Automated Configuration Testing Framework for KBase

Team: SDMAY21-26

Client/Advisor: Dr. Myra Cohen

Daulton Leach, Jake Veatch, Daniel Way, Caleb Meyer, Sergey Gernega, Hunter Hall

Table of Contents

Appendices

Project Design

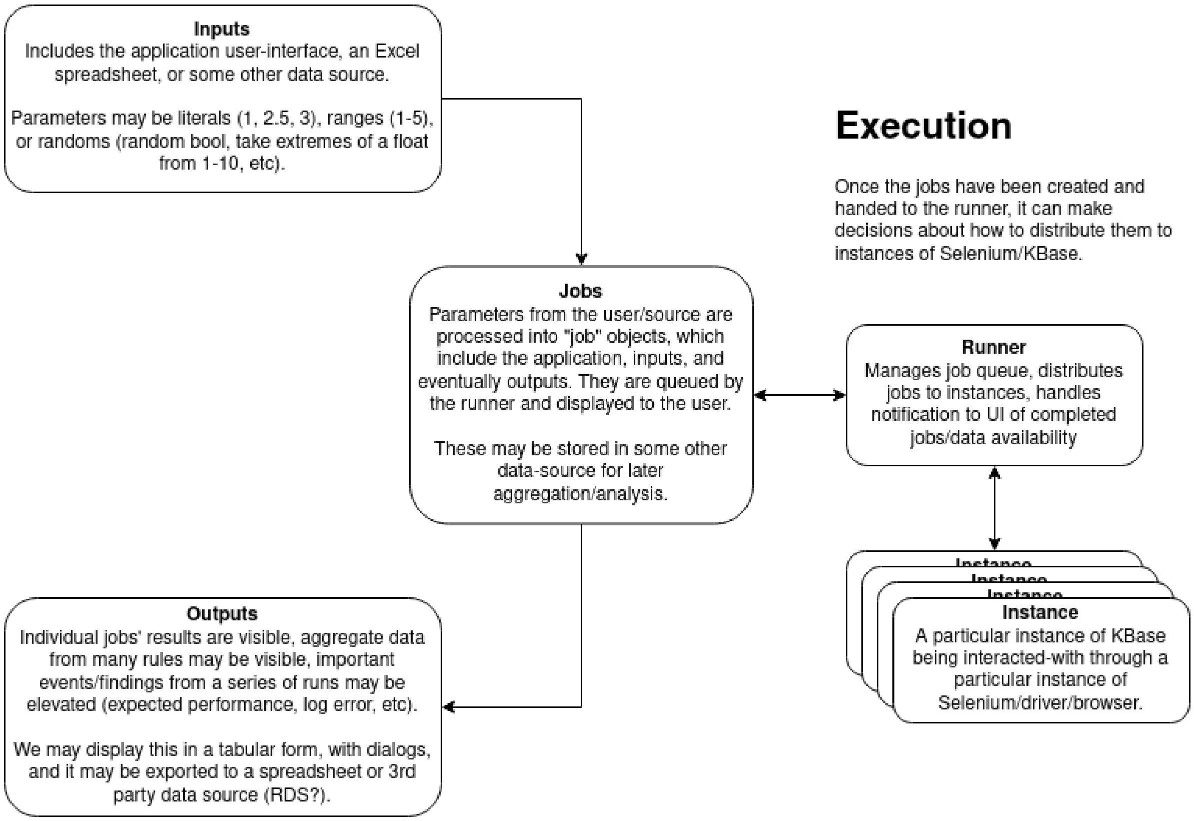
Design:

Problem: KBase is a platform used by biologists to simulate experiments based on a set of input parameters specified by the user. Those parameters are then run through calculations to give accurate results on the outcome of the experiments. The Biologists can then view the results and tailor the experiments virtually to get their desired results. There are numerous parameters and an exponential amount of combinations that can be executed. This creates a problem when testing and running simulations. For the biologists, an exponential amount of tests means running numerous experiments to get their results just right. KBase currently offers no concurrency or means to read tests from files. This can be taxing as each experiment takes roughly two minutes to complete. Biologists need a way to run potentially hundreds of simulations concurrently to become more efficient in their experimenting. The next problem this presents is directed towards the KBase developers. With an exponential amount of testing combinations, testing the predictive biology algorithms is not feasible at this point. Without the ability to run countless simulations concurrently the developers are unable to sufficiently test the system. A means for gathering output and flagging error producing combinations is needed to aid the development of KBase.

