**Filtering Reads Made Easy**

**<NAMES>**

1 [The Shmunis School of Biomedicine and Cancer Research](https://en-lifesci.tau.ac.il/lp-en-mcbb), George S. Wise Faculty of Life Sciences, Tel Aviv University, Tel Aviv 69978, Israel.

† To whom correspondence should be addressed:

Tal Pupko, Tel: +972 3 640 7693; Fax: +972 3 642 2046; E-mail: talp@tauex.tau.ac.il

Keywords: sequencing, webserver, DNA contamination, interactive webpage

# Abstract

Recent advances in sequencing and informatics technologies have led to a deluge of sequencings. This has dramatically changed our understanding of the genome and the understating of biology. While it is now relatively easy to sequence, extracting only meaningful reads is still a challenge. Here we introduce a new webserver to facilitate the filtering process. Three of the main features implemented in <Name> are: (1) automate updating of the latest bacteria database; (2) fast comparing the input reads to the database; (3) a simple interactive web page to further analyze the results.

# Introduction

Over the past decades, the price of sequencing has been decreasing rapidly (Check Hayden 2014). This allowed research groups all over the world to sequence different species and doubled the size of the GenBank every few months (Benson et al. 2018). The data provided by those sequencing is a vital resource in biology and clarified many unknowns.

With the increase of DNA sequencing, confounding contaminant DNA takes place as one of the main challenges. Instead of finding the genome of the required species, some of the reads are contaminated and originate in different species (Steinegger and Salzberg 2020). This contamination can have a significant effect on the later analysis including false positive SNP’s identification (Goig et al. 2020), incorrect labels on sequences in metagenomic studies (Kirstahler et al. 2018).

The filtering process requires the use of a search engine and downloading required genomes databases. Some research laboratories began implementing their in-house analysis pipelines, and later, different search engines emerge (Wood, Lu, and Langmead 2019). These applications require specific working environments (i.e., operating systems), computation power (multicore machines), more than basic technological skills (e.g., installation and running), and many parameters to configure. Other laboratories do not filter the reads and hence find false discoveries while wasting money and time.

Here we present <Name> web server. The web server was developed to easily filter genomics reads and make it more accessible for the scientific community. This is done by updating the latest database and searching each of the reads in it. No installation and no other prerequisites are needed. A simple and interactive graphical user interface (i.e., GUI) personalizes the user output. Visual and textual results that are ready for publication or further analysis.

# Materials and Methods

## Input

The <Name> web server requires a fasta or fastqc file containing the reads to be filtered. In the analysis stage, we ask the user for the species list he would like and for a k-mer percentage threshold to classify as contamination. Upon completion of the submission, a link to the results is sent to the user if they choose to provide their email address. The results remain available on the webserver for at least 3 months.

## Database

Each month, we download the full bacteria database from <NCBI?>. This is an automated process to verify our bacteria’s genomes are up to date. When the download is completed, the data is preprocessed based on the search engine.

## Search Engine

To conduct a fast search, we installed the Kraken 2 search engine (Wood, Lu, and Langmead 2019). With a given read file, each read is split into k-mers and then searched in the bacteria’s database. <DEFAULT PARAMETERS ARE USED?>. The output of the Kraken 2 search engine is a csv file containing all the reads with a list of species with the number of k-mers found in this read.

## Preprocess Kraken Output

For each read, we sum the amount of k-mers found for each of the species. Then, we save the species with the max k-mers, and the percentage of similarity to the species. This is done, by dividing the k-mers of the max species by the length of the read. To display the data in graphs fast, we transform the list of the reads into a matrix. The rows of the matrix contain the k-mers percentage (this value represents the similarity of the read to one of the species) and the columns of the matrix are the species names. The cells are the number of reads that fits the specific row and column <INSERT FIGURE>.

