list-style-type: none"> – Creates the necessary directories used by RawToProfile’s procedure to put the temporary files needed to do all the steps to create a profile .sgr file. • <i>initiateConversionStrings</i> <ul style="list-style-type: none"> – Defines all strings needed for the directories created when procedure is doing its work. Also defines a string for each step in the procedure, which gets passed to the corresponding execute methods.
<i>getRawFiles</i>	Constructs a File object with the parameter inFolder that should be a directory where the .fastq files that the procedure should run on are. returns an array of File objects with all the files procedure will be using.
<i>makeConversionDirectories</i>	Creates the necessary directories used by RawToProfile’s procedure to put the temporary files needed to do all the steps to create a profile .sgr file.
<i>initiateConversionStrings</i>	Defines all strings needed for the directories created when procedure is doing its work. Also defines a string for each step in the procedure, which gets passed to the corresponding execute methods.
<i>validateParameters</i>	Validates all parameters for the steps procedure should run on. Checks whether a step should be run. If so, validates that steps parameters, returns true if everything is correct.
<i>checkBowTieFile</i>	Checks that bowtie succeded to run and that the result is ok. Checks that bowtie created

8.5.2.3.1 BowTie

BowTie takes two raw *.fastq* files and converts them to *.sam* which is the first step to make the desired *.sgr* files. After a *.sam* file is converted the Linux command `sort` is run on both files which creates two sorted *.sam* files, it is sorted by chromosome and position as needed to use the scripts.

8.5.2.3.2 Used scripts

The different functions of the Perl scripts is explained below. They are explained in the same order that they are executed. All scripts take a directory of files to be processed as input parameter.

<i>sam_to_readgff_v1</i>	Makes a .gff file from a sorted .sam that have reads at each nucleotide positions. No input parameters except the directory of the sorted sam files are needed. The resulting files are put in the new folder <i>reads_gff</i> .
<i>readsgff_to_allnucsgr_v1</i>	Counts the reads from the previous script result. For each chromosome reads are read and each nucleotide position is incrementally counted with one when a read cover it. No parameters are needed for this script except the file path of the gff files. The resulting files are put in the new folder <i>allnucs_sgr</i> .
<i>ratio_calculation_v2</i>	Does ratio calculation on the processed files, for each position in the IP sample with at least one mapped read, a ratio of IP - input (on a log2 scale) is calculated. If the read count in the input is below the read count mean (in the input sample) is calculated it is set to the mean (or double mean (2 x mean) as user specified). If the input mean is below four the minimum input value is set to four (to avoid division by near zero values. Calculated as (read length x approximate total number of reads in input samples(9 millin))/ genome size (for Drosophila melanogaster 120381546)). A random number between -0.5 and 0,5 is added to the read counts before log2 conversion to make them discrete for statistical analysis. All ratio values are then adjusted by reducing each value by median of the ratios. This linear adjustment is carried out in order to compensate for differences in IP and input sequencing depth. Also, to visualize ratios distribution, ratios are plotted by binning ratios with user specified numbers of bins and minimum and maximum ratio values (200bins,minimum ratio value: -10, maximum ratio value:10). Ratio values are printed in sgr format.

8.5.2.4 Smoothing and stepping

The scripts that was provided was inefficient and in order to reduce ram usage and getting faster Raw-To-Profile conversion we rewrote the smoothing and step scripts into a built-in solution in the java server.

8.5.2.4.1 SmoothingAndStep

Smoothing means that we either calculate the trimmed mean value or median value for a position and its surrounding positions. The number of positions we should smooth on is called the Window Size. For example: if we have a window size of 10 we will calculate the smoothed value on position X by calculating on the interval (X-4, X+5). We also need to have a "minimum positions to smooth" number, which tells us that if we have fewer rows to calculate on than the "minimum positions to smooth" we shouldn't smooth at all. There's also one parameter called stepSize, if the stepSize is one the program will not do any stepping but if it's larger than 1 stepping will be done. Stepping is handled in this program by simply checking every time we are going to write to the new file if the current row's position is divisible with the stepSize, if it is we write to the file, otherwise the row is discarded.

The class SmoothingAndStep have one public method and many private ones. The public one called smoothing starts by validating the inparameters and setting up file readers/writers. It then reads as many rows from the file as the window size into an array. It then checks which values that have been read that should be smoothed, after this is done the initiation of the program is complete. From then on the program removes the first row from the array and add one new row to the array and then smooth the middle one in the array. This continues until the end of chromosome or end of file both of which are handled in a similar way. When the program approaches end of chromosome it smoothes as many values as it can until there's less values to smooth than "minimum positions to smooth". It then empties the array and refills it in the same way as it did in the beginning of the file.

The program can handle file corruption to some extent. If the file contains empty or wrongly formatted rows the program will not crash, it will simply ignore the corrupt rows.

The program can also calculate the total mean value of the whole file.

8.5.2.4.2 Tuple

The tuple class is a data carrier that represents one row of data in an sgr file. It consists of the fields chromosome, position, signal and newSignal. Where signal is the signal-value read from the infile and newSignal is the updated value after smoothing have been done. The methods in this class are all standard getters/setters except for the method toString which formats a row for the outfile and rounds of decimal numbers. The constructor is also of interest since it parse a row on tabs. Thus the fields in an infile needs to be seperated by tabs and not spaces. The constructor will throw an exception if the line it tries to parse is either null or if it does not consist of three columns separated by tabs where the first is a string and the second and third is a double.

8.5.2.5 ParameterValidator

Used by RawToProfileConverter to validate the parameters used by the steps in RawToProfileConverters procedure.

8.5.2.5.1 validateSmoothing

Validates all the parameters that will be used in RawToProfiles procedure to smooth files. All parameters needs to be integer numbers and be a positive number, smoothing takes five parameters.

8.5.2.5.2 validateStep

Validates parameters used in the implementation of the smoothing script for the stepping part, takes a string and parses it to become an array of string with each parameter in a index. Checks that there are two parameters in and that the second parameter is a number above zero.

8.5.2.5.3 validateRatioCalculation

Vaildates all parameters used in RawToProfiles procedure when it runs ratio calculation and smoothing on the files. Ratio calculation takes three parameters.

1. Needs to be "double" or "single".
2. Needs to be a positive integer.
3. Needs to be a positive integer.

8.5.2.6 RawToProfileChecker

A class used to calculate which steps in the raw-to-profile conversion to be run.

- calculateWhichProcessesToRun
 - Takes the parameter string array as input and calculates which process steps to be run
- shouldRunBowTie
 - returns a boolean which represents if bowTie should be run or not.
- shouldRunSamToGff
 - returns a boolean which represents if samToGff should be run or not.

- `shouldGffToAllnusgr`
 - returns a boolean which represents if `gffToAllnusgr` should be run or not.
- `shouldRunSmoothing`
 - returns a boolean which represents if smoothing should be run or not.
- `shouldRunStep`
 - returns a boolean which represents if Stepping should be run or not.
- `shouldRunRatioCalc`
 - returns a boolean which represents if ratio calculation should be run or not.

8.5.2.7 `SmoothingParameterChecker`

A class used to validate and convert the smoothing and stepping parameters into a string representation. Is used by the program to give the files the correct name which is needed by the ratio calculation script.

- `getWindowSize`
 - Returns the string representation of window size which is used when naming the file.
- `getMinProbe`
 - Returns the string representation of minimum probe which is used when naming the file.
- `getSmoothType`
 - Returns the string representation of smooth type which is used when naming the file.
- `checkSmoothParams`
 - returns a boolean which represents if the parameters are in the correct format.

8.5.2.8 `StartUpCleaner`

A class used at the initialisation phase of the server program. If the program crashed during a raw to profile processing this class removes the temporary files and directories that had been created.

- `removeOldTempDirectories`

- Removes temporary process directories and it's contents. Takes a string representation of the directory of which the temporary directories can be found as parameter.

8.5.2.9 ProcessHandler

The ProcessHandler is a controller that handles process-calls. Depending on the name of the process it handles it differently. It acts as an interface between the process-module and the rest of the program.

8.5.2.10 Logic & interface

The main logic in the ProcessHandler is a switch-case that switches on the name of the process being called. For example if the name of the process is "RawToProfile" is sets up a RawToProfile-converter and calls it.

processName	A string that tells the handler which kind of process should be executed.
procedureParams	A list of string with the parameters to the different external programs/scripts that will be called during the execution. The first element will be a string with parameters/flags for the first external program that will be called, and so on.
inFile	A string with a path to the directory containing the files that should be operated on.
outFile	A string with a path to the directory where the result .sgr files should be put.

8.5.3 File-transfer

In the current version of the program the desktop clients and the web clients connect to different software on the server. The desktop clients connect directly to the server communication software whilst the web clients connect via the apache server and all non web requests that is to be calculated using the server software is automatically redirected by apache. The redirect is setup in a way that all GET requests that have a /api/ tag in the URL will be redirected. The exception for the desktop clients are file up- and downloads which are done through the apache server.

The download and upload will work for all platforms although this will not be implemented for Android and iOS clients due to hardware limitations.

If the client wishes to upload a file to the server they first send a request to the server-system which authenticates the client and stores the annotations for the

file. The download and upload path is validated by the script to ensure that no invalid paths are sent to the scripts.

Two PHP-scripts are used for uploading and downloading files via Apache. If the user wants to upload a file, the PHP-script will try to store the file in a location on the server provided by the client. A script for downloading files from the GEO database and saving them on the server is currently being implemented, although not yet fully tested nor implemented by the clients. See Appendix F for examples when using the PHP-scripts.

In Figure 8.10 below it is shown how the systems handles the different types of messages the client-systems can send. The big square represents the Apache server with different parts of the Apache server within. The iOS and Android clients can only send some requests to the server-system. Meanwhile, the desktop client can send requests to the server-system and upload and download to/from the web server. The web client sends all its messages to the Apache server and if it is a request to do some sort of computation it will be redirected to the server-system and if it is a download, upload or web-page message it will be sent to the web server.

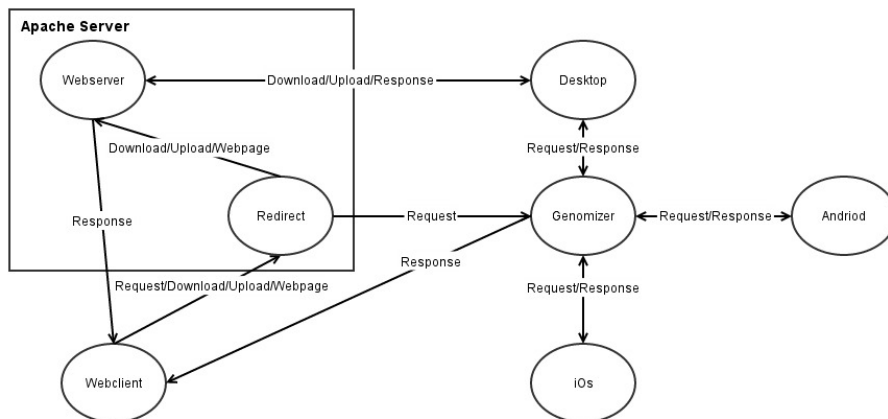


Figure 8.10: The different types of messages sent between the systems.

The current version of the system utilizes a file structure to organize HTML- and file requests on the server, the structure is illustrated in Figure 8.11. The Web-root folder contains the PHP-scripts for uploading and downloading files. The app folder contains the *Genomizer* web page. All folders of the experiments are located in the data folder, which contains folders for the different data-types.

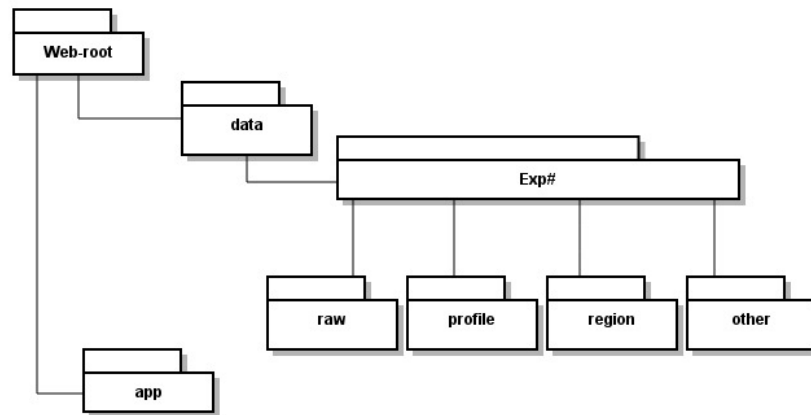


Figure 8.11: Illustrating the current file tree on the server machine.

8.5.4 Data Storage

In order to enable the annotation and subsequent searching for experiments and files the data stored on the server is complimented by a database of information.

Each file uploaded to or generated by *Genomizer* belongs to an experiment which is identified by the experiment ID (expID). Each experiment created by the end user results in an entry in the database's *Experiment* table.

Each experiment contains files that were either generated during the experiment (*raw* data) or processed from these files (*profile* or *region* data).

The full database schema is shown in Figure 8.12. The tables and columns currently not utilized by *Genomizer* are in grey. These have not been removed from the database under expectation of future development.

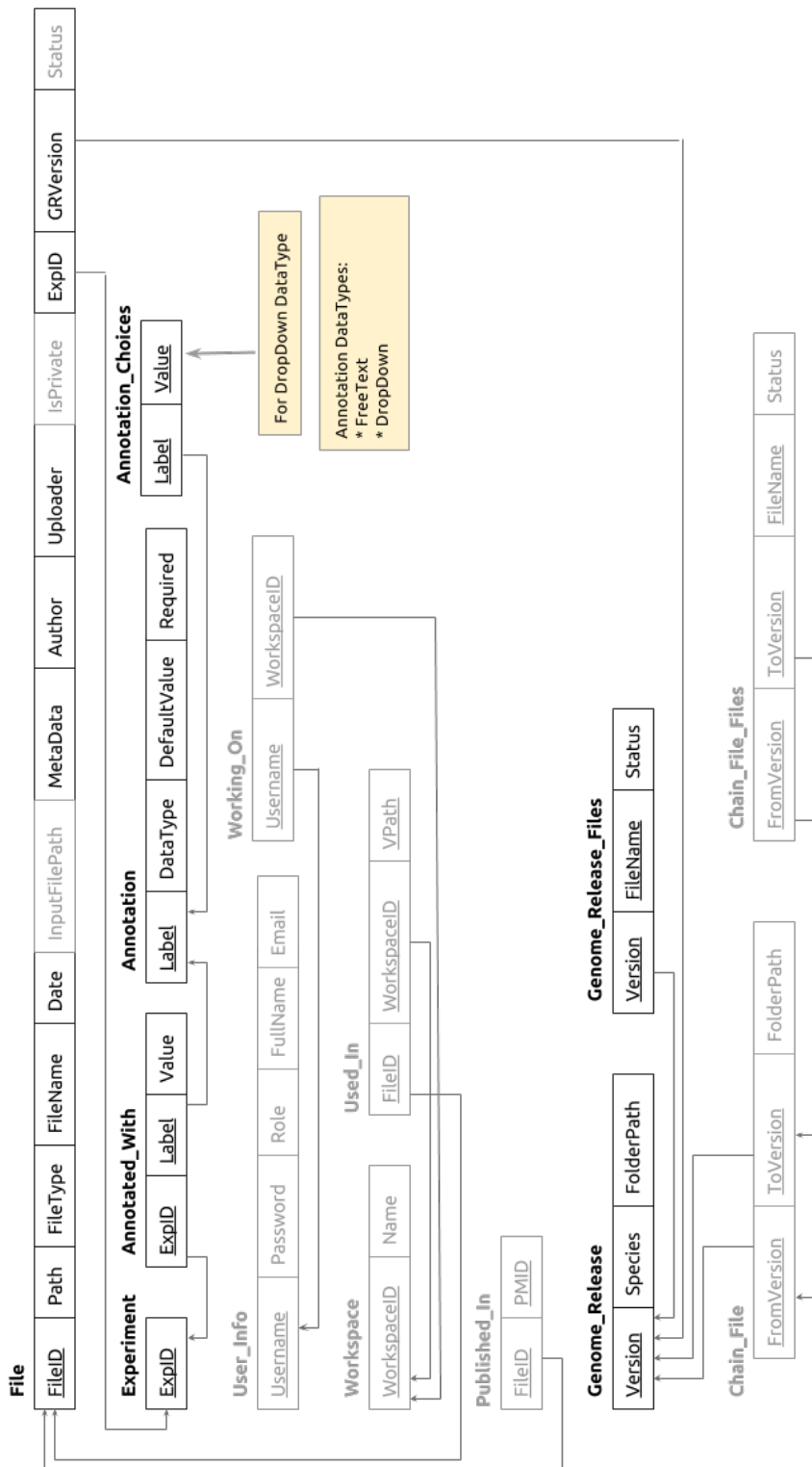


Figure 8.12: The database schema

8.5.5 Database Design

The following section will explain the less obvious columns and their intended use.

FileID is the identification number for a specific file. The data type **SERIAL** is used and will therefore be auto-generated by the database upon insertion.

Path is the path to the corresponding file in the file system, for example: `/var/www/data/Experiment1/raw/rawFile1.fastq`

MetaData is the string of parameters used in processing and should be **NULL** for all raw files.

Annotated_With is the table that enables the annotation of experiments, for example:

Experiment1	Species	Dog
-------------	---------	-----

Annotation is the table containing all the possible annotations a user can use to provide extra information about an experiment. This includes the type of annotation which is *Drop Down* for annotations where the user can choose from a drop down list, or *Free Text* where the user can enter the value freely. There is also support for a default value and annotation forcing where users are forced to provide the information, for example:

Species	DropDown	Human	T
---------	----------	-------	---

Annotation_choices is the table specifying the choices for *Drop Down* annotations, for example:

Species	Dog
Species	Fly

The **Genome_Release** table stores information about the *Genome Releases* available for use. This includes the unique version code for a *Genome Release*[20].

The **Genome_Release_Files** table stores the information about the files that make up the *Genome Release*.

8.5.6 The Data Storage Subsystem

All the classes used in the manipulation of the database and the creation of the file systems directory structure is contained in the java project's **database** package.

The other *Genomizer* subsystems execute all updates to the data storage through the `DatabaseAccessor` class. As a result there are many methods in this class, however most methods send the request on to the classes in the `database.subclasses` package. Here the methods that modify the different areas of the data storage system are broken up into different classes of a more manageable size. An UML diagram of the `DatabaseAccessor` class and its subclasses is available in Figure D.1 in Appendix D.

The `DatabaseAccessor` utilizes a number of classes in order to return information to the method caller. These classes are contained in the `database.containers` package and are as follows:

- `Experiment`
- `FileTuple`
- `Annotation`
- `Genome`

An UML diagram of these classes is also available in Figure D.2 in Appendix D.

8.5.7 Interaction

Below are examples of typical interactions with the `DatabaseAccessor` class.