Solution: Our proposed solution to the above problems is to use a browser automation tool to automate inputs from the user to send to KBase and lastly outputting the results back to the user. We utilize Selenium in our project, this tool allows us to use our custom made GUI to interact with KBase's interfaces. Our tool provides multiple means for the user to run experiments concurrently. The user interacts with our GUI/File reader and inputs their test(s). Selenium inputs the users test into the KBase interface and runs the tests with the desired inputs. After the execution of KBase's program, we return the results to the user and flag any errors that may have occurred. If the user entered multiple tests, then after the output of one we begin the next job in the queue. Sending jobs concurrently to KBase allows us to solve the efficiency problem the biologists face. We also solve the problem that the KBase developers face by collecting output from numerous tests and flagging any errors that occur. This will help them in testing the KBase application by giving them more code coverage overtime. The automation of the KBase program will in turn aid both the developers and the biologists who use it.

Intended Users: As discussed in the previous section, our application has two primary intended users. The first is biologists who need to run experiments on the KBase platform. It could take hundreds of simulated tests to get the exact results they seek. Our application will allow the biologists to run hundreds of experiments efficiently. The biologists will be able to tweak parameters to their liking by running many experiments before taking their experiments to physical laboratories. The second intended user of our application is the KBase developers. Our project can run hundreds of input combinations making it very useful for testing. The developers can test numerous input combinations and each test output will be appended to a file. The developers can then search for any failing input combinations and replicate these tests in their debugging. These are our two primary users, but in reality anyone with sufficient KBase credentials can use our application.

System Design: Below is a block diagram that illustrates our systems design. In this section we will provide a brief description of our system. KBase has a number of different inputs including: booleans, floats, and specific strings. We provide the user with the ability to interact with every input parameter offered by KBase. The user configures the parameters to interact with using our GUI, a file input, or the command line. These configurations are compiled as jobs. If more than one job is created we use a queue to store the future jobs. The first job is pulled from the queue and a runner takes the job. The runner takes the values from the parameters and hands them off to selenium. Selenium interacts with an instance of KBase and enters the parameters as specified by the user. The FBA test is run and the results are output back to the user in the form of a file. This process repeats until the job queue is empty and all the tests have been completed.



Inputs
Includes the application user-interface, an Excel spreadsheet, or some other data source.

Parameters may be literals (1, 2.5, 3), ranges (1-5), or randoms (random bool, take extremes of a float from 1-10, etc).

Jobs
Parameters from the user/source are processed into "job" objects, which include the application, inputs, and eventually outputs. They are queued by the runner and displayed to the user.

These may be stored in some other data-source for later aggregation/analysis.

Outputs
Individual jobs' results are visible, aggregate data from many rules may be visible, important events/findings from a series of runs may be elevated (expected performance, log error, etc).

We may display this in a tabular form, with dialogs, and it may be exported to a spreadsheet or 3rd party data source (RDS?).