## User Interactive Analysis

First, the matrix described above is loaded. The sum of each row represents the number of reads with this similarity <INSERT FIGURE>. While the sum of each column represents the number of reads associated with this species. One could find its optimal threshold of k-mer percentage by controlling the slider. In addition, one could control the species list that will be classified as contamination DNA. Those two parameters will immediately change the graphs thus, one could see the effect of the contamination. When the user is ready to export the results, the threshold for the k-mer percentage and the species list will be delivered to classify each of the reads. When the classification process is finished, the user will get a file containing all the unclassified reads.

### Implementation

<NAME> is implemented in Python using Flask framework. The source code is available at: [<GITHUB](https://github.com/orenavram/microbializer) LOCATION>. The web server jobs are processed on ProLiant XL170r Gen9 servers, equipped with 128 GB RAM and 28 CPU cores per node. The Gallery, Overview, and Frequently Asked Questions (FAQ) sections of the web server should help users get the most out of the web server. A running example (different from the case studies analyzed in the Gallery) is also provided.

# Test Case

<ADD ONE TEST CASE>

**References**

Benson, Dennis A, Mark Cavanaugh, Karen Clark, Ilene Karsch-Mizrachi, James Ostell, Kim D Pruitt, and Eric W Sayers. 2018. “GenBank.” *Nucleic Acids Research* 46 (D1): D41–47. https://doi.org/10.1093/nar/gkx1094.

Check Hayden, Erika. 2014. “Technology: The $1,000 Genome.” *Nature* 507 (7492): 294–95. https://doi.org/10.1038/507294a.

Goig, Galo A., Silvia Blanco, Alberto L. Garcia-Basteiro, and Iñaki Comas. 2020. “Contaminant DNA in Bacterial Sequencing Experiments Is a Major Source of False Genetic Variability.” *BMC Biology* 18 (1): 24. https://doi.org/10.1186/s12915-020-0748-z.

Kirstahler, Philipp, Søren Solborg Bjerrum, Alice Friis-Møller, Morten la Cour, Frank M. Aarestrup, Henrik Westh, and Sünje Johanna Pamp. 2018. “Genomics-Based Identification of Microorganisms in Human Ocular Body Fluid.” *Scientific Reports* 8 (1): 4126. https://doi.org/10.1038/s41598-018-22416-4.

Steinegger, Martin, and Steven L. Salzberg. 2020. “Terminating Contamination: Large-Scale Search Identifies More than 2,000,000 Contaminated Entries in GenBank.” *Genome Biology* 21 (1): 115. https://doi.org/10.1186/s13059-020-02023-1.

Wood, Derrick E., Jennifer Lu, and Ben Langmead. 2019. “Improved Metagenomic Analysis with Kraken 2.” *Genome Biology* 20 (1): 257. https://doi.org/10.1186/s13059-019-1891-0.

# Technical Description

## Overview

<VERIFY WITH ELYA>

The webserver is implemented using the Flask framework (instead of the CGI). This framework eases the implementation parts: uploading files, routing requests, returning files, and so on. The Apache is one more actor in the server flow controlled by the IT team. The university servers are all served by the same URL. The Apache routes the requests to the correct server and machine. To upload the server, one need to contact the IT and request to make a config for the Apache instance.

Frontend backend is the most common architecture used in webservers (and many other systems). This simple concept is dividing the system into two parts: client (frontend) and server (backend). An API constitutes a bridge between those parts. This allows modularity the frontend does rely on API but not on the backend. Our frontend is implemented with HTML and JavaScript, while our backend is implemented in Python and Bash commands. The communication between the frontend and the backend is by using a request and the other way is by using a response. Note that the client (frontend) runs at the client-side (usually a browser), and thus it cannot run complex things. Our backend purpose is to run the processes per user as requested, on the queues.

We have split the worked into three parts, as the design allowed:

frontend (client)

backend

processes

API

Elya

Edo

Michael

Response

Request

Listener

Create Jobs

HTML, JavaScript

Python

Python, Bash

## Frontend

## The frontend is a code that runs at the user's device. It can be a phone, a laptop, or any other computer. We cannot know the device's computational capabilities, so we should run only simple things. Our front is implemented in HTML and JavaScript.