8.5.7.1 Adding an experiment

In order to add an annotated experiment the following steps must be followed:

1. First the `addExperiment` method must be called. This will add one experiment to the database without any annotations set for that experiment. If you try to add one experiment that already exist then the addition will be refused and an exception will be thrown.
2. If there are no annotations that can be used to provide extra information about the experiment they must first be added by calling the `addFreeTextAnnotation` or `addDropDownAnnotation` methods.
If a *Drop Down* annotation already exists, but there is no suitable choice for the experiment a choice can be added by calling the `AddDropDownAnnotationValue` method.
3. An available annotation can be used to provide extra information about an experiment by calling the `annotateExperiment` method.

Now that an experiment has been added files can be added added to it.

8.5.7.2 Annotation Handling

`getChoices` gets all the available annotation choices connected to a specific label. For example the possible choices returned for the label "sex" might be "Male, Female and Unknown".

`getAnnotations` returns all annotation labels currently stored in the database. Examples could be "Sex,Species,Tissue,etc."

`getAllAnnotationObjects` Combines the two previous methods. Here an annotation object is returned that holds all the relevant information including the label, datatype, and the possible choices for a *Drop Down* annotation.

`changeAnnotationLabel` updates the given label in the database. This will change the label for all experiments that use it. For example changing "specie" to "Species".

`changeAnnotationValue` updates a value for a specific annotation label. For example changing "Human" to "Homosapien".

`updateExperiment` Updates an annotation for one specific experiment. Example: "experiment1, Species, Homosapien" can be changed to "experiment1, Species, Fly".

`deleteAnnotation` deletes an unused annotation from the database. This will also delete all the choices for that annotation.

`removeAnnotationValue` removes a single annotation value for a particular label.

8.5.7.3 File Handling

To add a file you will need to have an experiment added before you call the `addNewFile` method. Raw files usually come in pairs and so they can be added together by specifying the input file name.

`deleteFile` deletes the given file from both the database and the file system. This can be done by either specifying the path or the file's ID number.

Genome release files must be added one at a time by calling the `addGenomeRelease` method. This returns an upload URL.

`removeGenomeRelease` removes all the files associated with a genome release. This can only be done if there are no files that have been generated using the specified genome release.

8.5.8 Apache

The Apache HTTP Server, or commonly referred to as Apache, is the web server application which is used to upload and download files to and from the server. Apache is open source, which makes it free to use. Apache is a good choice because it is developed and maintained by an open community, that way all new versions and updates will become available for free. Since it is open source, the source code is open for everyone to read. Apache can be used on both Unix/Windows systems. In this case it is running on a Unix machine but can still communicate with all platforms.

8.5.8.1 Server user manual

The Apache server is controlled from the terminal. This can be done either directly from the server or remotely from another computer using Secure Shell (SSH). To use SSH from another computer, write

```
ssh username@address.to.server
```

in the terminal. Enter the password for the server and then write the commands directly in the terminal.

These are some of the most common commands for apache:

Action	Command
Start Apache	<code>sudo service apache2 start</code>
Stop Apache	<code>sudo service apache2 graceful-stop</code>
Restart Apache	<code>sudo service apache2 graceful</code>

Chapter 9

Implementation

This section contains the implementation of the different parts of the system and what tests has been used to ensure its functionality. Here developers can get an understanding of how and why the different parts of the server was completed.

9.1 Desktop application

The desktop client is implemented in java 7. The graphical part of the client is made with java swing and the external library swingx. The tree table which is used in the graphical interface is implemented using a modified version of the JxTreeTable found in swingx. The modifications made to the JxTreeTable is that a sorting mechanism has been added and it is possible for the user to choose which columns to show.

The communication with the server is handled with a http protocol involving JSON formatted bodies. The external library GSON and the Apache Http Client are used for the communication.

For dragging and dropping files into the upload tab, the desktop client uses a modified version of the class FileDrop, which was originally written by Robert Harder and Nathan Blomquist and was released as public domain.

9.1.1 Testing

The testing of the system has been quite varied since a large part of the desktop client consists of a graphical interface. The graphical part of the client was tested throughout the developing process and the customers also had a part in testing the interface. Another difficult part of the testing was the communication with the server. A part of it was tested with JUnit tests but the larger part of the testing was made manually by interacting with the GUI and communicating

manually with the server.

The client will be involved from an early point since the Scrum developing methodology relies in delivering functionality as early as possible. Because of this, it is given that the system will have bugs, and the client will be of assistance in finding these bugs and reporting them. Before each release of implemented functionality however, they are tested with the Test Cases enclosed to each User Story.

A small number of JUnit tests has been done concerning communication with server API.

9.2 Web application

9.2.1 Frameworks

To ease implementation a couple of frameworks have been used. The frameworks are described briefly below.

9.2.1.1 Backbone

Backbone[11] is a light-weight framework that loosely follows the **MVC** (model, view, controller) pattern. Out of the **MVC** components, backbone only has models and views, and the view behaves much like a view and a controller. **Models** are the parts of code that retrieves and populates data (for example, the model Experiment will obtain and populate the experiments resulting from a search). **Views** are the HTML representation of models, and they change as models change (When the Experiment model is populated, it is immediately presented on the view that contains that Experiment). Backbone makes use of **Events**, where other objects can trigger events and listen to them, which is an effective way to promote decoupling between components. It also uses **Collections**, that are ordered sets of models. A collection will automatically be provided with underscore array and collection methods for convenient set manipulations (You can, for example loop through a collection with `.each()` instead of writing a for-loop). We chose to use backbone because we wanted more structure in our web application. With more structure, it is easier to collaborate as we can divide up the work - keeping our Javascript in various model, collection and view files.

9.2.1.2 Bootstrap

Bootstrap[12] is a front-end framework that contains HTML and CSS-based design templates for typography, buttons, forms, navigations, and the like. Instead of creating our own buttons, deciding on colors, how big they are, and micro-managing how they fit with everything else on the page, we can use bootstraps

templates that handles all of that for us, leaving us able to focus on architecture. We chose to use it to save time on development and make the look of our web app easily customizable.

9.2.1.3 RequireJS

RequireJS[15] is a file and module loader for Javascript. RequireJS lets files require other files much like `#include` in Java. This is very handy for the programmer. It is used because it helps to structure the application.

9.2.1.4 JQuery

The purpose of JQuery[16] is to make it easier to use Javascript on a website. It takes a lot of common tasks that require many lines of Javascript code to accomplish, and wraps them into methods that you can call with a single line of code. It simplifies other things as well, like AJAX calls and DOM manipulation, both of which are in frequent use in our web application.

9.2.2 Technologies used

A couple of technologies have been used in the development and are described below.

9.2.2.1 AJAX

AJAX[13] stands for **Asynchronous Javascript and XML**. It is a technique for creating fast and dynamic web pages. Despite the name, the use of XML is not required; JSON is often used instead, as we have done in our web app. AJAX allows web pages to be updated asynchronously by exchanging small amounts of data with the server, so that you only update parts of a webpage without having to reload the entire page (like websites that don't use AJAX have to). For example, when "search" is clicked in our navigation bar, only the bottom half of the website is being updated, and displaying the search view. The navigation bar does not have to be reloaded, but remains as it is on top.

9.2.2.2 JSON

JSON[14] is short for **Javascript Object Notation** and is a format that is primary used to transmit data between a server and web application instead of using XML or other formats. JSON is formatted as text which is easy to read consisting of attribute-value pairs. JSON was used in this application because JSON uses the same syntax as Javascript and therefore we do not need to make

our own parser as we would have to do for e.g. XML. JSON also works very well together with Backbone as it has integrated methods using the JSON format.

9.2.3 Testing frameworks

For testing we have used three libraries to make testing easier: Chai, Mocha and Sinon. Together they let us make a page for testing where all tests and results will be shown visually. These libraries or testing frameworks will be discussed below.

9.2.3.1 Chai & Mocha

Mocha[18] is a test framework while Chai[17] is an expectation framework. While Mocha setups and describes test suites, Chai provides convenient helpers to perform all kinds of assertions against Javascript code. We use these frameworks to do unit testing on our models and collections.

9.2.3.2 Sinon

Sinon[19] is a framework used to “*fake environment*”. When doing unit testing, we don’t want to depend on things that are external to the unit of code that we are testing. We can use Sinon for stubbing and mocking external dependencies and to keep control on side effects against them. For example, we can use Sinon to create spies to see if an event has been triggered, and to create fake servers that respond with fake pre planned responses to our queries.

9.2.4 Our Tests

Unit tests have been performed on all model and collection files that contain non-trivial functions. All unit tests can be found in the root folder under `/tests/`, more specifically `/genomizer-web/tests/`. To run the tests, simply open the `index.html` in a web browser, and they will run. The views have not been unit tested since it is overly complicated; instead they have been continuously manually tested throughout the development process. In addition to these simple development tests more official system tests have also been done by the desktop group.

9.3 Android application

This section focuses on the communication between classes and how the Android application works.

9.3.1 Login request

When the user starts the application and is prompted for a login, the the following sequence of actions is performed by the system (See Figure 9.1).

- User starts the application and is prompted with a login. Then the username and password is passed on to the LoginFragment.
- LoginFragment sends a login request to the ComHandler, with the username and password.
- The ComHandler initializes Communicator and sends a setupConnection command to verify that connection can be made. Then initializes the MsgFactory and request a login-package using the createLogin method.
- MsgFactory responds to ComHandler with a login-package in JSON format. Then the ComHandler calls the sendRequest method in Communicator with the login-package and waits for reply.
- The Communicator connects to the remote Genomizer server with a HTTP-request containing the login-package as a JSON object. If the search is valid the server will respond with code 200 and a user-token as a JSON-object.
- The JSON-object is then returned to the ComHandler which will store the token and return a boolean to the LoginFragment informing if the login was successful or not.
- If the response was true the LoginFragment will startup the SearchFragment and present that view to the user. The Login Fragment will be terminated at that point.

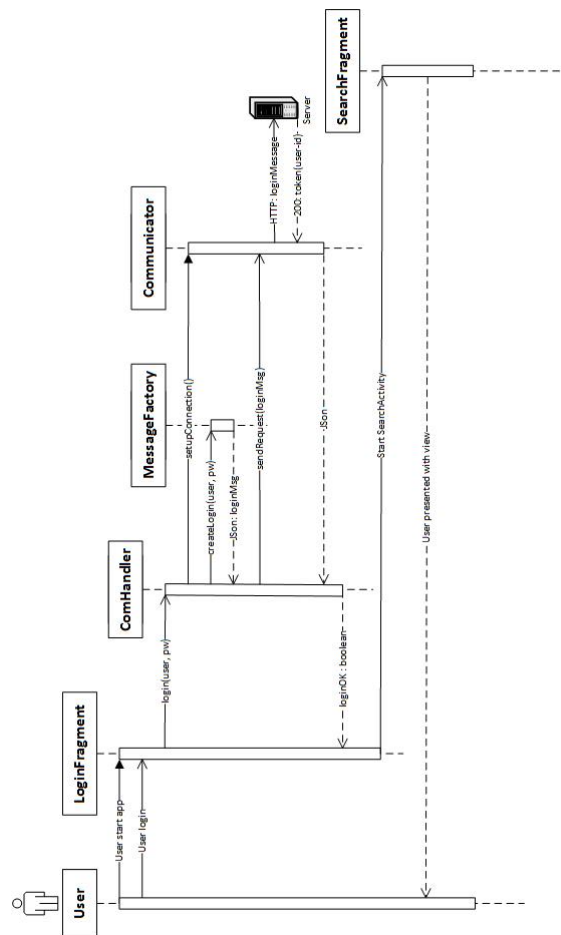


Figure 9.1: Sequence diagram for a login-request

9.3.2 Search request

When the user sends a search request using the search view in the application, the following sequence of actions is performed by the system (see Figure 9.2).

- User will make a search request on the screen and press on the search button. That will trigger the SearchFragment to startup the ExperimentListFragment with the search string sent in as a Intent-Extra variable.
- The ExperimentListFragment will then initialize the ComHandler and call the search method with the search-list provided by the SearchFragment.
- ComHandler initializes the Communicator and determines if a connection can be made.
- If connection can be made the ComHandler initialize the MsgFactory and calls the createRegularPackage method, which will return a pre-formatted JSON-object to be used with the search request to the server.

- ComHandler calls Communicator using the `sendRequest` method passing on the JSON-object containing the search-list, and waits for the reply from Communicator.
- Communicator connects to the Genomizer server with a HTTP request containing a JSON object with the search-list. The server will respond with code 200 and a JSON-object with the search result from the server.
- Communicator will reconfigure the JSON object to a GHTTP¹ to preserve both the head and body from the server response. Then the GHTTP is returned to the ComHandler.
- The ComHandler will initialize the `MsgDeconstruct` and send the collected JSON-object containing the search result from the server to the `deconstructSearch` method.
- `MsgDeconstruct` will parse the JSON-object to an `ArrayList` of experiments, and return that to the ComHandler.
- ComHandler will then return the search-results to the `ExperimentList-Fragment` that will present the results for the user on the screen.

¹Genomizer HTTP package

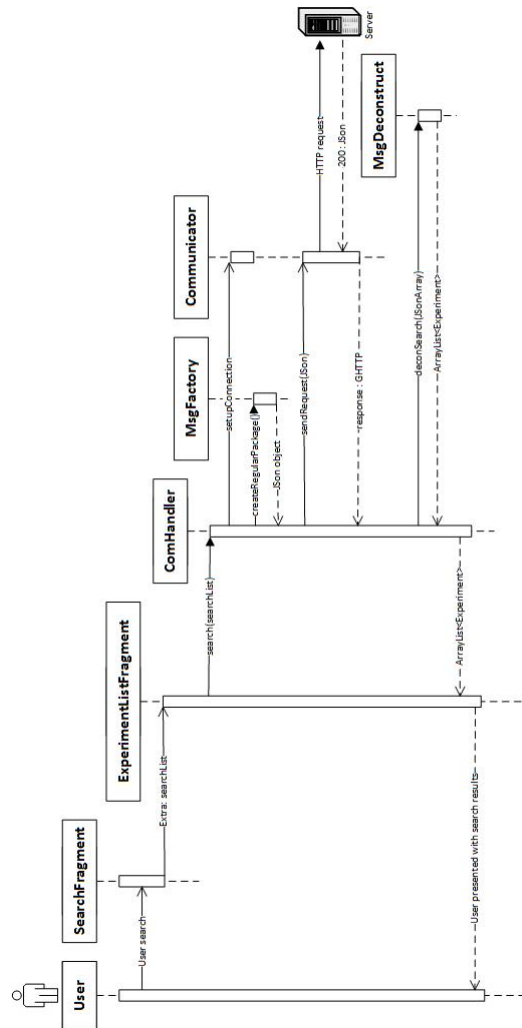


Figure 9.2: Sequence diagram for a search-request

9.3.3 Request for Genome releases from the server

In order to be able to perform a conversion the genome release version has to be supplied as a part of the parameters. This call will fetch all the available genome releases from the server to be presented to the user in the conversion menu. The flow of the retrieval of the genome releases follows these steps (see Figure 9.3).

- When the ConverterFragment is being created, the retrieval of the genome-releases is started. ConverterFragment calls the ComHandler with the getGenomReleases method and waits for response.
- The ComHandler will then initialize MsgFactory and call createRegularPackage that will return a JSON object to be used for the communication with the server. The ComHandler then initializes the Communicator and sends the JSON by calling the sendHTTPRequest method in the Communicator.
- The Communicator will setup a HTTP connection with the server and pass the JSON package to the server, which will respond with a JSON package in return. The response code and the JSON-body is then converted into a GHTTP package that is returned to the ComHandler.
- The ComHandler then will initialize the MsgDeconstruct and pass the package to that object with a call to the deconGenomeRelease method. The package is converted to an arrayList containing GenomeReleases and the passed back to the ComHandler, which the will return the arrayList to the ConverterFragment.
- The ConverterFragment then will setup a spinner with the numerous choices to be presented to the user when choosing which conversion parameters to use with a specific RAW file.

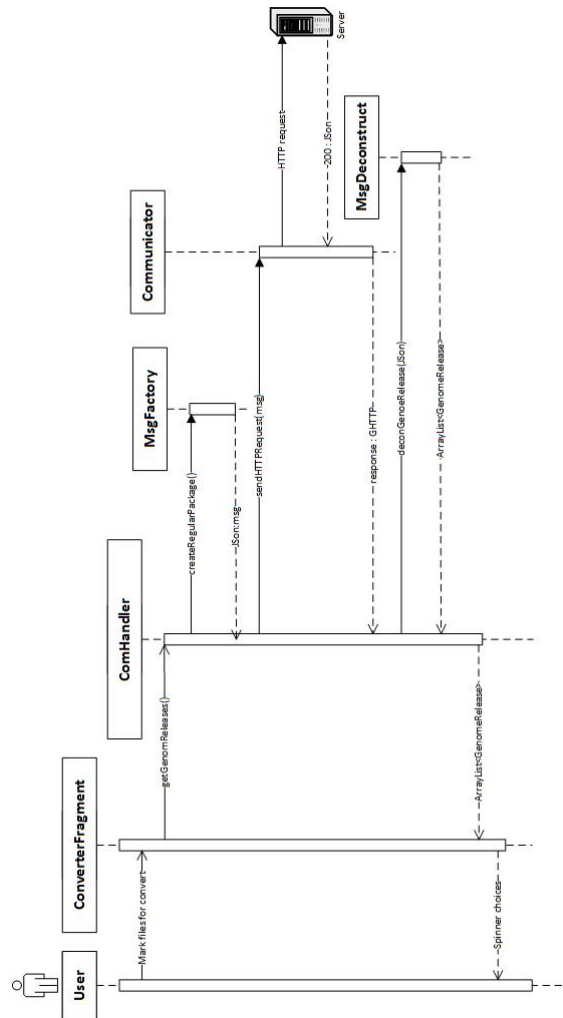


Figure 9.3: sequence diagram for a request of Genome-releases on the server

9.3.4 Request for conversion of RAW files to profile-data

When the user has chosen which conversion parameters to use and sends that request to the server, this is the flow of calls that follows from the program (see Figure 9.4).

- When the user click on the convert button, the ConverterFragment gathers all the parameters that the user selected for the conversion.
- The gathered parameters is sent to the ComHandler by calling the rawToProfile method. The Comhandler initializes MsgFactory and calls the createConversionRequest method and passing along the parameters. The MsgFactory then will return a preformatted JSON message, with the specific parmeters set to it
- The ComHandler initializes the Communicator and makes a sendHTTPrequest call with the created JSON message. The Communicator then will open a HTTP connection to the server and send the JSON message and wait for a response.
- When the response is received the Communicator will return the response as a GHTTP package to the ComHandler, which will retrieve the responsecode from the package. ComHandler then will return a boolean back to the ConverterFragment, true if the conversion started successfully otherwise false.
- The ConverterFragemnt will display the results of the started conversions to the user based upon the returned boolean from the ComHandler. If the conversion didn't start successfully the genefiles name will be displayed for the user, otherwise a toast with a summary about how many successfully started conversions will be displayed to the user.