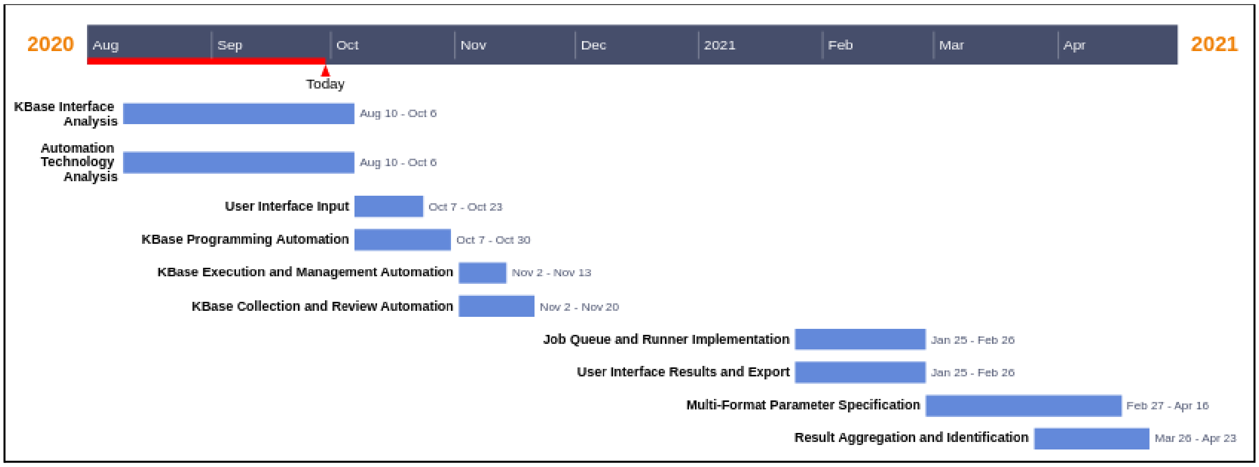
Execution

Once the jobs have been created and handed to the runner, it can make decisions about how to distribute them to instances of Selenium/KBase.

Runner
Manages job queue, distributes jobs to instances, handles notification to UI of completed jobs/data availability

Instance
A particular instance of KBase being interacted-with through a particular instance of Selenium/driver/browser.

Project Plan:



The project was scheduled to be developed over two semesters of a senior design course. The project was developed in an agile format, with each major feature estimated by the project plan above. Each major feature is outlined below.

- KBase Interface Analysis

Analysis of the KBase web application to better-understand the user's workflow and how the web pages are built. This will inform how we might interact with and automate the interface. Particular deliverables from this task may include specification of workflows for interaction with KBase and identification of particular DOM selectors or risks we could encounter as we automate.

- Automation Technology Analysis

Analysis of automation technologies like Selenium to understand which might best-fit our use-case. This will include considerations like the performance of the tool (can we parallel jobs?), support of the tool (does it accommodate modern browsers and devices?), and ease-of-use (will interaction with the framework require programming knowledge?). Deliverables should include identification of a best-fit automation technology and steps necessary to implement an MVP using the technology.

- User Interface Input Implementation

This project will involve the implementation of a basic Java GUI to accept parameter specification from our users with validation and potentially advanced range specification. This may be a simple Java Swing interface with inputs/form-controls curated for the particular KBase application we're targeting, which is a Flux Balance Analysis (FBA)

- KBase FBA Programming Automation

A Selenium routine must be developed to take the parameters specified through our Java Swing GUI and automation programming the KBase application (Flux Balance Analysis) with those parameters. Programming will involve a variety of challenges including accommodating timing of certain elements (dialogs with fade-outs, for instance) and interaction with a variety of primitive and rich inputs (checkbox vs. a multiple-selection from a set of media).

- KBase FBA Execution and Management Automation

Once programmed, we must automate execution and monitoring of a KBase Flux Balance Analysis application. This includes verifying the parameter programming is complete, interacting with the "Run" button, and monitoring the status updates to ensure processing is proceeding successfully and indicating to the collection step once processing completes.

- KBase FBA Collection and Review Automation

After processing completes successfully, we will need to collect results from the KBase Flux Balance Analysis by identifying and scraping particular elements from the webpage. These may include the objective value of the simulation and output logs from the job. These values should then be written to the output data structure of the job to be processed or visualized later.