<ANYTHING ELSE TO EXPLAIN?>

The following graph displays our main UI pages. The object that directs the user to the correct page is implemented at the \_\_init\_\_ object (will be explained at the backend section). Note, that every user has different pages as its input and the processes states are different.

Home:

* User upload file and email and then redirected to Process1 state
* Other links: about, Git, article

Process1 state:

* Displays the state of the process, when process finished, redirected to results page

Results (explained in detail below):

* Display the summary of the results reads
* Interactive page
* Users choose parameters for filtering and redirected to Process2 state

Process2 state:

* Displays the state of the process, when process finished, redirected to download page

Download:

* User can download the filtered reads

### Results Page

This is an interactive page contains two graphs: a histogram and a piechart. To reduce the amount of time between a user click and the update of the graphs, we used a 2d matrix containing the required information of the reads. Note, that the number of reads can be millions so passing and saving all this information will take a long time. Summarizing the information into a matrix reduces the amount of data saved and passed. This matrix makes it possible to do an interactive page.

The rows of the matrix are the k-mer-similarity (how much the sequenced read is like a genome at the searched database), and the columns are the species (which species the reads belong to). Each cell contains the number of the reads referred by this row and matrix. Building the histogram graphs requires summing all the rows. Note, that choosing a species to filter will change the histogram, as this species reads will now be not contaminated. Thus, recalculating the histogram requires removing the columns of the uncontaminated species from the contamination sum, and summing it in a different value. Building the piechart requires summing all the columns. Choosing the species not to filter sums those columns with the uncontaminated reads. Using this matrix allows short calculation that can be made in the client without requests additional information from the backend (only the loading of this matrix).

<APPEND IMAGES OF CLIENT AND A MATRIX EXAMPLE>

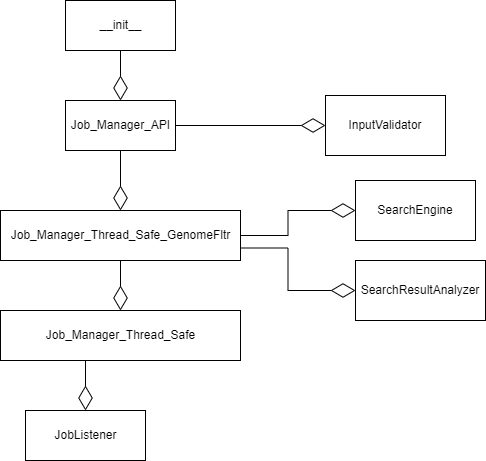
### 2-way communication

One of the disadvantages of using the old servers' implementation (CGI) is that the frontend can only communicate with the backend while the backend cannot send information to the client. Thus, the way to update the frontend pages is by refreshing the page every few seconds and hoping that the state has been changed. With the new Flask framework, it was important to us to stop these uncontrolled refreshments. There are many ways to fix this issue like WebSockets or push notification (which allows 2-way communication). Our implementation was simpler by adding a "stream" endpoint which all the clients listen to. When the process state of a client is changed, the process id is streamed to all the clients and if this id matches the client, it makes a refresh for the page. When the client refreshed the page, it sends a request to the backend and removes the process id from the list of processes that need to be updated. Note that now, the refreshments of the frontend happen once only when needed.

## Backend

The backend manages and handles the client requests. The main purpose is to wrap the PBS processes and manage them. It also, update the client on the changed state of those processes and organize the results of the processes.

The following UML diagrams displays the objects and their relations at the backend:



\_\_init\_\_:

* Route the client requests
* Alert user on errors
* Some verification on user input
* Download user file to server and send filtered file to user

Job\_Manager\_API:

* Wrap the management of the processes
* Returns results file to client
* Verify user input (using InputValidator)

InputValidator:

* Verify user input file
* Unzip user file (if needed)

Job\_Manager\_Thread\_Safe\_GenomeFltr:

* This object is different for the webservers to come
* Declare the different processes (by SearchEngine and SearchResultAnalyzer)
* Declare the updated of processes state

Job\_Manager\_Thread\_Safe:

