Technical documentation

Genomizer

Version 2.4

Publication date: 2015/06/02

Contents

Preface

This documentation describes the *Genomizer* project conducted during the spring of 2015. The project is a part of the course in software engineering named *Programvaruteknik(5DV151)* given by *Umeå University*. The course is given to a mix of students. Some studying for a master and some for a bachelor degree in computer science. Hence different kind of knowledge exist within the student groups.

The documentation has two purposes. **1**) To give the possibility for tutors to assess the project and grade the students on their work. **2**) To describe the project and all parts of the developed system . The target audience of the system description are three subgroups: end users, system administrators and developers.

End users are epigenetic researchers. Developers are students of the course (current and future). System administrators are both students and personnel responsible for maintenance of the system.

The origin of the project is a perceived need from the epigenetic research department at *Umeå University*. The wish is to make a more efficient pipeline for the computational parts of their research. An automated pipeline where knowledge of the parts involved is kept to a minimum. The automated pipeline would result in less time spent on data entry and more time available for analysing data or conducting experiments.

Changes since version 2.3

Preface:	Introduced the preface chapter.
Introduction:	Moved small parts to preface. Added some information.
Service Description:	Restructured to be non-technical about each part of the system.
Architectual design:	Moved as first chapter in development part as well as a bit more detailed.
Meaning of the text targeted at:	Decision taken in the design that needed to be right the first time to avoid time consuming alterations.
Interaction Design:	Introduction in chapter is written that explains content of the chapter.
Interaction Design:	Desktop section is more motivated on decisions, more pictures added.
Interaction Design:	Web section has added pictures, motivations.
System Design:	Desktop section fixed spelling, added pictures.
Appendix:	Moved deployment and maintenance and added information about the virtual machines.
Glossary:	Glossary added after bibliography

Acknowledgments

- Jonas Andersson: Great technical support and workflow support.
- Jonny Pettersson: Great support on the group dynamics.
- Jan-Erik Moström: Great feedback on documentation.

Developers spring 2015

Data Storage
Nils Gustafsson
Albin Råstander
Jimmy Sihlberg
Martin Larsson
Erik Samuelsson
Fredrik Uddgren

Processing
Adam Dahlgren Lindström
Carl-Evert Kangas
Emil Nylind
Mikhail Glushenkov
Saimon Marouki

Business Logic
Johannes Ekman
Alexander Frisk
Tim Hedberg
Mikael Johansson
Mikael Karlsson
Stefan Lindström
Robin Ramquist

Desktop
Viktor Bengtsson
Maximilian Bågling
Christoper Fladevad
Jonas Hedin
Petter Johansson
Marcus Lööw
Oscar Ottander

Mobile Applications
Erik Berggren
Jesper Bilander
Victor Bylin
Pål Forsberg
Petter Nilsson
Mattias Scherer

Website
Ludwig Andersson
Niklas Fires
Andreas Günzel
Pascal Hansson
Patrik Hörnqvist
Anna Jonsson
Björn Pers

Editorial Staff: Mikael Johansson, Erik Samuelsson, Carl-Evert Kangas, Maximilian Bågling, Mattias Scherer, Patrik Hörnqvist.

2015/06/02Mikael Johansson

1 Introduction

Genomizer is a system for storing and analysing *DNA*-sequence data. It was designed for researchers in the field of epigenetics, who are interested in where on a *DNA* string certain proteins 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. These files are hence refered to as *profile* data. *Genomizer* allows the researchers to upload *raw* files to a server and automate the generation of analysis data as well as store the generated analysis data in a database for later access.

The documentation contains three main parts. Introduction chapters that explain the goal of the project as well as a non-technical description of the project implementation. The development part of the document where the current implementation of each part of the project is explained how they look and work as well as an attempt to explain why certain design choices where done. Then finally there is a big collection of appendicies that goes deeper in their explanation of certain details of the implementation as well as maintenance guides.

2 Target group and needs

The *Genomizer* system was designed with a specific target group in mind: Epigenitic 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 *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 of molecular biology are interested in working with the data, however their access is limited to viewing and analysing 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 C on page 131. A overview of the requested system may be seen below in figure 2.1. Where orange colored nodes are must have features while gray nodes are visions of the clients that may be implemented if time allows for it.

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 analysed 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.

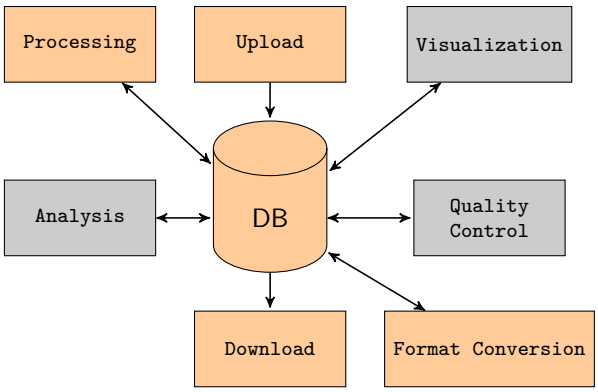


Figure 2.1: Overview of targeted system

2.2.1 Upload & Download

When conducting experiments the researchers generate *raw* data that generates what they call *Raw*-files. These files along with profile data, region data and genome release data may be added and related to an experiment. The requested functionality is to be able to upload these files to the database from multiple sources. The sources may be directly from an experiment conducted by the researchers or 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 an upload to *Genomizer* with the source of the upload beeing *GEO*.

2.2.2 Database

The `Database` module requested has the purpose to archive experiment data in a way of easy access. To allow for this the experiments and files associated with them needs to have information vital for good readability. This is solved with the help of annotations. The researchers must add annotations to files related to an experiment. This data is the foundation for further research and so must be stored securely. To ensure security the client requested a system for authorization that protects the data from outside tampering. To protect against hardware failure there exists a request for a backup system.

2.2.3 Processing

The unordered *raw* data gained from an experiment requires processing in order to be analysed. 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.

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.

It would also be an advantage if the system could carry out further processing from *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.

2.2.4 Format Conversion

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

It is not uncommon for errors in a new release to be discovered after publication. For this reason it is important to temporarily keep files that were generated using older genome releases after a new release is published.

3 Service description

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

3.1 Usage

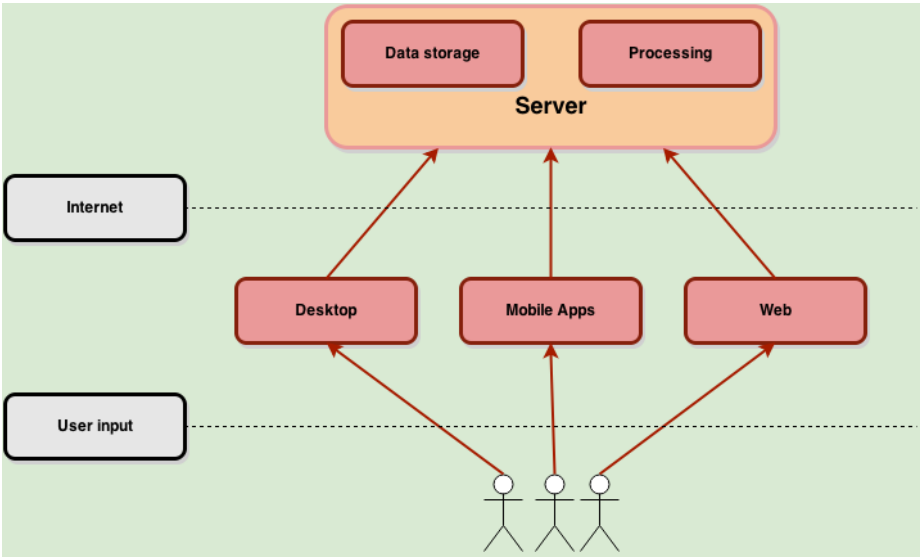


Figure 3.1: Communication diagram of the product

In order to give the users flexibility when using the service there are clients for many different platforms (Windows, Linux, OSX, Web, Mobile devices). When a user chooses a given task, for example start *raw* to *profile* processing. The task i sent via Internet to the server as shown in 3.1. The server then handles the request and return a response back to the user.

3.2 User Input

The user input in the *Genomizer*system may be done with four different clients: a desktop client, a web client, an *Android* client and a *iOS* client. The last two clients are collected under mobile application since they offer the same functionality.

3.3 Desktop

The desktop client is the main client for the *Genomizer* system. It offers all implemented functionality.

- Login and logout.
- Searching for experiments and different files.
- Create new experiments.

- Modify existing experiments.
- Upload files to experiments.
- Download files from experiments.
- Process files from raw to profile.
- Delete files and experiments from the database.
- Add annotations to experiments.
- Remove and modify annotations.
- Search annotations by name.
- Upload, remove and rename genome release files.

3.4 Web

The web client mimics the behaviour and functionality of the desktop client. Additionally, unlike the desktop client, it enables the user to log in and use the service remotely via Internet. This is useful if the user needs to, for example, download or upload files to the server but is not currently on the same local network as the server. The web client is easy to start using since it runs in a web browser and needs no installation.

3.5 Mobile application

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.

3.6 Server

The servers purpose is to take care of the organizations of files and experiments as seen in the top part of Figure 3.1 as the part *Data storage*. The server also serves as a processing tool of the files added to the *Data storage*.

3.6.1 Data 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.

3.6.1.1 Experiments

3.6.1.2 Annotations

Annotations is a way for the researchers to keep track of what an experiment consists of as well as files associated with an experiment. *Genomizer* has two kinds of annotations. There is *multiple-choice* annotations which have a defined name and choices. An example is the

annotation named *species* that have the choices human, fly and rat. Then there is *free text* annotations where the user may enter what they want.

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 if the user have the credentials. This includes the editing of *mulitple-choice* annotation choices and the removal of unused annotations.

> **Example 1** *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.*

3.6.2 Processing

Users can request that a *raw* file set be processed to *profile* files. This procedure is carried out by the server to avoid heavy workload and the requirements of certain programs on the clients side. 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.

Part I

Development

4 Architectual 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.

4.1 System overview

The *Genomizer* is a server-client architecture. It consists of four different clients: a web client, a desktop client and a two kinds of mobile clients. The mobile clients are implemented in *Android* and *iOS*. The server side consists of three components, a web server that acts as a proxy, a *Java* server that connects to the final component a *postgresql* database.

The web server is a *Apache* system in the current implementation.

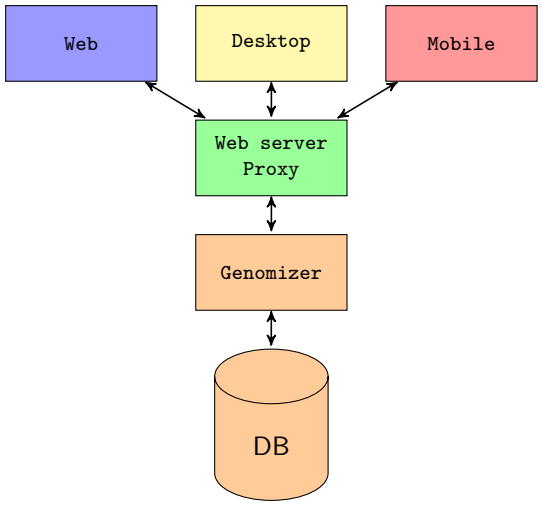


Figure 4.1: Overview of the system architecture

The connection to the server from any client goes through the web server which acts as a proxy as well. This to insure a *SSL* connection between the web server and the clients and a regular connection between the server and web server. The proxy protects the web client from *XSS* (Cross site scripting).

All of the clients use the *RESTful* protocol where the message body consists of a `JSON` object. These messages are requests that gets passed through the web server to the *Genomizer* server. The *Genomizer* server communicates with the database to perform the request. Then it generates a response following the *RESTful* protocol. The *RESTful* protocol is used to promise that the server will be state-less. State-less meaning that a request from a client can not lock the server for other clients requests.

The different requests that can be sent to the server are defined in the Appendix Appendix H on page 148 that goes into deeper detail about the API.

The architecture of the system can be seen in Figure 4.1 where the clients are the colors blue, yellow, and red. Web server is green and the *Genomizer* server and its database are colored orange.

4.1.1 Genomizer clients

All the clients that are part of the *Genomizer* service are implemented using the architectural design pattern MVC (`Model View Controller`). This pattern is based around keeping the logic and data separate from its representation and user interface. The basic idea is shown in figure Figure 4.2.

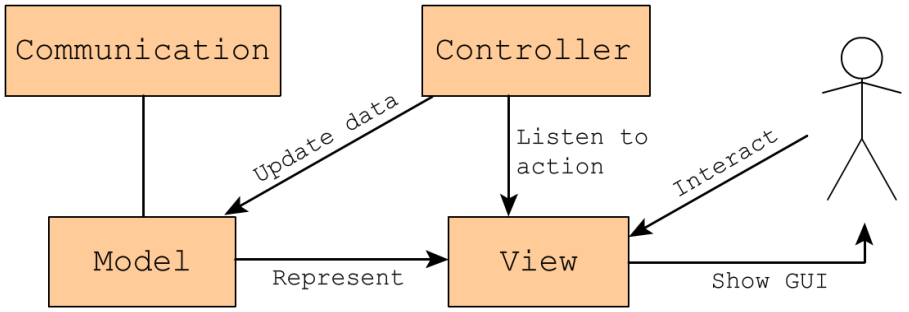


Figure 4.2: Basic MVC pattern.

The view will contain lists of experiments and annotations, data regarding process status, and other information. It will also present the user with windows and buttons that present and allow data to be manipulated. When a user interacts with the graphical interface the action will be handled by the controller using listeners. This in turn modifies the model; representing the data and information, as well as the operations available on it.

For this project a communication part is also necessary; often as a part of the model. As mentioned in chapter 4, the communication is performed via a *RESTful* protocol using `JSON` requests and responses. `JSON` is used because it is easy to parse and is directly compatible with the standard frameworks used in the web client.

The advantage of this approach lies in the separation of the interface- and logic-code; allowing one to be more easily replaced or modified. In our case the communication data and interface can be kept the same, while the client representation of it can be changed freely.

4.1.2 Genomizer server

The server is devided up in three main components, the front end, the data storage and the processing.

The front end consist of a handler for the *API* requests mentioned above. The handler interpret incomming request and passes them through to either processing or data storage depending request type.

Data storage makes up the back-end of the server. Its purpose is to communicate with file system and the database to return the correct information that was requested by the front-end.

The processing does the processing of files and convertation between file types. Processing is a heavy workload on the server and has to be scheduled for both time of day and parallel processing.

Figure 4.3 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. In the figure the front-end is the data transfer and communication node while data storage makes up the database node and processing the final process node.

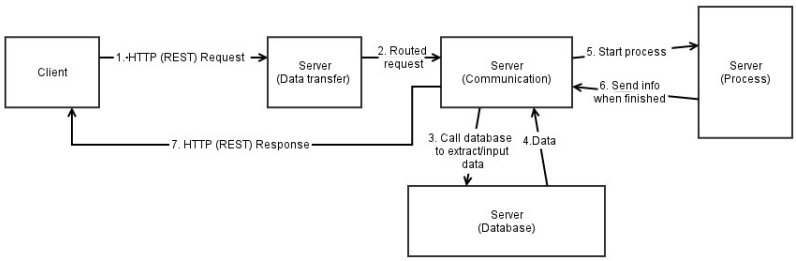


Figure 4.3: A simple flow diagram for the system

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 the communication part of the server.

When 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, for example, to process 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 closed and a new is opened with the next request.

5 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 divides into chapters based on the different clients.

5.1 General view

Since the Genomizer application is first and foremost about handling important data, it is crucial to allow the user to be in control but still to protect and preserve the data that is already in the system from non-authorized users. That is why Genomizer uses a log in screen to reassure that only authorized personnel can access the server data.

The workflow of Genomizer is intended to be as natural as possible for the users and to be easily integrated into their daily work routine. Furthermore, simplicity is favored, the clean and consistent design that stretches over platforms facilitating the tasks without adding the distraction of any unneccessary features.

5.2 Desktop client

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.

5.2.1 *Windows/OS X/Linux* application

The first thing a user will see when starting the desktop client is the login window (see Figure 5.1). The window prompts the user for user name, password, and a server IP to connect to. If the correct credentials are entered, the user will be logged in and taken to the next screen. If the user enters an invalid password, user name, or server, an appropriate error message will be displayed as seen in Figure 5.2. This feedback informs that user the login was unsuccessful.

After the user has been successfully logged in, the main window will appear (see Figure 5.3). The main window is built with tabs to simplify work by letting the user easily switch between different views for different work tasks. Each tab is described by appropriate name and contains related functionality. The main window also has a log out button. This button is of little importance, and is therefore located in the upper right corner.

At the bottom of the main window a status panel is located. The status panel gives feedback from different tasks executed by the user. When a task is executed successfully, the color of the status bar will turn green, and display a message. In case of an unsuccessful execution, the status bar will turn red, to indicate that something went wrong.

Figure 5.1: The login window.

Figure 5.2: The login window with an error message after a failed login.

From the search view (see Figure 5.4) the user can build search queries and look up existing experiments. The search view has been designed to have a similar layout and interaction as the advanced search tool on the site
http://www.ncbi.nlm.nih.gov/pubmed/advanced . The researchers are familiar with this site, so they can recognize interaction elements from it when using the search function of the desktop client.

To search for experiments the user can click the magnifying glass. This icon is well-known and often associated with searching. The icon is located next to the search field, so that the user can easily understand that the search field and the icon are connected. The user can also press the enter key to perform a search, without letting go of the keyboard, making the interaction faster. Next to the magnifying glass is a button for emptying the search field. The button has an icon depicting a trash can – a well-known metaphor for removing or emptying things.

From the upload view (see Figure 5.5) the user can create new experiments and upload files to them. When creating a new experiment the user is forced to fill in some fields. These fields have been given a bold-texted label, to indicate that they are of more importance than the others. A text below the fields also states that bold fields are forced. Non-forced fields are labeled with non-bold text.

To inform the user that information is missing, constraints has been put on the buttons for creating the experiment. If any forced field has not been filled in, or no files have been added for upload, these buttons will be grayed-out and cannot be clicked.

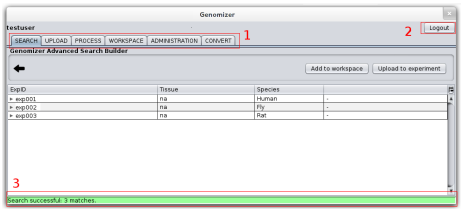


Figure 5.3: The Genomizer desktop client's main window. The window have tabs for different views (1), a log out button (2), and a status panel for feedback (3).

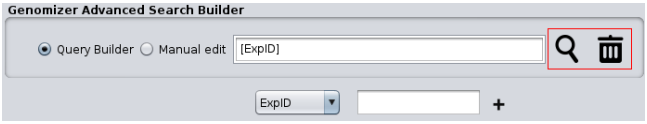


Figure 5.4: The search view of the Genomizer desktop client with the icons for search and emptying the search field highlighted.

For each file added to the experiment there is a progress bar. This bar gives the user feedback on upload process. Each file also has its size displayed after its name. This gives the user an idea of how time consuming the upload will be.

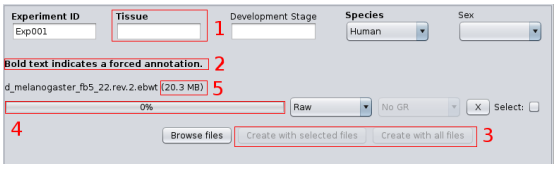


Figure 5.5: The upload view of the Genomizer desktop client. An empty forced field (1), as stated by the bold-texted label (2), makes the upload buttons (3) grayed-out. The upload progression bar (4) and the size of the file (5) gives the user an idea of how time consuming the uploading process will be.

The workspace tab lets the user easily manage files and experiments. Files and experiments in the work space are listed the same way as search results in the search view, making the design consistent throughout the system. The workspace view has easy access to the download and process functions.

The administration view (see Figure 5.6) is divided into two seperate views, one for managing annotations and the other one for managing genome releases. The division of the views makes the interface less cluttered and less confusing, and also increases the cohesion of the views. The user can easily switch between the views by clicking the the tabs on the left-hand side.

To improve the feedback when errors occur, an error dialog (see Figure 5.7) will be shown. The dialog explains what went wrong and why the error occurred. The user can find additional information about the error by clicking a button labled 'More info'. This information can be useful for system administrators, developers or support, but not for regular users (which is why it is initially hidden).

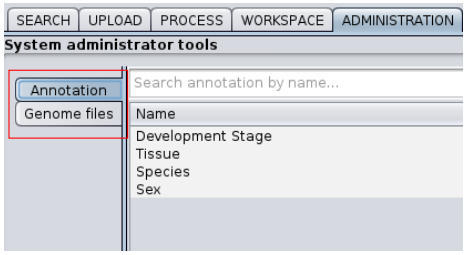


Figure 5.6: The administration view showing part of the view for managing annotation. The highlighted tabs to the left let's the user switch between views.

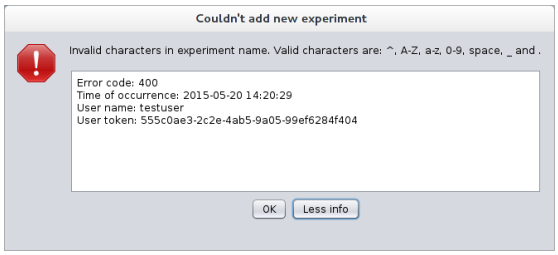


Figure 5.7: An error dialog that explains that the user have entered invalid characters for an experiment name.

5.3 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.

5.3.0.1 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.

- **Navigation bar:** A navigation bar is a menu that contains an overview of the functionality in the form of, in this case, tabs that leads the user to other views and that is always visible for the user to allow easy navigation. The navigation bar for the application can be seen in Figure 5.8. Since the most important parts of the application is to be able to upload, process and convert files, these are each a natural part of the navigation bar (although conversion is not yet implemented and therefore not shown in the figure).

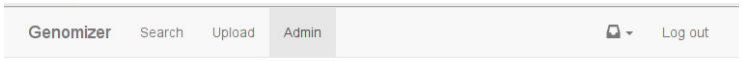


Figure 5.8: The web client navigation bar.

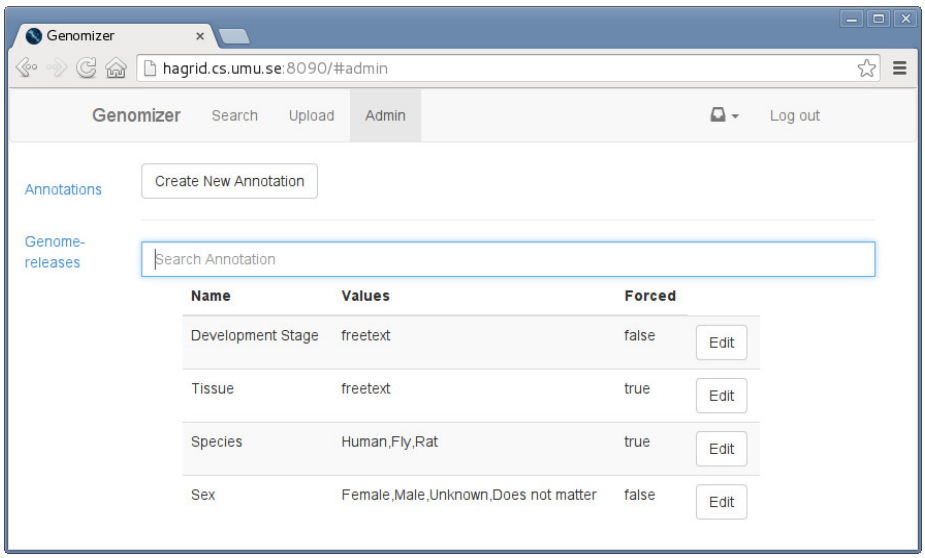


Figure 5.9: The web client administrator main view.

Figure 5.10: The login modal.

- **Main views:** A main view covers the entire page except for the navigation bar. 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. Figure 5.9 shows the administration main view.
- **Sub views:** A sub view is a part of a main view. In the case of the administrator view seen in Figure 5.9, 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, as does the annotation view in this case.
- **Modal views:** Modal views are opened on top of 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. An example of a modal view is the login view seen in Figure 5.10.
- **Panels:** Content that belongs 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 toolbars where operation controls available to the user are presented, for example, add and search experiment functionality in the upload view.
- **Popovers:** 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 navigation bar, that is, the main menu.

5.3.0.2 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 distinguish 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 element size. Light gray of varying brightness may also be used to highlight or distinguish elements.

5.3.0.3 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.

5.3.0.4 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 using checkboxes.

5.3.0.5 Processing

The interaction flow of the processing is adapted from the actual processing steps in order to help the researchers by increasing usability through providing a well-known but more optimized approach.

After having chosen an experiment for processing and entered the processing view, the user can choose the processing steps wanted and enter the correct files and parameters for each processing step as shown in Figure 5.11.

5.3.0.6 System administration

The admin page is built up by a number of components: the main view, the side bar, the create annotation view, the edit annotation view and the genome-release view. The first one is the main view which consists of a sidebar and an 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.

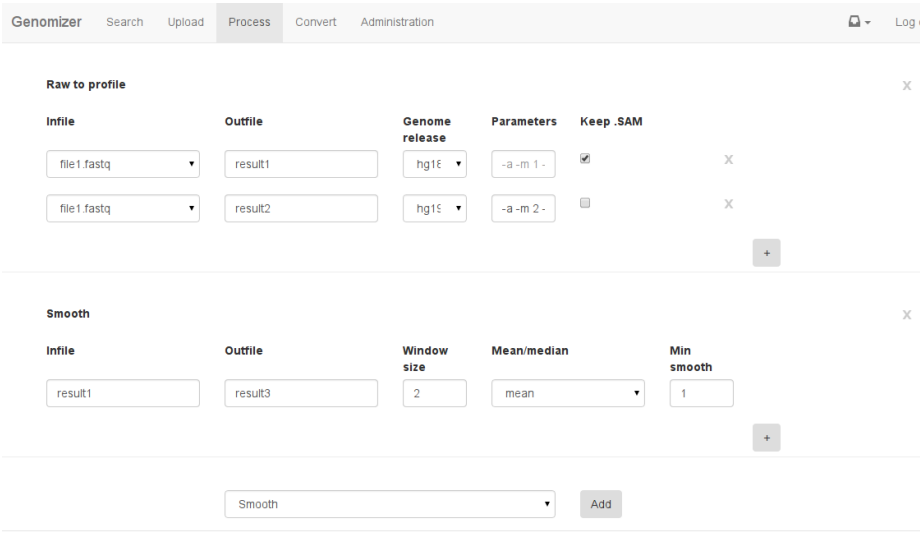


Figure 5.11: Files selected for upload.

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. Futher 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 from the database.For that reason we decided to ask if the user is sure of this action and of course 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.

5.4 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 plattforms. 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.

5.4.1 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.

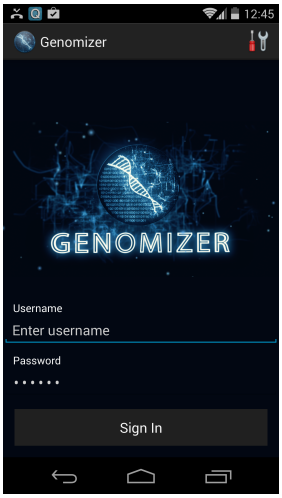


Figure 5.12: Login View

5.4.2 Search View

The design illustrated in Figure 5.13 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.

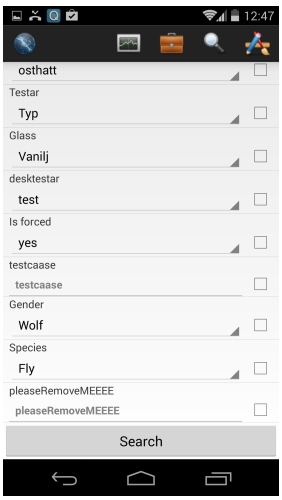


Figure 5.13: Search View

5.4.3 Search Results View

The design illustrated in Figure 5.14 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.