- Job Queue and Runner Implementation

The user may enter parameters with many permutations, so to automate execution we will develop a job queue to hold those parameter sets. This queue will be asynchronously linked to the GUI and runners, and will handle delegating jobs to runners and reporting the outcome of a job's processing to the GUI. The runners are responsible for executing the Selenium automation for a given job.

- User Interface Results Display

Once a job has completed, we will need to report this back to the user. This may include a basic tabular view, as well as a status view for jobs which have been queued but may not have completed processing yet. This view may indicate aggregate values as well, with a later task.

- User Interface Export Functionality

Once a set of parameters has completed execution, the user should have the opportunity to export that data for further evaluation in a tool like Excel. A set of jobs results should be exportable to the CSV file format from some GUI interaction, like an export dialog.

- Multi-Format Parameter Specification

Parameters may be specified for the application in a variety of formats depending on the parameter's type. Explicit values like “true” or “1” maybe be specified, a set of explicit values may be specified like “1, 4, 10”, a range may be specified like “10-100”, or randomness may be applied either uniformly or with preference for extremes. THe permutations of these ranges/inputs will need to be processed into a discrete set of jobs and parameters which can then be queued and processed.

- Result Aggregation and Identification

Once a set of jobs and parameters have completed processing, their results can be aggregated and outliers may be identifier for the user. These could be visualized through the UI or exported to some other format, like a CSV. We might attempt to programmatically identify outliers in the data and indicate them to the user.

How the Design Evolved from SE 491:

Throughout this project, we have maintained close communication with our client and advisor. This has benefited us tremendously because all parties have had insight into progress and vision for the project. The design of our project has, in large part, remained unchanged from SE 491. Our team spent significant time analyzing our problem and designing a solution and architecture which would best-enable us to address the problem, so the overall architecture of our project is very similar between SE 491 and 492. As we've continued to implement features, there have been additions to the design from SE 491. For instance, we have moved away from graphical interfaces and towards more data-driven interfaces, such as reading and exporting to JSON files. Thanks to the design work from SE 491, our core architecture has supported this transition. Additionally, we have expanded our graphical interface to include greater input complexity. An example of this would be the inclusion of randomization for input parameters. Overall, our project's architecture remains largely unchanged, though we have expanded it with greater functionality.

Requirements:

Functional Requirements:

Our project has many functional requirements that we must satisfy for our client. The first requirement is to be able to configure selenium to interface with KBase apps. The whole purpose of our project is to automate KBase, so we need some sort of a browser automation tool to achieve this. We decided to use Selenium to interface with the KBase FBA. From this we can send data and values to KBase from our application and then return output to the user. The next requirement is that users can set parameters to automate using Selenium. This requirement is simply getting values from our program (from the user) and sending those values to Selenium to input into the KBase FBA. Next, the app should be able to sample the configuration space from set parameters. Similarly, this requirement states that after those parameters are sent to KBase we run those set variables in the KBase application. Our next requirement is to add functionality to collect output data from KBase. This will be useful for both biologists and KBase developers. If a certain input space causes an error they can easily check that from the output files. This allows our users to check their experiments output to duplicate them physically and allow developers to be able to trace input combinations that fail. An additional requirement we have is to allow users the ability to randomize inputs. This will allow the developers to run a multitude of configurations to test the code base. Our last requirement is to have all inputs from the KBase GUI available. In order for our program to function exactly like KBase's version we must offer all the same input parameters in our program.

Non-Functional Requirements:

Our project has a few non-functional requirements that revolve around the system health and overall experience of KBase. Firstly, we must ensure that no noticeable KBase traffic flow disruption occurs so that other KBase users do not experience performance issues. Our application has the potential to create thousands of job narratives running concurrently. This has the potential to disrupt the daily work flow of the Kbase system and this is something we want to avoid. Secondly, The app GUI should have a similar layout as KBase GUI so that navigation will appear seamless and not foreign to its users.These are the only non-functional requirements we have identified in our project.