* This object is same for the webservers to come
* Manage the processes
* Manage the waiting list
* Gets the indication of processes state changed (from the JobListener)

JobListener:

* Listens to queues with “qstat” command
* Calls the required function when job stated changes

### Describe Job\_Manager\_Thread\_Safe Implementation

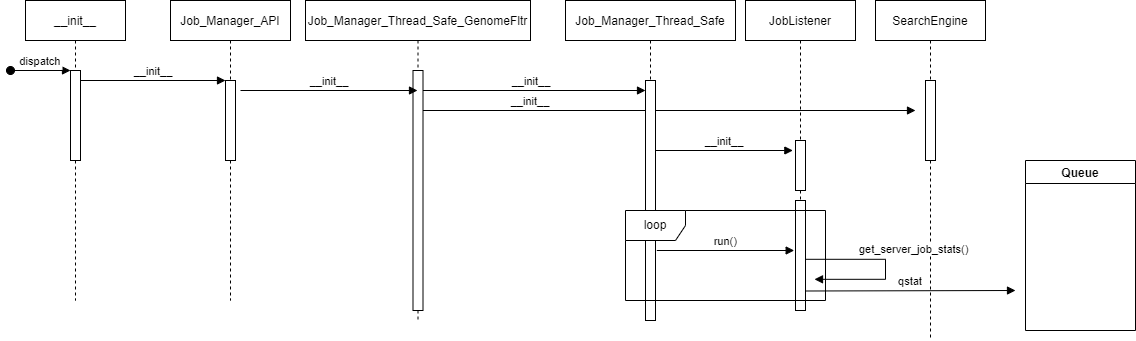
<COMPLETE>

## Sequences Diagrams

Sequences diagrams display the relations between the different objects in the process. Each object has its column (the title is at the top of the column). Arrows between objects refer to calling a function, and dashed arrows refer to values returned from the function. Loops are marked in rectangles.