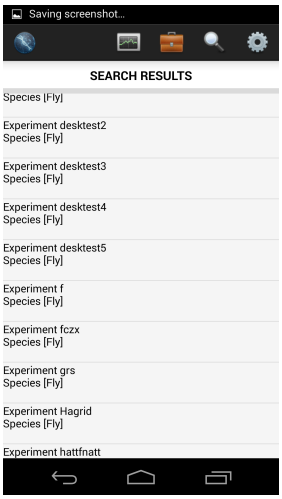


Figure 5.14: Search Result View

5.4.4 Experiment View

The design illustrated in Figure 5.15 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.

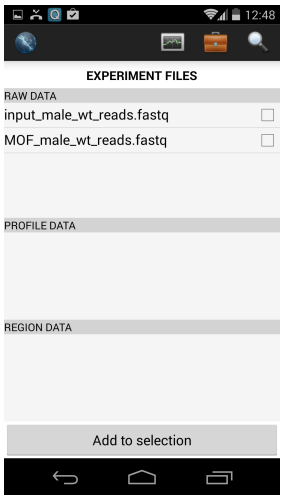


Figure 5.15: Experiment View

5.4.5 Search Settings View

The design illustrated in Figure 5.16 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.

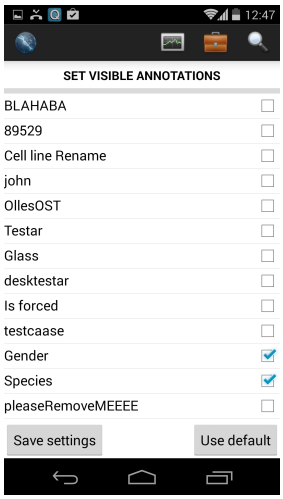


Figure 5.16: Search Settings View

5.4.6 Selected Files View

The design illustrated in Figure 5.17 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 the 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.

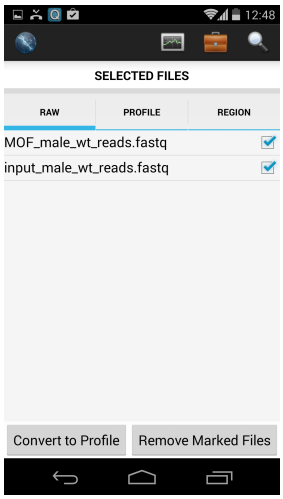


Figure 5.17: Selecting Files View

5.4.7 Convert View

The design illustrated in Figure 5.18 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 occuring. At the bottom of the view there is a button to press to start the conversion when all the parameters are set.

Figure 5.18: The Convert View

5.5 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.

5.5.1 Navigation bar

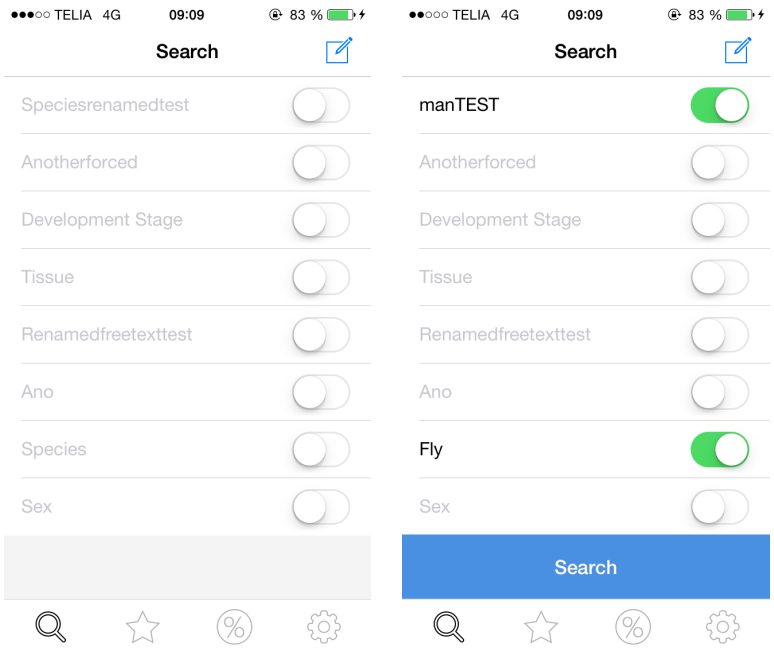
A navigation bar is used to make access to different main functionalities available at all times. Big and clear icons are used to show the user which view they all represent. It is also possible to simply swipe between the different views to increase the speed of which the advanced user can use the system.

5.5.2 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.

5.5.3 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 occurance 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 nonsence! The search button disappears when no annotation is selected to decrease the chance of user sending empty searches and to increase the understanding of the switches.



a b

Figure 5.19: The search screen.

Each annotation has a corresponding switch button as seen in Figure 5.19a-b. The button determines if the annotation should be included in the search request. This make 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 make more complex search queries and possibly make use of already accuried PubMed-search skills.

5.5.4 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 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 5.20. 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.

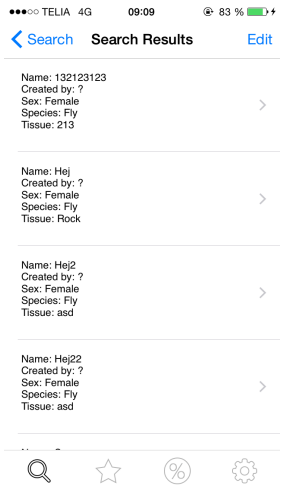
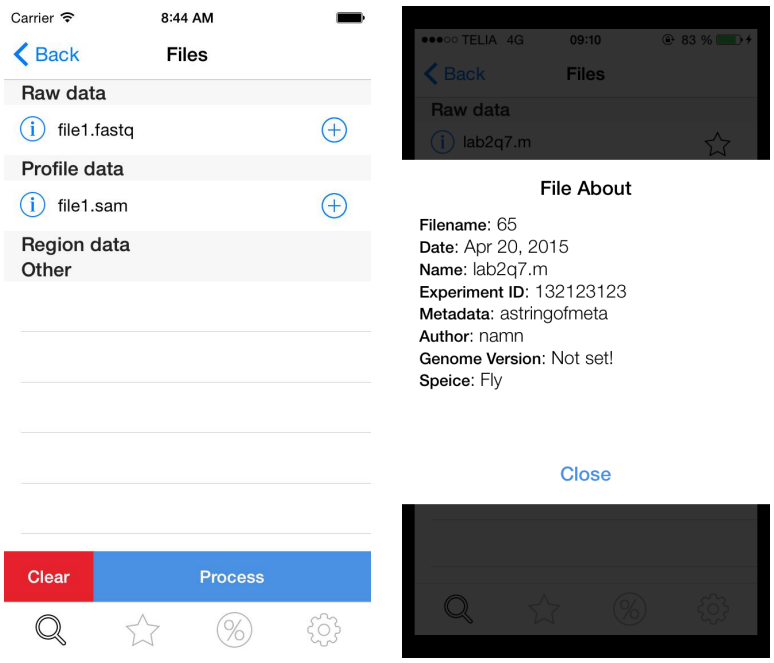


Figure 5.20: The search result view.

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 files view (see Figure 5.21a), 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. The purpose of the plus-symbol next to each file is to add as many files as the user wish to use when selecting processes. More information can be seen when the circled 'i' to the left of the filename is tapped, an example of this can be seen in Figure 5.21b.

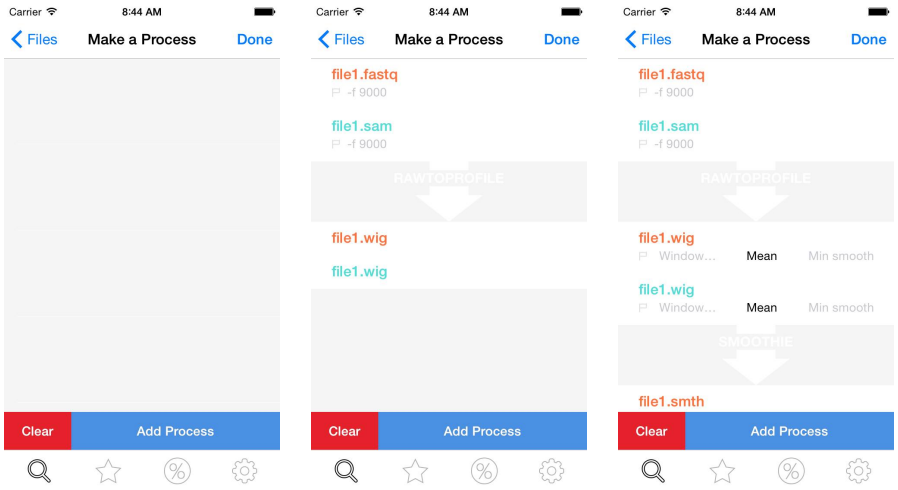


a b

Figure 5.21: The file view.

5.5.4.1 Create processes

The view to create a process is focused on the user's current workflow. The user wants to use the selected files from the files view as input and then select a specific process on those files, which creates output-files to be used as input-files in another process. This creates a sequence of processes which the server will execute one process at a time. Moreover each input-file for a process will be executed parallel with eachother. The input-files and the output-files are separated by the process which will be executed on the input-files. To make it easy to understand what will be executed on each step of the sequence the separator between input-files and output-files is the name of the process and an arrow from the input-files to output-files. The color of a output-file will have the same color as its corresponding input-file to track a file's conversion-process, from start to finish, see Figure 5.22a-c.



a b c

Figure 5.22: Creating a process

5.5.4.2 Processes

As visible in Figure 5.23, 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. The status is color coded to make it as easy as possible for the user to see which processes are in which state.

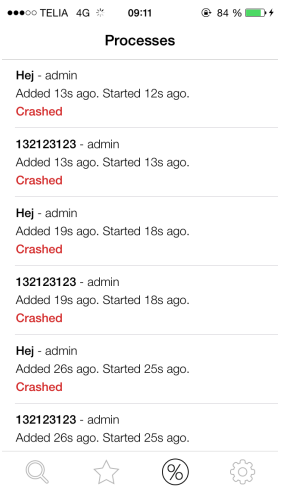


Figure 5.23: The process status view.

5.5.4.3 Alerts

Alerts are simple banners animating down from the top to give the user a headsup of what is going wrong and what is going the way it is expected. The user can tap the banner to dismiss it or if nothing is done it will animate away in 2 seconds. The banners were introduced to not stop the user with prompts which the user has to react with to be able to further use the system. A red banner, as seen in Figure 5.24, indicates an error and a white with a green icon, indicates something went as expected. The colors are used to allow the user to just notice which color pops down and give them somewhat of an understanding of what is going on. without having to read the text every time.

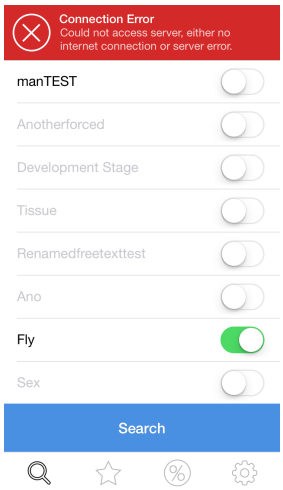


Figure 5.24: Examples of alert

6 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.

6.1 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 F a UML-diagram of the desktop client is presented. A basic overview can also be seen in Figure 6.1.

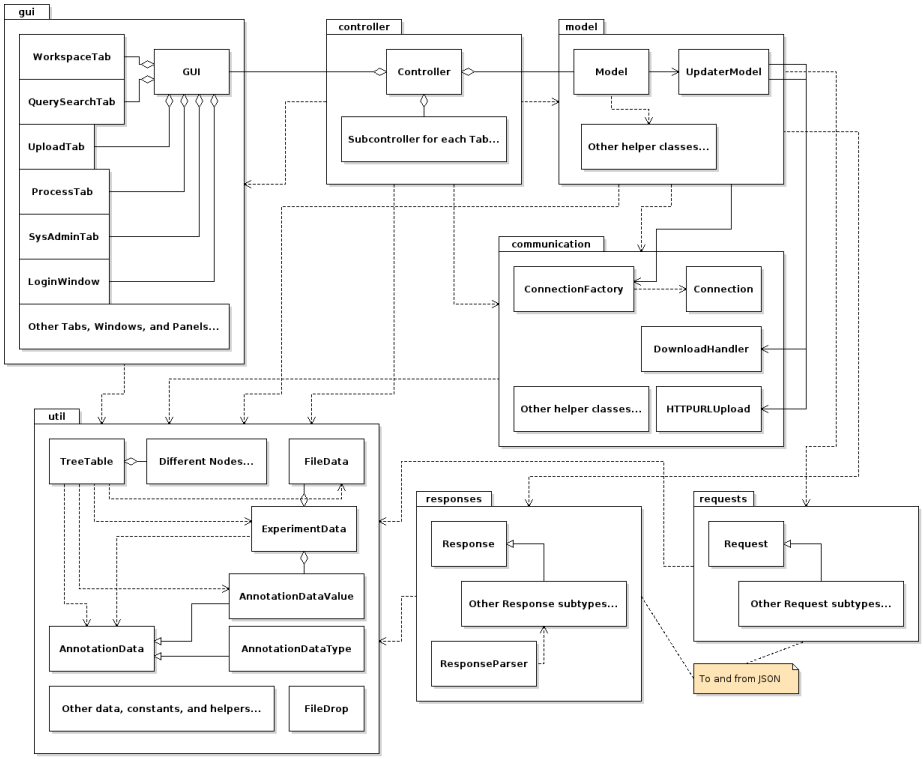


Figure 6.1: Overview of desktop client design. (Excluding SysAdmin related parts.)

6.1.1 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. In the *gui* package of Figure 6.1 these are shown.

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.

6.1.2 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 importing information such as the user token and list of ongoing downloads and uploads. This is shown in the *model* and *communication* packages of Figure 6.1.

6.1.3 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.

6.1.4 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.

6.1.5 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. Many of the actionlisteners have been divided into tab-specific classes.

6.1.6 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. As we can see in Figure 6.1, a lot of the other packages will use some of the data or funktionality within the *util* package.

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.

6.1.7 System Administration

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

Communication with the Server

All communication between the server and the system administration tab follows a line of steps. See Figure 6.2 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.

A communication example

As a more detailed example of Figure 6.2. Assume that the user clicks the 'Genome Files' tab in the 'ADMINISTRATION' tab. This will trigger an event (1) to be handled by the *SysadminTabChangeListener* (2) who will receive the event and execute the desired behavior of the tab, which is to directly show the available genome releases. This is done by sending a request to get available genome releases to the server and then parse the response.

In order to contact the server the *SysadminTabChangeListener* (2) calls the *SysadminController* (3) who uses a reference to the class *GenomeReleaseTableModel* (4) to call the method *getGenomeReleases()*. *getGenomeReleases()* will create a *GetGenomeRequest* using the *RequestFactory*. The request is then sent to the server through the *Connection* class (5). The

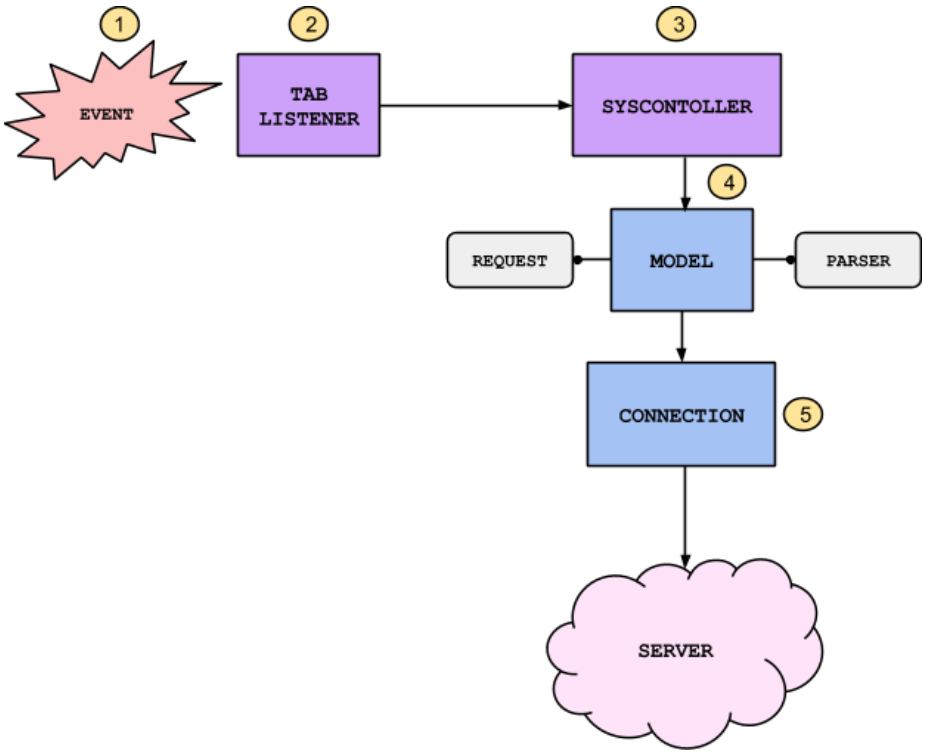


Figure 6.2: Communication Overview

response from the server is passed to the *ResponsParser* that parses the JSON respons into wanted *GenomeReleaseData[]* object. The genome release array is return all the way back to *SysadminController* (3) which updates *GenomeReleaseTableModel* with the new *GenomeReleaseData[]* and at last lets the GUI know that the data has changed through a new event (1). This will trigger the GUI to repaint and show the available data.

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.

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 desciption 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 recieved 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.

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 inconsistencies as well as making names easy to change.

6.1.8 Flow of the system

The sequence diagram in Figure 6.3 describes the flow of the system when the user presses the download file button and the diagram in Figure 6.4 describes how the desktop clients reacts to a login.

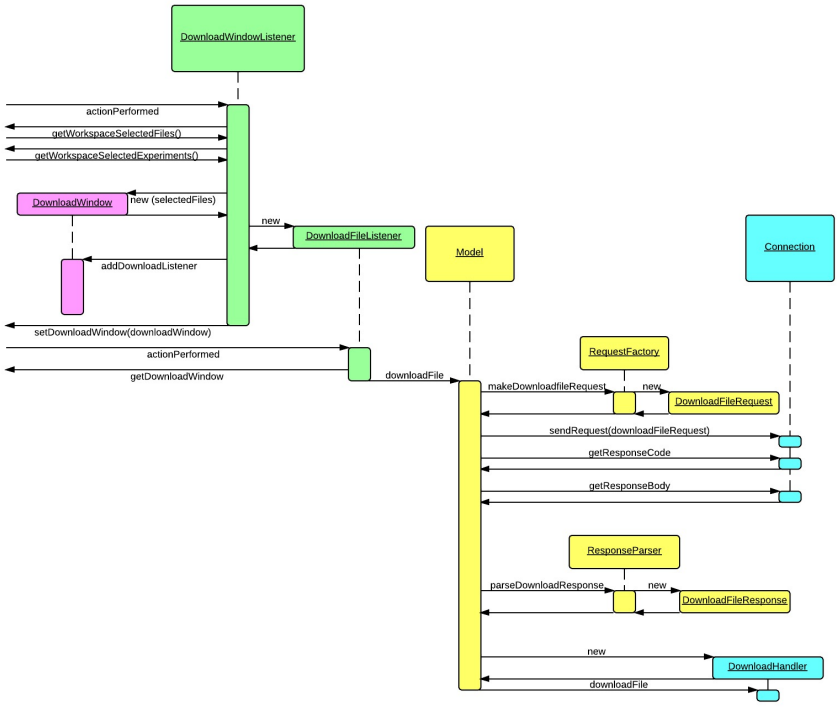


Figure 6.3: UML sequence diagram of downloading a file

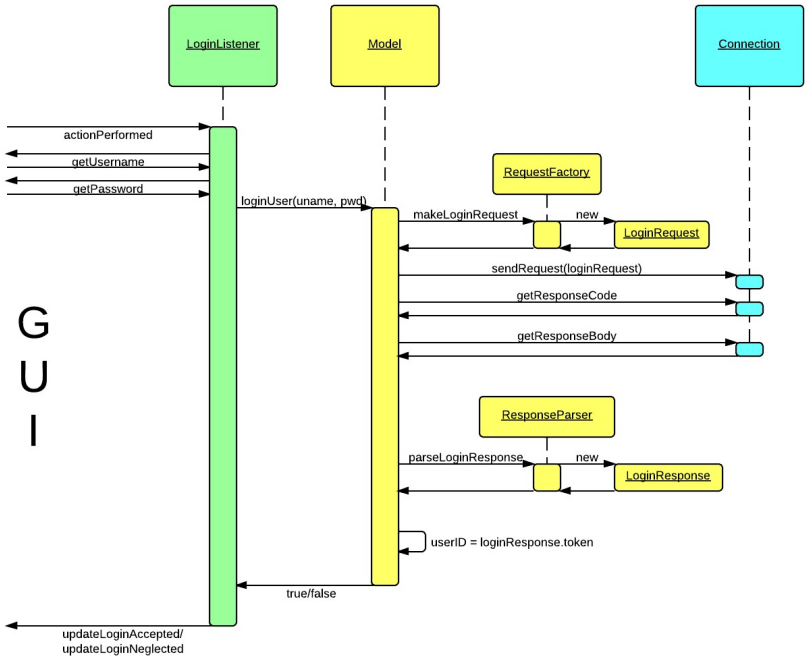


Figure 6.4: UML sequence diagram of login

6.2 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.

6.2.1 How the web application works

Figure 6.5 shows how the web application works in general. There is a user that interacts with a browser. A browser renders the DOM (Document Object Model, a convention for representing and interacting with objects in HTML) of the web application. How it does this is up to the browser. Different browsers might display it differently. The web app is based on the MVC pattern, but with the controller merged into the view and with a component called *Collection* being introduced. A collection is simply a ordered set of models. Models and collections will talk to the server to update themselves. 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 7.2.1 more information.

> **Example 2** *In the web app, there is a collection called* `Experiments` *which contains a set of* `Experiment` *models. Each of these models contains info about a specific experiment (for example, the name of the experiment and which files and annotations are incluced in it). The collection will retrieve experiments from*

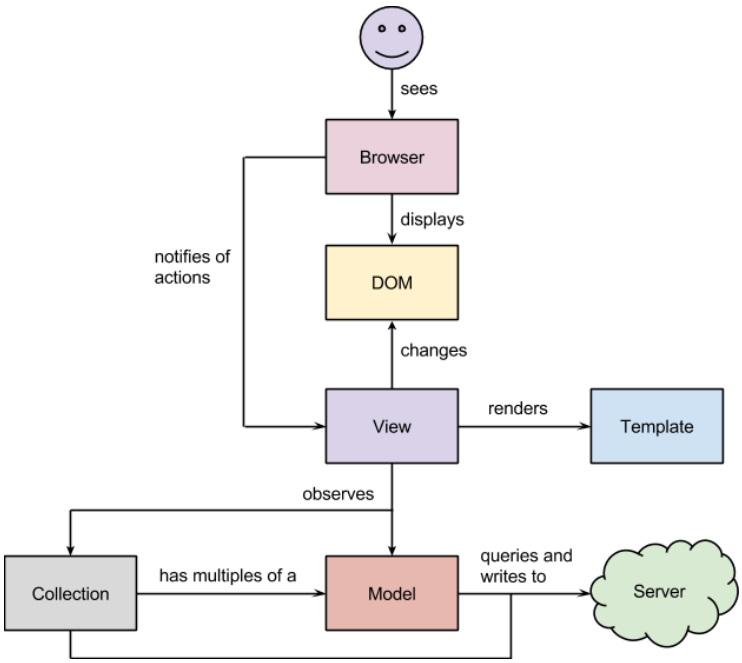


Figure 6.5: A general build of a backbone web app.

the server and update itself with a simple call to its `fetch()` method. After this, the collection is synced with the server data.

6.2.2 System overview

The web app is divided into the parts `Misc`, `Views`, `Collections` and `Models`. In Figure 6.6, an overview of the system is shown. The views are the parts in green, the collections the parts in yellow and the models the parts in red.

Example 3 *In Figure 6.6, the collection `Files` contains a set of `File` models. The model `Experiment` contains a `Files` collection. An `Experiment` model may be used by the collections `SearchResult` and `Experiments`.*

The parts in grey represent the *router* which belongs in the `Misc` category. It is responsible for rerouting links. This is done mainly when the user wants to go to another webpage by clicking a link. The router is "clever", however, and does not need to load a whole web page whenever a rerouting is triggered - instead it only loads the necessary parts of the new web page.

Example 4 *When a user clicks the search tab, the router navigates to `/search`. But instead of loading the whole `/search` on top of the page we are currently on, the router will keep the old navigation bar, open the search tab alone and put the search tab below the already existing navigation bar.*

The `Misc` category also holds the `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

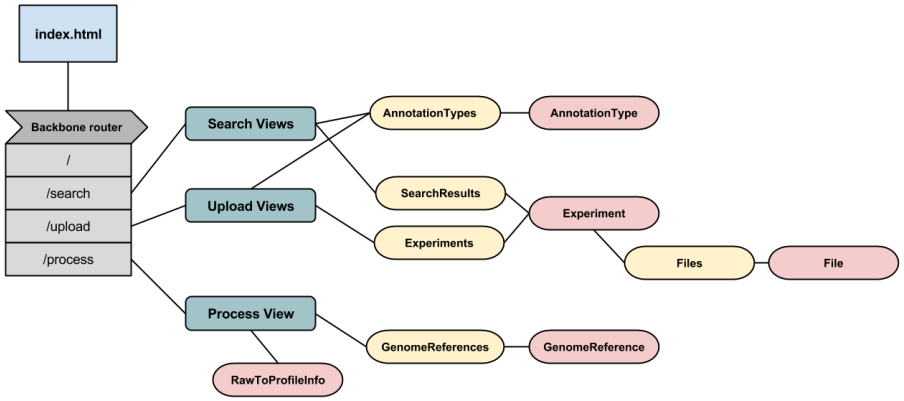


Figure 6.6: Overview of the relations between the different Javascript prototypes in the system.

events. The collections and models are responsible for holding the data.

6.2.3 Search

The search tab has three views that together make up the `Search Views` as they have been denoted in Figure 6.6. 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 experiments and files.

> **Example 5** *When searching for experiments, they and their contained files will be displayed. The user may choose to delete a file by marking it and then click a delete button. If this happens, the `Search Views` will receive the event and tell the model of the marked file to destroy itself. The model will then send a delete request to the server and disappear.*

In Figure 6.7 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` then triggers a "change event" that `SearchResultsView` listens to. When that event occurs, `SearchResultsView` knows that `SearchResults` has been changed, and rerenders itself.

6.2.4 Upload

The upload tab has three views, that together make up the "Upload Views" as we have denoted them in Figure 6.6. Unlike search (see section 7.2.1) 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.

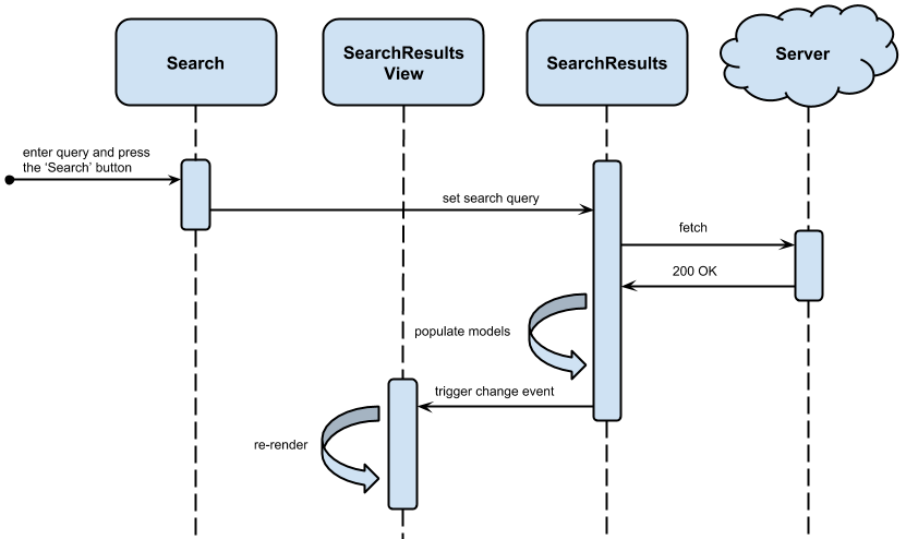


Figure 6.7: a sequence diagram showing what happens when a user enters a valid search query and results are fetched.