Environmental Requirements:

Our project also has a few system environment requirements. For starters, All users need a valid and active KBase account to use KBase. The users of the system must have access to a KBase account to successfully run our application. This will help us keep our application protected from potential malicious users. Another requirement we have is, Users' devices must be able to support and run Selenium inorder to interface with KBase apps. The application is built on top of Selenium, so the users system must support selenium. Lastly, A network connection is required.

Standards Used:

Our project follows a few popular IEEE standards in order to ensure we follow industry standards in some key areas. The standards that we follow are listed below along with a short description of each. We applied each of these standards to maintain consistency in aspects of our development/testing processes. Another reason for following these standards is to help ensure our project is of the highest quality. Lastly, we followed these standards to gain some relevant experience that we can apply to our post academic careers.

1.*IEEE 12207-2017 - ISO/IEC International Standard - Information Technology - Software Life Cycle Processes*

This standard holds the purpose of defining a lifecycle standard for software projects from design, development, deployment, testing, and acquisition standpoints. The standard can be used as a base for software projects at any point during their lifecycle. It defines specific stages of the lifecycle that can adhere to the standard.

2. *IEEE 1028-2008 - IEEE Standard for Software Reviews and Audits*

This standard defines five different types of software reviews and audits that can be used in combination to increase the quality of software projects at various stages of their lifecycle. The standard serves to provide a systematic guideline of how to review projects, carry out reviews, and use the results of the reviews.

3. *IEEE 2430-2019 - IEEE Trial-Use Standard for Software Non-Functional Sizing Measurements*

This standard defines how to size nonfunctional requirements in software. This means how to determine time/cost estimates for requirements that are non-functional in nature, such as user interface design, latency, and technical tools used in the project. The goal is to enable users to be able to more effectively be able to determine estimates of non-functional requirements.

Engineering Constraints:

The biggest constraints that we had was that we were required to automate the inputs of the Kbase gui through a web browser. This prevented us from using any tools provided by Kbase such as a SDK or an API. This constraint is what led us to using selenium, so that we could interact with the user interface directly. This lead to another constraint, because selenium uses the html page of the site that we are trying to automate if that html page changes then it could cause our whole system to fail therefore we had to make sure that are implementation was flexible enough so that any changes to the Kbase web site could be easily accounted for and make the appropriate changes. Finally the last constraint that we had was that it must be possible for future teams and independent individuals to be able to make their own contributions to this project once the semester is done. To solve this we are hosting the project on git to make the transfer of the project easy and we are following object oriented principles so that additions to the project will be easy.

Operational Environment:

The end product is expected to be run and available constantly to any KBase users. The environment the software application runs in will be the user's own system. Our application will be downloaded by each KBase user on their own machines. Our application is purely software therefore the environment is limited to the user's system. The only hazard that our project could encounter is no internet connection for the user. Our application needs an internet connection to operate. This environmental hazard is out primarily out of our control as developers.

Security Concerns/Countermeasures:

Our project is unique in that it is simply augmenting an existing solution for users. Rather than producing a standalone service which is accessed by users, we've produced a solution which augments the user's own abilities on their behalf. As a result of this configuration, we've identified one particular security concern for our project. In order to perform actions on our user's behalf, we require they provide their credentials as part of an authentication flow in our application. These credentials are simply used for Globus authentication prior to automation of a narrative within KBase itself. This is a security concern because if our application was somehow compromised (e.g. the distribution pipeline), the attacker would be able to gain access to our users' passwords. To mitigate this, we have provisioned a specific user account in Globus for automation and testing, and have additionally moved password configuration to a flat file which the user owns and can manage.