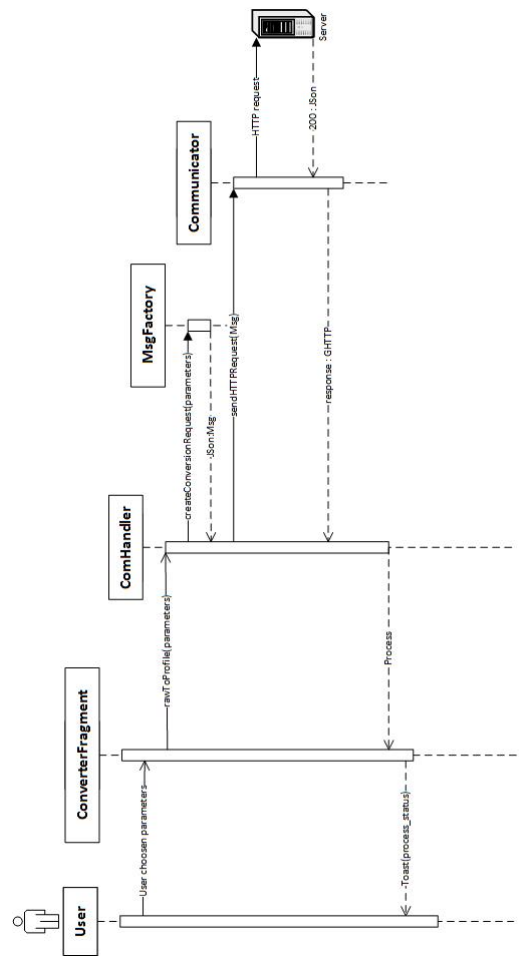


Figure 9.4: Sequence diagram for a RAW to Profile -data conversion request

9.3.5 Request for status on conversions on the server

This request is made after a conversion is sent to the server to be able to track the progress of started conversions, it can also be accessed from the menu in the application. To be able to receive current information from the server the ProcessFragment calls the following sequence on creation or when the refresh-button is pressed (see Figure 9.5).

- On creation or if the refresh-button is pressed the ProcessFragment will call the ComHandler through the getProcesses method, the call is made in a AsyncTask and will run in a separate thread.
- The ComHandler then initializes a MsgFactory and request a JSON package through the createRegularPackage method. Then initializes a Communicator and calls the sendHttpRequest method with the JSON package and a get command for the server.
- The Communicator then will setup a HTTP connection to the server and send the JSON package with a get command for the server. The response from the server is a JSON package with a response-code and a body with information, and is passed back to the ComHandler converted to a GHTTP object.
- The ComHandler then will initialize a MsgDeconstruct and call the deconProcessPackage and pass along the GHTTP object to that instance. MsgDeconstruct converts the package and returns the information in the form of an ArrayList with ProcessStatus objects.
- Then the ComHandler will pass the ArrayList to the ProcessFragment that will setup a listview with the information provided from the server, presenting information to the user about which processes there is, ETA and other information.

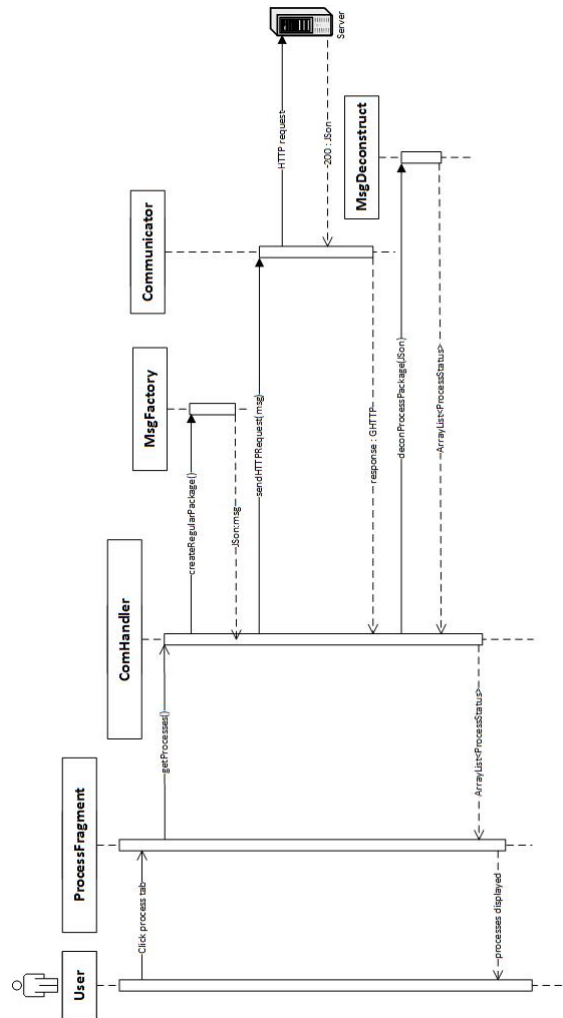


Figure 9.5: sequence diagram for a request for process status on the server

9.3.6 Testing

Testing has been done successively and the foremost type of testing has been JUnit tests. Although regular running and logs have been done too.

All fragments mentioned in subsection 8.3.1 have been tested visually and through running as no viable way of unit testing them was found.

Most of the classes labeled model in subsection 8.3.1 have been tested with a test driven development approach, except for the small classes such as Experiment, Annotation, GeneFile and GenomizerHttpPackage as they are very straight forward (And they are indirectly shown as working through the tests of the other classes).

Most tests are run against the mockup server. Albeit some are run against the real server as the authenticity of both the methods ability to handle data as well as their ability to get a response from the real server is relevant.

9.4 iOS application

The iOS application has been implemented using Objective C. The decision to use Objective C was made largely because it is the standard language used for writing iOS applications. In the following sections, details regarding the implementation of the iOS application are described using sequence diagrams. The section ends with a description of testing strategies.

9.4.1 Login

When the user tries to log in to the system, they enter username and password and clicks on the login button. The username and password is sent to the server which validates that they are valid. If they are valid, the user is presented with the main view of the application, otherwise an error message is displayed. This sequence is shown in Figure 9.6.

9.4.2 Search

Once the user is logged in to the system and they have reached the main view, they can immediately start searching by selecting a number of search criteria on the screen and pressing the search button. The search criteria are converted into a valid *PubMed*-style query which in turn is converted into a valid *HTTP request* and sent to the server. The server responds with all results matching the search criteria and the results are presented to the user in a *SearchResultView*. This sequence is shown in Figure 9.7.

9.4.3 Experiment Selection

In the *SearchResultTableView*, the user can click on an experiment to see which files are contained in an experiment. The file contents of an experiment is displayed in a *FileView*. The sequence is shown in Figure 9.8

9.4.4 File Selection

From the *FileView*, the user can select any number of files to add to a list of currently selected files by pressing the switch button next to each file name and pressing the *Add to Selected files* button. After that, the user can press the *Selected Files* button to go to the *Selected Files View*. This sequence is shown in Figure 9.9

9.4.5 Convert Request

As seen in Figure 9.10, convert requests are sent with very few steps.

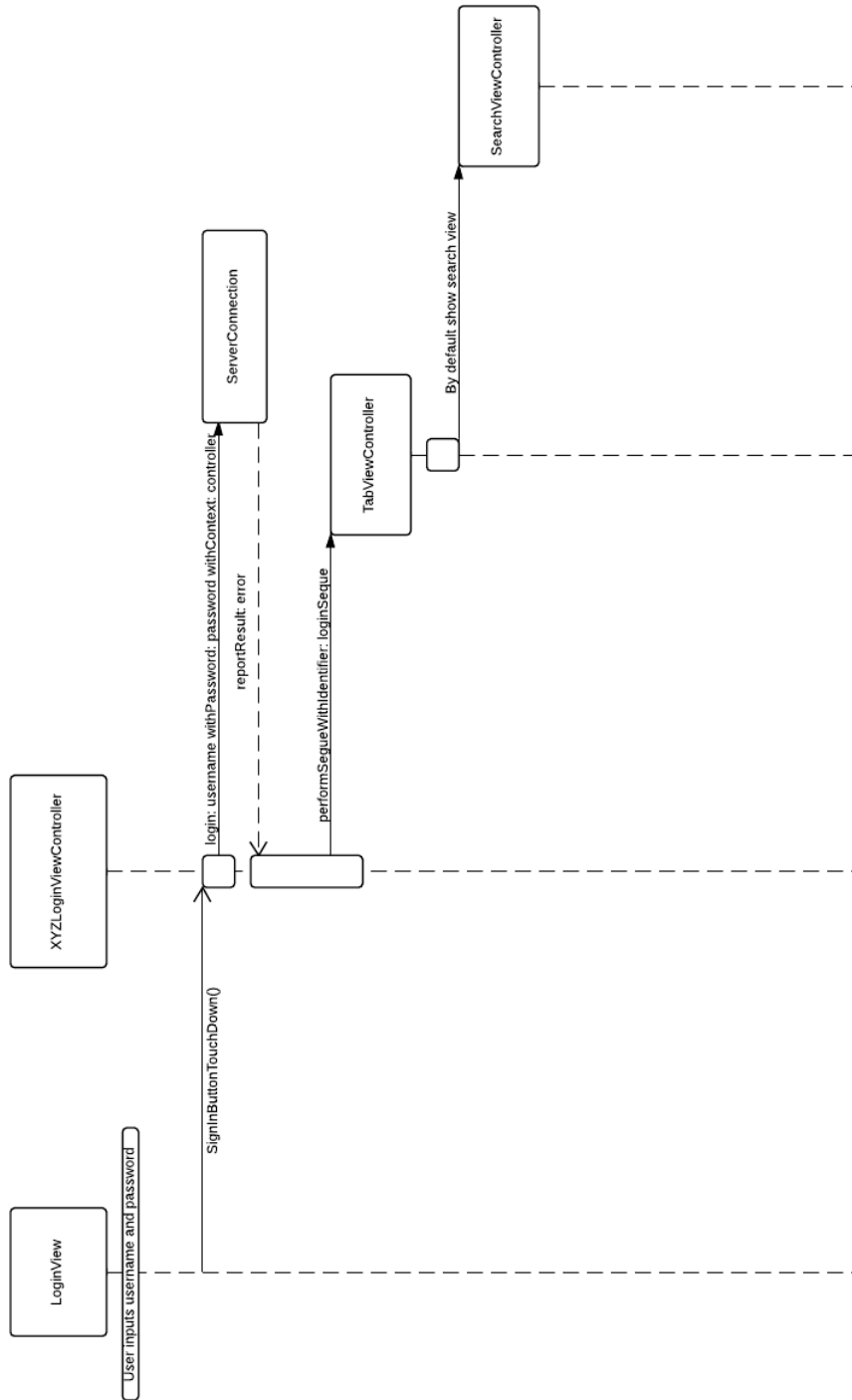


Figure 9.6: Login sequence diagram.

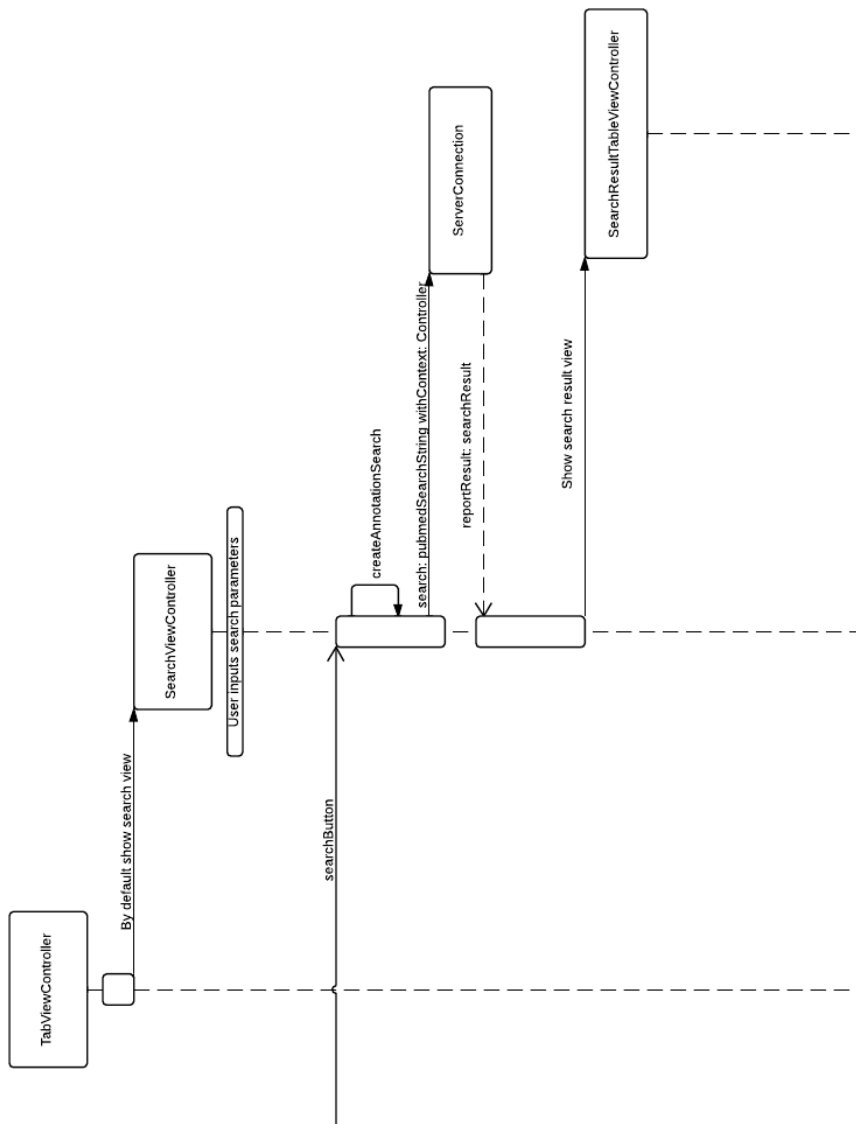


Figure 9.7: Search sequence diagram.

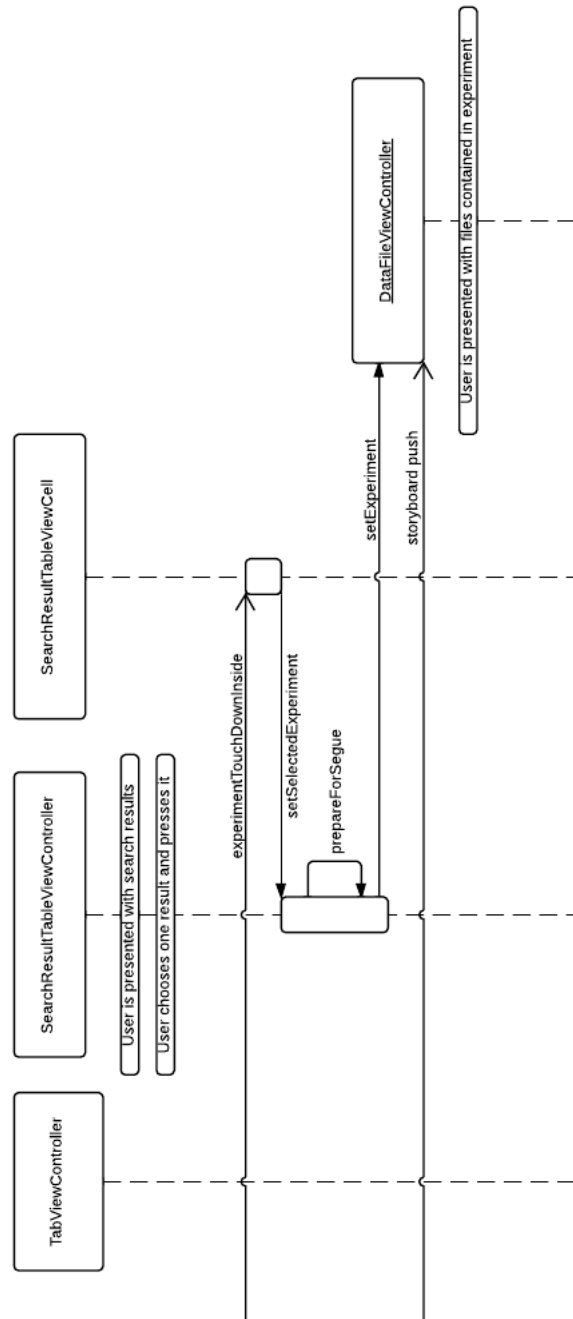


Figure 9.8: Experiment selection sequence diagram.

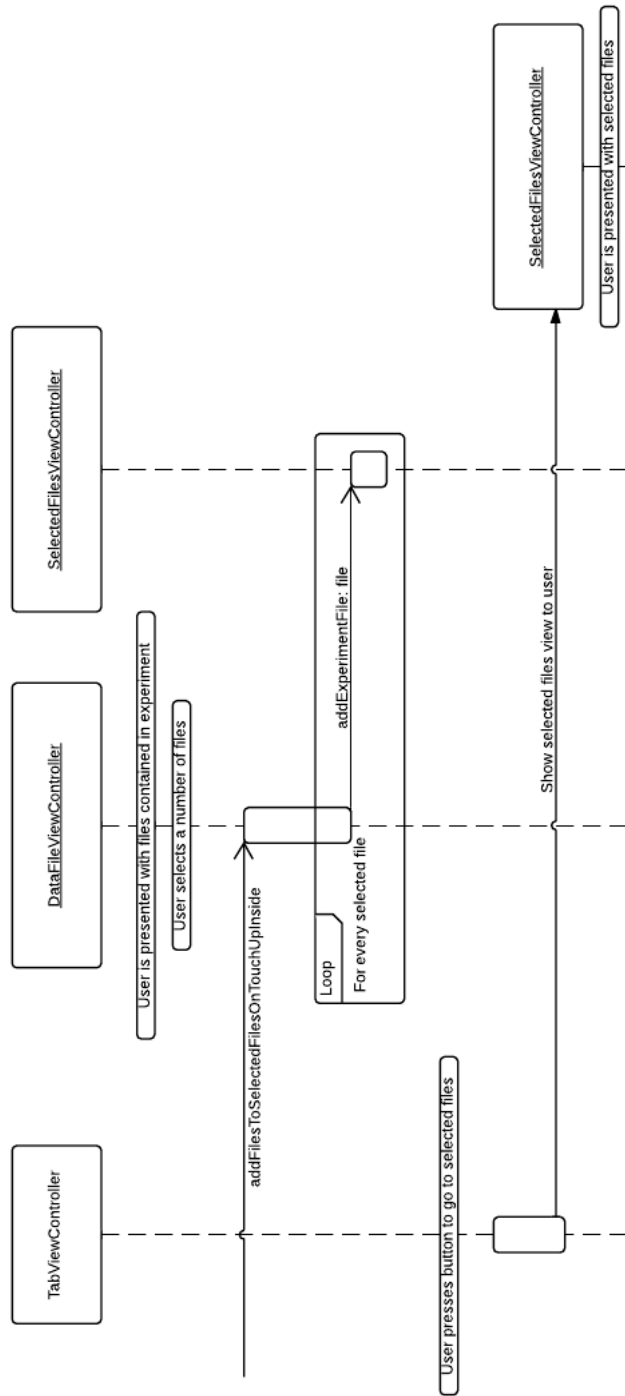


Figure 9.9: File selection sequence diagram.