6.2.5 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 species of an experiment.

6.2.6 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 case of uploading itself to the server and thereby also updates the progress bar through events that propagate up to the GenomeReleaseFiles collection.

6.3 Android application

The following sections describe the system 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.

6.3.1 System overview

The Android application is divided into eight *packages*. These packages are `default`, `com`, `login`, `model`, `processing`, `search`, `search_result` and `selected_files`. All packages (except for `com` and `model`) contains one or several pairs of activities and fragments. An `Activity` is a single, focused thing the user can do. A `Fragment` is an object that helps to modularize the code and brings more sophisticated user interfaces.

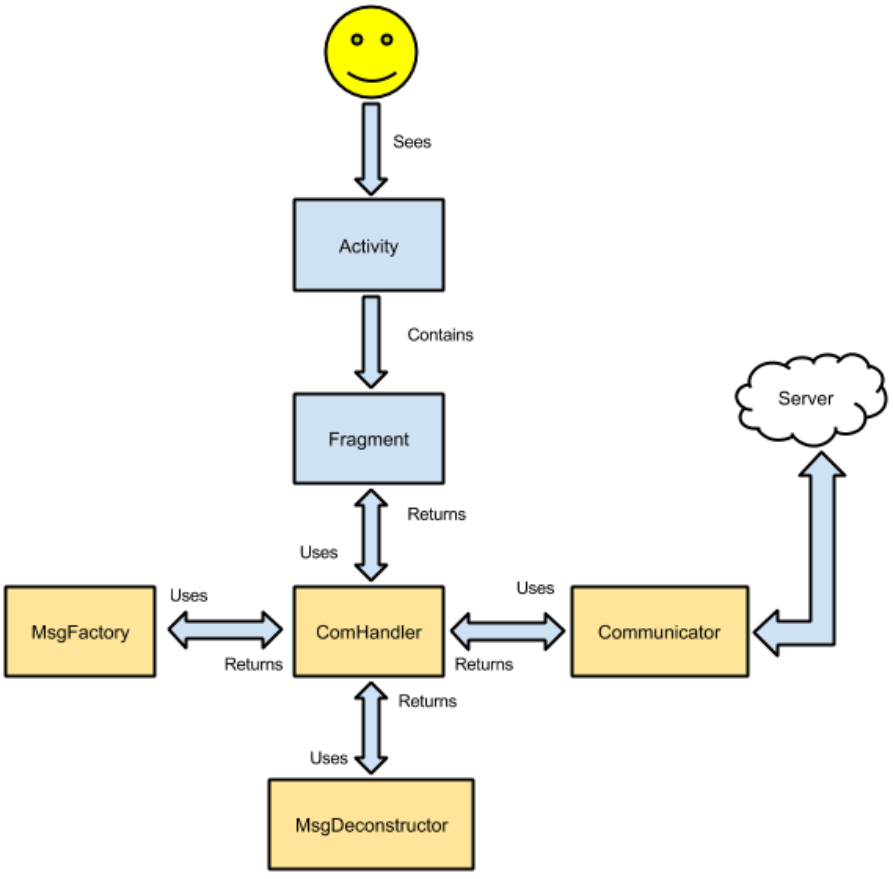


Figure 6.8: A generalization of how the Android application works.

The user will interact with an activity that holds a fragment. The fragment (which contains some logic) will tell the class `ComHandler` what action that should be performed. The `ComHandler` will construct a message by using `MsgFactory`. The message is then passed on to `Communicator` which sends the message to the server with REST. The `Communicator` returns the response to the `ComHandler` that parses it by using the class `MsgDeconstructor` and then returns it to the fragment. Hopefully, Figure 6.8 will bring some clarity.

6.3.2 Package overview

The `default` package only contains one class, `SingleFragmentActivity`. This class is the base of every screen in the application. It is responsible for the navigation in the application and to inflate the correct `ActionBar`. All other activities extends this class.

The `com` package is responsible for communication with the server. It also contains classes and methods for construction and deconstruction of JSON.

The `login` package contains the GUI and controller for the login screen. It also enables the user to select, add, edit and delete server URLs.

The `model` package holds information about experiments, annotations, files etc. found on the server.

The `processing` package is responsible for displaying processing parameters etc. for when the user wants to process a certain file.

The `search` package handles searches by either selecting annotations, or by manually typing in PubMed style.

The `search_result` package handles the results of the search by displaying the experiments found. When an experiment is selected, the files associated with the experiment is displayed.

The `selected_files` handles all the files that has been selected. They are sorted into raw, profile and region. The raw files can be selected for processing from here.

6.3.3 Class description

Fragment classes

FileListFragment	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.
SelectedFilesFragment	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.
RawFragment	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.
ProfileFragment	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.
RegionFragment	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

ConverterFragment	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 retreives 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.
ProcessFragment	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 retreive current conversion status from the server. The information is visualized in a listview.

Fragment classes

LoginFragment	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.
SettingsFragment	SettingsFragment is used for selecting which server to use. There is also choices for adding, deleting and editing server locations.
SearchListFragment	SearchListFragment is the search-view for the *Genomizer* app, the annotations that can be chosen are downloaded from the server and they are, as such, dynamic.
SearchPubmedFragment	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.
SearchSettingsFragment	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.
ExperimentListFragment	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.

Model classes

ComHandler	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.
Communicator	Communicator is used to manage the sending and receiving of messages between the server and the application using a http connection.
MsgFactory	MsgFactory creates the JSON messages that can be sent to the server.
MessageDeconstructor	MessageDeconstructor interprets JSON messages and returns appropriate information.
GenomizerHttpPackage	GenomizerHttpPackage stores the body and status code of an http-response.
GeneFile	GeneFile is used to store and transfer the information of a genome file.
Annotation	Annotation is used to store and transfer one or several annotations and their value.
Experiment	Experiment is used to store and transfer information about an experiment.
ProcessingParameters	ProcessingParameters is used to simplify the transfer of parameters in a processing request.
DataStorage	DataStorage is used to save lists of GeneFile objects on the device locally. DataStorage is a static class, which makes it possiblr to access the saved data anywhere in the application. This simplify the transfer of files between activities.
GenomeRelease	The GenomeRelease class is used to store information about a genomerelease.
ProcessStatus	ProcessStatus is used to store information about the status of a process.
Genomizer	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.

6.4 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.

6.4.1 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.

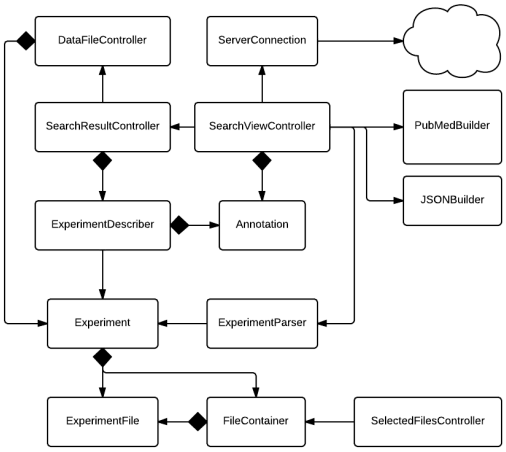


Figure 6.9: UML diagram.

Figure 6.9 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 6.1.

Class	**Description**
Annotation	Contains information about an annotation and can format the annotation name to an aesthetically more pleasing representation.
DataFileViewController	Controls the File view presented in Figure A.64. It contains a reference to an experiment and lists all its files in a table.
Experiment	A class that contains information related to an experiment, as well as its files.
ExperimentDescriber	Generates a description of an experiment using annotations chosen by the user.
ExperimentFile	Contains information about a file from an experiment.
ExperimentParser	Parses experiment information from a NSDictionary to an Experiment object.
FileContainer	Contains files and sorts them by file type.
JSONBuilder	Creates different JSON requests.
PubMedBuilder	Creates a pubmed search query.
SearchResultController	A controller class for the Search Results view presented in Figure A.62. 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.
SearchViewController	A controller class for the Search view, see Figure A.59. 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.
SelectedFilesController	A controller class for the The selected files view shown in **??**. The selected files controller contains information about files saved by the user.
ServerConnection	Sends and receives JSON objects to and from the server.

Table 6.1: 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.

6.5 Server

The design of the servers system is based around several parts. These parts consists of: communication, conversion, processing, storage and file transfer. Following is a more detailed description of each part.

6.5.1 Communication

The server is based around a `RESTful` protocol, where clients send requests on a non-persistent connection and the server responds to these requests. All communication is initiated 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 way the calls flow in may be seen in figure Figure 4.3 on page 4.

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 $file$s 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 H.

6.5.2 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 **??** 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.

6.5.2.1 Executor

The executor class, as seen in figure 5.2.1, is a 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.

executeCommand	*executeCommand* is a private method that is being used by the *executeScript*, *executeProgram* and *executeShell-Command* methods. Firstly a *processBuilder* 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 *stringBuilder* object. When the command has been executed the recorded string is sent back to the caller.
executeScript/executeProgram	Both methods are very similar. The difference is that *executeScript* 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. `shell resources/script.sh`.
parse	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 *processbuilder* takes a *String array* as argument. With the help of a tool called *stringTokenizer* the string is parsed into a *String array* separated on spaces.
cleanUp	Receives a *stack* 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.

6.5.2.2 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.

6.5.2.3 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`

procedure	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 then 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.
runBowtie	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.
sortSamFile	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. • *makeConversionDirectories* – 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. • *initiateConversionStrings* – 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.
getRawFiles	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.
makeConversionDirectories	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.
initiateConversionStrings	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.

validateParameters	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.
checkBowTieFile	Checks that bowtie succedded to run and that the result is ok. Checks that bowtie created the file it should and that the size of the file is not zero. If everything was correct it returns true.
validateInFolder	Perform a check on the parameter inFolder that should be a string with a path to where the files to be processed should be. If the string ends with a "/" it gets removed from the string.
runSmoothing	When implementing the scripts to create profile from raw we realized that the smoothing script used alot of memory to run, so we decided to convert it and try to optimise it. The result was a improvement and in this method we uses the smoothing a version of the smoothing that got approved from the customers. The method sets up all parameters the way the smoothing class wants them in and fixes all the paths then we run the smoothing, The method checks whether ratio calculation have been run before smoothing or not and sets the paths and parameters accordingly.
isSgr	Gets a string that should be a filename with the file extension and checks if the file extension is ".sgr", if so it returns true.
correctInFiles	Takes an array of File objects and checks that it contains a correct amount of raw files to process.
doRatioCalculation	Initiates a string using the incoming parameters and executes the the script to do Ratio calculation, uses the method executeScript to execute.
checkBowTieProcessors	Bowtie has a parameter where the number of processors used can be specified, we want to restrict the user from being able to run bowtie on all the cores on the server cause that would slow it down. Instead we make it so that bowtie runs on all but 2 of the available cores on the system.
verifyInData	Makes a initial check so that all the incoming parameters to the procedure method not is null. Also checks that the array string with parameters is correct size, not zero and not bigger then eight.

6.5.2.3.1 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.

6.5.2.3.2 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.

6.5.2.4 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.

6.5.2.4.1 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.

6.5.2.4.2 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.

6.5.2.5 ParameterValidator

Used by RawToProfileConverter to validate the parameters used by the steps in RawToProfile-Converters procedure.

sam_to_readgff_v1	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 *reads_gff*.
readsgff_to_allnucsgr_v1	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 `allnucs_sgr`.
ratio_calculation_v2	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.

6.5.2.5.1 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.

6.5.2.5.2 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.

6.5.2.5.3 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.

6.5.2.6 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.

6.5.2.7 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.

6.5.2.8 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.

6.5.2.9 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.

6.5.2.10 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.

6.5.3 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 **??** for examples when using the PHP-scripts.

In Figure 6.10 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.

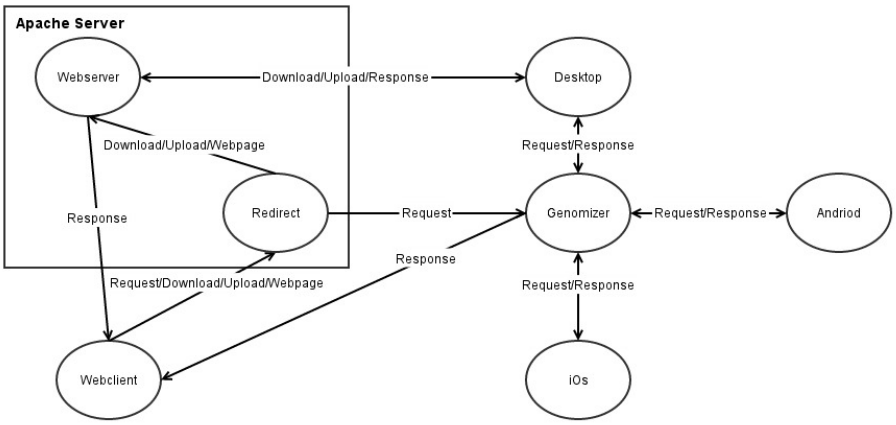


Figure 6.10: The different types of messages sent between the systems.

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 6.11. 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.

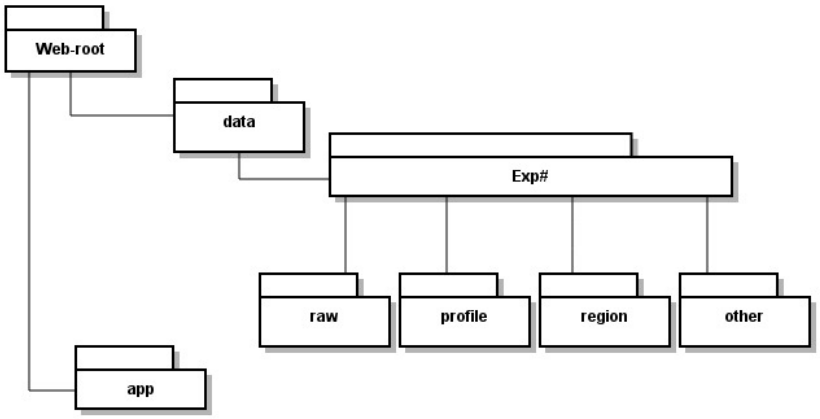


Figure 6.11: Illustrating the current file tree on the server machine.

6.5.4 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 6.12. 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.

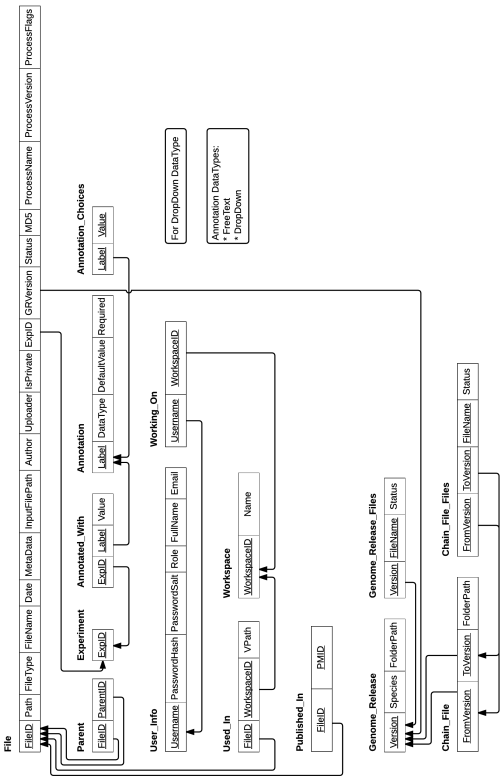


Figure 6.12: The database schema

6.5.5 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*.

6.5.6 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 G.1 in Appendix G.

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 G.2 in Appendix G.

6.5.7 Interaction

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

6.5.7.1 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.

6.5.7.2 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.

6.5.7.3 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.

7 Implementation

This section contains descriptions of the implementation and how different parts of the system are designed, 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 created.

7.1 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 grapical 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.

7.1.1 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.

Since the client were developed using the Scrum developing methodology, testing were an important part of the development. The client were built by using Test Driven Development, so all functionality should have been tested thoroughly during development. Use cases were later put together to further test the implementation and to make sure the program is working as intended.

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

7.2 Web application

7.2.1 Frameworks

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

7.2.1.1 Backbone

Backbone[11] is a light-weight framework that loosely follows the MVC pattern. Out of the MVC components, Backbone only has models and views, and the view behaves much like a combination of both a view and a controller. `Models` are the parts of code that retrieve and

populate data. `Views` are the HTML representation of models, and they change as models change.

> **Example 6** *When the `Experiment` model is populated, which may happen when the model fetches data from the server, it is immediately presented on the view that contains that experiment. The view itself does not have to manually get any data from the `Experiment` model.*

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). Backbone is used because it allows more structure in the web application. With more structure, it is easier to collaborate as the work can be divided - keeping the Javascript code in various model, collection and view files.

7.2.1.2 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 buttons from scratch, deciding on colors, how big they are, and micromanaging how they fit with everything else on the page, bootstraps templates that handles all of that, leaving the developers able to focus on architecture. Bootstrap is used to save time on development and make the design of the web app easily customizable.

7.2.1.3 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.

7.2.1.4 JQuery

JQuery[16] is a popular code library that handles AJAX calls and DOM manipulation, and makes the code more compact and readable.

7.2.2 Technologies used

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

7.2.2.1 AJAX

AJAX[13] (Asynchronous Javascript and XML) 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, which is the case in the Genomizer 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 do not use AJAX have to). For example, when the search button is clicked in the navigation bar of the web app, only

the bottom half of the website is updated, and displaying the search view. The navigation bar does not have to be reloaded, but remains as it is on top.

7.2.2.2 JSON

JSON[14] (Javascript Object Notation) is a format that is primarily used to transmit data between a server and a web application instead of using XML or other formats. JSON is formatted as easily readable text consisting of attribute-value pairs. JSON was used in this application because JSON uses the same syntax as Javascript and therefore no parsing is needed, as opposed to the usage of e.g. XML. JSON also works well together with Backbone as it has integrated methods using the JSON format.

7.2.3 Testing frameworks

Three libraries are used to make testing easier: Chai, Mocha and Sinon. Together they let the developers make a page for testing where all tests and results will be shown visually. These libraries or testing frameworks will be discussed below.

7.2.3.1 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.

7.2.3.2 Sinon

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

7.2.4 Web app 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 that 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.

7.3 Android application

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

7.3.1 Login request

When the user starts the application and is prompted for a login, the following sequence of actions is performed by the system (see Figure 7.1).

- 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 unpack the token and send it to Communicator for storage. ComHandler will 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 LoginFragment will be terminated at that point.

Figure 7.1: Sequence diagram for a login-request

7.3.2 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 7.2).

- 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 GHTTP[1] to preserve both the head and body from the server response. Then the GHTTP is returned to the ComHandler.
- The ComHandler will initalize the MsgDeconstruct and send the collected `JSON`/-object containing the search result from the server to the deconSearch method.

[1] Genomizer HTTP package

Figure 7.2: Sequence diagram for a search-request

- 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 ExperimentListFragment that will present the results for the user on the screen.

7.3.3 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 theese steps (see Figure 7.3).

- When the ConverterFragment is being creeated, the retrieval of the genome-releases is started. ConverterFragment calls the ComHandler with the getGenomReleases method and waits for response.
- The ComHandler will then initalize MsgFactory and call createRegularPackage that will return a `JSON`/ object to be used for the communication with the server. The ComHandler then initalizes 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 initalize 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 7.3: sequence diagram for a request of Genome-releases on the server

7.3.4 Request for conversion of RAW files to profile-data

When the user has choosen 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 7.4).

- 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 initalizes 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 initalizes 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 retreive the responsecode from the package. ComHandler then will return a boolean back to the ConverterFragment, true if the conversion started successfully otherwise false.
- The ConverterFragement 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 7.4: Sequence diagram for a RAW to Profile -data conversion request

7.3.5 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 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 7.5).

- 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 initalizes a MsgFactory and request a `JSON`/ package through the createRegularPackage method. Then initalizes a Communcator 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 respons 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 7.5: sequence diagram for a request for process status on the server

7.3.6 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 **??** have been tested visually and through running as no viable way of unit testing them was found.

Most of the classes labeled model in **??** have been tested with a test driven developlment 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.

7.4 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. Libraries and framework that has been used is mostly Apple's standard framework for iOS applications called UIKit. A few custom interface object has been created to better suit our and our customers needs.

Communication with the server is done with a HTTP REST protocol and JSON objects are sent and received.

7.4.1 Testing

The logic in the application has been tested with XCTest which exists in the program Xcode. The graphical interface has both been tested throughout development both by the developers and by the customers. Furthermore a systemized testscript has been implemented using UI Automation which tests the application in the form of a user and runs through a set of scenarios. UI Automation is an instrument in Xcode which uses scripts written in JavaScript to simulate user interaction with the app.

7.5 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.

7.5.1 Communication

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

7.5.1.1 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*.

7.5.1.2 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 created when a user sends a login is stored in the server memory

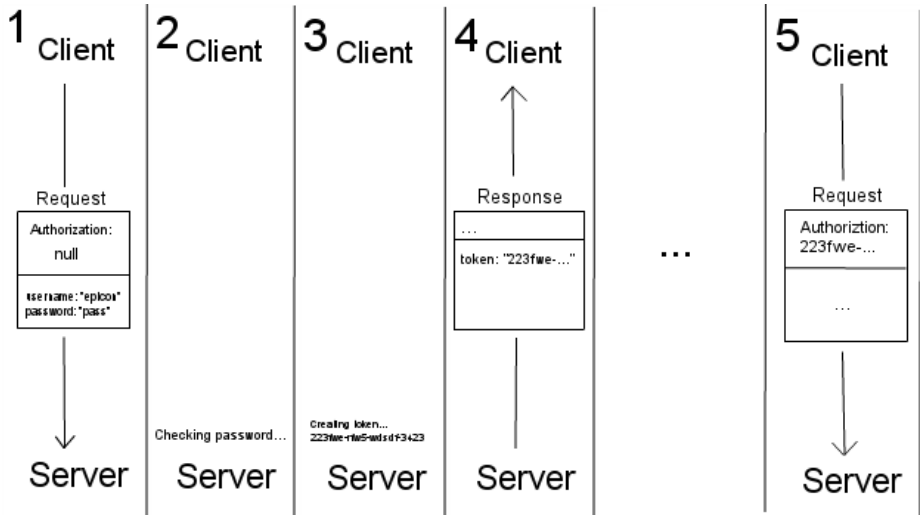


Figure 7.6: 1. The user sends a login request without any authorization token. 2. The server checks the given password. 3. The server creates a unique token for the user. 4. Server sends the token back to the user in a response. 5. Now that the user has a unique token, the token is placed in the header whenever the user sends another request.

until the user sends a logout request.

7.5.1.3 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 InactiveUuidsRemover 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 of with the flag "-nri".

7.5.1.4 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

7.5.1.4.1 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 doorman which then sends the response to the client.

7.5.1.4.2 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.

7.5.1.5 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.

7.5.1.6 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.

7.5.1.7 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.

7.5.2 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.

7.5.2.1 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.

7.5.3 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.

7.5.4 Data Storage

The following text describes the different classes the Genomizer server uses to communicate with the database and the file system. All the communication with the database happens through the DatabaseAccessor-class, which uses several helper classes that contain methods with the actual logic. These relationships are visualized in Figure 7.7.

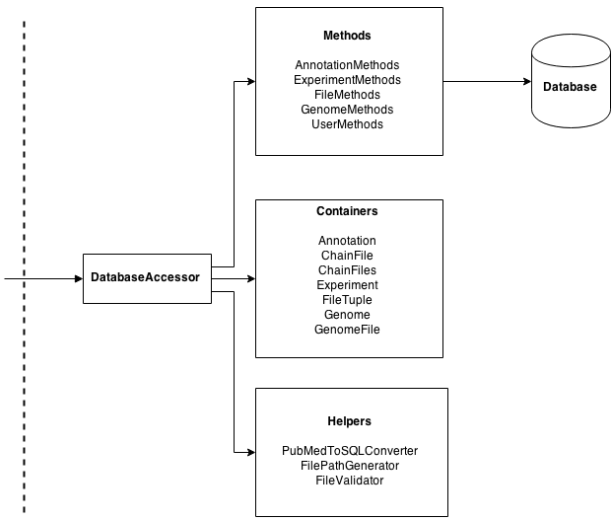


Figure 7.7: Conceptual overview - The implementation of data access and storage

7.5.4.1 DatabaseAccessor

`DatabaseAccessor` is a class that serves as an API for the *Genomizer* database - 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. For more details see the class diagram, Figure G.1, in Appendix G.

7.5.4.2 Containers

The classes grouped under the name Containers are used for representation of domain specific objects. An `Experiment` is a container for an experiment's annotations and their values as well as a list of files corresponding to that experiment. These files are contained in `FileTuple` objects which holds information regarding filename, type, size etc. The `Annotation` class contains information about label, whether or not the label is required, the default value and annotation choiches for drop down menus. More information can be found in the class diagram Figure G.2 in Appendix G.

7.5.4.3 Methods

The `DatabaseAccessor` uses helper classes to execute *SQL* queries. These helper classes are grouped by their area of responsibility:

- Annotation methods
- Experiment methods
- File methods
- Genome methods
- User methods

These classes execute the basic *SQL* functions create, read, update and delete (CRUD). The use of *prepared statements* (also known as parameterized queries) safeguards against *SQL* injection.

7.5.4.4 Helpers

The creation of folders is handled by the `FilePathGenerator` class. Folders are created for a new experiment when the experiment is added to the database, including subfolders in preparation of files to be uploaded. All files are divided into folders corresponding to experiment id, except for genome release files that are divided in regards to species.

The `PubMedToSQLConverter` class converts a *PubMed* string to a *SQL* query. The simplest format for a *PubMed* string is *value[Label]* but the user can enter the annotation labels and values in combination with the the logical operators `AND`, `OR` and `NOT`. Parentheses are used for disambiguation.

> **Example 7** *To search for all raw files that Per created the PubMed string should be:*
>
> *raw[FileType] AND Per[Author]*

Searches can be made on labels corresponding to file attributes as well as experiment id. Searching for the empty string will return all experiments in the database.

To validate filename a check against a *regex* is done in the `FileValidator` class.

7.5.4.5 Testing

WARNING! The unit tests in the `database.test` package have a database dependency. Do not run any of the tests found in this package on a database that is in use. All tuples are removed from the database upon test completion. Every instance of unit testing should start with an empty database and finish with an empty database to avoid test interdependency.

All the unit tests use the JUnit package and 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.test.unittests` package. The scripts for adding the test tuples and clearing the test database tables can also be found in the `sql` package.

7.6 Limitations

The *Genomizer* system has some limitations and known problems that needs to be mitigated. In Appendix J 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". *BioViz*. 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) 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>

Nomenclature

Bowtie Program the preforms parsing of the raw data and counts base pairs

Chain Files Genome release files with small alterations to previous genome releases.

Genome Releases Constant research gives more understanding, new genome versions are often found.

Genomizer Collective name for the project.

GEO Centralized database where article data can be found.

LiftOver Converts genome release versions.

Profile Data converted to a human readable file for analysis.

Raw Collection word for files that are the result from a *DNA*-sequencing machine.

Region Region data is small parts of the profile data.

A 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.

A.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.

A.1.1 Login and startup

When you start this application the first thing that's displayed is a login screen, as illustrated in Figure A.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.

Figure A.1: Screenshot of the login screen.

The application is built with tabs, as illustrated below in Figure A.2. Each tab contains separate features of the application. There are five tabs: Search, Upload, Process, Workspace and Administration.

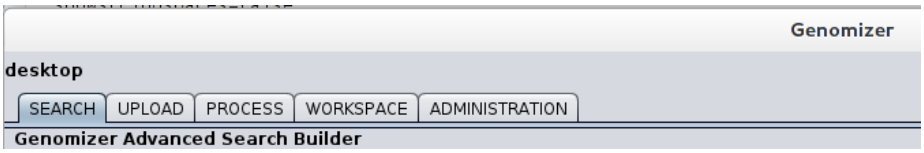


Figure A.2: Illustration of the different tabs of *Genomizer*Desktop and displaying the Search tab.