Implementation Details:

Our project was designed early-on with a core architecture which allows us to extend the system easily. This core architecture decouples automation of the KBase narrative from particular parameter sets, their origin, or the export format. In summary, we collect parameters from the user either via a graphical user interface or a file which is imported through the interface. Once we've loaded the parameters, we will apply any randomization specified by the user and create a Job for that particular parameter set. Once a Job is instantiated, that job is queued with a JobManager whose responsibility is to assign jobs to runners. A runner handles automation of KBase narratives by being assigned Jobs, programming KBase's parameters from that Job, executing the simulation, and scraping the results of that simulation and storing them within the Job. Once complete, a job can be exported to the file system where the user has the opportunity to perform further analysis. The most difficult aspect of this architecture is the runner's automation. We have to navigate through a series of complex flows to authenticate the user, open the narrative, reset a narrative application, program the application, execute the application, and collect results from the application. Each of those steps may involve a series of actions which are reactive to behaviors from KBase itself. Thanks to our architecture, however, we have been able to develop the automation code in-parallel and extend the system to accommodate new features and enhancements.

Testing:

Unit Testing:

The first type of testing that our team performed was unit tests. We perform unit tests on our data inputs, selenium runners and data outputs. For instance, once a job has had its parameters set, it's handed off to a selenium runner for execution. Our Selenium runner will move through several phases as it configures the KBase interface, executes the job, and collects results from the Flux Balance Analysis's output. These individual steps may be tested in isolation through unit tests to validate that the correct actions are being performed in KBase through the Selenium API, and that the output collected from the application's DOM is being correctly instantiated into an output data structure for later processing. In order to continue to run all of the unit tests we created a continuous integration pipeline that would automatically run all of the tests that we have written. We did this by using the maven test command that will run all of the test files that are in the project upon a build.

Interface Testing:

Another type of testing that our group did was interface testing. Currently, our interface testing is all done manually and is not extensive. Our interface changes periodically and is minimalistic so our team has decided that interface testing would not be as beneficial as other types of testing. In the future, some broad interface testing that is automated by Selenium. Our most frequent type of testing is our acceptance testing.

Acceptance Testing:

Our accepting testing is mostly manual performance tests that are performed by our team while working on the project. This means that each of the team members are responsible for manually testing the code that they plan to implement to ensure that it works in accordance with project requirements. In addition, the people who review the pull request of any given team member should make sure that the code works either by asking the team member who created the pull request or by manual testing that the code works themselves. Lastly we demo the project every two weeks to our client so that they can assess the project and determine if it meets their requirements.

Testing Results:

The results of our implemented testing strategy allowed us to catch most of the errors that were created by our software before making it into the main branch. This provided the user with a better experience and exceeded the expectations of the client.

References

IEEE SA - The IEEE Standards Association - Home, IEEE Standards Association, standards.ieee.org/.

Arkin AP, Cottingham RW, Henry CS, Harris NL, Stevens RL, Maslov S, et al. KBase: The United States Department of Energy Systems Biology Knowledgebase. Nature Biotechnology. 2018;36: 566. doi: 10.1038/nbt.4163