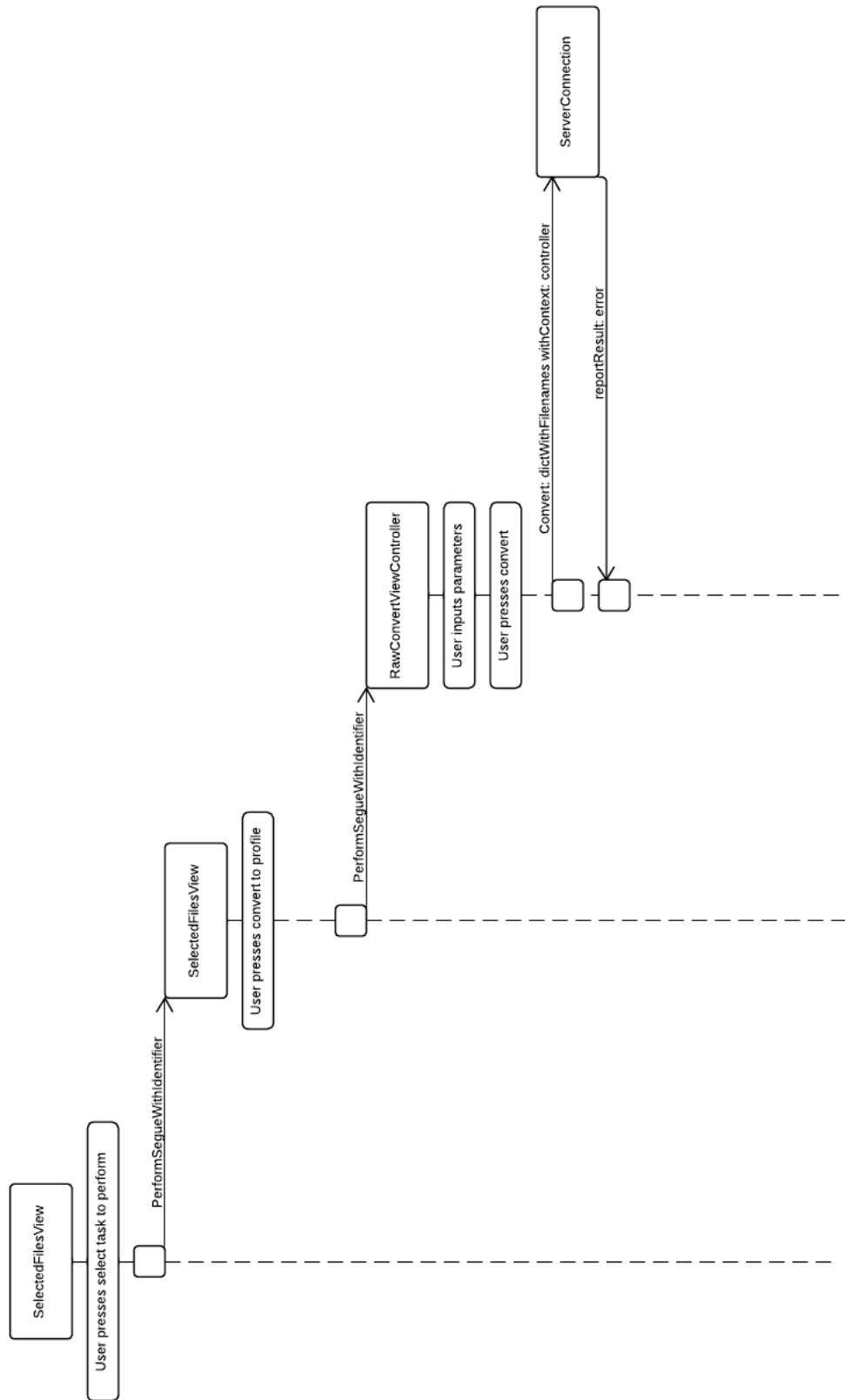


Figure 9.10: Send convert request.

9.4.5.1 Segue Control

A segue control package has been implemented to avoid multiple segues being executed at the same time. When a segue is started, a static *BOOL* is set to *YES* in *XYZSegueController*. When the segue is finished, the *BOOL* in *XYZSegueController* is set to *NO*. This means that the *BOOL* is *YES* when a segue is being executed, and *NO* otherwise. Every time a segue is going to be executed, it first checks the value of the *BOOL*. If a segue is already being executed, i.e. the *BOOL* is *YES*, the new segue is aborted.

9.4.6 Testing

We have worked using the principles of TDD, which means that Unit-tests have been written for nearly all underlying classes. This includes *JSONBuilder*, *ExperimentDescriber*, *Experiment*, *ExperimentFile*, and *ExperimentParser*. These tests check all required functionality, including but not limited to proper object creation. Exactly what all the tests do is explained in the test names and comments. *ServerConnection* has not been tested, this is work for next years workers. User interface functionality and integration testing has been done using exploratory methods. Some UI functionality has been tested using automated tests, where we recorded a certain combination of clicks, and let the computer do them instead.

9.5 Server

In this section the server and its different subsystem are displayed. Information about how the software design was realised in code will be provided.

9.5.1 Communication

This section explains implementation details of certain bits of the communication/control part of the system.

9.5.1.1 Doorman

The doorman is a class which handles all incoming connections and requests. The doorman reads the header and checks what kind of HTTP method it is (GET, PUT, PUSH, PULL or DELETE). A switch statement switch on these different methods.

After switching on the different methods another switch statement is used to switch on the different types of commands, for example */experiment*, */file*, */search* or */process*. From that point a specific command object is created

corresponding to the correct command, for example **GET** `/experiment` will create a *getExperimentCommand*.

9.5.1.2 Authorization

The communication between a client and the server is authorized by a user-unique token which is created when the user sends a login request. A token is created when a user has logged in successfully and the token is sent back to the user so that the user can thereafter use this in future requests. The token

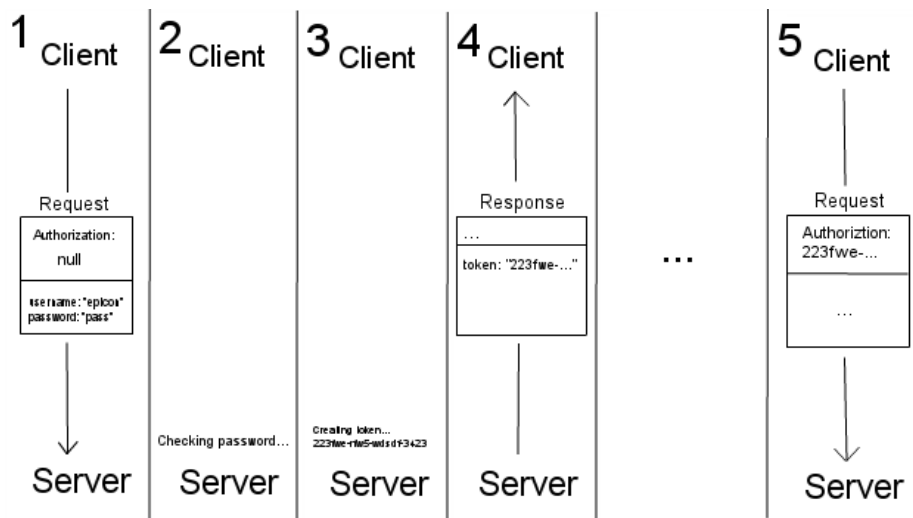


Figure 9.11: 1. The user sends a login request without any authorization token. 2. The server checks the given password. 3. The server creates a unique token for the user. 4. Server sends the token back to the user in a response. 5. Now that the user has a unique token, the token is placed in the header whenever the user sends another request.

created when a user sends a login is stored in the server memory until the user sends a logout request.

9.5.1.3 Removing inactive tokens

The server has a function which removes inactive tokens after a set limit of time. This is done because a client sometimes skips sending a logout request when shutting down the client program. The *InactiveUidsRemover* class is used to achieve this goal. In a thread it sleeps for one hour before checking all clients. If any client hasn't sent a request for 24 hours, the client token is removed from the server memory.

This feature may be turned off with the flag `"-nri"`.

9.5.1.4 Command object

The command object represent a specific command. It is created from the RESTful header and/or the JSON body sent from the client. The JSON API provides methods for automatic parsing of the JSON body into an object. The fields in the command object created must match the attributes in the JSON body. This match is case sensitive

9.5.1.4.1 Execute

Every command object must implement a execute method. This method is the part of the command which uses the system interface to perform the task that corresponds to the command.

The execute method returns a response object which is sent up to the door-man which then sends the response to the client.

9.5.1.4.2 Validation

Every command must implement a validate method. This method is run after the command is created but before the command is executed.

The validate method returns a boolean. If the command is correctly parsed with correct data the method returns true, otherwise false. This validate is used to prevent unnecessary communication.

9.5.1.5 Heavy work thread

For heavy work a queue, namely work handler is used. The command which is put in this queue is ProcessCommand. All the command objects which is in the queue, are executed one at a time in the order first in first out. This execution is done by another thread with the only responsibility to do this kind of heavy work. The thread constantly checks whether the queue is empty or not and if there is a command object in the queue the thread polls the command object and executes it.

Which command that is put in the queue or not is determined in the method processNewCommand in the class CommandHandler.

9.5.1.6 Response object

There are different response objects for different kind of responses since the form of the response to the client depends on the command the client initially sent.

The response object contains all the data necessary to create a RESTful header and a JSON body for the response.

9.5.1.7 Testing

Testing has been done in multiple steps. The first step is unit testing, where individual methods are tested. This is often difficult due to the fact that the responsibility of handling client requests is shared by multiple classes. To catch these test cases a client dummy has been frequently used, which is the next step. It simulates a client by sending HTTP requests and examines the response from the server. It is used manually to test a particular use case, and to see that the server behaves as intended for that request. After a feature has passed the client dummy it is pushed to a test server, where it is open for other clients to test and debug. If no bugs are found the feature is declared complete and can be released.

9.5.2 Conversion

This section will explain the implementation of the *SmoothingAndStep* subroutine used in the conversion of files from *raw* to *profile*. The basic algorithm is a dynamic arrayList which carries the rows that are relevant at a given time, smoothing on the first row is performed. The newly smoothed value is shifted out and replaced with a fresh row. This becomes a dynamic window that traverses the entire file one row at a time.

9.5.2.1 Methods

- **smoothing** : The one public method of the class. It controls the whole process and calls the other methods. It takes in the following parameters:
 - **int[] params**: An array with 5 integers representing parameters.
 - params[0]: Window Size, the number of signal values that the smoothing should be calculated on.
 - params[1]: Whether the smoothing should be trimmed mean (0) or median (1)
 - params[2]: Minimum numbers to smooth. A number that says how many signal values the program at least need in order to smooth one row. This number must be smaller than windowSize.
 - params[3]: Can either be 1 or 0. If 1 the program will calculate the total mean value for all rows and print those.
 - params[4]: Print zeroes. If the program should print rows where the signal value is 0 the flag should be (1), if (0) the program will not print the zeroes.
 - **String inPath**: A filepath to the source file.

- String `outPath`: A filepath to either an existing file to be overwritten or of a location and name that will become the path to a newly created file.
- int `stepSize`: An integer larger than 0 that tells if there should be stepping. No stepping will be done if the number is 1.

The method will also return the total mean of every row in the file if that flag is set properly.

- `smoothOneRow`: Checks whether smoothing should be trimmed mean or median and calls the corresponding method, after this is done it calls the method that writes to the new file.
- `smoothTrimmedMean`: Extracts the first position from the data array and initiates it's value to min and max values. We do this because trimmed mean means that we should remove the largest and smallest number from the mean value in order to get a more reliable/stable result. We then check that we have more numbers in the data array than the minimum numbers to smooth number. In order to avoid doing unnecessary calculations.
- `smoothMedian`: This method tries to fill an array with window size number of signal values and then pass this array to a method that finds which number is the median.
- `writeToFile`: This method does three different things. It check whether we should print zeroes in the outfile. It also check whether the current position is divisible with `stepSize` to determine if the row should be written to the outfile or skipped. After these two checks it either writes the row to the new file or not.

It also check whether we want to print the total mean of the whole file and/or if we should then it counts up the proper variables.

- `shiftLeft`: Removes the first row from the data array and adds one row to the end of it. It then checks whether the new row is of a different chromosome than the others, if so it calls the special method `chromosomeChange`.
- `chromosomeChange`: This method knows that the last element in the data array is a new chromosome. It then reads and smooths as many rows as it can before hitting the cutoff number (minimum number of rows to smooth). It then writes and removes these values from the data array as well. It's important to note that so far it doesn't add new values to the array. Afterwards the method tries to refill the array with the new chromosome until it has window size number of rows.

9.5.3 File-transfer

To handle all downloads and uploads to and from the clients, two PHP-scripts have been written.

Both scripts use a token provided by the client to authenticate the user. This token is sent to the server, which will send a code back to the script. The code

will be '200' if the client has provided a valid token, and '401' if the token is invalid. The upload script gets the token in an 'Authorization' header from all clients, while the download script gets the token in an 'Authorization' header from the desktop and mobile clients and in an 'Authorization' parameter from the web client.

When the client downloads or uploads a file, it will send a path to the script in a 'path' parameter. This path will be validated against the database. It will check if the file is 'Done' when downloading and that the file is not 'Done' when uploading.

When an upload has been finished and validated, the script will change the status for this file to 'Done' and then send a '201' response code to the client. When a download request has been validated the script will send the file as an octet-stream as a response to the client.

9.5.4 Data Storage

The following text describes the different classes the server uses to communicate with the database and update the file system.

9.5.4.1 Annotation

When a user wants the properties of an *annotation* they will retrieve this object. It holds information about the data type, label, forced annotation, default value and annotation choices for drop down menus. With this object the graphical interfaces can set up a search view dynamically.

9.5.4.2 Experiment

A container of an experiments annotations and their values. The annotation labels and values are contained in a *hashmap* with label as key. All files corresponding to the actual experiment lies in a list of *FileTuple* objects.

9.5.4.3 FileTuple

A simple class that holds all information of a file in variables found in the database. Holds links for download and upload of the specific file.

9.5.4.4 FilePathGenerator

The creation of folders is handled by the `FilePathGenerator` class. When a new experiment is added to the database a folder is also created for the new

experiment. Sub folders are also created in preparation for the uploading of files.

When a user requests to upload a file, the *FilePathGenerator* class will, if required, generate a new folder to house the file.

When a new *genome release* is added to the database, the *FilePathGenerator* will create a folder to house the associated files. Genome releases are divided into folders corresponding to species.

A new folder is also created for each process request and houses the resulting file set.

9.5.4.5 PubMedToSQLConverter

This class converts a *PubMed* string to an *SQL query*. A typical *PubMed* search string takes the form: `raw[FileType] AND Per[Author]`. In this case the search is for *all raw files that Per created*. The user enters the annotation labels and values together with the logical operators *AND*, *OR* and *NOT*. Parentheses are used for disambiguation.

All the variables in the *pubMed* string are bound to variables in the *WHERE* section of the *SQL* query to avoid *SQL injection*.

When the `search` method is called in the *DatabaseAccessor*, the *PubMed* string is checked for file attributes by calling the `hasFileAttributes` method in the *PubMedToSQLConverter*. This is done so that even empty experiments are returned when searches do not contain a file specific attribute.

9.5.4.6 DatabaseAccessor

A class that serves as an API for the server and is used for all database access. It handles all connections and queries to the database. The methods simplify queries to the database by removing the need to write *SQL* in any other packages. The *DatabaseAccessor* utilizes its subclasses as shown in ?? where the methods are divided up into more specific areas. *Prepared Statements* are created from queries and parameters to avoid *SQL injection*. Methods that have some resemblance are grouped together for easier navigation.

9.5.4.7 Testing

OBS! Do not run any of the tests found in the `database` package on a database that is in use. All tuples are removed from the database upon completion of testing. All unit testing should start with an empty database and finish with an empty database to avoid dependency between tests.

The database package was created using *Test Driven Development* (TDD). The full test suite, `AllTests`, can be found in the `database.TestSuite` package.

Most of the unit tests utilize the `TestInitializer` class. This simplifies the process of connecting to the test database, filling it with test tuples and clearing the test database and closing the connection when the test class is finished. The individual unit tests can be found in the `database.TestSuite.UnitTests` package. The scripts for adding the test tuples and clearing the test database tables can also be found in the `database.TestSuite` package.

Stress tests are also implemented with the goal to determine functionality by simulating multiple users working in parallel.

9.6 Limitations

The *Genomizer* system has some limitations and known problems that needs to be mitigated. In Appendix K a detailed description of known problems for each of the *Genomizer* subsystems are listed.

Bibliography

- [1] Langmead, Ben and Trapnell, Cole and Pop, Mihai and Salzberg, Steven L and others. *Ultrafast and memory-efficient alignment of short DNA sequences to the human genome*. *Genome Biol* 10(3). 2009.
- [2] Zhan, Xiaowei. "LiftOver". *Center For Statistical Genetics*. February 14, 2014. Web. May 30, 2014. <<http://genome.sph.umich.edu/wiki/LiftOver>>
- [3] Norris, David. "Integrated Genome Browser". *Bio Viz*. Web. May 31, 2014. <<http://bioviz.org/igb/>>
- [4] *technoweenie*. "Release Your Software". *GitHub*. July 2, 2013. Web. May 29, 2014. <<https://github.com/blog/1547-release-your-software>>
- [5] Preston-Werner, Tom. "Semantic Versioning 2.0.0". Web. May 29, 2014. <<http://semver.org/>>
- [6] *National Center for Biotechnology Information*. "PubMed Advanced Search Builder". *U.S. National Library of Medicine*. October 28, 2009. Web. May 29, 2014. <<http://www.ncbi.nlm.nih.gov/pubmed/advanced>>
- [7] "Authentication and Authorization". *The Apache Software Foundation*. Web. May 21, 2014. <<http://httpd.apache.org/docs/2.2/howto/auth.html>>
- [8] Dudler, Roger. "git - the simple guide". Web. May 29, 2014. <<http://rogerdudler.github.io/git-guide/>>
- [9] Davis, Adam. "Git for beginners: The definitive practical guide". *Stackoverflow*. May 21, 2012. Web. May 29, 2014. <<http://stackoverflow.com/questions/315911/git-for-beginners-the-definitive-practical-guide>>
- [10] "Generating SSH Keys". *Github*. May 16, 2014. Web. May 29, 2014. <<https://help.github.com/articles/generating-ssh-keys>>
- [11] Backbone.js documentation: <http://backbonejs.org/> Retrieved 8/5 -14
- [12] Bootstrap documentation: <http://getbootstrap.com/> Retrieved 8/5 -14
- [13] AJAX on wiki: [http://en.wikipedia.org/wiki/Ajax_\(programming\)](http://en.wikipedia.org/wiki/Ajax_(programming)) Retrieved 8/5 -14

- [14] JSON on wiki: <http://en.wikipedia.org/wiki/JSON> Retrieved 8/5 -14
- [15] RequireJS documentation: <http://requirejs.org/> Retrieved 8/5 -14
- [16] JQuery documentation: <http://jquery.com/> Retrieved 8/5 -14
- [17] Chai documentation: <http://chaijs.com/> Retrieved 9/5 -14
- [18] Mocha documentation: <http://visionmedia.github.io/mocha/> Retrieved 9/5 -14
- [19] Sinon documentation: <http://sinonjs.org/> Retrieved 9/5 -14
- [20] "List of UCSC genome releases". *UCSC*. Web. May 30, 2014. <<https://genome.ucsc.edu/FAQ/FAQreleases.html>>

Appendix A

User Stories

A *User Story* is a description of functionality in non technical terms. It describes the wishes of a certain user group and a motivation for why the function is needed.