A.1.2 Search

The first tab you see after logging in is the Search tab, illustrated in Figure A.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 build the query. 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 plus button is only available in the last row and it adda another row to the query. The minus button is used to remove a row and it exists on every row except if there is only one row in the query. 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 A.3 below.

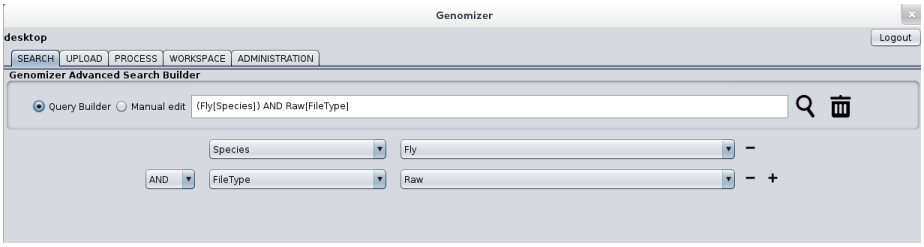
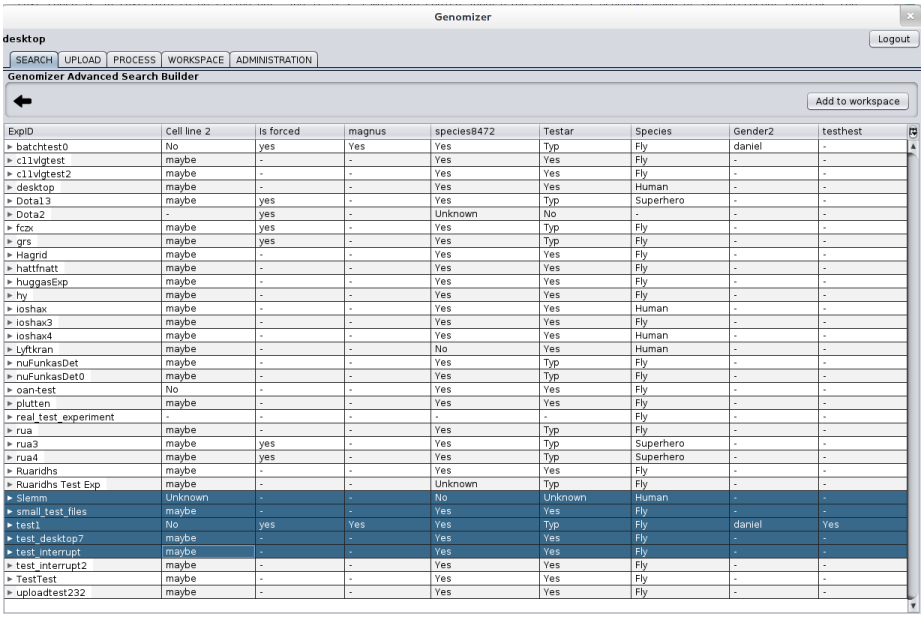


Figure A.3: Illustration of a query, made by the query builder.

A.1.2.1 Search results

When you press the search button the search tab will change it's view to display the search results as illustrated in Figure A.4. The results are displayed as experiments in a tree table. Each row is an experiment that cn be expanded to show more information and 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. You 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, you can click the *Back* button. There is also a button called *Add to workspace* for adding the selected files or experiments to the workspace. The last button, *Upoad to experiment* is used to upload more files to an experiment.



ExpID	Cell line 2	Is forced	magnus	species8472	Testar	Species	Gender2	testhest
batchtest0	No	yes	Yes	Yes	Typ	Fly	daniel	-
c11vlgtest	maybe	-	-	Yes	Yes	Fly	-	-
c11vlgtest2	maybe	-	-	Yes	Yes	Fly	-	-
desktop	maybe	-	-	Yes	Yes	Human	-	-
Dota13	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	-	-
hy	maybe	-	-	Yes	Yes	Fly	-	-
ioshax	maybe	-	-	Yes	Yes	Human	-	-
ioshax3	maybe	-	-	Yes	Yes	Fly	-	-
ioshax4	maybe	-	-	Yes	Yes	Human	-	-
Lyftkran	maybe	-	-	No	Yes	Human	-	-
nuFunkasDet	maybe	-	-	Yes	Typ	Fly	-	-
nuFunkasDet0	maybe	-	-	Yes	Typ	Fly	-	-
oan-test	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	-	-
Ruaridhs	maybe	-	-	Yes	Yes	Fly	-	-
Ruaridhs 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 A.4: Illustration of search results.

A.1.3 Upload

If you need to upload files to the database it can be done through the upload tab. When the tab is pressed you get presented with a button to create a new experiment shown in figure A.5.

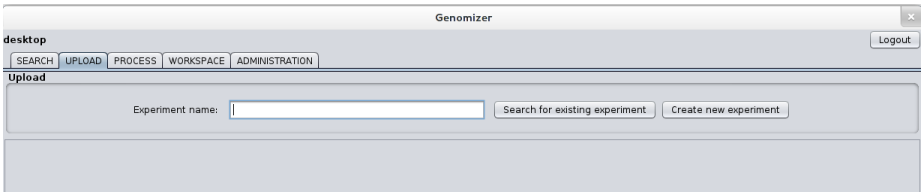


Figure A.5: Illustration of the starting view of the upload tab.

A.1.3.1 Existing experiment

In order to upload files to an existing experiments you need to search for the experiment in the search tab and then press the *Upload to experiment* button. When this is done the experiment information get retrieved from the server and presented to you. For an existing experiment no editing of the annotations can be done at the moment. So after retrieving the wanted experiment you can press the "Browse files"-button to add files to the experiment. Then a file browser window pops up, it is illustrated in figure A.8. Here you cn select the files you want to add to the experimant. The files will be added to the upload tab and there will be some new choices available for you. Each file will be associated with one file row, this is also shown in A.6. 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 you simply click 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.

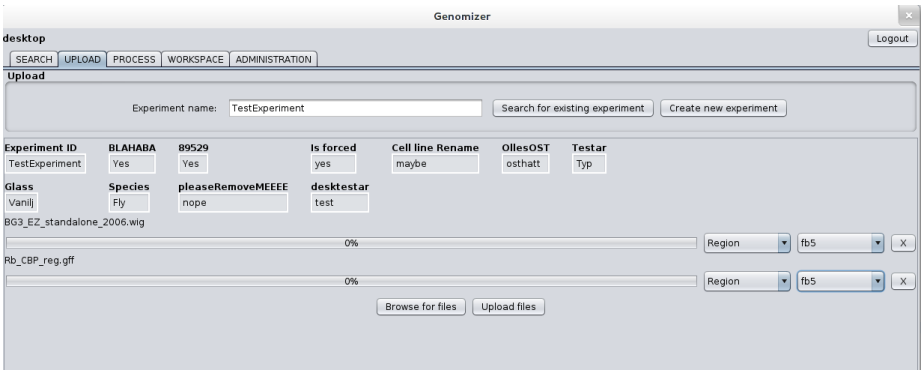


Figure A.6: Illustration of the add to existing experiment part of the upload tab.

A.1.3.2 New experiment

The first thing you need 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 an textfield to be filled out, and if it's a multiple choice annotation there is a dropdown menu of the different choices. The annotations who have bold text 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 A.7. In order to add files to this experiment you need to press the "Browse files"-button where a filebrowser appears and choose 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 A.1.3.3 below. The other three button are used in the same manner as in section A.1.3.1 above. When all the annotations that are needed is filled and the associated files are added you press the "Create with all files"-button to create the experiment. The "Create experiment with selected files"-button is discussed in section A.1.3.3 below.

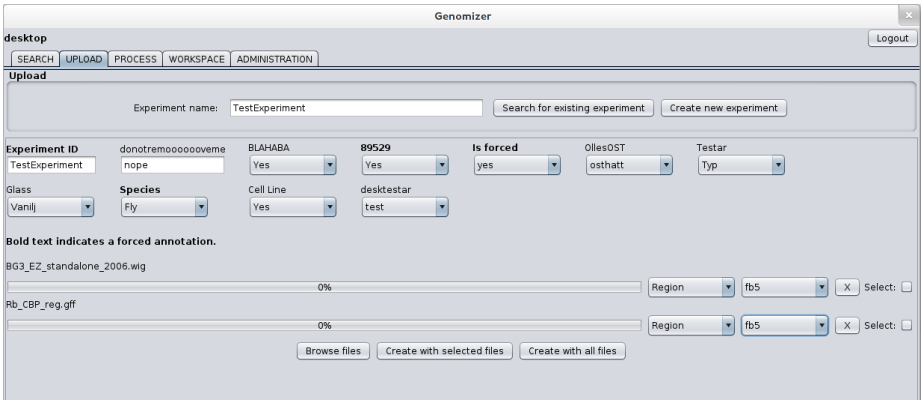


Figure A.7: Illustration of the create new experiment part of the upload tab.

A.1.3.3 Batch upload experiments

In order to batch upload experiment the workflow of this application is suggested as follows: You start of as when uploading one experiment, as explained in A.1.3.2. But instead of choosing the wanted files for that experiment you choos all the files that are supposed to be uploaded to all the different experiments that are supposed to be created. You then starts to fill in the annotaions for the first experiment and selects the files that is going to be used in the experiment by checking the select field. After the files are selected you need to press the "Create experiment with selected files"-button. This creates the first experiment and starts to upload the selected files to it. And after the experiment is created, you can change the annotations needed for the second experiment and then select the files for that experiment in the same manner. You can then click the "Create experiment with selected files"-button again and change the annotations to match the third experiment and the select the files for it and starts the upload. Every file that is uploaded is removed from the list of files.

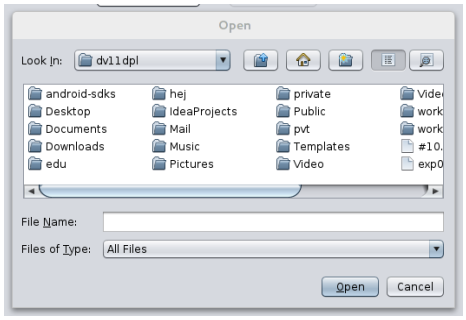


Figure A.8: Illustration of the file browsing window.

A.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 Figure A.12. From this list you can mark RAW-files and choose to create profile data by left clicking on the files they will be marked. If you left click 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 you enter the create profile data tab and presses the Start process button which is visible in the right side of the tab see Figure A.9, all the files that are marked will now be processed to profile data. This list of files will be empty unless you have 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 you have selected some RAW files you have the option to change the processing parameters that is above the Start process button as illustrated in Figure A.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 A.10. For you to reach this window you need 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 ,the process will not be able to start until it is fixed. Depending on what format you choose to process different parameters will be enabled. For example ratio calculation parameters cant be set unless SGR format is used.

If you have selected some RAW-files and then press 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 A.9. If you want to perform a ratio calculation while processing a file you have the option to press the *Use ratio calculation* button. When pressed a popup window appears and you get the option to fill 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 you have 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 you have started a process you 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 A.9.

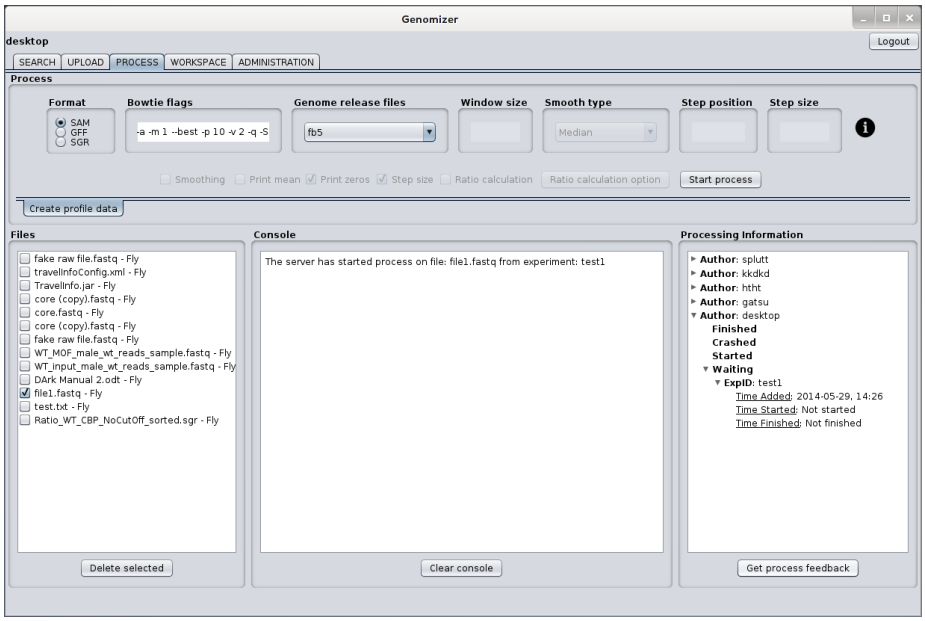


Figure A.9: Screenshot of the process tab in the program.

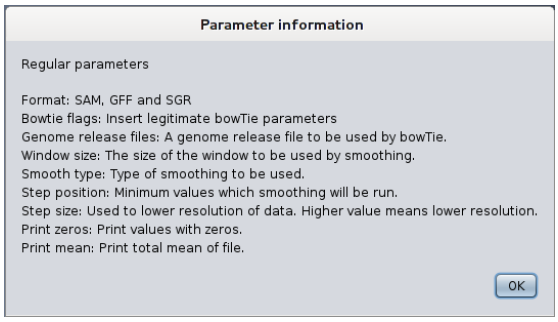


Figure A.10: The parameter information popup window.

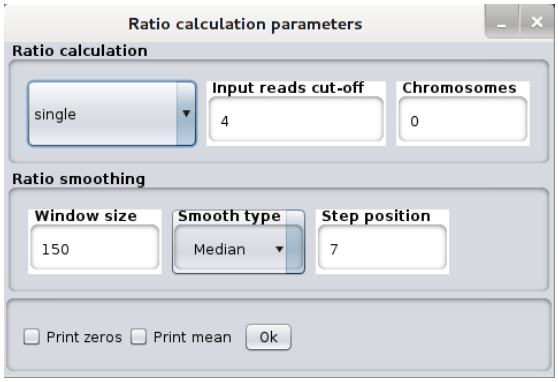


Figure A.11: The popup window for ratio calculation parameters.

A.1.5 Workspace

The workspace Tab seen in Figure A.12 is a tab where you 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*.

A.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.

A.1.5.2 Upload to

If you want to upload files to an experiment you have in the workspace, you 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.

A.1.5.3 Process

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

A.1.5.4 Download

You can make the choice to download files to their local computer. If you press the *Download* button seen in Figure A.12, you get to choose a directory where you want to save the files. 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 A.13.

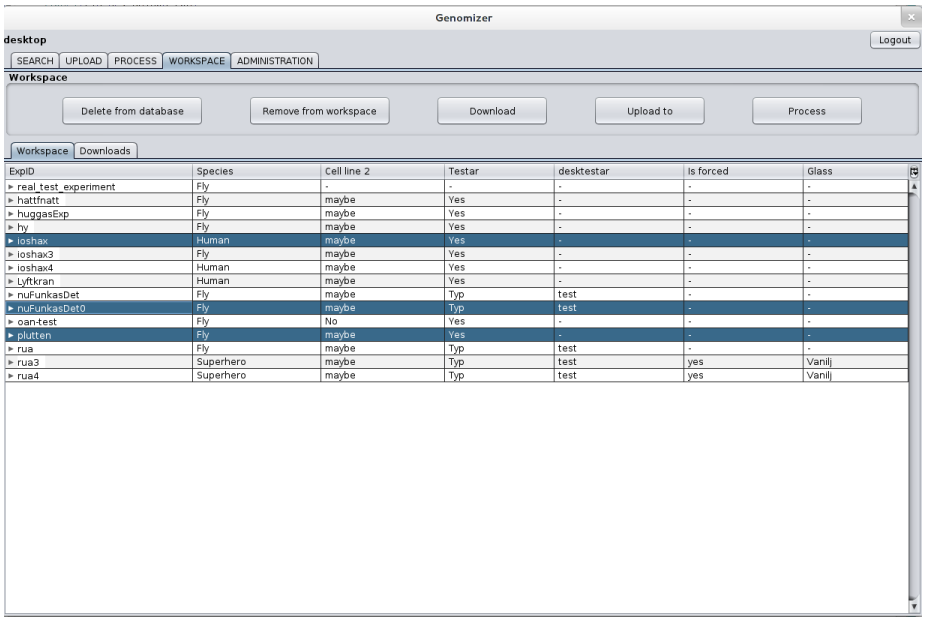


Figure A.12: Screenshot of the workspace tab in the program.

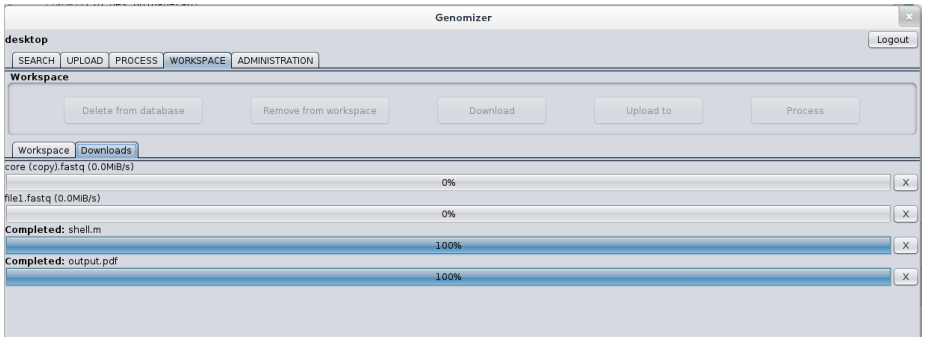


Figure A.13: The downloads tab of the workspace

A.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 shuld have an annotation called "species" with the value "rat". The Annotations sub tab in the Administration tab gives you the tools to create, edit and remove annotations and annotation values.

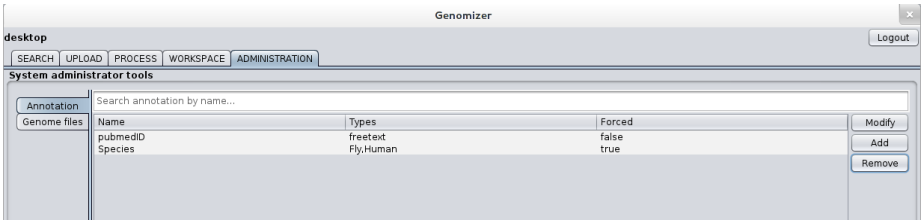


Figure A.14: The annotation view

In the annotations tab, when you select 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 A.15

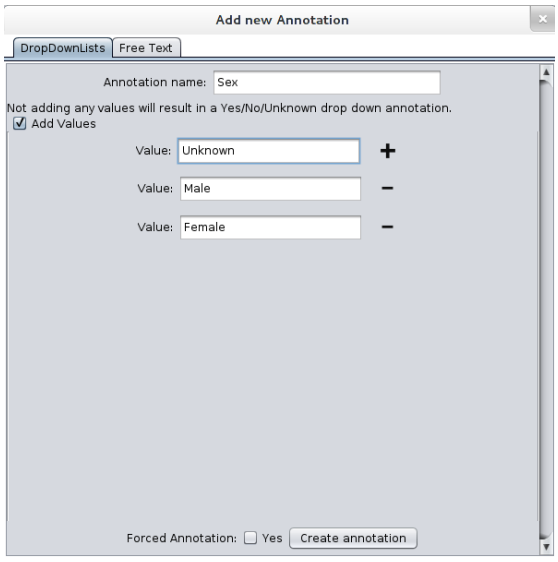


Figure A.15: The add annotation popup

If you want 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. You might then want to use a freetext annotation, which allows you to type any value you want. To create a freetext annotation you click on the freetext tab on the "add" popup.

To remove an annotation, you select an annotation from the table in the center of the view, and click on the remove button on the right side. You then have to confirm this deletion. After

this the annotation is completely removed and cannot be brought back to life, see Figure A.16. Some annotations cannot be removed for security reasons, 'Species' is such an annotation. Trying to remove it will generate an error message.

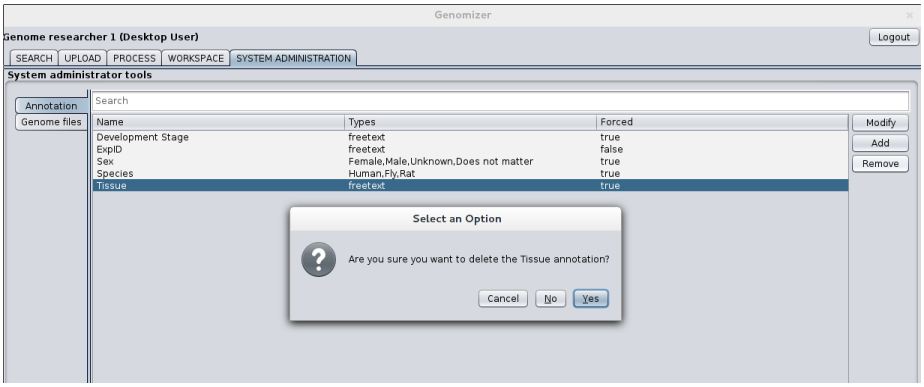


Figure A.16: The remove annotation popup.

The genome files tab shown in Figure A.17 contains a table with information about which genome release versions are stored on the server. If you click 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. You can name the new genome release in the text field and you are 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 you press "Upload", the upload of the selected files will commence and you can follow the upload progress from the progress bars. After the upload is finished, you 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.

If you want 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. You simply writes the name of the new species and presses the "add" button and the species will be added to the "Species" annotation.

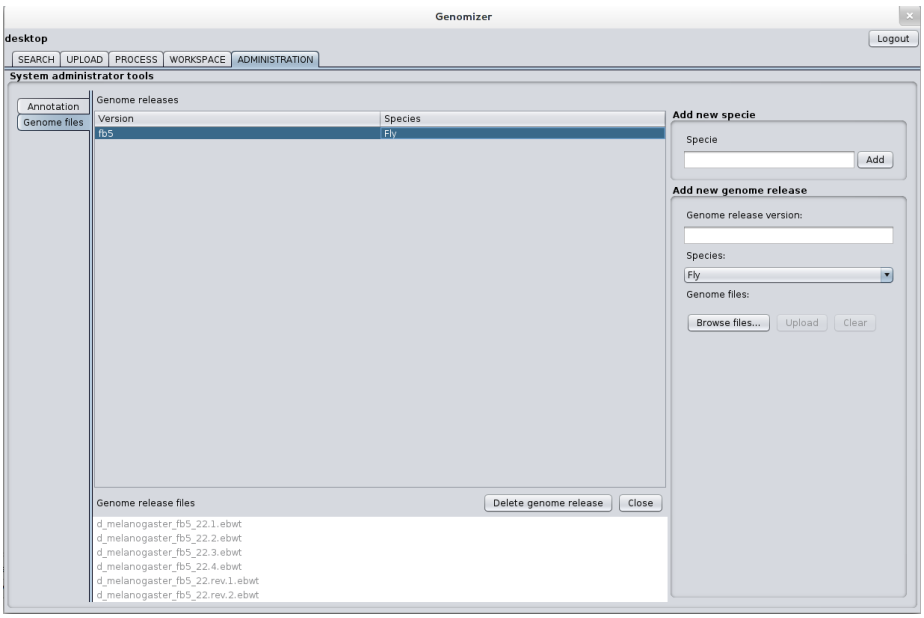


Figure A.17: The genome release view.

A.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: `https://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 has been implemented so far.

A.2.1 Using the interface

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

A.2.1.1 Start view

When first entering the web page, the login pop-up window in Figure A.18 is shown and the user will have to enter their username and password to gain access to the application.

When the user has logged in, the user is taken to the search page as shown in Figure A.19.

The navigation bar at the top has a number of 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.

Figure A.18: The login pop-up window.

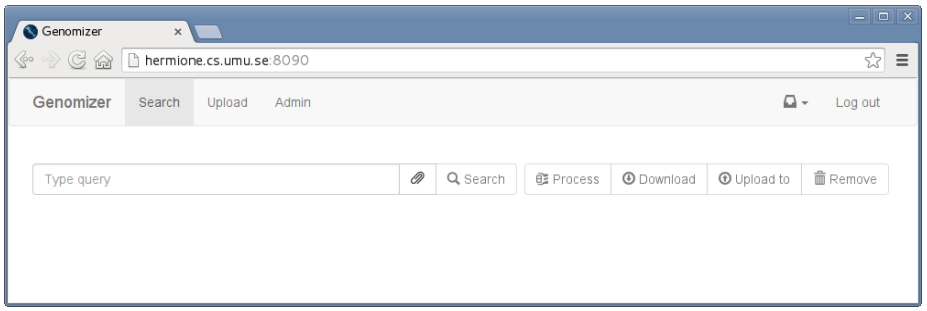


Figure A.19: The start view of the web page.

- The “Process” button will bring up the process view where the user can select an experiment to process.
- The “Administration” 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 dropdown list which displays the statuses of files currently being processed.
- The “Log out” button will log out the user.

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

A.2.1.2 Search view

In the search view, below the navigation bar, a “search-and-functionality” bar is visible. There is a search field and there are seven buttons that are explained below, starting with the left-most button:

- ”Query builder”, represented by a paperclip, brings up a query builder, shown in Figure A.20, that helps unexperienced users construct a valid query used for searching experiments. 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.
- “Search” searches for the query in the search field.
- “Process” processes the selected files. This feature is demonstrated further in Figure A.23.

- "Convert" converts the selected files. This feature is demonstrated further in section A.2.1.4.
- "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.

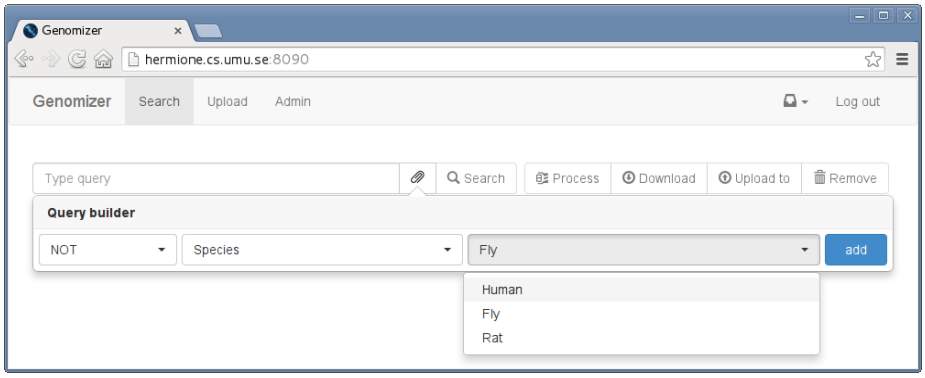


Figure A.20: The query builder.

When first entering the search view, only the query builder button and the search button are clickable. The rest of the buttons become clickable once the user selects experiments or files. To search, the user can either write a pubmed style query (for example: *"Fly[Species]"* to search for every experiment with "Fly" as value of the annotation "Species") or use the query builder. When clicking the search button, a loading screen is shown. The experiment data is displayed once it has been retrieved from the server.

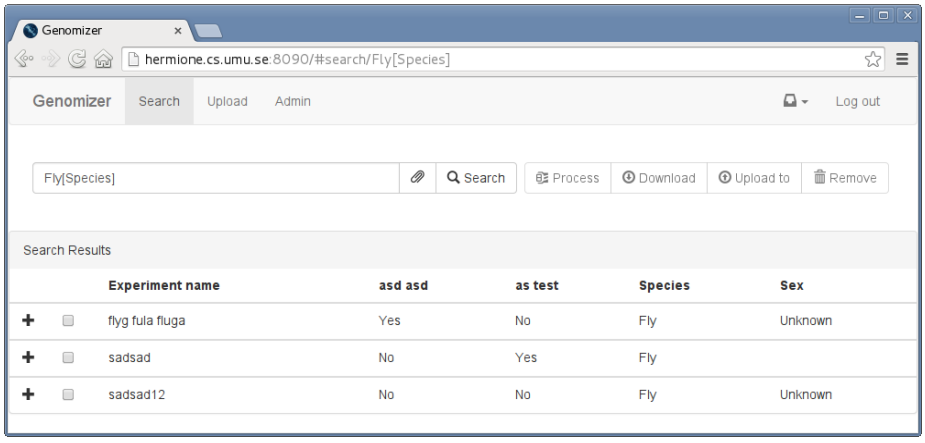


Figure A.21: The search tab after searching for *"Fly[Species]"*.