Appendices

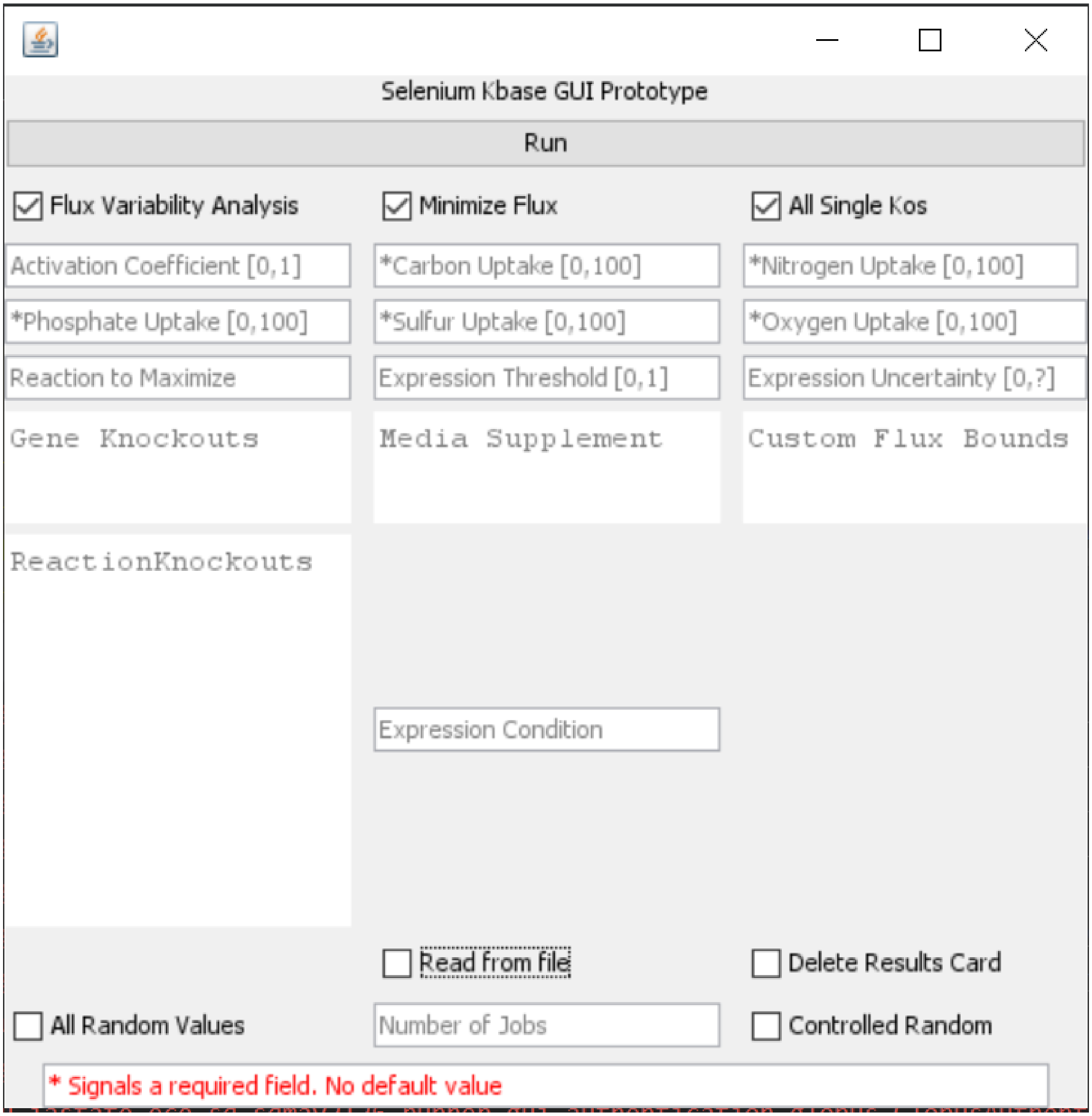
Appendix I - Operation Manual

The application is highly configurable and may be run in different modes as desired. The application is bundled into releases in the form of JAR files. These files may be run, according to the user's operating system. Typically this means double-clicking the JAR file.

FBA Application jobs can be configured through a GUI, using a configuration file or by manually setting the variables, or through the command line via a configuration file. The configuration files can be used to queue multiple jobs from a single file.

- Running the application GUI

To run the application from the GUI, double click on the bundled JAR file. This will open a new window. This window will allow the user to configure the application through the use of a configuration file, or manually setting values to be programmed into KBase.



Selenium Kbase GUI Prototype

Run

☑ Flux Variability Analysis ☑ Minimize Flux ☑ All Single Kos

Activation Coefficient [0,1]

*Carbon Uptake [0,100]

*Nitrogen Uptake [0,100]

*Phosphate Uptake [0,100]

*Sulfur Uptake [0,100]

*Oxygen Uptake [0,100]

Reaction to Maximize

Expression Threshold [0,1]

Expression Uncertainty [0,?]