A.1 Implemented user stories

Annotation
To structure the data files the researchers want to be able to annotate the data files.
Single download
To scrutinize a single data file the researchers want to be able to download a specific file.
Single upload
To store a single data file the researchers want to be able to upload a specific file.

Search for data

To analyse data
the researchers
want to be able to search for specific types of data.

Batch upload

To analyse, share and have greater access to data
the researchers
want to be able to upload multiple files and batch annotate them to a shared location.

Raw to profile

To be able to analyze
the researchers
want to process raw data to profile data (using bowie and then Philge's code).

Delete data

To save space
the researchers
want to be able to delete data from the database.

File traceability

To be able to access the underlying raw data or profile data
the researchers
want the raw data files to be traceable from profile files and the profile files to be traceable from the region data (if available) when the files have been generated on the server.

Change annotation

To correct and update annotations
the researchers
want to be able to change data annotations.

Backup

To prevent loss of data
the researchers
want the data to be backed up.

Password protected

To protect the database from unauthorized use
the researchers
want the application to be password protected.

Add genome release / reference genome

To be able to annotate the data properly and extract genome reference
the researchers
want to be able to add genome releases and reference genome.

Add chain file

To be able to convert between genome releases
the researchers
want to upload chain files (LiftOver).

Batch download

To scrutinize several data files
the researchers
want to be able to download multiple files at once.

A.2 Product backlog

Convert common file formats

To get data in a certain convenient file format
the researchers
want to convert between common file formats (WIG, SGR, GFF3, BED).

Convert genome release

To easier handle files
the researchers
want to convert files between genome releases (LiftOver).

Extract genome reference sequence

To analyze the reference genome
the researchers
want extract the reference genome sequence for a given region data.

Advanced batch upload

To simplify mass upload 500 files
the researchers
want to batch annotate files to be uploaded in a spreadsheet.

Profile to region

To be able to find regions of interest
the researchers
want to process profile data to region data (Per's code).

Workspace

To be able to save work in a convenient way
the researchers
want to have some sort of workspace view where all kind of results/data can
be saved.

Unread results

To avoid missing results
the researchers
want to see which results are unread.

Sort search results

To avoid missing results
the researchers
want to see which results are unread.

Preview of file

To correctly annotatate a file
the researchers
want to preview a portion of a file

Work scheduling

To strategically spread the servers workload over time
the researchers
want to be able to schedule the processing/analysis of data

Work queue

To reduce server load
the researchers
want to queue time consuming work.

User rights

To allow invitation of guests (postgraduate students or other researchers
etc.)
the researchers
what to have different users types with different rights.

Time estimation

To warn for time consuming jobs
the researchers
want to have a time estimation for jobs.

Plot overlap analysis

To see region overlap of genomes
the researchers
want to plot an overlap analysis (see separate user story)

Plot average regions

To view data
the researchers
want to plot average of regions with the profile data.

IGB Session

To be able to make IGB analyzes
the researchers
want to retrieve a IGB session file.

Combine regions

To find interesting regions
the researches
want to select multiple files and combine their regions (union, intersect).

Create region subset

To retrieve certain parts of regions
the researchers
want to create region subsets using reference points.

Calculate average of region

To find the average protein binding value for a region
the researchers
want to calculate average of regions with the profile data. (Possibly split into a number of bins).

Overlap analysis

To conduct overlap analysis
the researchers
want to divide regions bins, either by value or by order.

Save analysis results

To be able to return to previous work
the researchers
want to save analysis results(in workspace).

Appendix B

Android application: *UML*-diagrams

In this appendix the *UML*-Class-diagrams of the Android application will be presented.

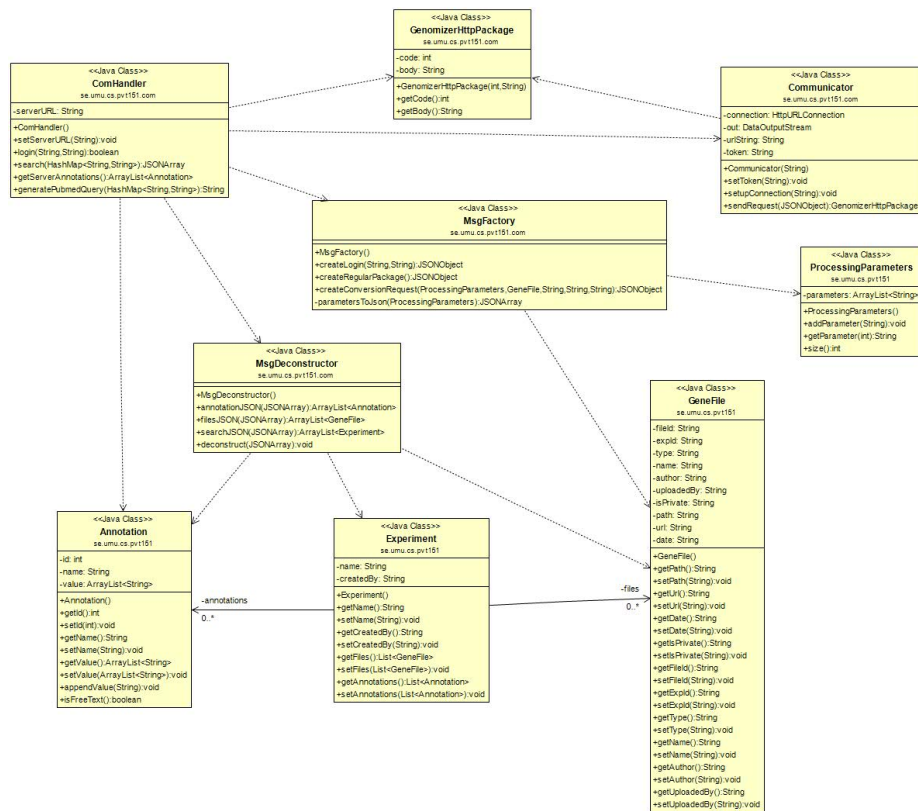


Figure B.1: Android UML of model

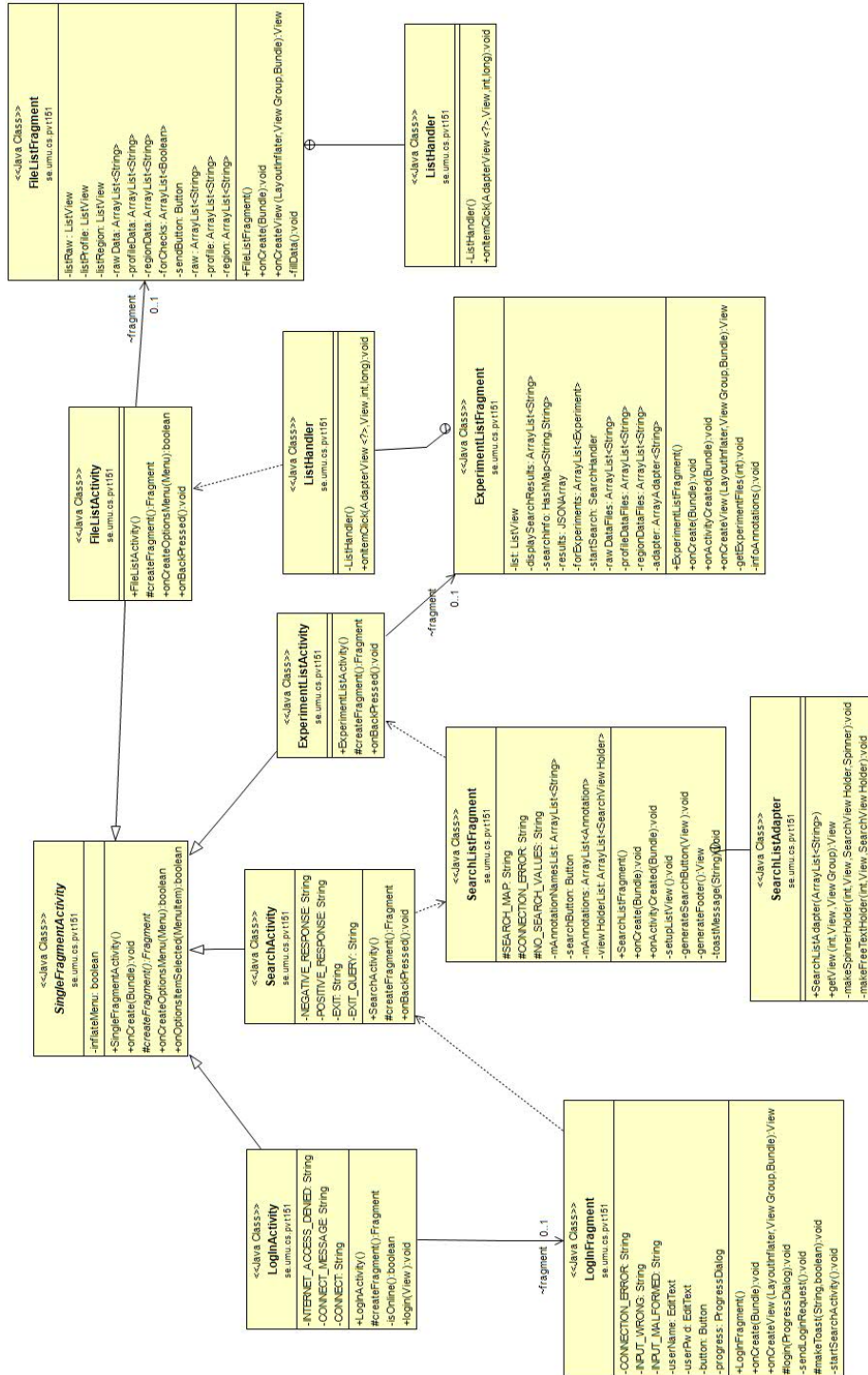


Figure B.2: Android UML without model

Appendix C

Desktop application: *UML*-diagrams

Appendix D

Data Storage: *UML*-diagrams

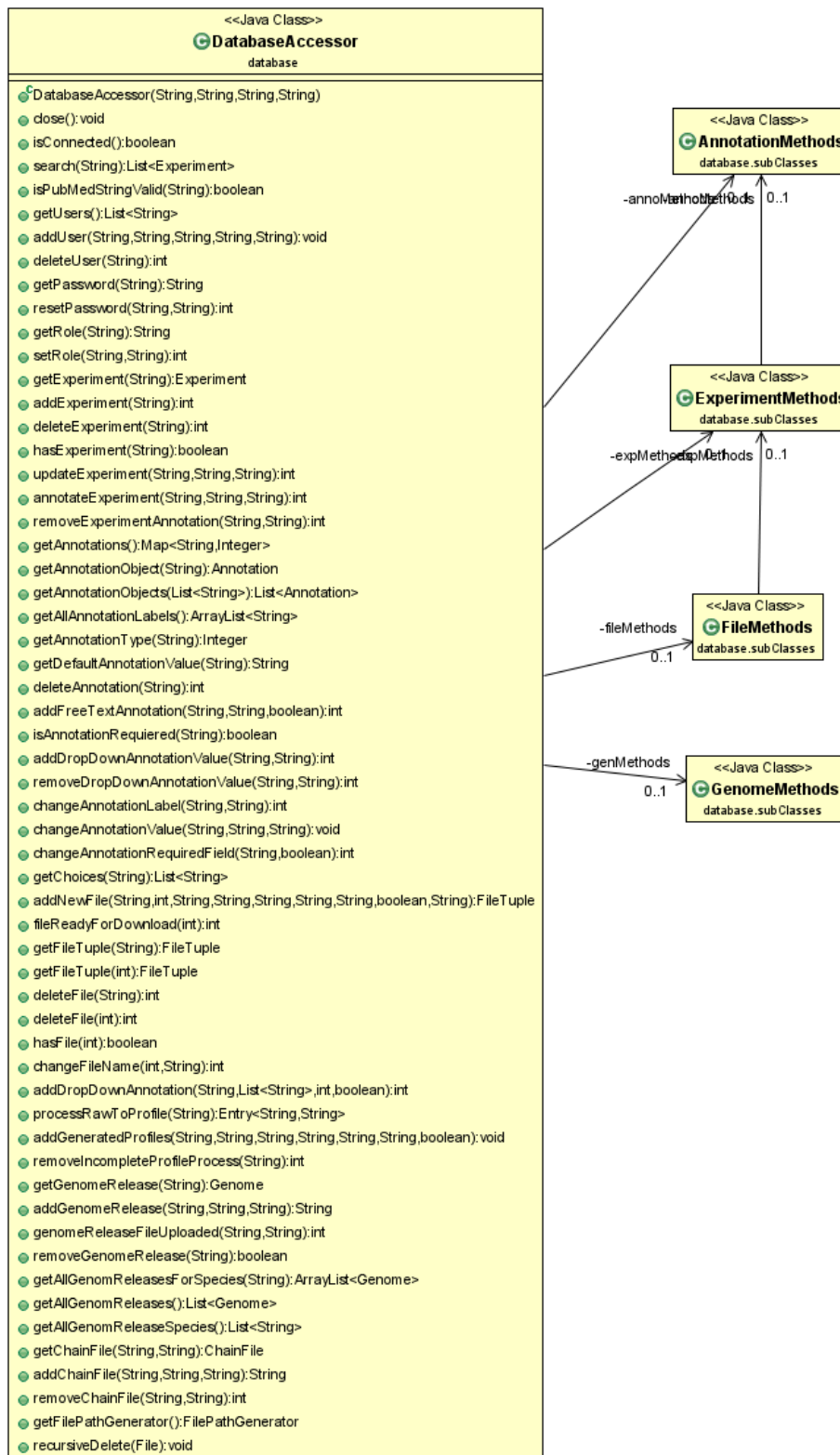


Figure D.1: DatabaseAccessor and its subclasses

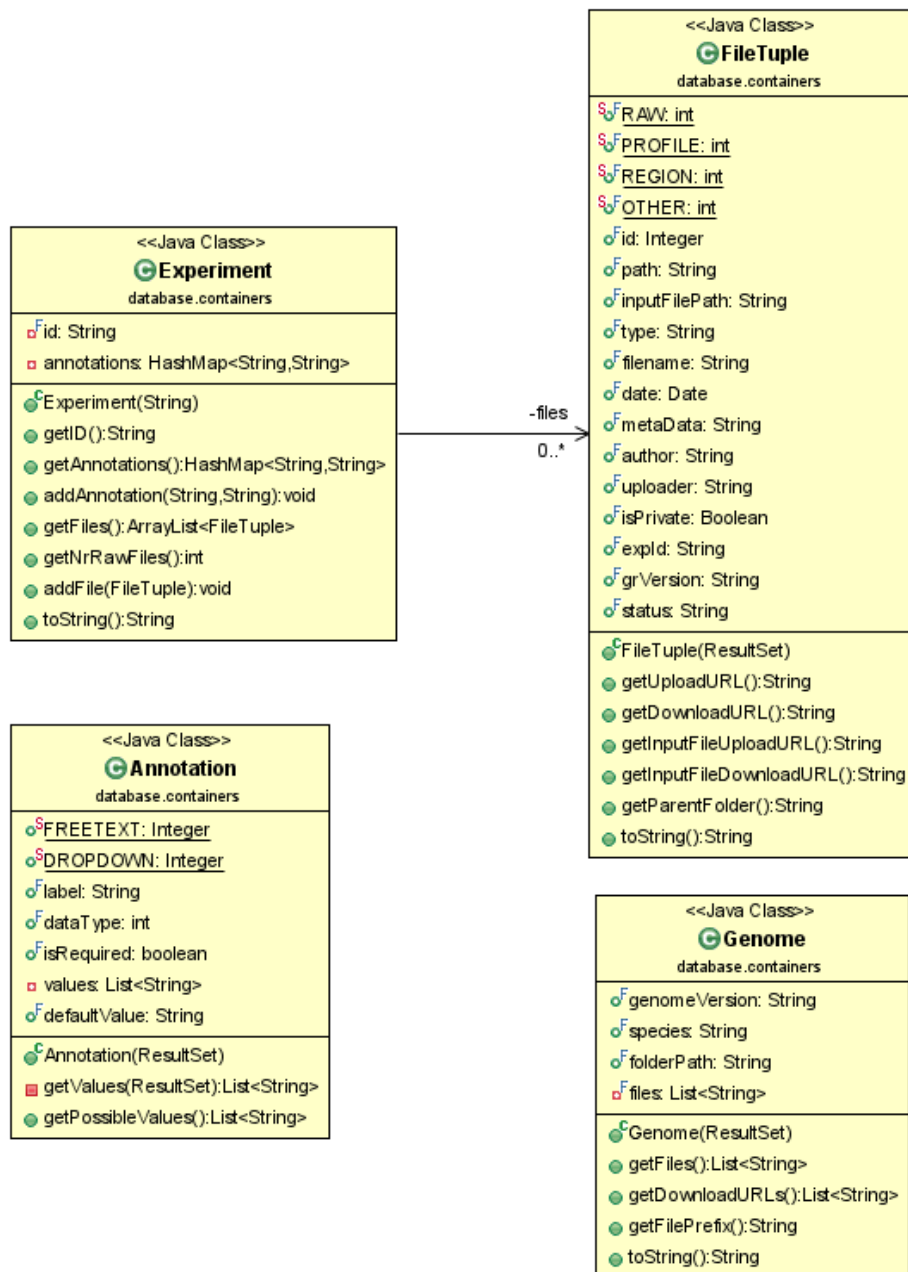


Figure D.2: The container classes used to return information

Appendix E

Server API

Connection

Login/Logout to and from the server. When a user has been verified a token (user-id) is supplied in the response. The token is generated from the current date and the users password. It is then hashed and given an expiration date. The token should be supplied in the Authorization header for each request in order to identify the user.