The view in Figure A.21 is shown when the user has searched for the query *"Fly[Species]"*. The displayed list contains all experiments returned from the search and a header on top with all annotation types. Every experiment can be expanded by clicking it to show the file types it contains. Each file type can be further expanded to show all files of that type in the experiment. Every file and experiment has a checkbox next to it that is used to select it. In Figure A.22, an experiment called `Ratiotest` and its contained collection of `raw` files have been expanded. Furthermore, the files `test.fastq` and `test2.fastq` have been selected. These files can now,

for example, be processed or removed by using the buttons in the "search-and-functionality" bar.

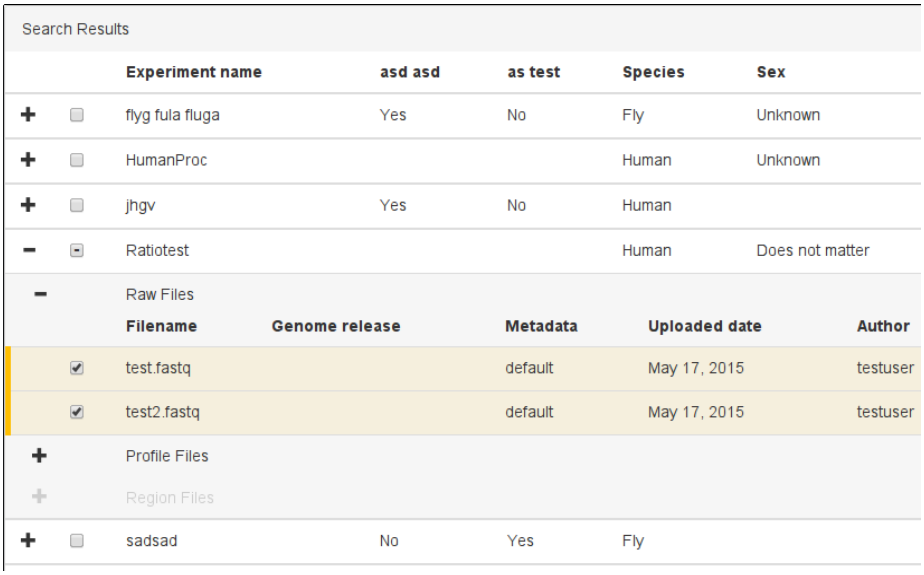


Figure A.22: The search results table zoomed in, displaying the information of a `raw` file after having expanded an experiment.

If no experiments match the search query, the Search Results table will be empty stating "No search results found".

A.2.1.3 The processing modal

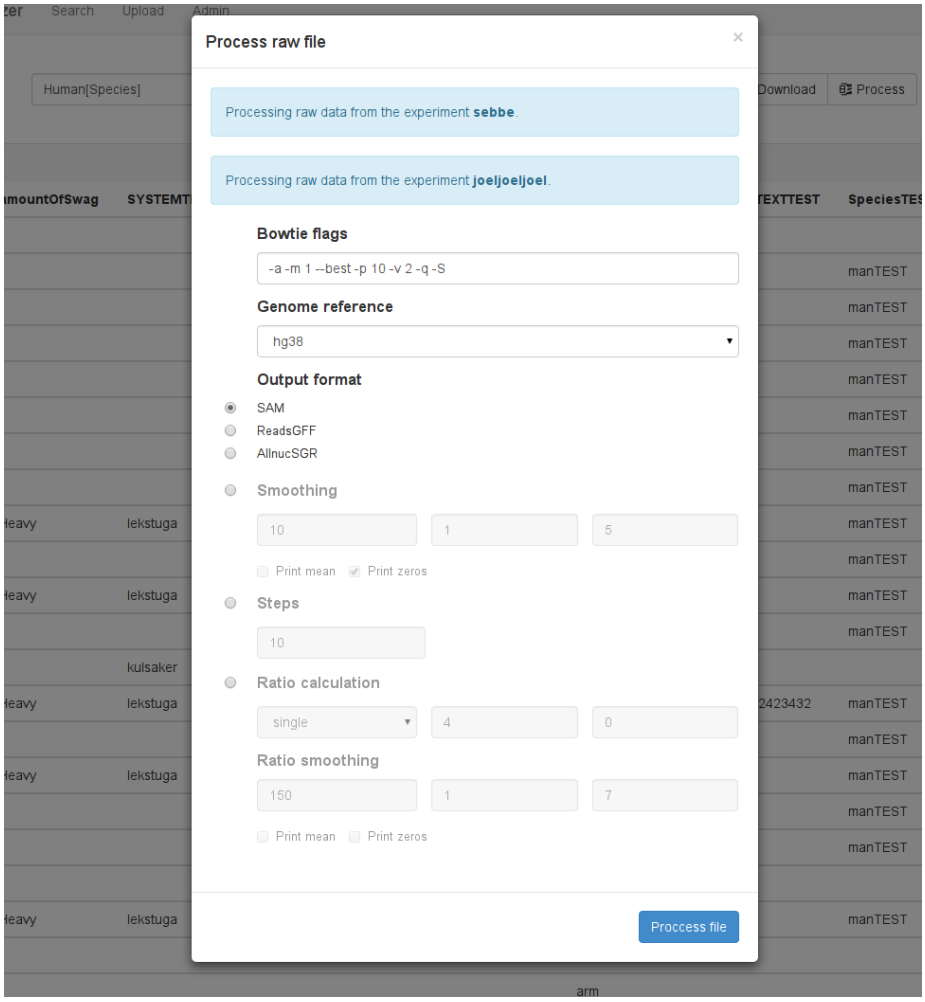


Figure A.23: The processing modal.

If the user wants to process files from an experiment they first have to enter the search-view and search for the experiment in which they wish to process files.

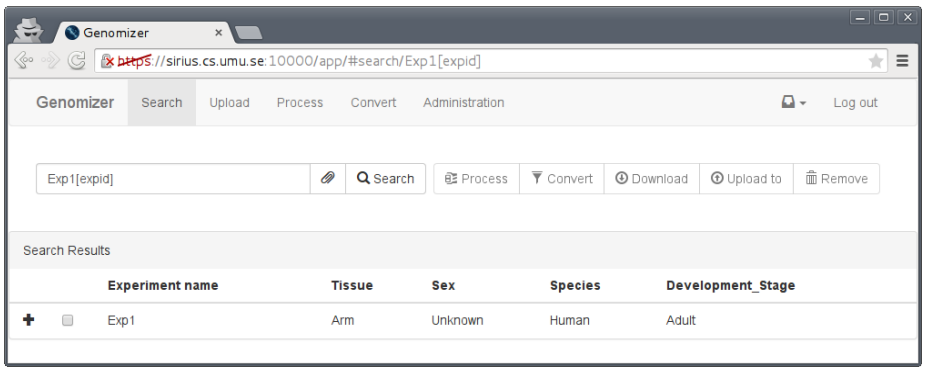


Figure A.24: Searching for Exp1.

Check the box for the experiment and then click the process-button. When the process view is

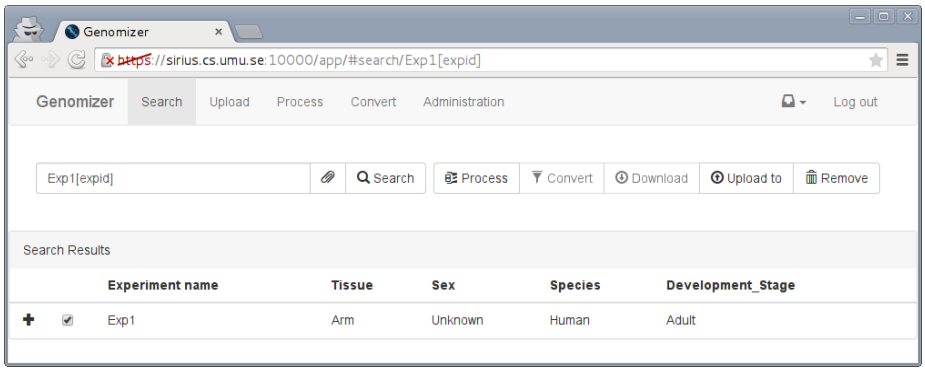


Figure A.25: Selected Exp1.

opened press choose which process step you want to do in the scrollbar and then press the add button. To and a new file to the selected process step press the "+" button. After pressing the add button the view will now show a new set of processing options. The infile scrollbar chooses which file is going to be processed. Outfile sets the name for the new processed file. Genome release sets which genmoe release to use for the process and parameters sets which parameters to be used. If the user wants the .SAM file to be saved the "Keep .SAM" box has to be checked.

When the user has selected some files that are going to be processed the user will be presented with the view from Figure A.23. 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.

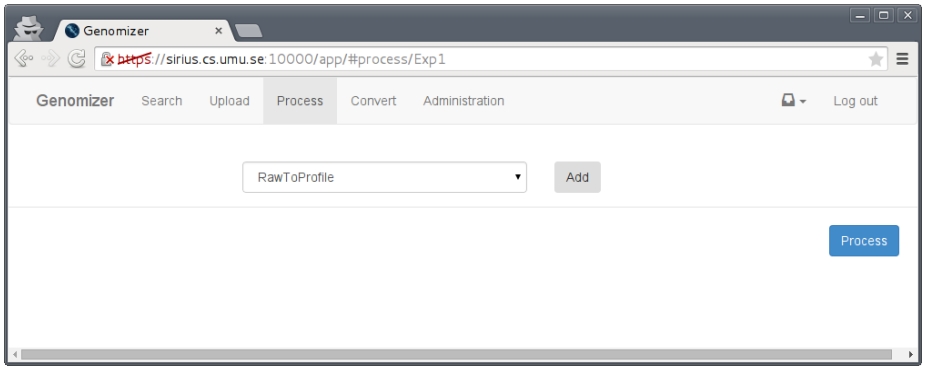


Figure A.26: The process view opened with Exp1 selected.

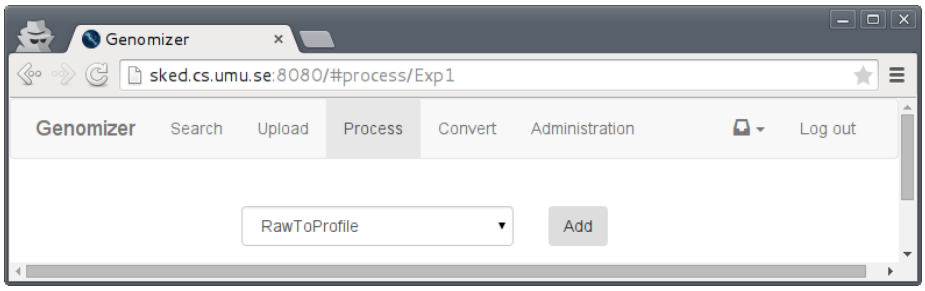


Figure A.27: Scrollbar which shows the different process steps.

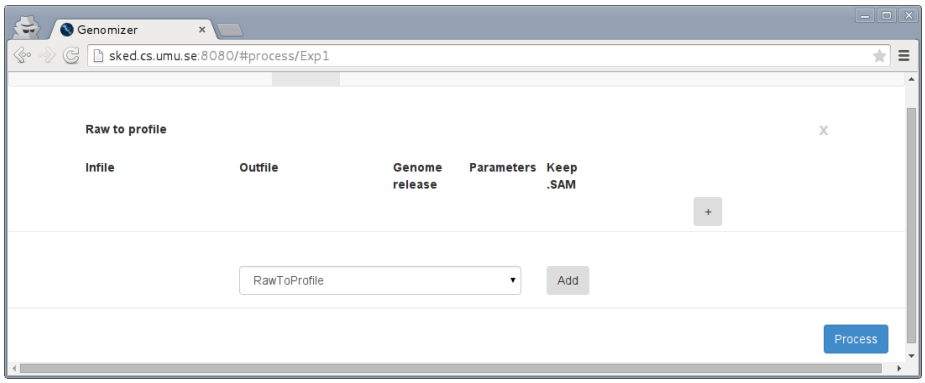


Figure A.28: The process view opened with Exp1 selected.

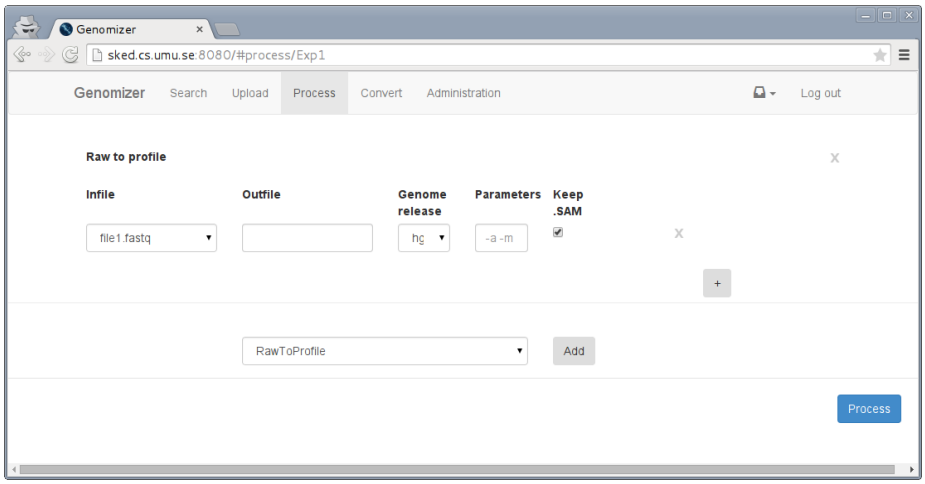


Figure A.29: The process view opened with Exp1 selected.

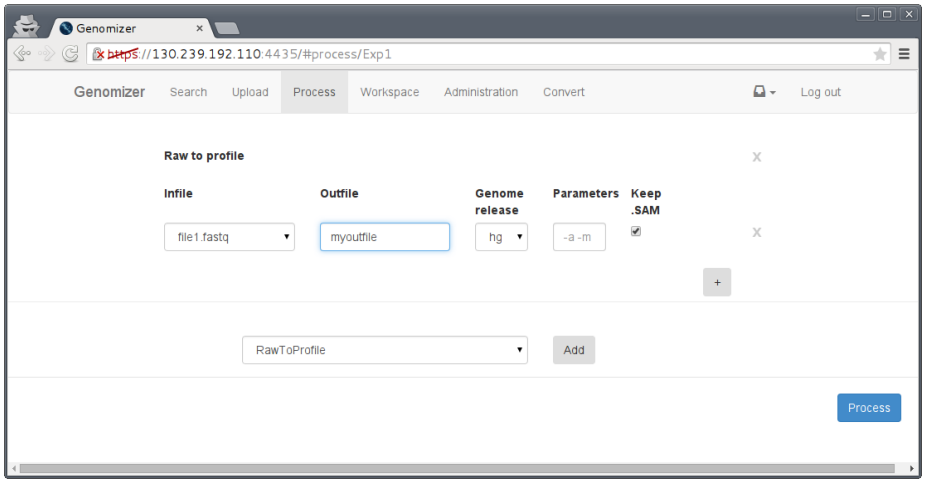


Figure A.30: The process view opened with Exp1 selected.

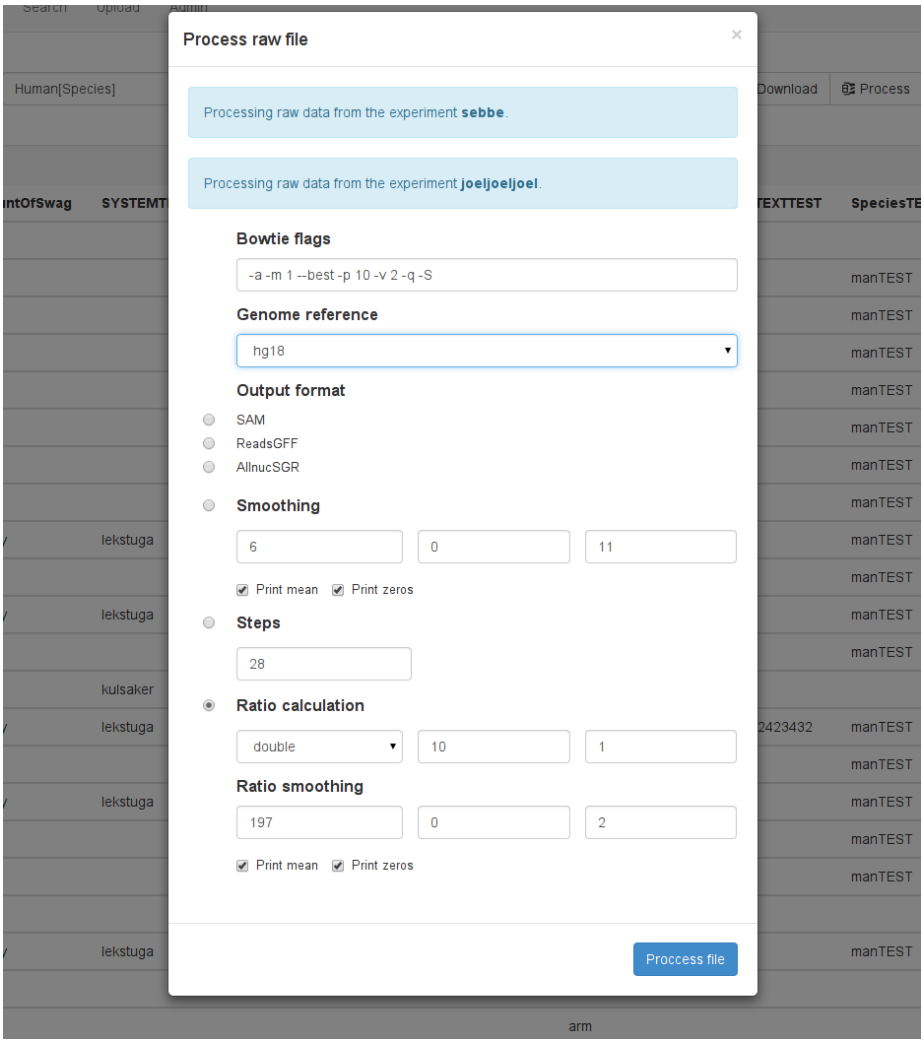


Figure A.31: The process modal with selected parameters.

When the user has decided the parameters as shown in Figure A.31 and wants to start the processing the process button in the bottom right should be pressed.

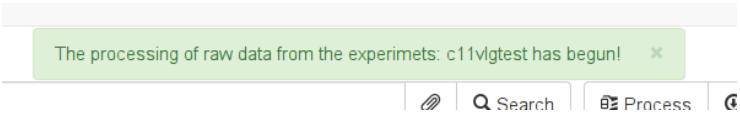


Figure A.32: Success message.

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 A.32. If some of the files that was going to be processed did for some

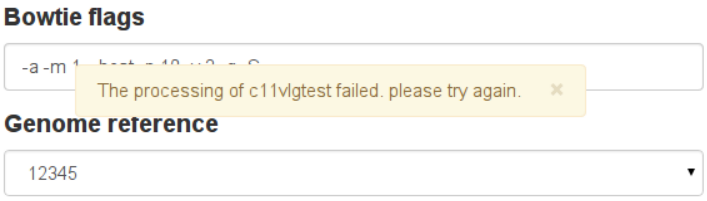


Figure A.33: Fail message.

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 A.33. 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.

A.2.1.4 The convert view

The web application allows conversion between a number of file formats which both are of profile type. More specifically, the user may convert any of the file formats `.sgr, .wig, .bed` and `.gff` to either `.wig` or `.sgr`, with the exception that a file cannot be converted to the same file format.

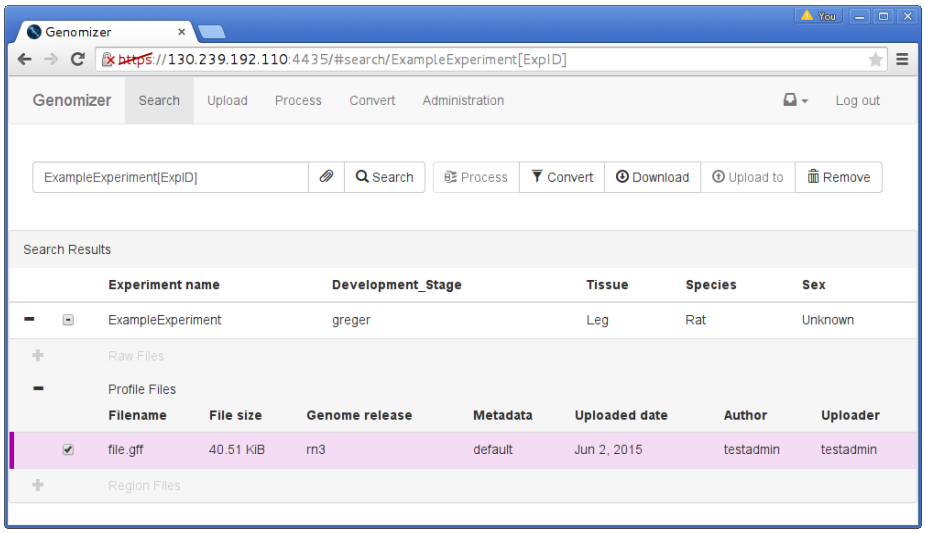


Figure A.34: Selecting a file to convert.

Assume that there is an experiment `ExampleExperiment` which contains a profile file `file.gff`. Then the experiment will show up in the search view, see Figure A.34, when typing "Example-Experiment[ExpID]" as search query and then clicking the search button. If the user wants to convert `file.gff` to a new file of format `.wig` called `file.wig`, the following steps can be taken:

- From the view in Figure A.34, select `file.gff` by clicking in its checkbox.
- Click the "Convert" button next to the search text field.
- The user will now be taken to the convert view shown in Figure A.35.
- Select `file.gff` by clicking on it.
- Mark the `.wig` checkbox in "Convert to" and click the "Convert" button.

The file conversion will now start on the server. Once the conversion is done, the user will be able to see `file.wig` listed together with the old file `file.gff` when searching for the experiment.

If multiple files are selected for conversion, all of them will appear as a list in the convert view. If the user quickly wants to select all files of a specific file format, for example all `.wig` files, the GFF option can be marked in "Select all". Then all files in the list of the format `.wig` will automatically be selected.

A.2.1.5 The remove pop-up window

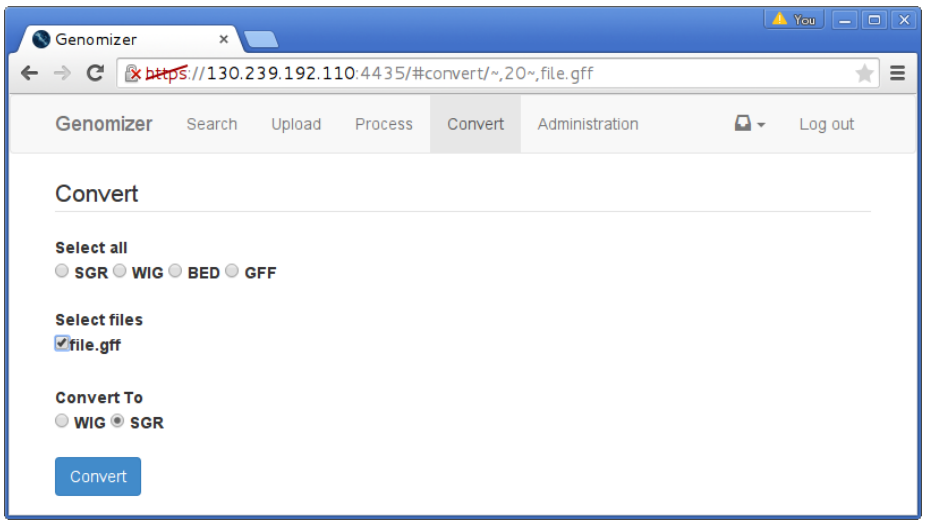


Figure A.35: The convert view.

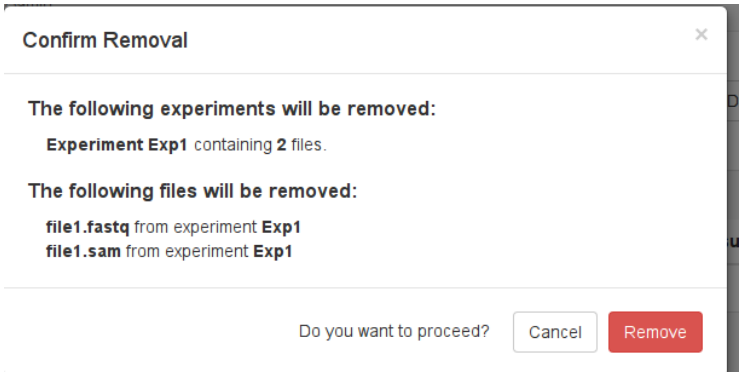


Figure A.36: The remove pop-up window.

When the remove button is pressed the pop-up window in Figure A.36 is shown displaying which files and experiments will be removed when the remove button is pressed.

A.2.1.6 The process status dropdown

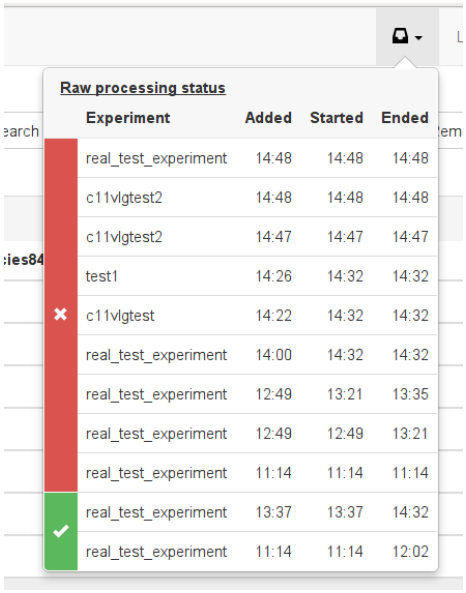


Figure A.37: The process status dropdown.

When pressing the inbox icon, a dropdown is shown as in Figure A.37 displaying the status of experiments currently being processed. There are four different statuses a processing can have, all grouped into colors: Waiting (yellow), Running (blue), Complete (green) and Failed (red). For example, in the figure, the two bottom experiments are complete and the rest have failed. If there are no experiments being processed, the dropdown will simply display "No process status available".

A.2.1.7 The upload view

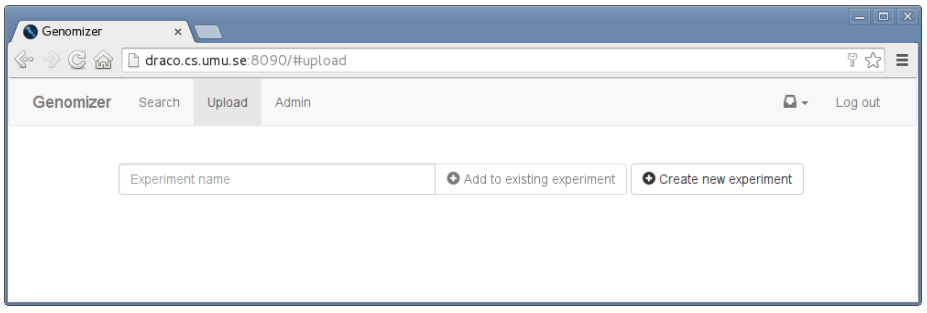


Figure A.38: The upload view.