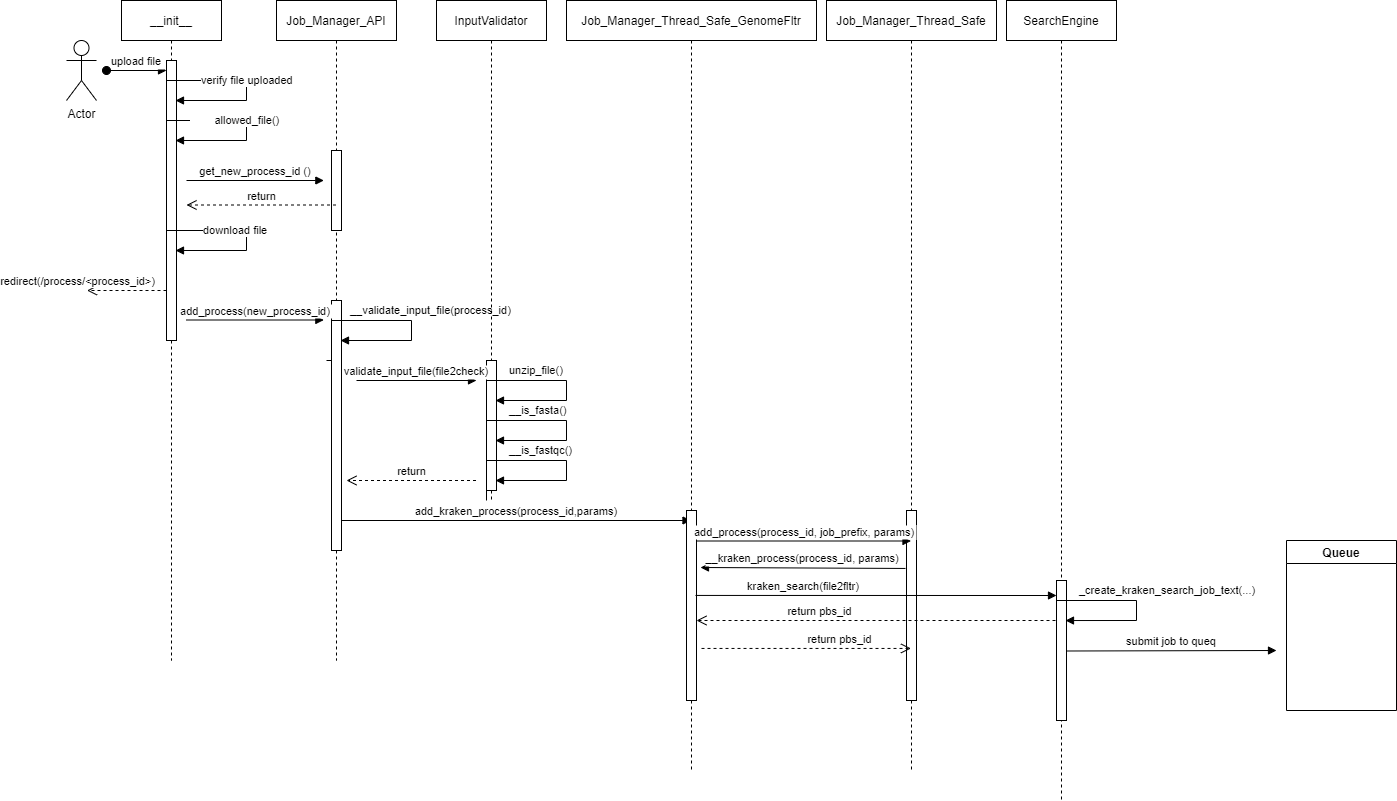
### Init Flow

What happens in the server initialization:



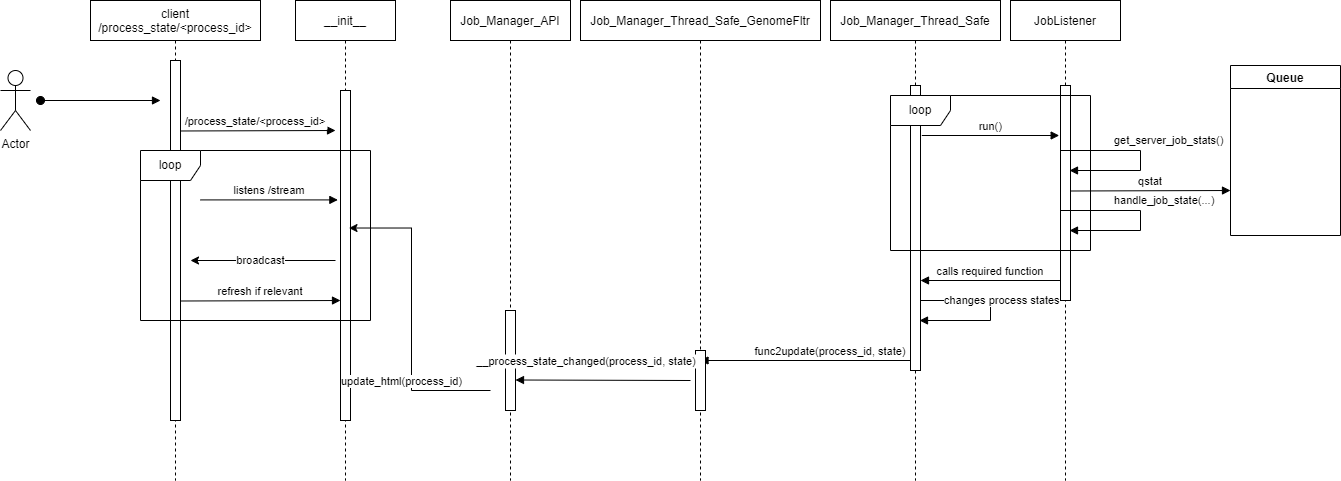
### Upload File Flow

What happens when user upload file:



### Job State Update Flow

What happens when the process state is changed:



### Kraken Search Process

<COMPLETE>

### Results Analyzer Process (postprocess)

<COMPLETE>

## Important Parameters to Configure

Probably in shared const.

## Creating another webserver

Stages for creating another webserver.