\login

Login to the server [POST]
+ Request (application/json)

```
{
  "username": "uname",
  "password": "pw"
}
```

+ Response 200 (application/json)

```
{
  "token": "user-id"
}
```

Logout from the server [DELETE]
+ Request

+ Header

```
authorization: user-id
```

+ Response 200

Experiment

An experiment containing annotations and files. 'experiment-id' in the header of the request is the unique id (name) of the experiment.

```
+ Parameters
  + name ... Name and id of the experiment
  + created by ... Which user created the experiment
  + annotations
    + pubmedId
    + type
    + specie
    + genoRelease
    + cellLine
    + devStage
    + sex
    + tissue
    + ...
```

\experiment

Add an Experiment [POST]

+ Request (application/json)

+ Headers

Authorization: token

+ Body

```
{
  "name": "experimentId",
  "createdBy": "user",
  "annotations":
  [
    {
      "name": "pubmedId",
      "value": "abc123"
    },
    {
      "name": "type",
      "value": "raw"
    },
    {
      "name": "specie",
      "value": "human"
    },
  ],
}
```

```
        {
          "name": "genome release",
          "value": "v.123"
        },
        {
          "name": "cell line",
          "value": "yes"
        },
        {
          "name": "development stage",
          "value": "larva"
        },
        {
          "name": "sex",
          "value": "male"
        },
        {
          "name": "tissue",
          "value": "eye"
        }
      ]
    }
  }
}

+ Response 201
```

`\experiment\<experiment-id>`

Retrieve an Experiment [GET]

+ Request

+ Headers

Authorization: token

+ Response 200 (application/json)

+ Headers

Authorization: token

+ Body

```
{
  "name": "experimentId",
  "createdBy": "user",
  "files": [
    {
```

```
    "fileId": "id",
    "experimentID": "id",
    "fileName": "name",
    "type": "raw",
    "metaData": "metameta",
    "author": "name",
    "uploader": "user1",
    "isPrivate": "bool",
    "grVersion": "releaseNr",
  },
  {
    "fileId": "id",
    "experimentID": "id",
    "fileName": "name",
    "type": "raw",
    "metaData": "metameta",
    "author": "name",
    "uploader": "user1",
    "isPrivate": "bool",
    "grVersion": "releaseNr"
  }
],
"annotations":
  [
    {
      "name": "pubmedId",
      "value": "abc123"
    },
    {
      "name": "type",
      "value": "raw"
    },
    {
      "name": "specie",
      "value": "human"
    },
    {
      "name": "genome release",
      "value": "v.123"
    },
    {
      "name": "cell line",
      "value": "yes"
    },
    {
      "name": "development stage",
      "value": "larva"
    },
    {
      "name": "sex",
```

```
        "value": "male"
      },
      {
        "name": "tissue",
        "value": "eye"
      }
    ]
  }

### Update an Experiment [PUT]
+ Request (application/json)

+ Headers

    Authorization: token

+ Body

{
  "name": "experimentId",
  "createdBy": "user",
  "annotations":
  [
    {
      "name": "pubmedId",
      "value": "abc123"
    },
    {
      "name": "type",
      "value": "raw"
    },
    {
      "name": "specie",
      "value": "human"
    },
    {
      "name": "genome release",
      "value": "v.123"
    },
    {
      "name": "cell line",
      "value": "yes"
    },
    {
      "name": "development stage",
      "value": "larva"
    },
    {
      "name": "sex",
      "value": "male"
    }
  ]
}
```

```
    },  
    {  
      "name": "tissue",  
      "value": "eye"  
    }  
  ]  
}
```

+ Response 201

Remove an Experiment [DELETE]

+ Request

+ Header

Authorization: token

+ Response 200

Files

Add/remove files in experiment. 'file-id' specifies the unique id of the file in the header.

+ Parameters

+ 'fileName' ... Name of the file

+ 'experimentID' ... Name of the experiment associated with the file

+ 'size' ... File size

+ 'type' ... Type of data (raw/profile/region)

+ ('URL' ... An URL to the file, added when the file has been uploaded)

\file

Add file to experiment [POST]

+ Request (application/json)

+ Header

Authorization: token

+ Body

```
{  
  "experimentID": "id",  
  "fileName": "name",  
}
```

```
    "type": "raw",
    "metaData": "metameta",
    "author": "name",
    "uploader": "user1",
    "isPrivate": "bool",
    "grVersion": "releaseNr"
  }
```

+ Response 200

```
{
  "URLupload": "url"
}
```

\file\<<file-id>

Get file from experiment [GET]

+ Request

+ Headers

Authorization: token

+ Response 200 (application/json)

```
{
  "experimentID": "id",
  "fileName": "name",
  "type": "raw",
  "metaData": "metameta",
  "author": "name",
  "uploader": "user1",
  "isPrivate": "bool",
  "grVersion": "releaseNr"
}
```

Update file in experiment [PUT]

+ Request (application/json)

+ Header

Authorization: token

+ Body

```
{
  "experimentID": "id",
```

```
    "fileName": "name",
    "type": "raw",
    "metaData": "metameta",
    "author": "name",
    "uploader": "user1",
    "isPrivate": "bool",
    "grVersion": "releaseNr"
  }
```

+ Response 201

Delete file from experiment [DELETE]

+ Request

+ Header

```
Authorization: token
```

+ Response 200

Search

Searching using annotations. The annotations is included last in the request header. The results from the search is contained in the JSON document in the response. Results are an array of **files** linked to their respective **experiments**.

`\search\?annotations=<pubmedStyleQuery>`

Search for experiments [GET]

+ Request

+ Headers

```
Authorization: token
```

+ Response 200 (application/json)

```
[
  {
    "name": "experimentId",
    "created by": "user",
    "files": [
      {
        "id": 25,
        "path": "/var/www/data/Exp1/raw/file1.fastq",
```



```
    "url": "http://scratchy.cs.umu.se:8000/download.php?path\u003d/var/www",
    "type": "Raw",
    "filename": "file1.fastq",
    "date": "May 8, 2014",
    "author": "Ume? Uni",
    "uploader": "user1",
    "expId": "Exp1"
  },
  {
    "id": 26,
    "path": "/var/www/data/Exp1/raw/file1.fastq",
    "url": "http://scratchy.cs.umu.se:8000/download.php?path\u003d/var/www",
    "type": "Raw",
    "filename": "file1.fastq",
    "date": "May 8, 2014",
    "author": "Ume? Uni",
    "uploader": "user1",
    "expId": "Exp1"
  },
  {
    "id": 27,
    "path": "/var/www/data/Exp1/raw/file1.fastq",
    "url": "http://scratchy.cs.umu.se:8000/download.php?path\u003d/var/www",
    "type": "Raw",
    "filename": "file1.fastq",
    "date": "May 8, 2014",
    "author": "Ume? Uni",
    "uploader": "user1",
    "expId": "Exp1"
  }
],
"annotations":
[
  {
    "name": "pubmedId",
    "value": "abc123"
  },
  {
    "name": "type",
    "value": "outdoor"
  },
  {
    "name": "specie",
    "value": "human"
  },
  {
    "name": "genome release",
    "value": "v.123"
  },
  {
```

```
        "name": "cell line",
        "value": "yes"
    },
    {
        "name": "development stage",
        "value": "larva"
    },
    {
        "name": "sex",
        "value": "male"
    },
    {
        "name": "tissue",
        "value": "eye"
    }
]
}
```

Processing

API for executing commands such as file conversions.

`\process`

Get status of all processes [GET]

+ Request

+ Header

Authorization: token

+ Response 200 (application/json)

```
[
  {
    "experimentName": "Exp1",
    "status": "Finished",
    "outputFiles": [
      "file1",
      "file2"
    ],
    "author": "yuri",
    "timeAdded": 1400245668744,
    "timeStarted": 1400245668756,
    "timeFinished": 1400245669756
  }
]
```

```
    },
    {
      "experimentName": "Exp2",
      "status": "Finished",
      "outputFiles": [
        "file1",
        "file2"
      ],
      "author": "janne",
      "timeAdded": 1400245668746,
      "timeStarted": 1400245669756,
      "timeFinished": 1400245670756
    },
    {
      "experimentName": "Exp43",
      "status": "Crashed",
      "outputFiles": [
        "file1",
        "file2"
      ],
      "author": "philge",
      "timeAdded": 1400245668748,
      "timeStarted": 1400245670756,
      "timeFinished": 1400245671757
    },
    {
      "experimentName": "Exp234",
      "status": "Started",
      "outputFiles": [
        "file1",
        "file2"
      ],
      "author": "per",
      "timeAdded": 1400245668753,
      "timeStarted": 1400245671757,
      "timeFinished": 0
    },
    {
      "experimentName": "Exp6",
      "status": "Waiting",
      "outputFiles": [],
      "author": "yuri",
      "timeAdded": 1400245668755,
      "timeStarted": 0,
      "timeFinished": 0
    }
  ]
}
```

`\process\rawtoprofile`

Parameters

- [1] Bowtie parameters
- [2] Empty string
- [3] `y/` - If you want the file in GFF format
- [4] `y/` - If you want the file in SGR format
- [5] Smoothing parameters 1
- [6] `y/` X - If you want stepping parameters and with stepsize X
- [7] Ratio calculation parameters
- [8] Smoothing parameters 2

Convert a file from raw to profile [PUT]

+ Request

+ Header

Authorization: token

+ Body

```
{
  "expid": "Exp1",
  "parameters": [
    "-a -m 1 --best -p 10 -v 2 -q -S",
    "",
    "y",
    "n",
    "10 1 5 0 0",
    "y 10",
    "single 4 0",
    "150 1 7 0 0"
  ],
  "metadata": "astringofmetadata",
  "genomeVersion": "hg38",
  "author": "yuri"
}
```

+ Response 200

Annotation handling

,

Used to add, modify and delete annotations.

`\annotation`

Get information about annotations [GET]

+ Request

+ Header

Authorization: token

+ Response 200 (application/json)

```
[
  {
    "name": "pubmedId",
    "values": ["freetext"],
    "forced": true
  },
  {
    "name": "type",
    "values": ["freetext"],
    "forced": true
  },
  {
    "name": "specie",
    "values": ["fly", "human", "rat"],
    "forced": true
  },
  {
    "name": "genome release",
    "values": ["freetext"],
    "forced": true
  },
  {
    "name": "cell line",
    "values": ["yes", "no"],
    "forced": true
  },
  {
    "name": "development stage",
    "values": ["larva", "larvae"],
```

```
    "forced": true
  },
  {
    "name": "sex",
    "values": ["male", "female", "unknown"],
    "forced": true
  },
  {
    "name": "tissue",
    "values": ["eye", "leg"],
    "forced": false
  }
]
```

`\annotation\field`

```
### Add annotation field [POST]
```

```
+ Request (application/json)
```

```
+ Header
```

```
    Authorization: token
```

```
+ Body
```

```
{
  "name": "species",
  "type": [
    "fly",
    "rat",
    "human"
  ],
  "default": "human",
  "forced": false
}
```

```
+ Response 201
```

```
### Rename annotation field [PUT]
```

```
+ Request (application/json)
```

```
+ Header
```

```
    Authorization: token
```

```
+ Body
```

```
{
```

```
        "name": "species",
        "newName": "mouse"
    }

+ Response 200

## /annotation/field/<field-name>
### Remove annotation field [DELETE]
+ Request

    + Header

        Authorization: token

+ Response 200
```

`\annotation\value`

```
### Add annotation value [POST]
+ Request (application/json)

    + Header

        Authorization: token

    + Body

        {
            "name": "species",
            "value": "mouse"
        }

+ Response 201

### Rename annotation value [PUT]
+ Request (application/json)

    + Header

        Authorization: token

    + Body

        {
            "name": "species",
            "oldValue": "mouse",
            "newValue": "rat"
        }
```

+ Response 201

`\annotation\value\<field-name> \<value-name>`

Remove annotation value [DELETE]

+ Request

+ Header

Authorization: token

+ Response 200

Genome release handling

Used to get, add and delete genome releases.

`\genomeRelease`

Get all genome releases, no matter species[GET]

+ Request

+ Header

Authorization: token

+ Response 200 (application/json)

```
[
  {
    "genomeVersion": "hy17",
    "specie": "fly",
    "path": "pathToFile",
    "fileName": "nameOfFile"
  },
  {
    "genomeVersion": "u12b",
    "specie": "human",
    "path": "pathToFile2",
    "fileName": "nameOfFile"
  },
  {
```



```
    "genomeVersion": "wk1m",
    "specie": "human",
    "path": "pathToFile3",
    "fileName": "nameOfFile"
  },
  {
    "genomeVersion": "fg2b",
    "specie": "rat",
    "path": "pathToFile4",
    "fileName": "nameOfFile"
  },
  {
    "genomeVersion": "abc1",
    "specie": "rat",
    "path": "pathToFile5",
    "fileName": "nameOfFile"
  }
]
```

Add genome release [POST]

+ Request (application/json)

+ Header

```
Authorization: token
```

+ Body

```
{
  "fileName": "nameOfFile",
  "specie": "human",
  "genomeVersion": "hx16"
}
```

+ Response 201

```
{
  "URLupload": "url"
}
```

`\genomeRelease\<species>`

Get all genome releases for specific species[GET]

+ Request

+ Header

Authorization: token

+ Response 200 (application/json)

```
[
  {
    "genomeVersion": "hy17",
    "specie": "fly",
    "path": "pathToFile",
    "fileName": "nameOfFile"
  },
  {
    "genomeVersion": "u12b",
    "specie": "fly",
    "path": "pathToFile2",
    "fileName": "nameOfFile"
  },
  {
    "genomeVersion": "wk1m",
    "specie": "fly",
    "path": "pathToFile3",
    "fileName": "nameOfFile"
  }
]
```

`\genomeRelease\<species>\<genomeVersion>`

Delete genome release [DELETE]

+ Request

+ Header

Authorization: token

+ Response 200

Appendix F

Server commands

program: apache2
port: 8000
login: null
password: null
info: webserver <http://scratchy.cs.umu.se:8000/>

program: ssh passphrase
port: 0
login: null
password: *****
info: ssh key for server in mc333, (used for git)

program: JBDC
port: 0
login: null
password: null
info: include own jars: /usr/local/lib/psql_jbdc4.jar

program: postgresql database
port: 6000
login: postgres
password: *****
info: psql -h hostname -p 6000 dbname username

program: phppgadmin
port: 8000
login: postgres
password: *****
info: <http://scratchy.cs.umu.se:8000/phppgadmin> use for remote psql management

program: SSH tunnel port

port: 0
login: null
password: null
info: %> ssh -g -R (wanted port on scratchy):(host of server):(server port) -N
-f (cs-user)@scratchy.umu.se

program: SSH to server
port: 2222
login: null
password: *****
info: ssh pvt@scratchy.cs.umu.se -p 2222

program: host/download.php
port: 8000
login: database user
password: database pass
info: parameters: "path" - path to the file on the server. Example:
/var/www/data/Exp1/raw/humanarm.fastq
Example URL:
http://scratchy.cs.umu.se:8000/download.php?
path=/var/www/data/Exp1/raw/humanarm.fastq

program: host/upload.php
port: 8000
login: database user
password: *****
info: parameters: "path" - path to the file on the server Example:
/var/www/data/Exp1/raw/humanarm.fastq

Appendix G

Ubuntu 14.04 Installation and configuration manual

G.1 Introduction

This document will guide a user through the process to configure the server machine needed for the *Genomizer* server software. This guide is created while configuring a newly installed machine running Ubuntu 14.04. Other Linux or UNIX operating systems could have other commands to install different software. Some experience with a terminal and the UNIX environment is presumed.

Be sure to have a fully functioning Internet connection to the server machine with the possibility to direct ports to it before continuing.

G.2 Installation and Configuration

The server machine must run a Linux or UNIX operating system. In order to follow this guide in the easiest manner possible, use any Ubuntu distribution.

G.2.1 Java

Firstly Java must be installed on the system to be able to run some of the server software. The software requires Java 1.7 or later.

To install Java JDK open a terminal and enter the following command:

```
sudo apt-get install openjdk-7-jdk
```

G.2.2 OpenSSH

To be able to access the server machine remotely OpenSSH must be installed.

Enter the following command to install OpenSSH properly.

```
sudo apt-get install openssh-server openssh-client  
openssh-sftp-server
```

G.2.3 Apache2

In order to handle web requests and file transferring the server will use Apache2.

To install Apache2 with necessary software, use this command:

```
sudo apt-get install apache2 apache2-utils
```

After installation of Apache2 some configuration is needed. Follow the steps below.

G.2.3.1 Configure listening port

The default port for listening is set to 80. To change port follow the steps below.

1. Open file with following command:

```
sudo nano /etc/apache2/ports.conf
```

2. Edit the value in the file after “Listen” to the preferred port to use for listening. For example:

```
Listen 80
```

3. Save and close the file.
4. Restart the Apache server with:

```
sudo service apache2 graceful
```

G.2.3.2 Set document root

The document root on the Apache server is where the root folder for the server is located. When a user is connecting to the server it will request the content from the root folder of the server.

The *Genomizer* server uses `/var/www/` as the document root. As default for the Apache server the document root is set to `/var/www/html/`. To change the root folder for the Apache server, do the following steps:

1. Open the configuration file for the document root:

```
sudo nano /etc/apache2/sites-enabled/000-default.conf
```

2. Edit the second string in the line starting with the string “DocumentRoot” to the root directory to be used.

```
DocumentRoot /var/www/
```

3. Save and close the configuration file.
4. Restart the Apache server:

```
sudo service apache2 graceful
```

After these steps the document root is changed. Please note that in step two the root directory can be set to something else. If these steps are followed precisely the document root will be set to `/var/www/`.

G.2.3.3 Add system user

To add a system user, you first have to create a new file containing the user names and their corresponding passwords by using a terminal and writing:

```
htpasswd -c /etc/apache2/passwords username
```

Change `username` to the username wanted for the new user.

This will create a password file in `/etc/apache2/`. The path to the password file should not be accessible for the clients. Then you will be asked to enter the password for the user:

```
New password: mypassword
Re-type new password: mypassword
```

Instead of `mypassword`, enter the password wanted for the new user. When everything is done you will get the message:

```
Adding password for user username
```

The passwords in the file will be stored encrypted. To add more users, the following command must be used:

```
htpasswd /etc/apache2/passwords username
```

If the `-c` flag is used, a new file will overwrite the old one so all users will be overwritten. For more information, see [7]

G.2.3.4 Setup protected folders for users

The Apache server software have functionality to make protected directories to restrict access to its content. To set this up it is necessary to create a user for the Apache server (*read G.2.3.3*) to be able to access the files.

To restrict users from accessing the folders you can make them password protected. To do this, open the file `apache2.conf` that is located in `/etc/apache2/`. In the file, new folders can be added. These folders will be password protected. To add a directory, new directory tags `<Directory></Directory>` must be added among the others. A step-by-step instruction of how to password protect a folder follows:

1. Open the file by the following command:

```
sudo nano /etc/apache2/apache2.conf
```

2. Paste in the following in the file:

```
<Directory /path_to_document_root/path_to_restricted_folder/>
  AuthUserFile /etc/apache2/passwords
  AuthName "This is a protected area"
  AuthType Basic
  Require valid-user
</Directory>
```

Make sure that the `/path_to_document_root/` is set to the document root set in G.2.3.2. Then `path_to_restricted_folder/` needed to be changed to the actual path to the folder that is to be protected. For more information see [7].