When the user clicks the upload tab in the navigation bar, the view in Figure A.38 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 A.39 will appear. Here the user can input the annotations for the experiment through either freetext fields or dropdown lists. If a freetext field has a red border around it, that annotation is required and the experiment cannot be uploaded before all required fields have been filled in and at least

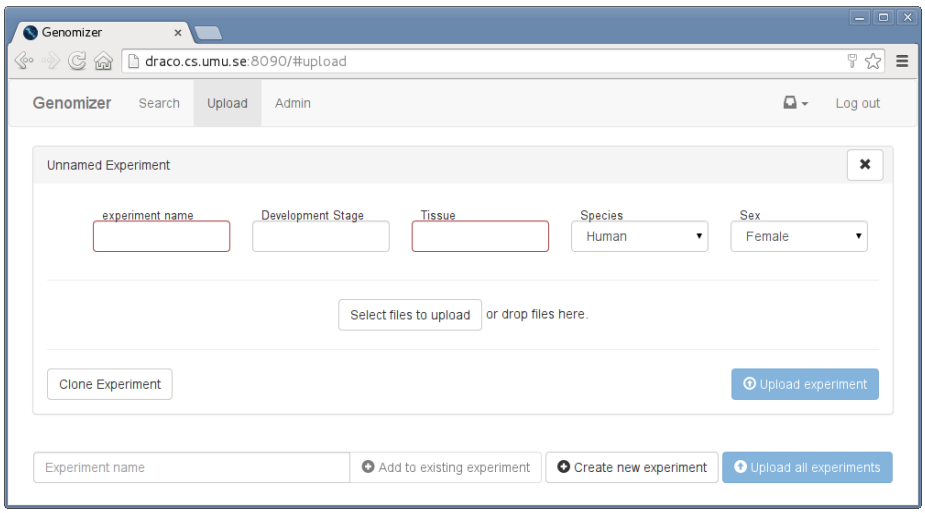


Figure A.39: Creating a new experiment.

one file has been added.

The user can create more experiments by clicking the "Create new experiment" button and a new empty experiment will be placed below the first experiment. The user can also clone an experiment by clicking the "Clone Experiment" button. What happens in this case, is that the every filled-in annotations gets copied to the new experiment.

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 them 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 A.40. 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. 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 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".

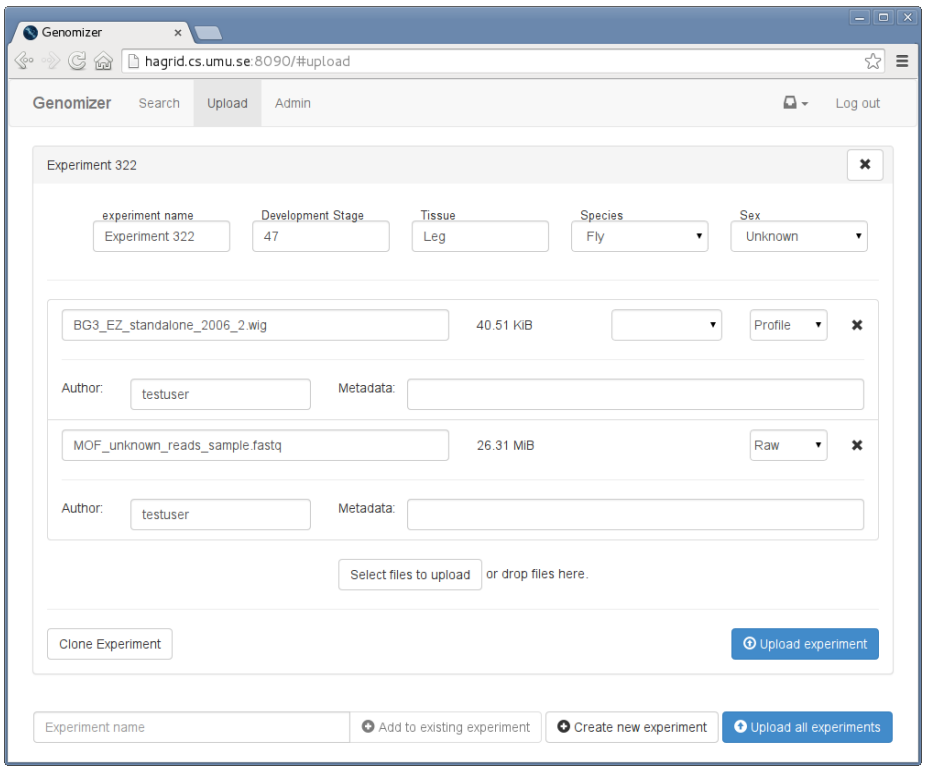


Figure A.40: Files selected for upload.

A.2.1.8 System administration view

This part of the web application is only accessible if the user has administrator rights. It is integrated with the rest of the web user interface and accessible through the "Administration" 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 Annotation" button, a list of existing annotations in the database and an edit button per existing annotation. The view looks like in Figure A.41.

For each annotation in the annotations list, an "Edit" button is available. When pressed, it will take the user to a page in which they can edit the selected annotation to change its name and what values the dropdown list will have if it is not a freetext field (see Figure A.42).

In the edit page, the admin can see the attributes of the chosen annotation and is able to delete the chosen annotation or change the information of it. The "Delete Annotation" button will delete the whole annotation, and for that reason two pop-up windows will appear to confirm that the administrator is sure of the action.

The administrator can change the list of annotation values. 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 the "Create new annotation" button from the admin start page, another view will open with the following structure:

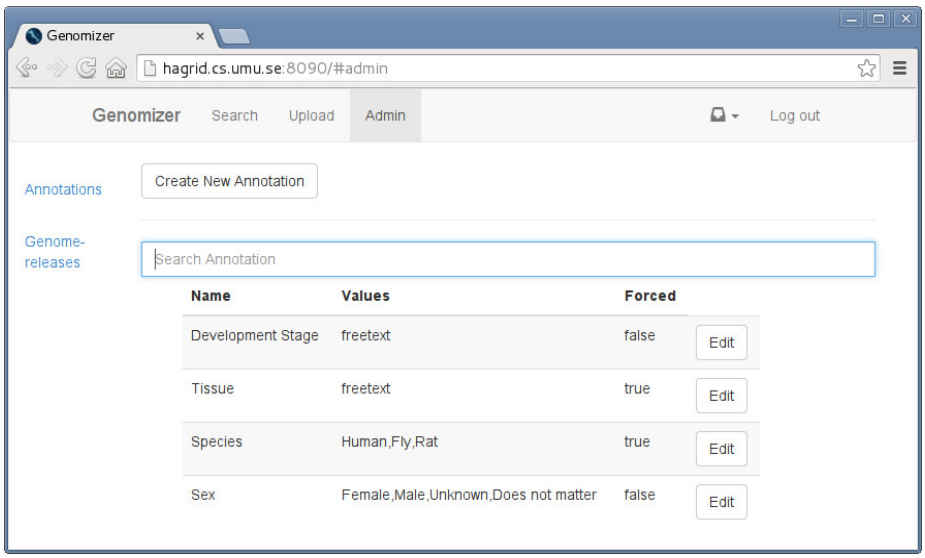


Figure A.41: The start page for the administrator in the web client.

- Annotation Name
  Admin can enter a name for the annotation.
- Annotation Types
  Yes/No/Unknown - creates a dropdown list with those three options.
  freetext - creates an annotation where the users will be able to enter anything.
  Dropdown 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 required by users to enter.

A Create Annotation will, if all necessary information has been entered, result in a popup (see Figure A.43) 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.

The example in Figure A.43 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.

A back button which takes the user back to the annotations start page is also available in this view. In Figure A.44 the create annotation view can be seen.

The “Genome-releases” link on the sidebar takes the administrator to a page where it is possible to add and remove genome releases to and from the server (see Figure A.45.

The button “Select files to upload” opens the native file explorer where the user can select one ore multiple files and click on “OK”. This will open a popup-window, seen in Figure A.46, showing what files that where 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.

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.

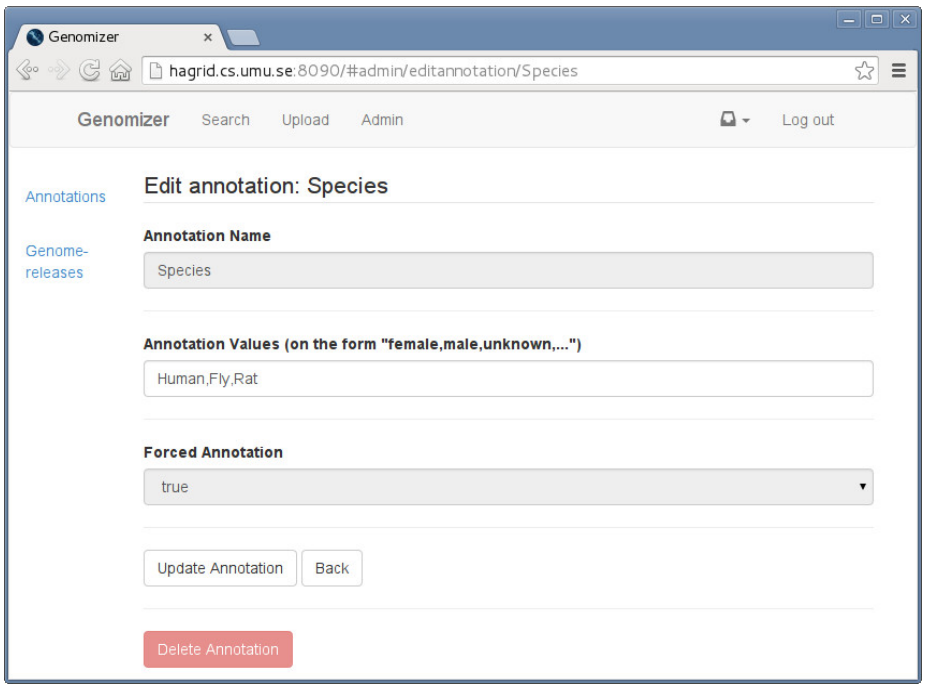


Figure A.42: The edit annotation view.

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

A.2.2 Setting up the application

To setup the application, move the content of the folder `genomizer-web/app/` to the desired location from where the application should be run. To run the web page, open a web browser and enter the url to the folder which contains the `index.html` file (where the content of app was placed). For example, given that the `genomizer-web` folder is placed in the home folder of the Umeå university CS user `c12abc` and that user wants to put the web app in a folder called `public/html/` which is also in the home folder of the user. In Linux, do the following steps:

1. Navigate to the app folder: `"cd /genomizer-web/app/"`
2. Move the contents of app to the folder `public/html`: `"mv * /public_html/"`
3. Given that the url to `public/html` is: `"www8.cs.umu.se/ c12abc/"`
4. To run the application start a web browser and type `"www8.cs.umu.se/ c12abc/"`

This will open the web page in the browser.

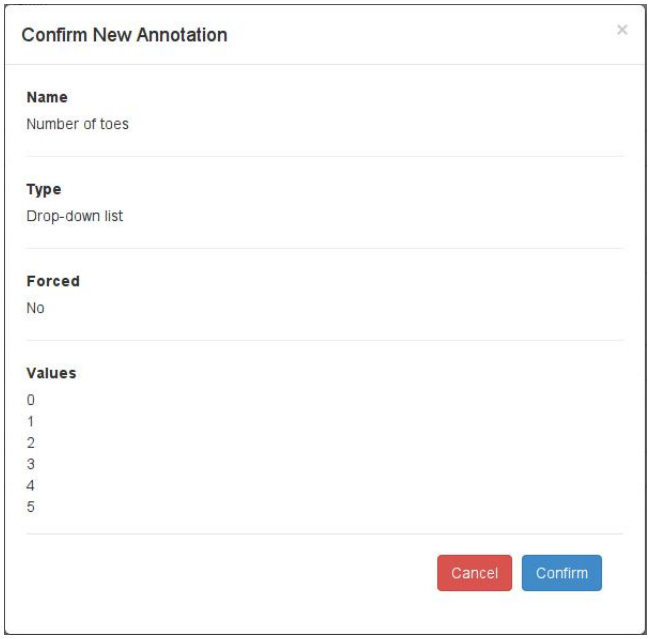


Figure A.43: The confirm annotation pop-up.

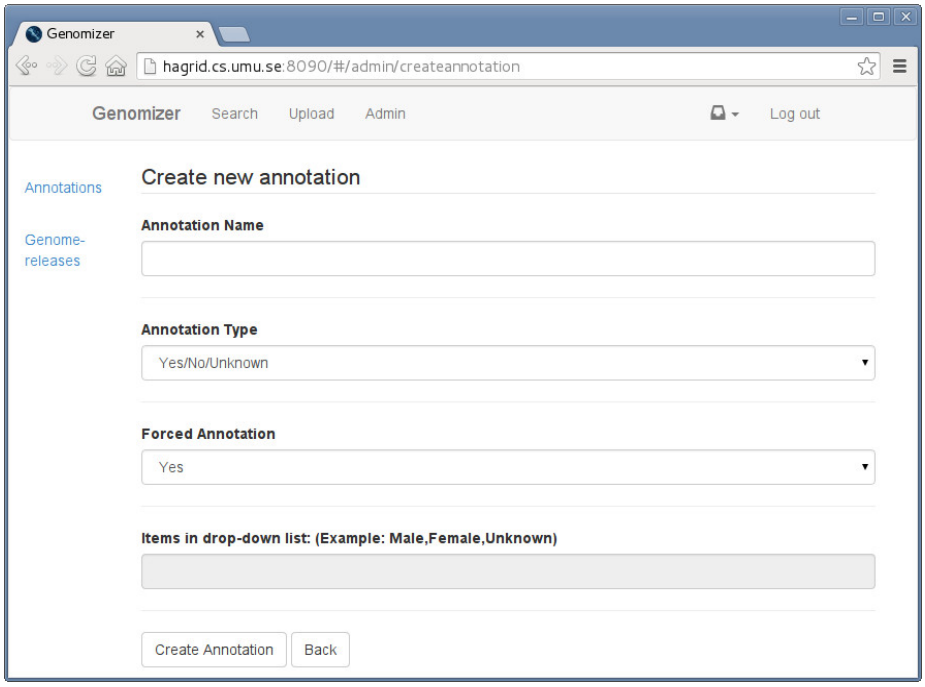


Figure A.44: The view for administrators where new annotations can be created.

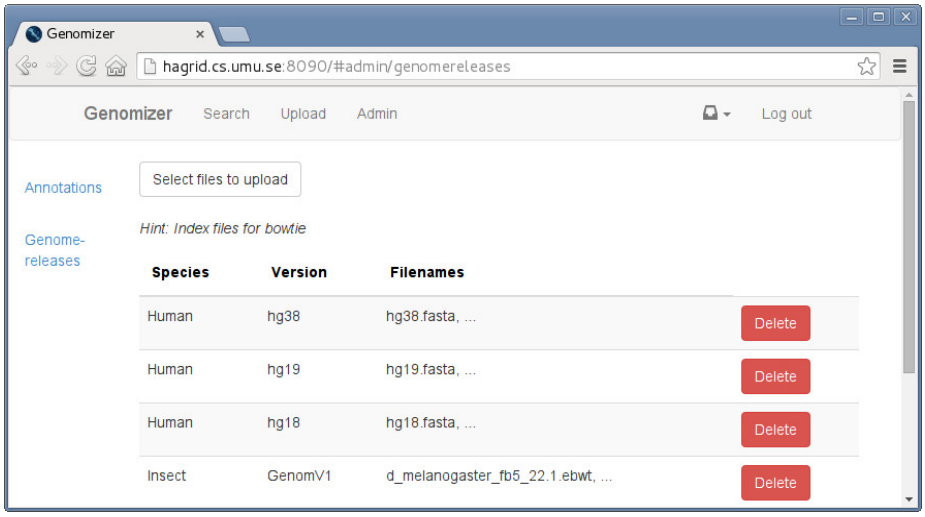


Figure A.45: The genome-release view.

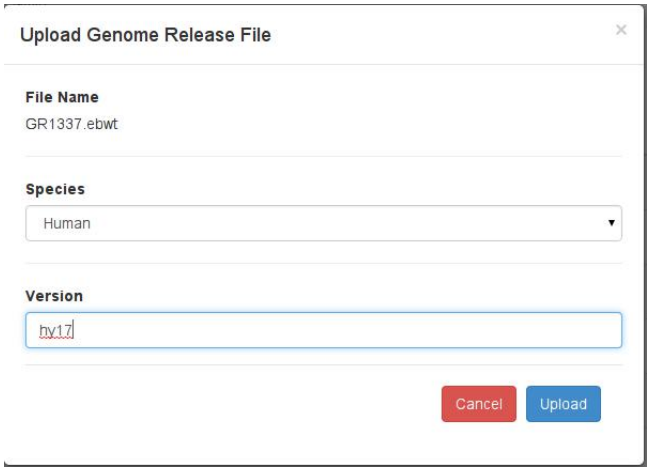


Figure A.46: Popup for uploading genome releases.

A.3 Android application

In this section instructions for the usage of the *Genomizer* Android application is presented. In subsection A.3.1 there is a description on how to start the application and subsection A.3.3 gives instructions on how to search for experiments.

A.3.1 Start the Application and Login

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

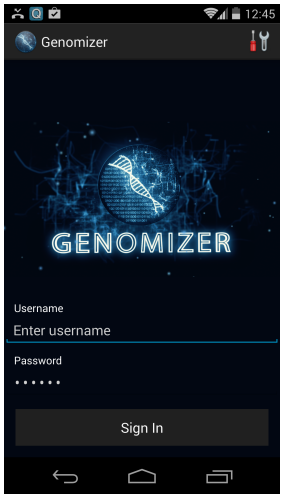


Figure A.47: Login View

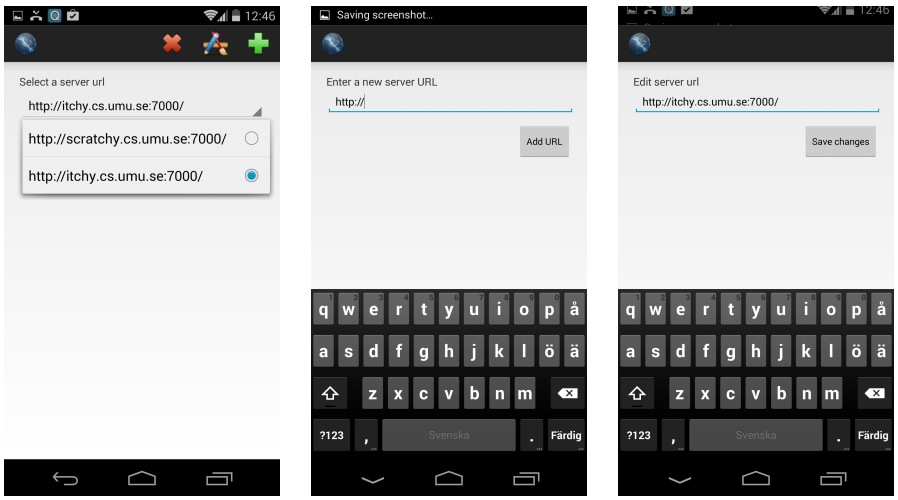
A.3.2 Settings

In Figure A.47, the tool button in the upper right corner leads to the *Settings View.l_ settings*. The *Settings view* acts to enable the user to choose which server to connect to when using the Genomizer application. In the current release of the application, 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 leftmost image in Figure A.48 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 A.48. The button in the middle with a paint brush icon will on selection show the server URL edit view, as illustrated in the rightmost image in Figure A.48. 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.



a b c

Figure A.48: Settings View

A.3.3 Searching for files

When entering the Search View, as illustrated in Figure A.49 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 A.3.4.

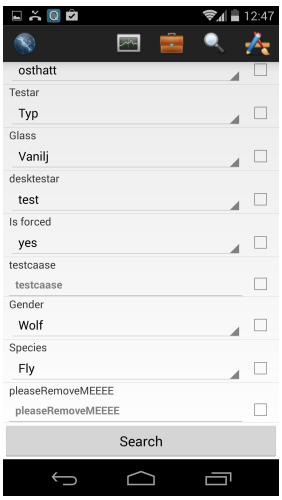


Figure A.49: The Search View

A.3.4 Pubmed Search

The Pubmed Search view provide the means of free-text search using Pubmed-Style queries as seen in Figure A.50. 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.

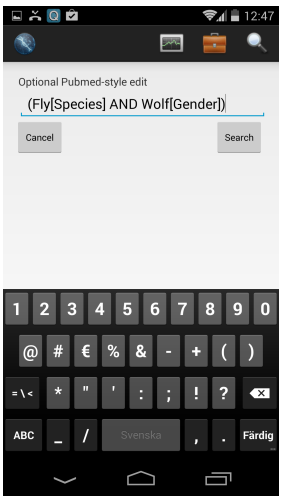


Figure A.50: The Pubmed Search View

A.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. When 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 A.3.6.

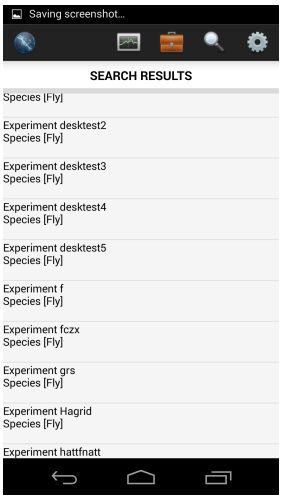


Figure A.51: The Search Results View

A.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 A.52 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.

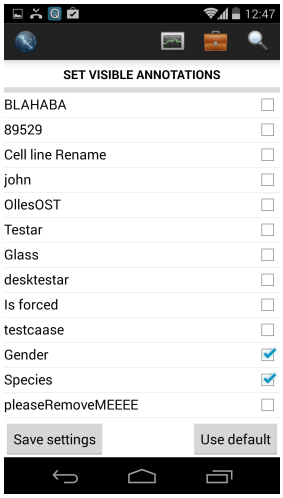
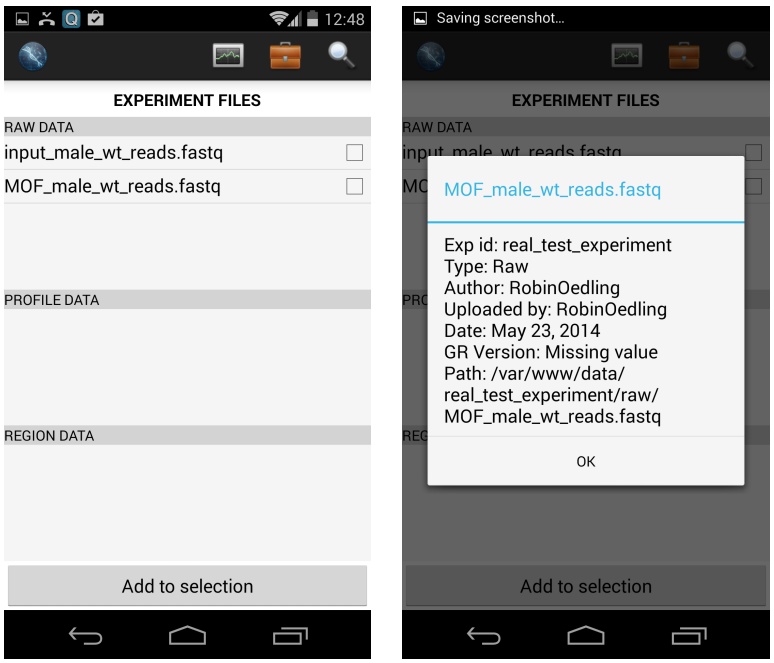


Figure A.52: Search Settings View

A.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 A.3.8, by marking the checkbox of the desired files, as done in Figure A.53, 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 A.53.



a b

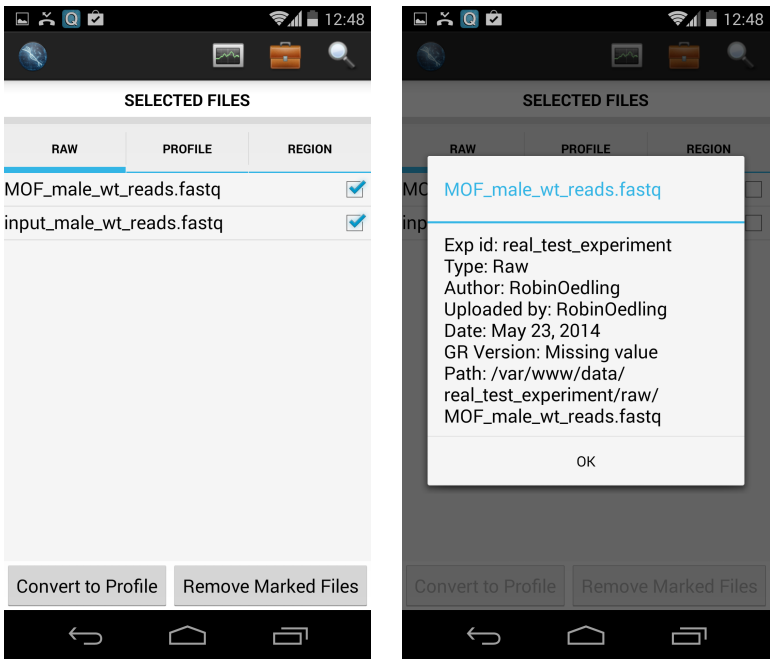
Figure A.53: The Experiment File View

A.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 A.54. 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 diplay all the Raw files that the user has choosen 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 choosen 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.



a b

Figure A.54: Selected Files View

A.3.9 Converting Files

When the user has choosen a file (or several files) for conversion, the user will be presented with the Conversion View as seen in Figure A.55. 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.

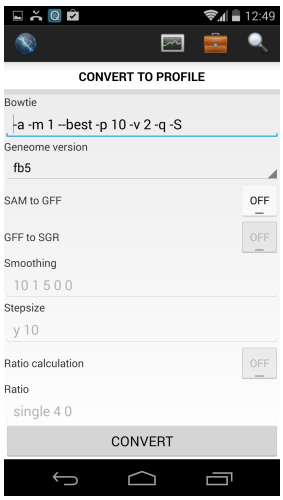


Figure A.55: The Conversion View

A.3.10 Process View

The process view, as illustrated in Figure A.56 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

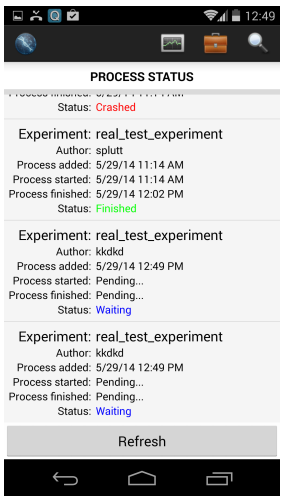


Figure A.56: The Process View

A.4 iOS application

A.4.1 How to run the app in Xcode

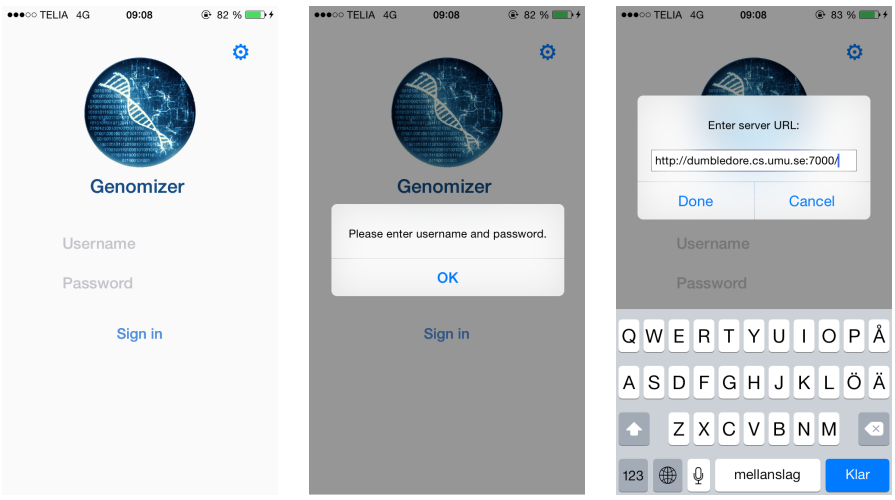
In order to use the program, import the project from github into Xcode from the following repository: `https://github.com/genomizer/genomizer-iOS.git`

To compile and run the program, press ***cmd+R***. A simulator will start and the login screen will be shown as seen in Figure A.57a.

A.4.2 How to login

1. Tap the settings button in the upper right corner and enter the url and port for the server you want to use and press ***Done***. See Figure A.57a,c.
2. Tap on the ***Username*** textfield and enter your username.
3. Tap on the ***Password*** textfield and enter your password.
4. Tap on ***Sign in*** to sign in.

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.



a b c

Figure A.57: The login screen.

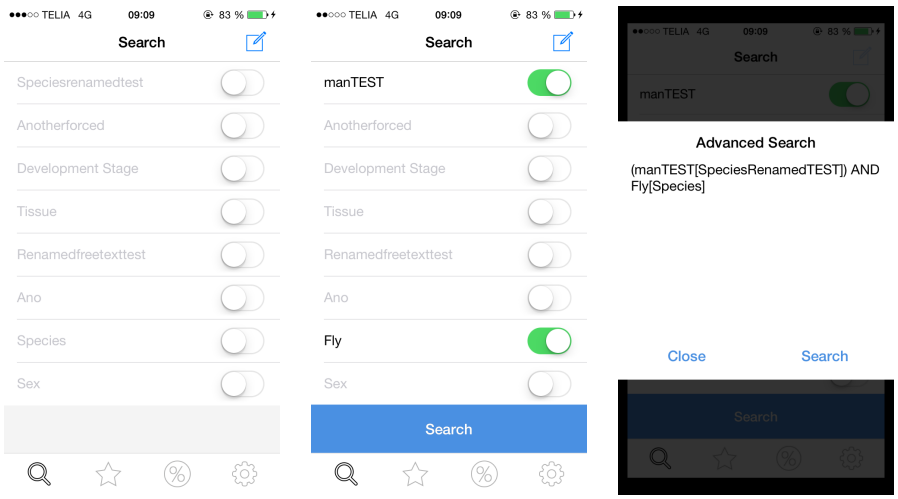
A.4.3 How to logout

1. Tap ***Gear***-symbol on the tab bar and a *Setting*-screen will appear. See Figure A.58
2. Tap ***Logout*** to logout.

Figure A.58: The settings screen.

A.4.4 How to search for experiments

1. Tap on leftmost button(magnifying glass) on the tab bar which you can find on the bottom of the screen to get to the *Search*-view.
2. Tap on the annotation you want to search for and a spinning wheel with options will appear from the bottom of the screen. See Figure A.59a.
3. Drag the wheel up or down to select the option you want.
4. Enable the annotation to use when searching by toggling the switch to the right of the annotation. See Figure A.59b.
5. Do steps (2)-(5) for more search criteria.
6. Tap ***Search*** to search.



a b c

Figure A.59: The search screen.

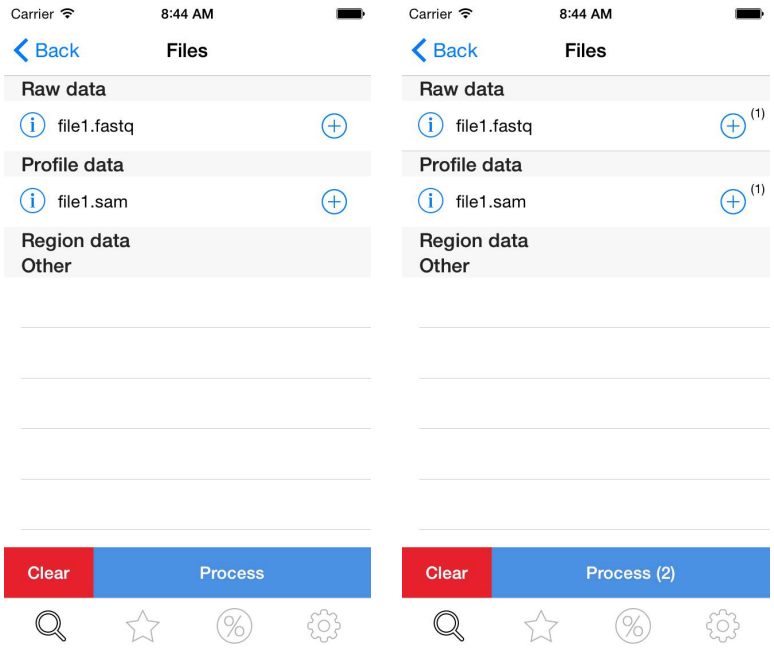
A.4.5 How to use advanced search

1. Tap on leftmost button(magnifying glass) on the tab bar which you can find on the bottom of the screen to get to the *Search*-view.
2. Tap on the symbol to top right of the screen and a new view will appear with the title *Advanced Search*. See Figure A.59c.
3. Write your search criteria in PubMed-style and tap ***search***

The annotations you select on the search-screen will also show in advanced search.

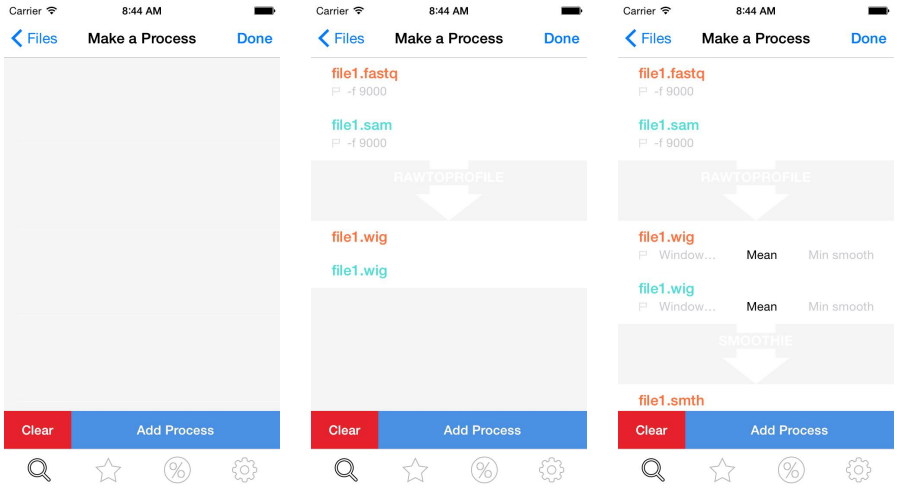
A.4.6 How to process files

1. Search for experiments
2. In *Search Results*-screen tap on an experiment and the *Files*-screen will appear showing the files which belongs to the experiment. See Figure A.60a
3. Tap the ***Plus***-symbol next to the file you want to process.
4. Do step (3) for every file you want to process. Same file can be used more than once. A counter will appear next to the Plus-symbol to keep track on how many times the file will be used in the process-step, see Figure A.60b.
5. Tap the ***Process***-button and *Make a process*-view will appear with every file you have selected. See Figure A.61a
6. Tap ***Add Process*** and select what kind of process you want to do on the selected files.
7. After you have selected a process the input files and output files will appear with the selected process separating them. You can add parameter values by tapping ***Flag***-symbol below the input files. The output files can be renamed by tapping on the filename. See Figure A.61b
8. Do step (6) to create a sequence of processes to be made on the selected files. See Figure A.61c for an example of a process sequence.
9. Tap ***Done***-button to send the sequence of processes to the server.



a b

Figure A.60: Files view.

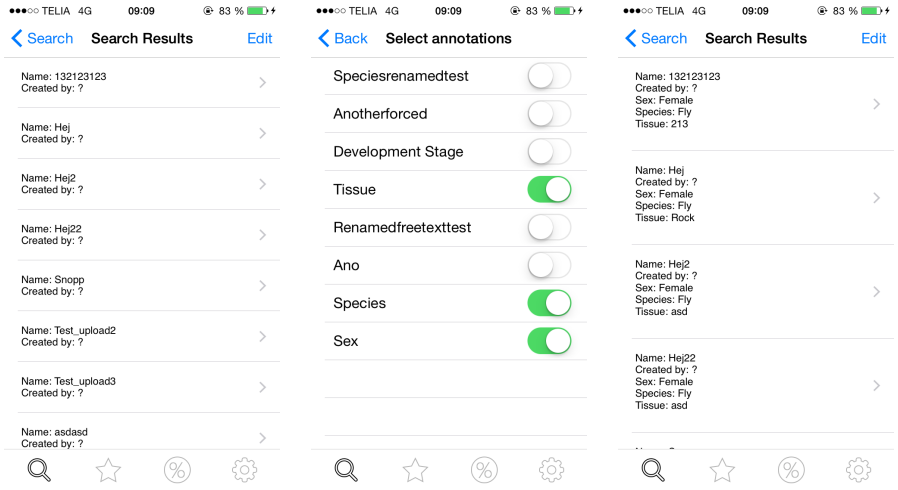


a b c

Figure A.61: Create processes.

A.4.7 How to set which annotation to be visible on Search Results

1. In Search Results view, tap ***Edit*** and *Select Annotations*-screen will appear
2. Select which annotations to show by toggle the switch next to each annotation.
3. Tap ***Back*** to go back to *Search Results*. See Figure A.62a-c



a b c

Figure A.62: Select annotation

A.4.8 How to view process status on the server

1. Tap ***Process***-symbol(the symbol between the star and the trash can) to view the processes on the server.
2. To refresh the view, drag the view down until an activity indicator icon is visible below the title of the screen and release. See Figure A.63

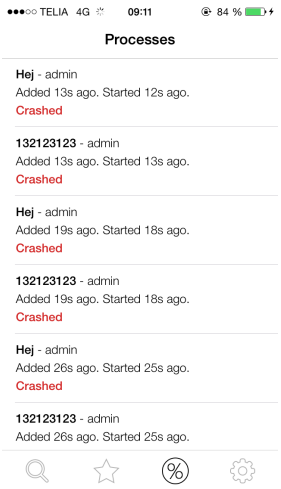
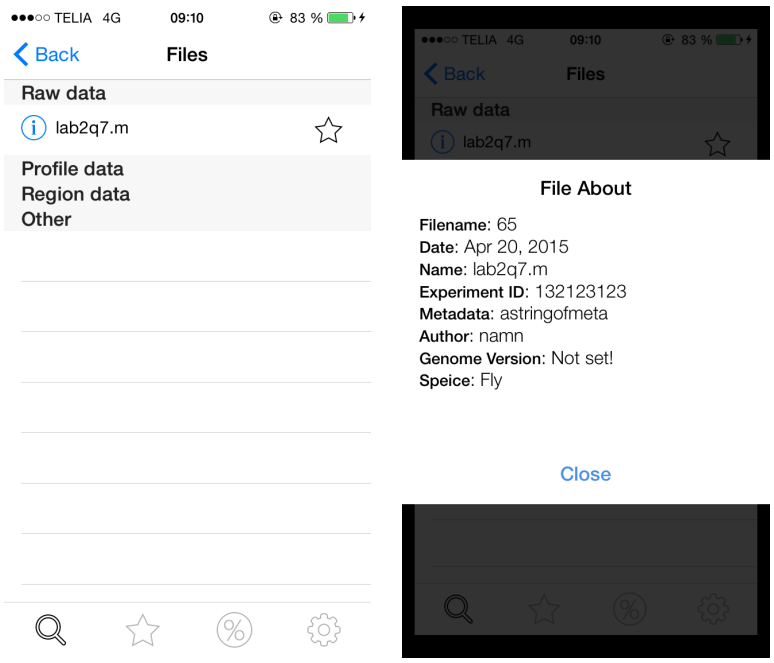


Figure A.63: The select task screen.

A.4.9 How to view information about a file

1. Tap ***information***-symbol next to the file to view information of the file.
2. Close by tapping ***Close***. See Figure A.64a-b



a b

Figure A.64: The files screen.

B 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.

B.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.

B.2 A brief introduction to vagrant

B.2.1 Basic usage

Vagrant is configured by a `Vagrant` file, which defines how the virtual machine is constructed. This should not be modified unless you understand what you are doing.

To start a vagrant instance, run `vagrant up`. The vagrant instance will now either (a) or (b):

a) Create a new virtualmachine with correct configuration
b) Start an already built virtual machine

If the instance is stale or needs to be rebuilt, it can be achieved with

```
vagrant destroy
vagrant up && vagrant ssh -c 'bash startup.sh'
```

If you simply run the server using `vagrant up`, then `vagrant ssh -c 'bash startup.sh'` starts the genomizer server in the virtual machine, inside a screen session labeled `server`.

B.2.2 Modifying the configuration

To make vagrant set up a specific branch, modify the corresponding script, such as `scripts/provision/install_genomizer_server` to checkout `yourBranch` instead of `develop`.

If your feature requires changes to the settings.cfg it is located at `scripts/provision/config/settings.cfg` and is installed along with a fresh instance of the virtual machine.

B.2.3 Entering the vagrant virtual machine

If you need to reach the environment using SSH, simply go to the environment you require with

```
cd development-1
vagrant ssh
```

and you will be logged in using ssh to the virtual machine.

B.3 Systems overview of production

The production server runs currently on `130.239.192.110`. The production server is constructed of several virtual machines, configured by vagrant[1].

The system is entirely scripted, that is, everything can be set up using the scripts located in the vagrant git[2]. The scripts located under `scripts/provision` deal with the virtual environments, while the scripts under `scripts/environment` deal with the server, such as setting up the firewall and external directories used by the virtual machines.

B.3.1 Using the toolchain

This is a short introduction to the tools used in the environments.

[1] Vagrant is a virtual machine configuration manager. The manual is available at https://docs.vagrantup.com/v2/

[2] git.cs.umu.se, access required.

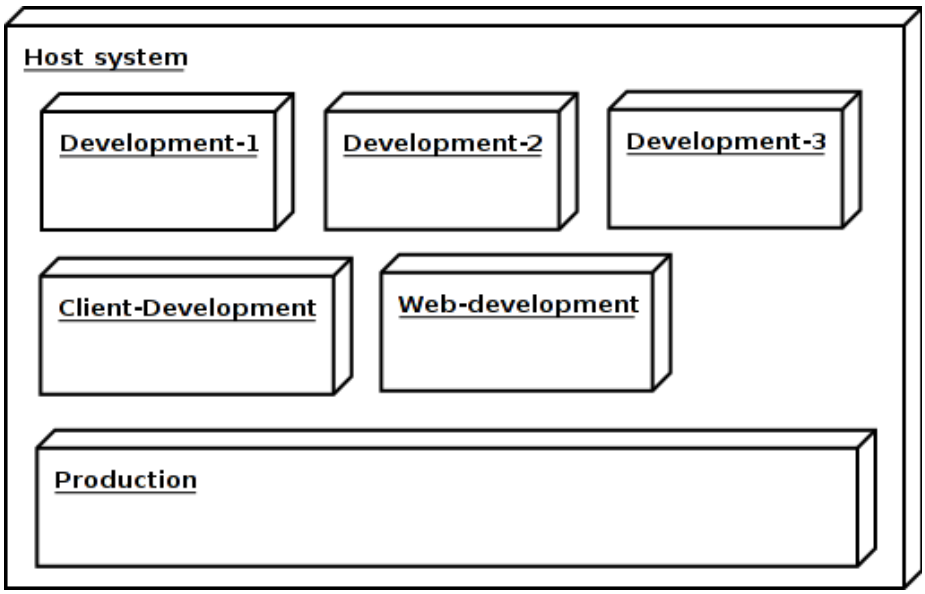


Figure B.1: Illustration of system setup

B.3.1.1 Using gnu screen

To ensure processes run even when nobody has a terminal open to the server `screen` is used. `screen` is a terminal multiplexer, which could be viewed as a in-terminal window manager. To start a new screen you run the command

```
$ screen
```

which then starts a `screen` and attaches your current terminal to it. `screen` is controlled by prefixing a command with a modifier. For example, to detach from a `screen` window, you use `<C-a>` which you then release, indicating you wish to send a command to `screen`. You then press `d`, which is the hot-key for *detach*. You are now detached from the screen.

You might not be surprised when I tell you that several `screen` windows can be running at the same time. To list the currently running `screen` sessions, you run the command

```
$ screen -list
```

If there is only one session running, you can attach to it using simply the command

```
$ screen -r
```

to attach. If there are several running `screens`, you must specify which session to attach to. The following is an example of this. It starts two sessions, `first` and `second` in a detached state. The `-dmS` flag means they are started in a detached state. We then list the sessions and see that several are running, as expected.

```
[vagrant@localhost ~]$ screen -dmS "first"
```

```
[vagrant@localhost ~]$ screen -dmS "second"
[vagrant@localhost ~]$ screen -list
There are screens on:
    15073.second    (Detached)
    15053.first (Detached)
2 Sockets in /var/run/screen/S-vagrant.
```

We can now attach to these sessions by running the command, replacing `<session-name` with the name of your session.

```
$ screen -S <session name>
```

To terminate a `screen`, attach to it and run `exit`.

B.3.2 Configured environments

The environments as currently configured are described below.

B.3.2.1 Production

This environment runs the production code. Hands off if you are unsure of or have not communicated your change clearly to everyone else.

- Virtual hardware
  - 45000 MB RAM
  - 8 CPU Cores
- Ports
  - Genomizer-server: 7000
  - Http: 80
  - Https: 443
- Local IP: 192.168.33.10
- Directories
  - Temporary files
    - External directory: `/Data/tmp`
    - Internal directory: `/tmp`
  - Data files
    - External directory: `/Data/production-data`
    - Internal directory: `/data`

B.3.2.2 Development-1

This environment is assigned to the `database` group.

- Virtual hardware
  - 4000 MB RAM
  - 1 CPU Core
- Ports

  - Genomizer-server: 7001
  - Http: 8081
  - Https: 4431
- Local IP: 192.168.33.11
- Directories
  - Temporary files
    - External directory: `/Data/development-1-tmp`
    - Internal directory: `/tmp`
  - Data files
    - External directory: `/Data/development-1-data`
    - Internal directory: `/data`

B.3.2.3 Development-2

This environment is assigned to the `business-logic` group.

- Virtual hardware
  - 4000 MB RAM
  - 1 CPU Core
- Ports
  - Genomizer-server: 7002
  - Http: 8082
  - Https: 4432
- Local IP: 192.168.33.12
- Directories
  - Temporary files
    - External directory: `/Data/development-2-tmp`
    - Internal directory: `/tmp`
  - Data files
    - External directory: `/Data/development-2-data`
    - Internal directory: `/data`

B.3.2.4 Development-3

This environment is assigned to the `processing` group.

- Virtual hardware
  - 4000 MB RAM
  - 1 CPU Core
- Ports
  - Genomizer-server: 7003
  - Http: 8083
  - Https: 4433
- Local IP: 192.168.33.13
- Directories
  - Temporary files

        * External directory: `/Data/development-3-tmp`
        * Internal directory: `/tmp`
    - Data files
        * External directory: `/Data/development-3-data`
        * Internal directory: `/data`

B.3.2.5 Client-development

This environment is assigned to the `desktop` group.

- Virtual hardware
    - 4000 MB RAM
    - 1 CPU Core
- Ports
    - Genomizer-server: 7004
    - Http: 8084
    - Https: 4434
- Local IP: 192.168.33.14
- Directories
    - Temporary files
        * External directory: `/Data/development-client-tmp`
        * Internal directory: `/tmp`
    - Data files
        * External directory: `/Data/development-client-data`
        * Internal directory: `/data`

B.3.2.6 Web-development

This environment is assigned to the `web` and `app` groups.

- Virtual hardware
    - 4000 MB RAM
    - 1 CPU Core
- Ports
    - Genomizer-server: 7005
    - Http: 8085
    - Https: 4435
- Local IP: 192.168.33.15
- Directories
    - Temporary files
        * External directory: `/Data/development-web-tmp`
        * Internal directory: `/tmp`
    - Data files
        * External directory: `/Data/development-web-data`
        * Internal directory: `/data`

B.3.3 The important scripts

The machines are configured using a myriad of scripts with various responsibilities. The `Vagrantfile` defines how the virtual machine is built. This includes which provisioning files are to be executed, and how much memory and resources to give the machine. It also configures the port forwarding into the machine, and which shared data folders are available to the machine. Messing with these configurations is inadvisable.

The `Vagrantfile` runs, as previously said, configuration scripts. Their names and basic responsibilities are listed below. Unless otherwise specified, they are located under `scripts/provision`.

1. `install_apache.sh` - Installs and configures the apache server. It installs the config files `httpd.conf`, `ssl.conf`, and `proxy.conf` located under `scripts/provision/config`, along with running the `install_certificates` script.
2. `install_postgresql.sh` - While the name may seem confusing, this installs `puppet` and the `puppet/postgresql` module. This allows the `Vagrantfile` to run puppet provisioning on the machine to install postgresql with the settings provided in `manifests/build.pp`.
3. `install_certificates.sh` - This is an expect script, which is run by the `install_apache` script. This is introduced to deal with the fact that you need to respond to questions to generate a SSL certificate. It generates a SSL certificate, and moves it to the correct location.
4. `install_utils.sh` - There are some tools and utilities which were requested or needed by the scripts or persons working on the project. They are installed from this script.
5. `install_startup_scripts.sh` - Installs various scripts that are used for server administration.
6. `install_genomizer_server.sh` - Installs the genomizer-server
7. `install_genomizer_webclient.sh` - Installs the genomizer-web client to the apache server.

If you are confused by what a script does, reading it can give you a clearer picture of what it does.

B.3.4 Creating a new environment

B.3.4.1 Considerations

Before setting up a new environment, you should consider the memory and CPU footprint inherent in running a virtual machine. The host machine as currently configured can run 6 environments with no noticeable slowdown, with five machines at 1 cpu core and 4GB of RAM, and a production environment that is assigned 8 CPU cores and 45GB of RAM. It is not recommended to exceed this configuration, as unexpected side-effects could occur.

B.3.4.2 Checking out the correct files

When building a new environment, you may wish to configure which versions of the server software and web client software the script installs. By default, the scripts are configured to fetch the `master` branch of the `genomizer-server` and the `master` branch of the `genomizer-web` application. These settings are modified by editing the `scripts/provision/install_genomizer_server.sh` and `scripts/provision/install_genomizer_webclient.sh` respectively.

In order to change the version of the `genomizer-server`, edit the `scripts/ provision/install_genomizer_server.sh` script. Locate the line(s)

```
1 # We checkout the master version by default
2 git checkout master
```

and replace them with

```
# We checkout the <desired branch> version by default
git checkout <desired branch>
```

To change the version of the `genomizer-web` client, edit the `scripts/provision/ install_genomizer_webclient.sh` script. Locate the line(s)

```
# We run the master branch by default
cd genomizer-web
git checkout master
```

and replace them with

```
# We run the <desired branch> branch by default
cd genomizer-web
git checkout <desired branch>
```

B.3.4.3 Setting up the files

When setting up the new environment, you may (or may not) want to run it with the same settings as the production environment. In this section all relevant settings are explained and motivated.

B.3.4.3.1 Editing the Vagrantfile

In the vagrant file, the primarily interesting settings are concerning *port forwarding*, *local ip*, *synced folders*, and *memory*. Given that you wish to customize the environment, the *provision* settings might be of use as well. The `Vagrantfile` is written in a Ruby DSL.

The port forwarding settings looks as follows.

```
config.vm.network "forwarded_port", guest: 80, host:8080
config.vm.network "forwarded_port", guest: 7000, host:7000
config.vm.network "forwarded_port", guest: 443, host:4443
```

The `host` field defines which port the virtual machine binds on in the host machine, while the `guest` defines the port the virtual machine it binds on internally.

To define which **local IP** the virtual machine runs on, you modify the line below to suit your needs. Note that the IP must be unique, and it is recommended to stay inside the `192.168.33.x` subnet.

```
config.vm.network "private_network", ip: "192.168.33.10"
```

You may also need to mount a folder from the host machine as a drive inside the virtual machine. This is achieved by editing or adding to the lines shown below. The first argument is the hosts path to the folder to share, and the second is where the virtual machine should mount it.

```
config.vm.synced_folder "/Data/production-data", "/data"
config.vm.synced_folder "/Data/production-tmp", "/tmp"
```

The actual hardware specifications given to the virtual machine is defined by these lines:

```
config.vm.provider "virtualbox" do |vb|
    vb.memory = "45000"
    vb.cpus = 8
end
```

These lines define that the machine should run with 45000 MB of RAM, and run on 8 CPU cores. If other settings are desired, you modify these lines. For example, the development environments might look like

```
config.vm.provider "virtualbox" do |vb|
    vb.memory = "4000"
    vb.cpus = 1
end
```

B.3.4.3.2 Editing the settings.cfg

The `settings.cfg` file contains the settings that are installed to the genomizer-server. These should need no modification, except perhaps to edit the tunneling settings.

If you however do want to edit some settings, note that the database settings must correspond to the ones configured in `build.pp`, else the server will fail to connect to the database.

The settings available in the server settings are straightforward, and are commented such that no further explanation is required.

B.3.4.3.3 Editing the httpd.conf

The `httpd.conf` file is dead-standard, except for the very last few lines. These lines assures that SSL is forced, and that any traffic connecting to the non-SSL apache is told to reconnect with SSL.

```
<VirtualHost *:80>
    RedirectPermanent / https://130.239.192.110
</VirtualHost>
```

You may wish to edit where this reroutes to as needed.

B.3.4.3.4 Modifying the build.pp

The `build.pp` file details how the postgresql database is constructed, along with which sql files are automatically run on the server. It is not recommended to change how this file functions except to change the password and modify which sql files are run.

B.3.4.3.5 Creating the `tmp` and `data` folders

When constructing an environment, the machine requires a place to store temporary files, and a place to store large data files. These are by default placed in `/Data/<environment>-tmp` and `/Data/<environment>-data`. To create these folders, and give them the correct permissions, you should run the following commands. The `<your-user>` should be replaced with the user that runs the virtual machines, and `<environment>` with the name of your environment.

```
[user@your-environment]$ sudo mkdir /Data/<environment>-tmp
[user@your-environment]$ sudo mkdir /Data/<environment>-data
[user@your-environment]$ sudo chown -R <your-user> /data/<environment>-tmp
[user@your-environment]$ sudo chmod -R 1777 /data/<environment>-tmp
[user@your-environment]$ sudo chmod -R 777 /data/<environment>-data
```

B.3.4.4 Running the new environment

Once you have created an environment, you need to allow vagrant to construct the associated virtual machine. This is done by running `vagrant up` in the directory associated with your environment.

Vagrant should now start building the virtual machine, along with provisioning it. It will write a lot of information to the terminal you run it in, and this is perfectly normal. The build takes on average 5-10 minutes to complete.

Once it has finished, the environment is up and running, except for one detail: `genomizer-server` is not running. You can start it by typing the following into the terminal.

```
[user@environment]$ vagrant ssh
Last login: Tue May 12 12:4032 2015 from 10.0.2.2
[vagrant@localhost]$ bash startup.sh
```

If you wish to view the server process, this can be done by running

```
[vagrant@localhost]$ screen -r
```

which attaches to the screen created by `startup.sh`.

B.3.5 Modifying an existing environment

While it is very useful to build an environment from scratch, you do not always want to do everything from scratch. You want to *modify* an existing environment. Modifying an environment is fairly straight forward, you simply edit the configuration files, as shown in *Creating a new environment*. Once you have modified the files according to your preferences, you run

```
[user@your-environment]$ vagrant destroy && vagrant up
```

if you wish to *completely* rebuild the machine. If you simply want the machine to reload the `Vagrantfile` settings, you can run

```
[user@your-environment]$ vagrant reload
```

B.3.5.0.1 Checkout new branches

If you decide that you wish to run another branch in the environment, and do not wish to rebuild the environment, you can simply ssh into the vagrant machine and checkout using git as you normally would, compiling and running as you would normally.

```
[user@your-environment]$ vagrant ssh
[vagrant@localhost]$ cd genomizer-server
[vagrant@localhost]$ git stash
[vagrant@localhost]$ git checkout <branch>
[vagrant@localhost]$ git stash pop
[vagrant@localhost]$ ant clean build jar
[vagrant@localhost]$ java -jar server.jar
```

In this context, the `stash` ensures that our modified `settings.cfg` is preserved. If for some reason `settings.cfg` is overwritten, you can get a fresh copy from `/vagrant/scripts/provision/config/settings.cfg` inside the virtual machine.

B.3.6 Rebuilding an environment

Rebuilding an environment is dead simple, simply run

```
[user@your-environment]$ vagrant destroy && vagrant up
```

B.3.7 Deleting an environment

NOTE: The changes explained in this chapter are irreversible!