3. Make sure your setup is correct.
4. Save and close the file.
5. Restart the Apache server to make changes:

```
sudo service apache2 graceful
```

G.2.3.5 Setup restricted folders for all users

It is also possible to make a folder restricted for all users and only accessible through the server machine or the PHP scripts. This is done almost exactly as

in G.2.3.4. The difference is that you add something else between the directory tags `<Directory></Directory>`. Follow step 1 to 5 in G.2.3.4 but instead paste this to the file at step 2:

```
<Directory /path_to_document_root/path_to_restricted_folder/>  
    Require all denied  
</Directory>
```

G.2.3.6 Setup proxy redirect

To allow tunneling through the Apache server to the *Genomizer* server software, a proxy pass has to be set up. To enable the module for Apache, enter the following command in the terminal:

```
sudo a2enmod proxy_http
```

After the module is loaded a proxy pass has to be set up, the proxy pass works on a url and sends all requests to that url to the proxied address. Start by opening the file:

```
sudo nano /etc/apache2/apache2.conf
```

Scroll down to the end of the file and enter this row (don't forget to modify the url and the proxy to your server setup):

```
#ProxyPass  
ProxyPass /anyurlyouwant/ http://your.server.address:port/
```

In this server setup this line looks like this:

```
ProxyPass /api/ http://scratchy.cs.umu.se:7000/
```

This means that all requests sent to `http://scratchy.cs.umu.se:8000/api/Login` will be proxied (tunneled) to `http://scratchy.cs.umu.se:7000/Login` for example. **Do not forget to restart the Apache server:**

```
sudo service apache2 restart
```

G.2.4 Git

This project uses gitHub for easy sharing of the code between all collaborators. For this to work, the server machine must have git installed to be able to clone the repositories with code.

This document will not specify how to use git, instead please read the two guides [8] and [9]

To install git on the server machine, enter the following line in the terminal:

```
sudo apt-get install git
```

For easy access to gitHub, SSH-keys can be added to easily execute push and pull of repositories. See gitHub's guide [10].

G.2.5 Ant

Ant is a build system to compile java code. It is used to build a runnable jar file for the server. To install, run the following command in a terminal:

```
sudo apt-get install ant
```

G.2.6 PHP5

The Apache server uses PHP scripts to upload and download files. Therefore it is necessary to install PHP5 on the server machine. To install PHP5, just enter the following command in a terminal:

```
sudo apt-get install php5-curl
```

The following file needs to be configurated

```
/etc/php5/apache2/php.ini
```

with values:

```
max_execution_time to 0
```

```
max_input_time to -1
```

```
upload_max_filesize to 0
```

```
post_max_size to 0
```

G.2.7 SRA Toolkit

One of the PHP scripts will need the application SRA Toolkit installed. To install this application, enter the following in the terminal:

```
sudo apt-get install sra-toolkit
```

This application is used to convert .sra files to .fastq files. To manually use SRA Toolkit enter the following in the terminal:

```
fastq-dump /var/www/test/SRR869740.sra
```

This will open the SRR869740.sra file to a SRR869740.fastq file in the same directory of the original .sra file.

G.2.8 PostgreSQL

For the server machine, PostgreSQL is required for the server to work as intended. To install PostgreSQL, enter the following command (note: the version 9.3 may vary):

```
sudo apt-get install postgresql-9.3 postgresql-client-9.3  
postgresql-contrib-9.3
```

An admin account needs to be set up for the database, to do this follow these instructions:

1. Login to the PostgreSQL server by typing

```
sudo -u postgres psql
```

where `postgres` is the default user for the database

2. Enter the following command while inside `psql` to set up a password for the user `postgres`:

```
ALTER ROLE postgres WITH ENCRYPTED PASSWORD password;
```

and change `password` to whatever password is wanted.

To grant access to the database from non-local machines, the following file must be changed (note: the version 9.3 may vary):

```
sudo nano /etc/postgresql/9.3/main/postgresql.conf
```

Find the segment *CONNECTIONS AND AUTHENTICATION* in the top part of the file and change the lines "listen_addresses" and "port":

```
#-----
# CONNECTIONS AND AUTHENTICATION
#-----

# - Connection Settings -

listen_addresses = '*' # what IP address(es) to listen on;
                      # comma-separated list of addresses;
                      # defaults to 'localhost';
                      # use '*' for all
                      # (change requires restart)
port = 6000            # (change requires restart)
```

Port of the server can be changed to whatever is wished. Now the access needs to be changed. To do this add the following lines to the file (note: the version 9.3 may vary):

```
sudo nano /etc/postgresql/9.3/main/pg_hba.conf
```

Make sure that there exists 2 lines that look like the following (change existing lines or add new ones):

```
# "local" is for Unix domain socket connections only
local  all          all                               md5
# IPv4 local connections:
host  all          all          127.0.0.1/32    md5
```

Restart PostgreSQL by typing:

```
sudo service postgresql restart
```

G.2.8.1 Clone database

If there exists an old database that is wished to be migrated to the new database the following command can be executed on the machine where the database is presently:

```
sudo pg_dump -U dbUserName -d dbName -h localhost -p
dbPort > backupfile.sql
```

1. Change dbUserName to the username you have setup for PostgreSQL

2. Change `dbName` to the name of the database that is wished to be migrated
3. Change `dbPort` to the PostgreSQL port which it is setup to listen to
4. Change `backupfile.sql` to whatever filename is wished

This creates a backup SQL file. Now transfer the file to the server where the database is migrated to and type in the following command to inject it into the database:

```
psql -U dbName -h localhost -d dbName -p dbPort < backupfile.sql
```

1. Change `dbName` to the username you have setup for PostgreSQL
2. Change `dbName` to the name of the database that is wished to be migrated to
3. Change `dbPort` to the PostgreSQL port which it is setup to listen to
4. Change `backupfile.sql` to whatever it is named

Restart PostgreSQL by typing:

```
sudo service postgresql restart
```

G.2.9 PgAdmin

PgAdmin is a software which provides a graphical interface towards the PostgreSQL server and can be installed with following command:

```
sudo apt-get install pgadmin3
```

G.2.10 PhpPgAdmin

PhpPgAdmin (Figure G.1) is a user friendly web interface that connects to the server PostgreSQL database. This is recommended to be installed if you are not very comfortable working with the database using the terminal interface or wish to only configure the database on the local server machine using PgAdmin (G.2.9).

G.2.10.1 Setup PhpPgAdmin

Then install the required software by typing in the following command in the terminal:

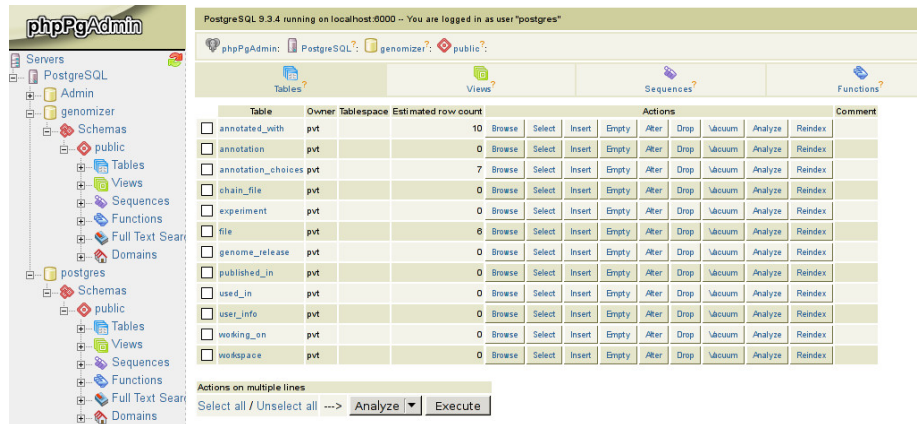


Figure G.1: PhpPgAdmin web interface

```
sudo apt-get install phpPgAdmin
```

Then this needs to be included by the Apache2 software, which is done by editing the file:

```
sudo nano /etc/apache2/apache2.conf
```

and adding this line to the end of the file after the other includes:

```
#Include Phppgadmin
Include /etc/apache2/conf.d/phppgadmin
```

Then we need to change the access settings to the phppgadmin via the Apache software, this is done by changing the file:

```
sudo nano /etc/apache2/conf.d/phppgadmin
```

In the top part of the file a section is displayed as below:

```
order deny,allow
deny from all
allow from 127.0.0.0/255.0.0.0 ::1/128
#allow from all
```

Change this section so that it looks like this:

```
order deny,allow
#deny from all
allow from 127.0.0.0/255.0.0.0 ::1/128
allow from all
```

Now an account must be set up with the PhpPgAdmin. Make sure you have the *htpasswd* software installed (comes with Apache2-utils). Then to set an account, enter the following command in the terminal:

```
sudo htpasswd -c /etc/phpPgadmin/.htpasswd <username>
```

Change the username to what you want the user to be called. After that a prompt will be shown to enter a password, enter the password twice and then the account is setup.

Now the Apache server needs to be told where to look for the users. This is done by editing the file:

```
sudo nano /etc/apache2/sites-enabled/000-default.conf
```

Then add this to the end of the file:

```
<Directory "/usr/share/phpPgadmin">
    AuthUserFile /etc/phpPgadmin/.htpasswd
    AuthName "Restricted Area"
    AuthType Basic
    require valid-user
</Directory>
```

Now PhpPgAdmin needs to be told which port to connect to the PostgreSQL on (see configurations of the PostgreSQL server). To do that changes need to be made to the file:

```
sudo nano /etc/phpPgadmin/config.inc.php
```

Then change the following post to what corresponds to your server setup:

```
// Database port on server (5432 is the PostgreSQL default)
    $conf['servers'][0]['port'] = 6000; //PostgreSQL port here
    .
    .
    .

// passworded local connections.
    $conf['extra_login_security'] = false; //True as standard
```

Restart Apache and phpPgadmin by typing:

```
sudo service apache2 restart
sudo service phpPgadmin restart
```

Appendix H

Debian 7.5 Installation and configuration manual

H.1 Introduction

This document will guide a user through the process to configure the server machine needed for the *Genomizer* server software. This guide is created while configuring a newly installed machine running Debian 7.5 Wheezy. Other Linux or UNIX operating systems could have other commands to install different software. Some experience with a terminal and the UNIX environment is presumed.

Be sure to have a fully functioning Internet connection to the server machine with the possibility to direct ports to it before continuing.

H.2 Installation and Configuration

The server machine must run a Linux or UNIX operating system. In order to follow this guide in the easiest manner possible, use any Debian distribution.

H.2.1 Installation of Debian

Since Debian is more stable for running server applications than other Linux distributions such as Ubuntu and Linux Mint, it is recommended to use any Debian release. When partitioning the hard drives make sure to assign at least 2 times the amount of ram as swap section, in our case we used 250GB. This is done to prevent the server from crashing in case the ram RAM is filled out.

The swap is a hardware RAM section that the system can dump to if the RAM

is filled.

H.2.2 Configure Debian repositories

To allow the system to download software via terminal a few repositories changes must be made. To do this open the file:

```
sudo nano /etc/apt/sources.list
```

Then the line

```
deb cdrom:[Debian GNU/Linux 7.5.0 _Wheezy_ -  
Official amd64 CD Binary-1 20140426-13:37]/ wheezy main
```

needs to be commented away from the file to avoid errors when the system tries to fetch software from an non-existent cd rom. Then four repositories should be added to allow installation of software. Add the following lines to the end of the file:

```
deb http://ftp.acc.umu.se/debian/ wheezy-updates main  
deb-src http://ftp.acc.umu.se/debian/ wheezy-updates main  
  
deb http://ftp.acc.umu.se/debian/ wheezy main  
deb-src http://ftp.acc.umu.se/debian/ wheezy main
```

Try this out by typing:

```
sudo apt-get update
```

and make sure there is no errors.

H.2.3 Create a super user

When Debian is installed there only exists one super user on the computer, and that is "root". To give other users on the system root access and super user rights a configuration file must be changed, to open the file you need to login as root temporary (this is not recommended to do for other things).

To login as root, type:

```
su
```

Enter the password for the root, as setup in the installation. Then type the following to open the config file:

```
visudo
```

Then add this line under the line where root is set.

```
username ALL=(ALL:ALL) ALL
```

Where username is the user that will be given super user rights.

H.2.4 Locales

If there is a problem with the locales settings that looks something like this:

```
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
LANGUAGE = "en_GB:en",
LC_ALL = (unset),
LC_COLLATE = "sv_SE",
LC_MEASUREMENT = "sv_SE",
LC_PAPER = "sv_SE",
LANG = "C"
    are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

Try this fix:

```
export LC_ALL=en_GB.UTF-8
sudo /usr/sbin/locale-gen
sudo dpkg-reconfigure locales
```

Then reboot the server.

H.2.5 Java

Firstly Java must be installed on the system to be able to run some of the server software. The software requires Java 1.7 or later.

To install Java JDK open a terminal and enter the following command:

```
sudo apt-get install openjdk-7-jdk
```

H.2.6 OpenSSH

To be able to access the server machine remotely OpenSSH must be installed.

Enter the following command to install OpenSSH properly.

```
sudo apt-get install openssh-server openssh-client  
openssh-sftp-server
```

H.2.7 Apache2

In order to handle web requests and file transferring the server will use Apache2.

To install Apache2 with necessary software, use this command:

```
sudo apt-get install apache2 apache2-utils
```

After installation of Apache2 some configuration is needed. Follow the steps below.

H.2.7.1 Configure listening port

The default port for listening is set to 80. To change port follow the steps below.

1. Open file with following command:

```
sudo nano /etc/apache2/ports.conf
```

2. Edit the value in the file after “Listen” to the preferred port to use for listening. For example:

```
Listen 80
```

3. Save and close the file.
4. Restart the Apache server with:

```
sudo service apache2 graceful
```

H.2.7.2 Set document root

The document root on the Apache server is where the root folder for the server is located. When a user is connecting to the server it will request the content from the root folder of the server.

The *Genomizer* server uses `/var/www/` as the document root. As default for the Apache server the document root is set to `/var/www/html/`. To change the root folder for the Apache server, do the following steps:

1. Open the configuration file for the document root:

```
sudo nano /etc/apache2/sites-enabled/000-default.conf
```

2. Edit the second string in the line starting with the string “DocumentRoot” to the root directory to be used.

```
DocumentRoot /var/www/
```

3. Save and close the configuration file.
4. Restart the Apache server:

```
sudo service apache2 graceful
```

After these steps the document root is changed. Please note that in step two the root directory can be set to something else. If these steps are followed precisely the document root will be set to `/var/www/`.

H.2.7.3 Add system user

To add a system user, you first have to create a new file containing the usernames and their corresponding passwords by using a terminal and writing:

```
htpasswd -c /etc/apache2/passwords username
```

Change `username` to the username wanted for the new user.

This will create a password file in `/etc/apache2/`. The path to the password file should not be accessible for the clients. Then you will be asked to enter the password for the user:

```
New password: mypassword
Re-type new password: mypassword
```

Instead of `mypassword`, enter the password wanted for the new user. When everything is done you will get the message:

```
Adding password for user username
```

The passwords in the file will be stored encrypted. To add more users, the following command must be used:

```
htpasswd /etc/apache2/passwords username
```

If the `-c` flag is used, a new file will overwrite the old one so all users will be overwritten. For more information, see [7]

H.2.7.4 Setup protected folders for users

The Apache server software have functionality to make protected directories to restrict access to its content. To set this up it is necessary to create a user for the Apache server (see H.2.7.3) to be able to access the files.

To restrict users from accessing the folders you can make them password protected. To do this, open the file `apache2.conf` that is located in `/etc/apache2/`. In the file, new folders can be added. These folders will be password protected. To add a directory, new directory tags `<Directory></Directory>` must be added among the others. A step-by-step instruction of how to password protect a folder follows:

1. Open the file by the following command:

```
sudo nano /etc/apache2/apache2.conf
```

2. Paste in the following in the file:

```
<Directory /path_to_document_root/path_to_restricted_folder/>
  AuthUserFile /etc/apache2/passwords
  AuthName "This is a protected area"
  AuthType Basic
  Require valid-user
</Directory>
```

Make sure that the `/path_to_document_root/` is set to the document root set in H.2.7.2. Then `path_to_restricted_folder/` needed to be changed to the actual path to the folder that is to be protected. For more information see [7].

3. Make sure your setup is correct.
4. Save and close the file.
5. Restart the Apache server to make changes:

```
sudo service apache2 graceful
```

H.2.7.5 Setup restricted folders for all users

It is also possible to make a folder restricted for all users and only accessible through the server machine or the PHP scripts. This is done almost exactly as

in H.2.7.4. The difference is that you add something else between the directory tags `<Directory></Directory>`. Follow step 1 to 5 in H.2.7.4 but instead paste this to the file at step 2:

```
<Directory /path_to_document_root/path_to_restricted_folder/>
  Require all denied
</Directory>
```

H.2.7.6 Setup proxy redirect

To allow tunneling through the Apache server to the *Genomizer* server software, a proxy pass has to be set up. To enable the module for Apache, enter the following command in the terminal:

```
sudo a2enmod proxy_http
```

After the module is loaded a proxy pass has to be set up, the proxy pass works on a url and sends all requests to that url to the proxied address. Start by opening the file:

```
sudo nano /etc/apache2/apache2.conf
```

Scroll down to the end of the file and enter this row (don't forget to modify the url and the proxy to your server setup):

```
#ProxyPass
ProxyPass /anyurlyouwant/ http://your.server.address:port/
```

In this server setup this line looks like this:

```
ProxyPass /api/ http://scratchy.cs.umu.se:7000/
```

This means that all requests sent to `http://scratchy.cs.umu.se:8000/api/Login` will be proxied (tunneled) to `http://scratchy.cs.umu.se:7000/Login` for example.

Do not forget to restart the Apache server:

```
sudo service apache2 restart
```

To allow apache to upload and download files to the system a user called "www-data" must be added to the group of the user created for the system. If you don't remember what user you have setup you can write:

```
groups
```

The group name should be present there. now type the following line in the terminal:

```
sudo gpasswd -a www-data groupname
```

where groupname is the user you have setup for the system.

H.2.8 Git

This project uses gitHub for easy sharing of the code between all collaborators. For this to work, the server machine must have git installed to be able to clone the repositories with code.

This document will not specify how to use git, instead please read the two guides [8] and [9]

To install git on the server machine, enter the following line in the terminal:

```
sudo apt-get install git
```

For easy access to gitHub, SSH-keys can be added to easily execute push and pull of repositories. See gitHub's guide [10].

H.2.9 Ant

Ant is a build system to compile java code. It is used to build a runnable jar file for the server. To install, run the following command in a terminal:

```
sudo apt-get install ant
```

H.2.10 PHP5

The Apache server uses PHP scripts to upload and download files. Therefore it is necessary to install PHP5 on the server machine. To install PHP5, just enter the following command in a terminal:

```
sudo apt-get install php5-curl
```

The following file needs to be configurated

```
/etc/php5/apache2/php.ini
```

with values:

```
/etc/php5/apache2filter/php.ini
```

```
max_execution_time to 0
```

```
max_input_time to -1
```

```
upload_max_filesize to 0
```

```
post_max_size to 0
```

H.2.11 SRA Toolkit

One of the PHP scripts will need the application SRA Toolkit installed. To install this application, enter the following in the terminal:

```
sudo apt-get install sra-toolkit
```

This application is used to convert .sra files to .fastq files. To manually use SRA Toolkit enter the following in the terminal:

```
fastq-dump /var/www/test/SRR869740.sra
```

This will open the SRR869740.sra file to a SRR869740.fastq file in the same directory of the original .sra file.

H.2.12 PostgreSQL

For the server machine, PostgreSQL is required for the server to work as intended. To install PostgreSQL, enter the following command (note: the version 9.3 may vary):

```
sudo apt-get install postgresql-9.1 postgresql-client-9.1  
postgresql-contrib-9.1
```

An admin account needs to be set up for the database, to do this follow these instructions:

1. Login to the PostgreSQL server by typing


```
sudo -u postgres psql
```

where `postgres` is the default user for the database

2. Enter the following command while inside `psql` to set up a password for the user `postgres`:

```
ALTER ROLE postgres WITH ENCRYPTED PASSWORD 'password';
```

and change `password` to whatever password is wanted.

To grant access to the database from non-local machines, the following file must be changed (note: the version 9.3 may vary):

```
sudo nano /etc/postgresql/9.3/main/postgresql.conf
```

Find the segment *CONNECTIONS AND AUTHENTICATION* in the top part of the file and change the lines `"listen_addresses"` and `"port"`:

```
#-----  
# CONNECTIONS AND AUTHENTICATION  
#-----  
  
# - Connection Settings -  
  
listen_addresses = '*' # what IP address(es) to listen on;  
# comma-separated list of addresses;  
# defaults to 'localhost';  
# use '*' for all  
# (change requires restart)  
port = 6000 # (change requires restart)
```

Port of the server can be changed to whatever is wished. Now the access needs to be changed. To do this add the following lines to the file (note: the version 9.3 may vary):

```
sudo nano /etc/postgresql/9.3/main/pg_hba.conf
```

Make sure that there exists 2 lines that look like the following (change existing lines or add new ones):

```
# "local" is for Unix domain socket connections only  
local all all md5  
# IPv4 local connections:  
host all all 127.0.0.1/32 md5
```

Restart PostgreSQL by typing:

```
sudo service postgresql restart
```

H.2.12.1 Clone database

If there exists an old database that is wished to be migrated to the new database the following command can be executed on the machine where the database is presently:

```
sudo pg_dump -U dbUserName -d dbName -h localhost -p  
dbPort > backupfile.sql
```

1. Change `dbUserName` to the username you have setup for PostgreSQL
2. Change `dbName` to the name of the database that is wished to be migrated
3. Change `dbPort` to the PostgreSQL port which it is setup to listen to
4. Change `backupfile.sql` to whatever filename is wished

This creates a backup SQL file.

H.2.13 Inject database copy

To inject a database backup into a new database do the following command.