To delete an environment it is not sufficient to simply remove the folder containing the environment. The virtual machine must also be deleted. The following steps removes an environment cleanly.

1. Enter the environment folder.
2. Destroy the virtual machine with `vagrant destroy`.
3. Remove the `.vagrant` folder.
4. Remove the environment folder.
5. Remove the environments `tmp` and `data` folders.

The `tmp` and `data` folders are by default located under `/Data/<environment>-tmp` and `/Data/<environment>-data`.

B.3.8 Configuring the host system

Please note, all of these things can be achieved by running the script `scripts/ environment/setup_host_environment.sh`, provided a CentOS host system. **Never run this script without reading it through. Seriously.** Even if you want to configure manually, reading this script is useful to get examples of the commands that are relevant.

B.3.8.1 External folders and their permissions.

The host machine is required to make available storage space for the virtual machines. By default this is assumed to be `/Data`. In this folder, you should place `tmp` and `data` folders – one for each environment – with the permissions 1777 for `tmp`. The `data` folder should have the permissions 777. The `tmp` folder must be owned by the vagrant user.

B.3.8.2 Configuring the firewall

The firewall needs to allow *at least* the ports `80`, `443` and `7000`. There is a script which configures these settings under `scripts/environment/iptables_config.sh`. The only advanced setting here is a workaround to patch reserved ports into the virtual machines without requiring root permissions.

The firewall is instructed to reroute traffic to port `80` to the port `8080` and traffic to the port `443` to the port `4443` which then the virtual machine can bind on.

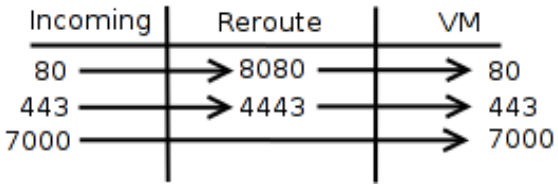


Figure B.2: Illustration of rerouting

Specifics on how to run these commands are illustrated in `iptables_config.sh`.

B.3.8.3 Prerequisite software

The setup requires `vagrant` and `virtualbox` to be installed.

B.3.8.3.1 Virtualbox

To install virtualbox, you can run the following commands. Replace `$VAGRANT_USER` with the vagrant user.

```
# Get the EPEL repo
rpm -Uvh https://download.fedoraproject.org/pub/epel/7/x86_64/e/\
    epel-release-7-5.noarch.rpm

# Install kernel headers and devel tools
yum -y install kernel-devel kernel-headers dkms
yum groupinstall "Development Tools"
yum update

# Install oracle public key
wget http://download.virtualbox.org/virtualbox/debian/\
    oracle_vbox.asc
rpm --import oracle_vbox.asc
rm -rf oracle_vbox.asc

# Add the virtualbox repo
wget http://download.virtualbox.org/virtualbox/rpm/el/\
    virtualbox.repo -O /etc/yum.repos.d/virtualbox.repo
yum update # Update repo info for safety

# Time to actually install virtualbox
yum -y install VirtualBox-4.3
service vboxdrv setup # Setup vbox driver
usermod -a -G vboxusers $VAGRANT_USER # Add VB user
```

B.3.8.3.2 Vagrant

To install vagrant, you can run the following commands.

```
wget https://dl.bintray.com/mitchellh/vagrant/\
    vagrant_1.7.2_x86_64.rpm
rpm -ivh vagrant_1.7.2_x86_64.rpm
rm -rf vagrant_1.7.2_x86_64.rpm
```

B.4 Administer the database

This chapter contains instructions for setting up the postgresql database and user accounts.

B.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_adresses = '<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
    ```

B.4.2 Upload SQL Script to server

1. In a termainal 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
    ```

B.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.

B.5 Set up processing

To be able to run the processes such as raw to profile convertion 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.

B.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. If these are already installed refer to B.6.1 on how to download the source files.

B.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.

B.6.2 Creating a runnable JAR file

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.

B.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*.

C 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.

C.1 Implemented user stories

Annotation

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

Figure C.1: Annotation user story

Single download

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

Figure C.2: Download user story

Single upload

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

Figure C.3: Upload user story

Search for data

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

Figure C.4: Search for data user story

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.

Figure C.5: Batch upload user story

Raw to profile

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

Figure C.6: Raw to profile user story

Delete data

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

Figure C.7: Delete data user story

Change annotation

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

Figure C.8: Change annotation user story

Backup

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

Figure C.9: Backup user story

Password protected

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

Figure C.10: Password protected user story

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 C.11: Add genome release / reference genome user story

Add chain file

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

Figure C.12: Add chain file user story

Batch download

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

Figure C.13: Batch download user story

C.2 Product backlog

File traceabillity

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 C.14: File traceabillity user story

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).

Figure C.15: Convert common file formats user story

Convert genome release

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

Figure C.16: Convert genome release user story

Extract genome reference sequence

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

Figure C.17: Extract genome reference sequence user story

Advanced batch upload

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

Figure C.18: Advanced batch upload user story

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 C.19: Profile to region user story

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.

Figure C.20: Workspace user story

Unread results

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

Figure C.21: Unread results user story

Sort search results

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

Figure C.22: Sort search results user story

Preview of file

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

Figure C.23: Preview of file user story

Work scheduling

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

Figure C.24: Work scheduling user story

Work queue

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

Figure C.25: Work queue user story

User rights

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

Figure C.26: User rights user story

Time estimation

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

Figure C.27: Time estimation user story

Plot overlap analysis

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

Figure C.28: Plot overlap analysis user story

Plot average regions

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

Figure C.29: Plot average regions user story

IGB Session

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

Figure C.30: IGB Session user story

Combine regions

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

Figure C.31: Combine regions user story

Create region subset

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

Figure C.32: Create region subset user story

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).

Figure C.33: Calculate average of region user story

Overlap analysis

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

Figure C.34: Overlap analysis user story

Save analysis results

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

Figure C.35: Save analysis results user story

D Android application: *UML*-diagrams

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

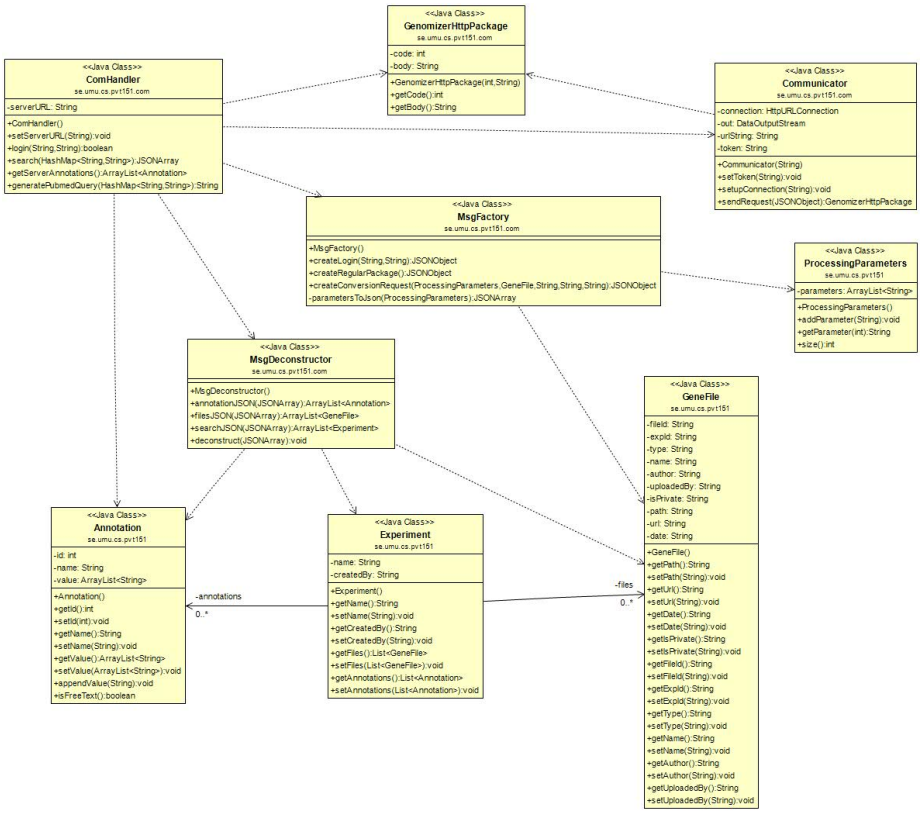


Figure D.1: Android UML of model

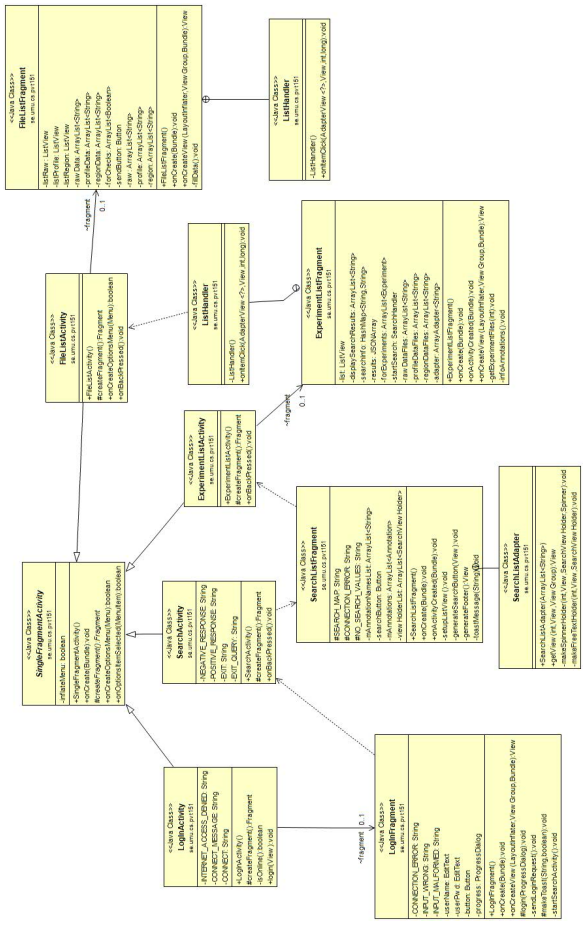


Figure D.2: Android UML without model

E iOS application: *UML*-diagrams

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

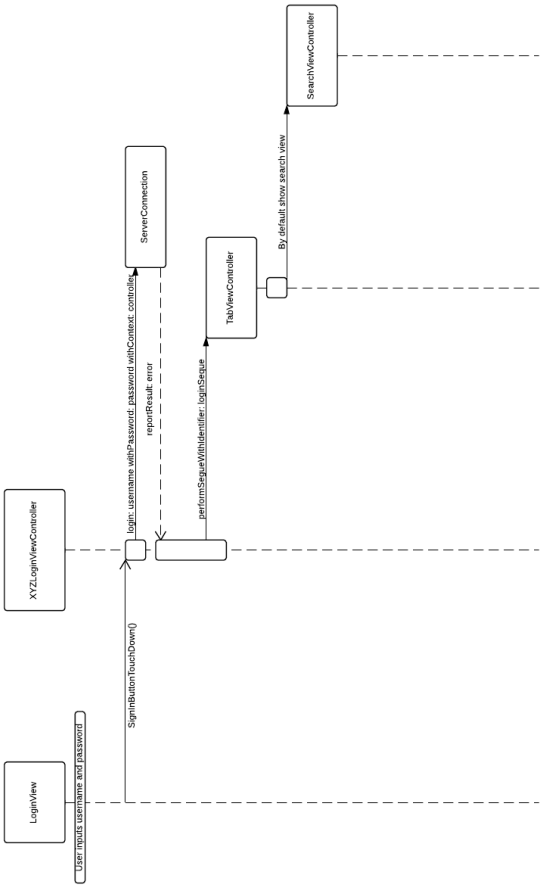


Figure E.1: Login sequence diagram.

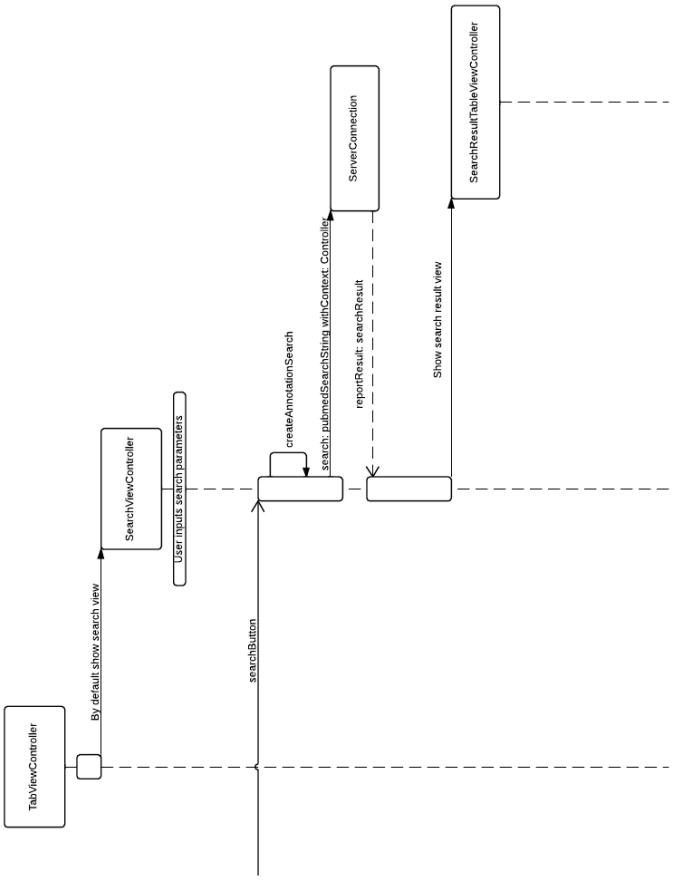


Figure E.2: Search sequence diagram.

F Desktop application: *UML*-diagrams

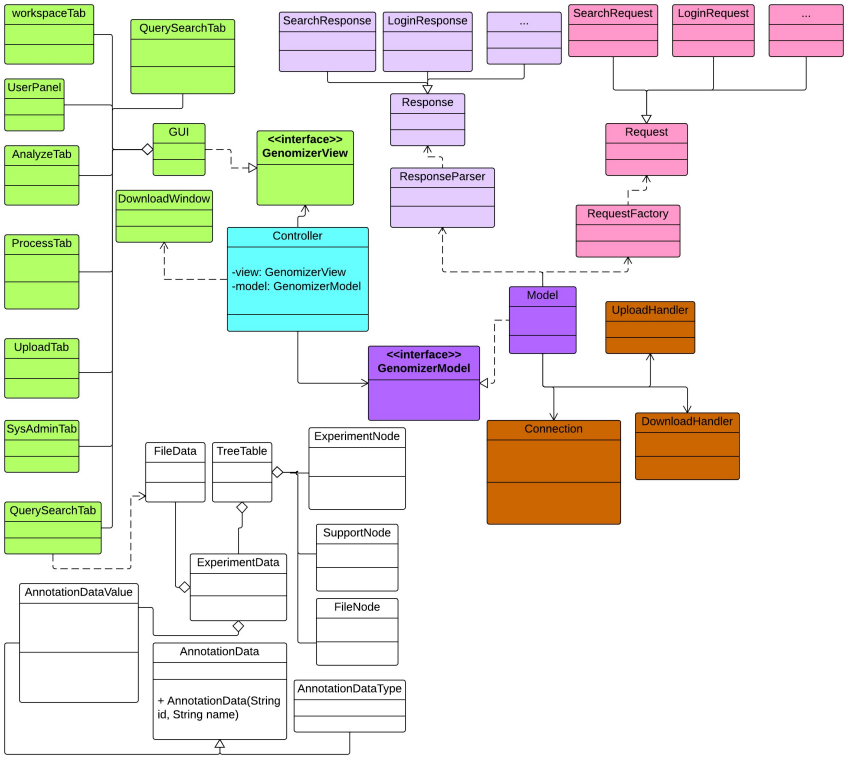


Figure F.1: `Desktop Overview` with major class relations.

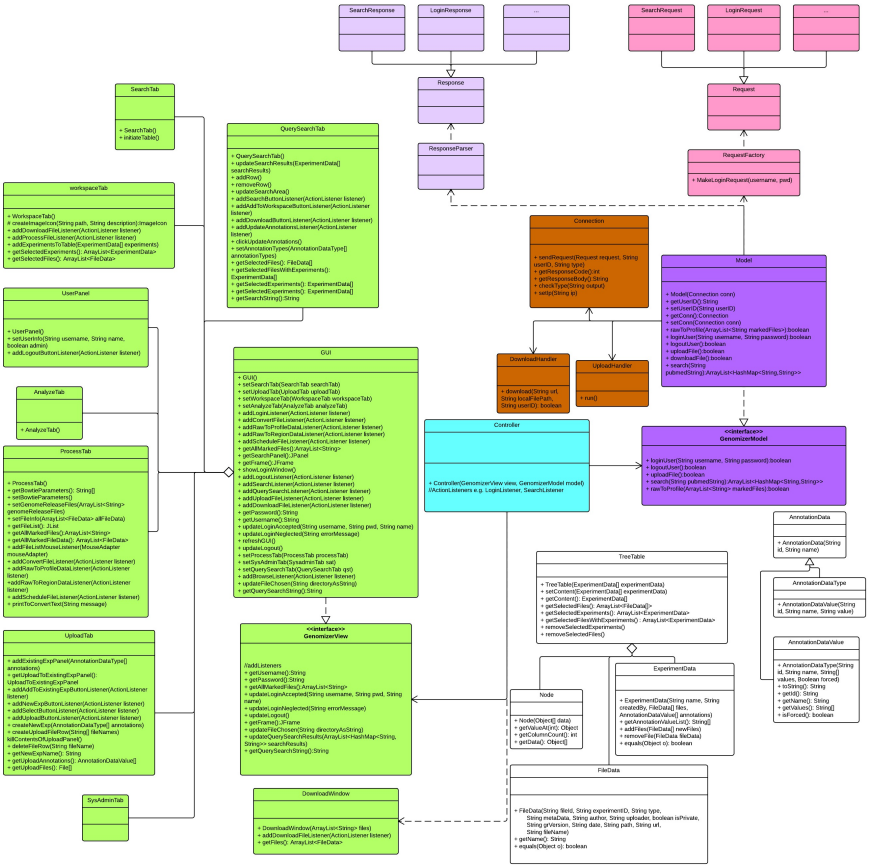


Figure F.2: `Desktop Overview` with major class relations.

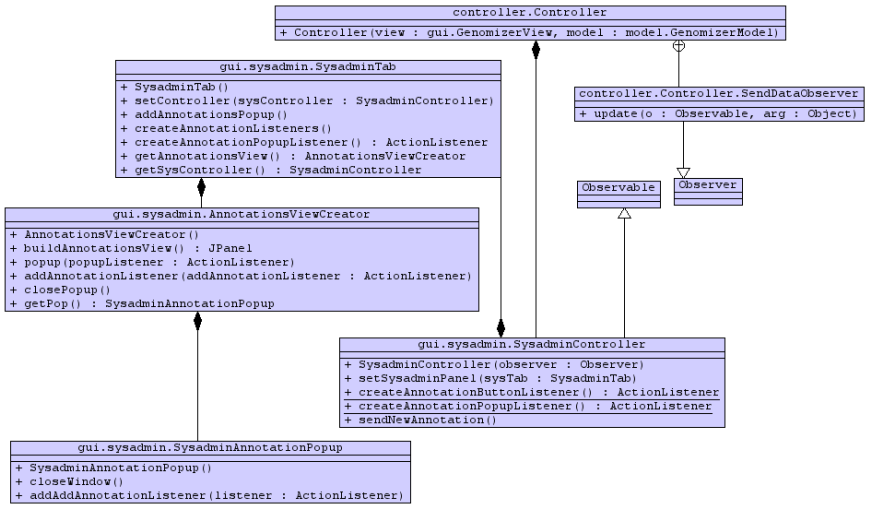


Figure F.3: The `Controller` and `Sysadmin` class relations.

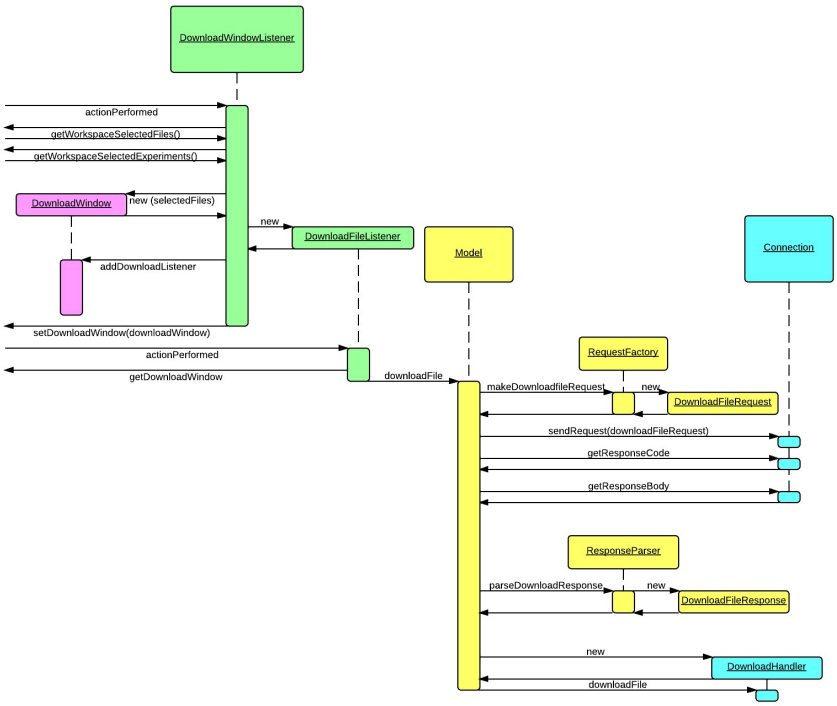


Figure F.4: The download sequence.

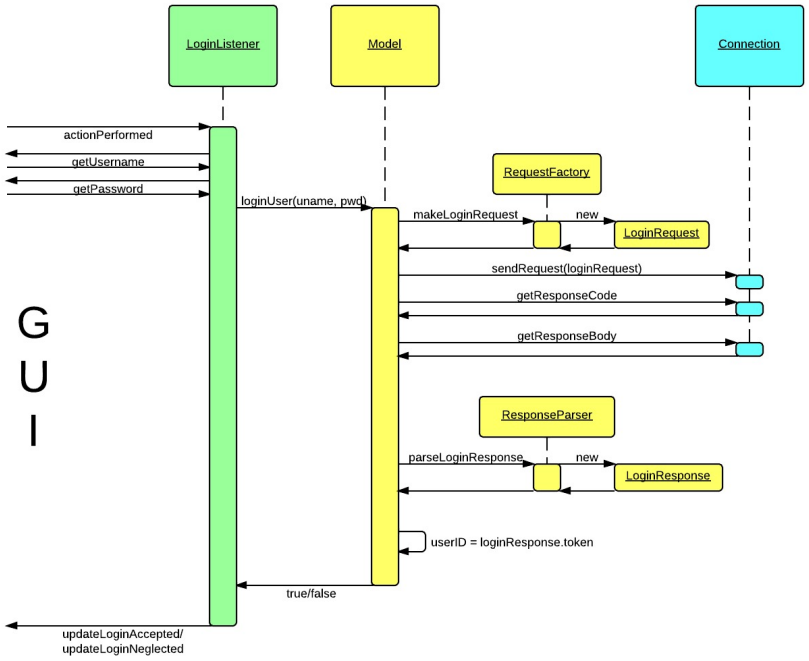


Figure F.5: The login sequence.

G Data Storage: *UML*-diagrams

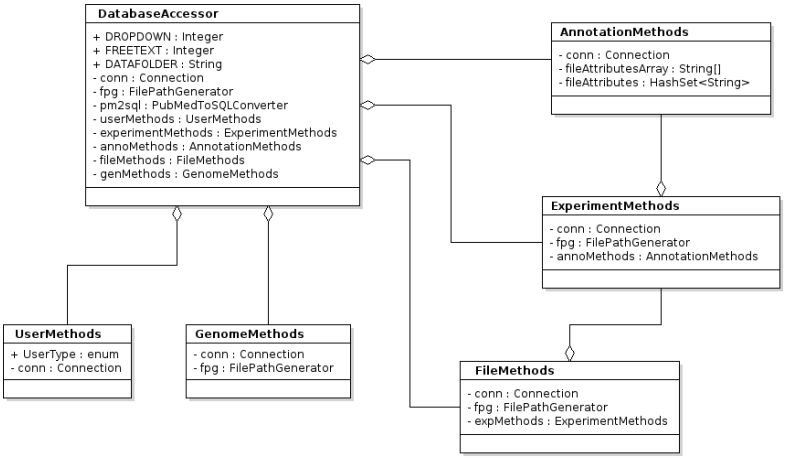


Figure G.1: The wrapper class `DatabaseAccessor` and the method subclasses

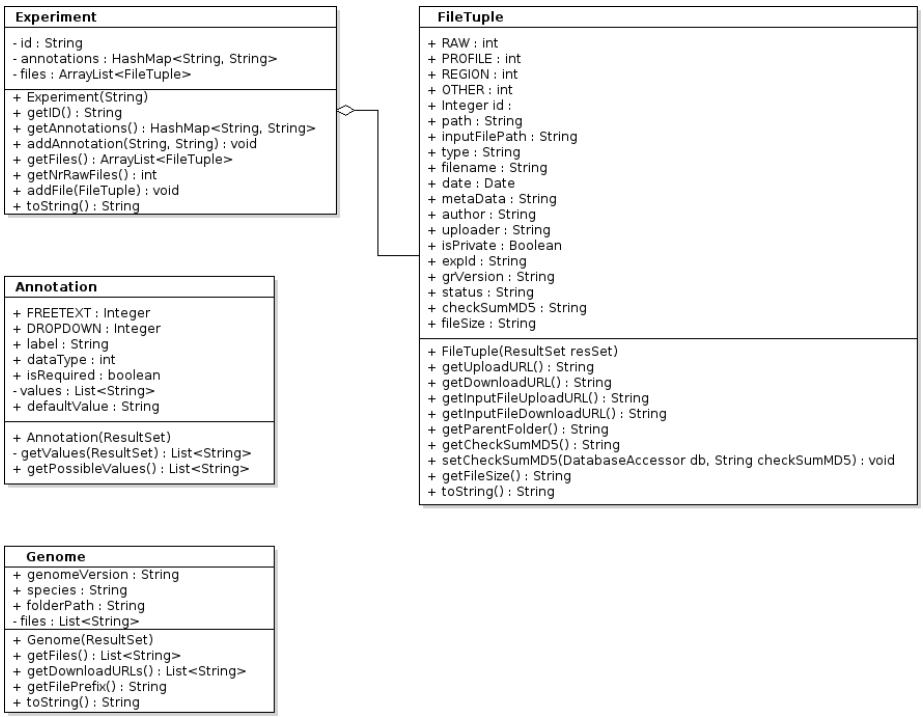


Figure G.2: The container classes used to return information

H 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/data/Exp1/raw/f
                         "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/data/Exp1/raw/f
                         "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/data/Exp1/raw/f
                         "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 (appliaction/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 (appliaction/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
```

I *Backup*

I.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
```

I.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/blcmdl1_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
```

I.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 passdword for the user of the postgresql database.

I.4 Chrontab

To enable the server to automatically do syncronizations 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.

J Known problems

J.1 Web application

J.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

J.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.

J.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 dissapear.

Show login modal without redirecting to root url and save errorous ajax-request and resend it when the login has been completed.

J.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.

J.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.

J.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.

J.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.

J.1.8 Sidebar on adminpage dosen't stay vertical

The sidebar items goes horizontally when page width is 2480px.

CSS the sidebar better.

J.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

J.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.

J.2 *iOS* application

J.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.

J.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.

J.2.3 No administrative features

The app does not have any of the administrative functions of the web and desktop clients.

J.3 Server

J.3.1 Communication and control

The communication between the server and the clients have some limitations and security holes. These limitations are described below.

J.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.

J.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.

J.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.

J.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.

J.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.

J.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/*.

J.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.

J.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.

J.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 windowsize. 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.