```
psql -U dbUserName -h localhost -d dbName -p dbPort < backupfile.sql
```

1. Change `dbUserName` to the username you have setup for PostgreSQL
2. Change `dbName` to the name of the database that is wished to be migrated to
3. Change `dbPort` to the PostgreSQL port which it is setup to listen to
4. Change `backupfile.sql` to whatever it is named

Restart PostgreSQL by typing:

```
sudo service postgresql restart
```

H.2.14 PgAdmin

PgAdmin is a software which provides a graphical interface towards the PostgreSQL server and can be installed with following command:

```
sudo apt-get install pgadmin3
```

H.2.15 PhpPgAdmin

PhpPgAdmin (Figure H.1) is a user friendly web interface that connects to the server PostgreSQL database. This is recommended to be installed if you are not very comfortable working with the database using the terminal interface or wish to only configure the database on the local server machine using PgAdmin (H.2.14).

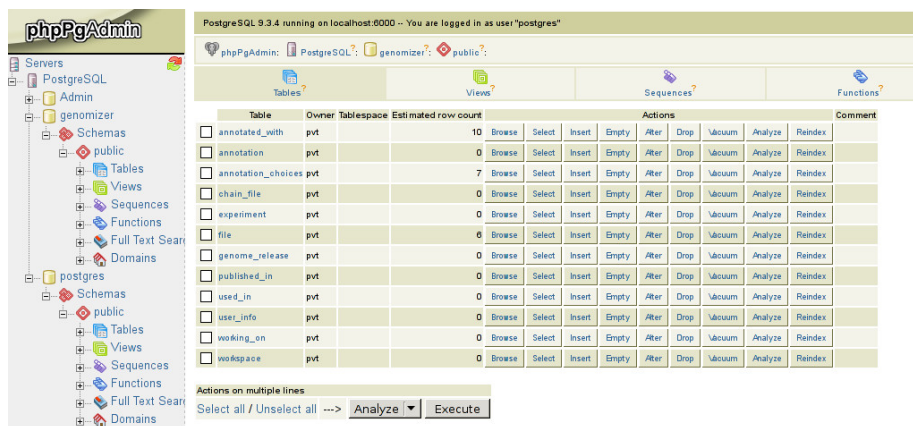


Figure H.1: PhpPgAdmin web interface

H.2.15.1 Setup PhpPgAdmin

Then install the required software by typing in the following command in the terminal:

```
sudo apt-get install phppgadmin
```

This needs to be included by the Apache2 software, which is done by editing the file:

```
sudo nano /etc/apache2/apache2.conf
```

and adding this line to the end of the file after the other includes:

```
#Include Phppgadmin
Include /etc/apache2/conf.d/phppgadmin
```

When that is done, we need to change the access settings to the phppgadmin via the Apache software, this is done by changing the file:

```
sudo nano /etc/apache2/conf.d/phppgadmin
```

In the top part of the file a section is displayed as below:

```
order deny,allow
deny from all
allow from 127.0.0.0/255.0.0.0 ::1/128
#allow from all
```

Change this section so that it looks like this:

```
order deny,allow
#deny from all
allow from 127.0.0.0/255.0.0.0 ::1/128
allow from all
```

Now an account must be set up with the PhpPgAdmin. Make sure you have the *htpasswd* software installed (comes with Apache2-utils). Then to set an account, enter the following command in the terminal:

```
sudo htpasswd -c /etc/phppgadmin/.htpasswd <username>
```

Change the username to what you want the user to be called. After that a prompt will be shown to enter a password, enter the password twice and then the account is setup.

Now the Apache server needs to be told where to look for the users. This is done by editing the file:

```
sudo nano /etc/apache2/sites-enabled/000-default.conf
```

Add this to the end of the file:

```
<Directory "/usr/share/phppgadmin">
    AuthUserFile /etc/phppgadmin/.htpasswd
    AuthName "Restricted Area"
    AuthType Basic
    require valid-user
</Directory>
```

Now PhpPgAdmin needs to be told which port to connect to the PostgreSQL on (see configurations of the PostgreSQL server). To do that changes need to be made to the file:

```
sudo nano /etc/phpPgAdmin/config.inc.php
```

Then change the following part to what corresponds to your server setup:

```
// Database port on server (5432 is the PostgreSQL default)
$config['servers'][0]['port'] = 6000; //PostgreSQL port here
.
.
.

// passworded local connections.
$config['extra_login_security'] = false; //True as standard
```

Restart Apache and phpPgAdmin by typing:

```
sudo service apache2 restart
sudo service phpPgAdmin restart
```

H.2.16 Genomizer configuration

The server requires some parameters to be set before it can be used. They should be stored in a file called “settings.cfg” in the same directory as the server JAR. It can look like this:

```
databaseuser      = admin
databasepassword  = password
databasehost      = localhost:6000
databasename      = genomizer
publicaddress     = http://www.genomizer.se
apacheport        = 8000
downloadurl       = /download.php?path=
uploadurl         = /upload.php?path=
genomizerport     = 7000
passwordsalt      = genomizer
passwordhash      = 2fd26e9aea528153a865257a723f6d4859e9f6c4a6775c003ae91297f619c6e8
```

Each setting should be on a separate line, and be separated by a '=' sign. The number of spaces does not matter, neither does case on the setting names. Case does matter on the setting values however, password and Password are different things.

H.2.16.1 Database settings

The settings `databaseuser`, `databasepassword`, `databasehost` and `databaseName` are all connected to the SQL database, and should be the same as the ones used when setting up the database.

H.2.16.2 Addresses and ports

The settings `publicaddress`, `apacheport`, `downloadurl`, `uploadurl` and `genomizerport` decide how the clients connect to the genomizer system. `downloadurl`, `uploadurl` and `genomizerport` should generally stay the same as the example file, but `publicaddress` and `apacheport` depends on how the server is set up.

H.2.16.3 Password handling

`Passwordsalt` is used to increase the security of passwords. It is combined with the password and hashed so that the password does not need to be saved in plaintext. `Passwordhash` is the result of this operation, and this parameter should not be set manually. There is a script called `changePwd.sh`, which when called with the password as parameter will update `settings.cfg` with the result.

H.2.16.4 Flags

In addition to the settings file, a set of flags may be set when starting the server from terminal. These are:

- `-p [port]`
This flag sets the listening port.
- `-debug`
If this flag is set the server will print output to terminal for every request and response. Also other outputs are written aswell.
- `-f [file]`
Uses file to read settings instead of the default file `settings.cfg`.
- `-nri`
If this flag is set the server will not remove inactive users which are logged in on the server.

Appendix I

Migration of the Genomizer system

I.1 Introduction

This manual will present the steps in the process to migrate the *Genomizer* system to a new server machine. This document will guide an experienced Linux user through the process of a migration from one server machine to a new one. The migration can be seen as a manual backup if the steps in I.2 is followed.

I.2 Steps of migration

1. Run `pg_dump` command on the server machine to migrate. This creates a copy of the database. For help, see *Appendix H.2.12.1*
2. Copy the `/var/www/` folder on the current server machine and save the copy to a removable storage device.
3. Install the new server machine with operating system and necessary software. For help with this see Appendix H.
4. When installation is done, paste the copied `/var/www/` folder from step 2, to the same location on the new server machine.
5. Insert the database copy into the database on the new machine. For help, see *Appendix H.2.13*
6. Copy `settings.cfg` and make sure it is properly set up for the new machine. See *Appendix H.2.16*.

7. Now the system is ready for the first startup. Make sure to have a new version of the *Genomizer* server.jar on the machine.

Appendix J

Backup

J.1 Introduction

To allow the server to create a mirror of the file system on to a remote backup server, some settings needs to be added. First of make sure that there exists one backup computer with internet access and port forwarding to the ssh port (22 by default) running any Linux distribution.

Make sure both computers have the rsync software by typing in:

```
man rsync
```

and make sure there is no error message.

Check if the genomizer server computer have "crontab" installed by typing:

```
man crontab
```

and make sure there is no error message. If one of the softwares aren't installed just type:

```
sudo apt-get install crontab
```

OR

```
sudo apt-get install rsync
```

J.2 File backup

To synchronize the computer's file systems a simple bash script has to be edited to wanted effect, the script is located at `/var/www/scripts/backup.sh` and looks like this:

```
#!/bin/bash
PORT=
USER=
IP=
READPATH=/var/www/data
SAVEPATH=/var/www/data
rsync --ignore-existing --delete --update
      -avze 'ssh -p '$PORT $READPATH $USER@$IP:$SAVEPATH
```

make sure the rsync commad is on one line.

- Set the PORT variable to the ssh port on the backup server.
- Set the USER variable to login in the backup server.
- Set the IP variable to the ip to the backup server.

The flags "-avze" should be present for the script to work as intended, there are many flags available to change the behaviour of the program rsync. Flags to add to create a direct "mirror" of the system is:

- "-ignore-existing" - Does not upload files that already exists on the backup server.
- "-delete" - Deletes files on the backup server that does not longer exist on the genomizer server.
- "-update" - Updates files if they have been changed.

To see more available flags check: http://linux.about.com/library/cmd/blcmd11_rsync.htm

To give rsync the right to create folders on the backupserver the `/var/www` folder needs have its user changed to the user of the system. To do this go to the folder `/var/` and type in :

```
sudo chown -R username www/
```

change the username for the username of the system, now the folders belong to the user and not root.

Finally make the script file runnable by typing:

```
sudo chmod -x backup.sh
```

J.3 Database backup

To synchronize the computer's database a simple bash script has to be edited to wanted effect, the script is located at `/var/www/scripts/sqlback.sh` and looks like this:

```
#!/bin/bash
DATE=$(date -I)
DBUSER=
DBNAME=
DBPORT=
SAVEFILE=SqlBackup- $\$$ DATE.sql
pg_dump -w -U  $\$$ DBUSER -h localhost -p  $\$$ DBPORT  $\$$ DBNAME > tmp
echo pvt | sudo -S cp tmp /var/www/sqlbackup/ $\$$ SAVEFILE
```

- Set the DBUSER variable to the username of the database
- Set the DBNAME variable to the name of the database
- Set the DBPORT variable to the the port of the databse

to allow the script to access the database a file named

```
.pgpass
```

needs to be created in the home folder of the user for example:
`/home/username/.pgpass`.

This file should contain the following information:

```
localhost:PORT:DATABASE:USERNAME:PASSWORD
```

where:

- PORT is changed to the port of the database.
- DATABASE is the database to be cloned.
- USERNAME is the username of the postgresql database.
- PASSWORD is the passwdword for the user of the postgresql database.

J.4 Crontab

To enable the server to automatically do synchronizations to the backupserver one line have to be added to a crontab config file. open the file by typing:

```
sudo crontab -e
```

In the end of the file add a line that looks like this:

```
1 0 * * * /var/www/scripts/backup.sh
1 0 * * * /var/www/scripts/sqlback.sh
```

Explanation of the settings that executes the script:

1. Setting is the minute (Present: first minute).
2. Setting is the hour (Present: midnight).
3. Setting is the week (Present: every week in each month)
4. Setting is the month (Present: every month)
5. Setting is the weekday (0=Sunday,1=Monday, Present: every day)

So this backup will run on the first minute of every midnight all year around.

Appendix K

Known problems

K.1 Web application

K.1.1 Moving backwards in the browser does not hide modal windows

When navigating to a modal window the URL is updated, and added to the browser history. When using the browser back-button the modal is not closed, but the URL is still updated.

Modify ModalAC and the router

K.1.2 Error handling when uploading experiments

If there is an error when adding files to an experiment the experiment collapses so the user won't get a chance to correct it's mistake

Start uploads of files when all files have been added without errors.

K.1.3 Old authorization token causes page redirect

If the authorization token expires the user will be sent to the login screen and any input entered will disappear.

Show login modal without redirecting to root url and save errorous ajax-request and resend it when the login has been completed.

K.1.4 Code duplication in SearchResults and Experiments

The collections SearchResults and Experiments represents the same models. But are different collections as they have different URL. It might be better to have the both use the same collection.

Merge SearchResults and Experiments one collection.

K.1.5 No warning when closing tab during upload.

If a upload is in progress, there is no warning when closing the tab and the upload is canceled directly.

There are some code for this in view/Upload.js, but it's currently broken.

K.1.6 Uploading genome release - does not update list automatically

After adding a genome release the list does not update automatically.

Build functionality to render when upload is done.

K.1.7 The annotation list can't be sorted

The annotation list should be re-ordered by clicking on table headers.

Rebuild list to match design of GenomeReleaseView. We had some problems as we are using a separate list for the search-bar.

K.1.8 Sidebar on adminpage dosen't stay vertical

The sidebar items goes horizontally when page width is 2480px.

CSS the sidebar better.

K.1.9 Missing error check on annotation values

It is possible to add one space as annotation value (and name)

The server should check for such faults, otherwise some regex code should go around line 70 in NewAnnotationView

K.1.10 No warning when closing tab during uploading genome releases

If a upload is in progress, there is no warning when closing the tab and the upload is canceled directly.

K.2 *iOS* application

K.2.1 Unspecified behaviour on loss of internet connection

The application is based around having an active internet connection, and may crash if the connection is lost.

K.2.2 Lack of security

Currently, the only security feature is that a valid password is required to log in. All communication is done in plain text. This should be fixed as soon as possible.

K.2.3 No administrative features

The app does not have any of the administrative functions of the web and desktop clients.

K.3 Server

K.3.1 Communication and control

The communication between the server and the clients have some limitations and security holes. These limitations are described below.

K.3.1.1 Don't use the same username as someone else

When someone logs in with the same username as someone who is already logged in with that username, this could create problems. The problem occurs if one of the clients logs out while the other is communicating with the server. When one

of the clients has logged out, the other will get an error response code UNAUTHORIZED telling the client that he/she is not logged in.

To avoid this problem, never use the same username as someone else.

K.3.1.2 Communication in plain text

A major security hole in the system is that all communication between the server and the clients are in plain text. These HTTP-packages can be read by outside people.

To fix this problem, implement a cryptographic protocol, like Secure Sockets Layer(SSL) or Transport Layer Security (TLS), which makes the communication between the server and the clients unreadable by outsiders.

K.3.1.3 Only one process at a time

The server can only run one process at a time. This is because only one thread is set to search the queue and if there is a process waiting the single thread takes this process and executes it before going back to search the queue again.

A solution to this would be to create some kind of thread pool with a user defined amount of threads. Either the size of the threadpool could be decided when starting the server, or in some way be decided by an administrator during runtime to give ability to increase or decrease the number of possible simultaneously running processes.

K.3.1.4 No way to stop a running process

When a process is started there is no way to end the processing without shutting down the server. So if the user would start a process with wrong parameters or on the wrong files, there is no way to just stop that and start a new process.

A solution to this would be to keep track of all working threads and give a user the possibility to terminate these through the user interface. When a thread is terminated a cleanup should be executed to remove created folders and files.

K.3.1.5 No way to see if a process is stuck

In the case that a process for some reason would get stuck while running, there is no feedback to the user to show that the process is stuck. The only feedback the user is given is that the process is currently being executed.

This is a hard problem to solve since there is no good way to know if the

processing is just taking a long time to complete or if it is stuck. A bad solution would be to check how long the process have been running and end it if the time exceeds some defined number.

K.3.2 Upload and download

The two scripts used for file transferring in the Genomizer system have some limitations. These will be presented below. Please note that both the scripts reads the settings.cfg file to get information to be able to access the database. Make sure to put a copy of the settings.cfg file into the */var/www/*.

K.3.2.1 Upload script

When a user tries to upload a file and the upload is interrupted the file entry will remain in the database but the file will not exist in the file system. The file will have the status 'In Progress' but will never be uploaded if the user do not try to upload the file again. Furthermore the script will not return good error messages to the user if a file transfer is interrupted.

K.3.2.2 Download script

If a file download is interrupted the user will not receive a error message from the script containing and explaining the reason for the interrupt.

K.3.3 Process limitations

- Ratio calculation has a limitation that it requires processing to be run on both files and that one of the files needs to be named input.
- One known problem with the smoothing subprogram, is that if a chromosome is smaller than the window size. The program will then smooth that chromosome together with the following chromosome. In practice this problem should never occur on a regular file when doing smoothing once.

However, if stepping is done on a file with a step size of, for example 10 000. And we want to smooth the new file again with a window size of 100. Then the shortest chromosome in the original file must be atleast 1 000 000 rows. From what we have seen of the melanogaster data the shortest chromosome there is roughly 200 000 rows.

Therefor a user should be cautious when smoothing the second time on file that have been stepped with a large number.

It seems unlikely that stepping will be done with a step size of 10 000.