Gene Knockouts

Media Supplement

Custom Flux Bounds

ReactionKnockouts

Expression Condition

□ Read from file □ Delete Results Card

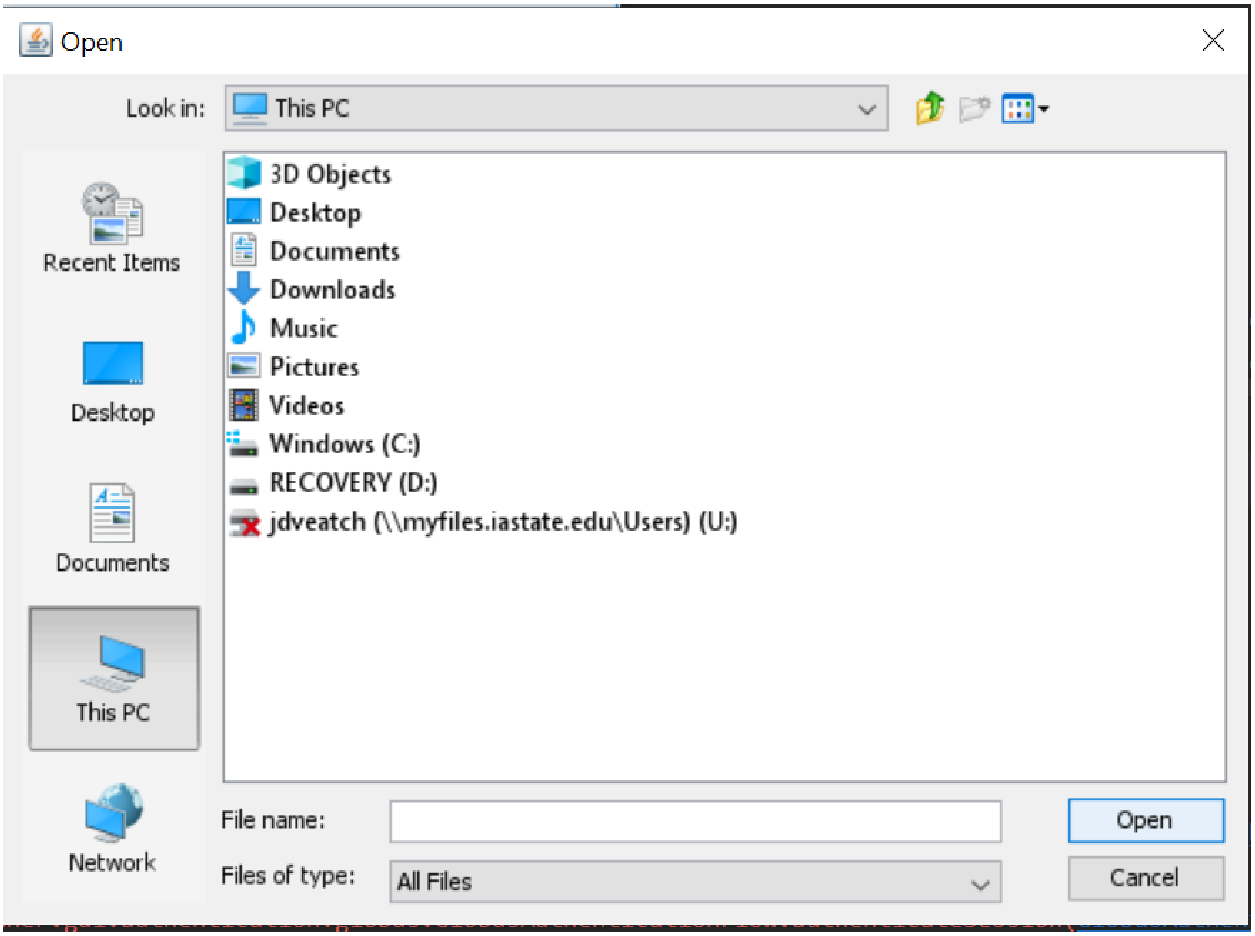
□ All Random Values

Number of Jobs

□ Controlled Random

* Signals a required field. No default value

Optionally, “Read from file” may be checked to allow the user to select a file on their machine to program variables to KBase, instead of programming the input boxes manually.



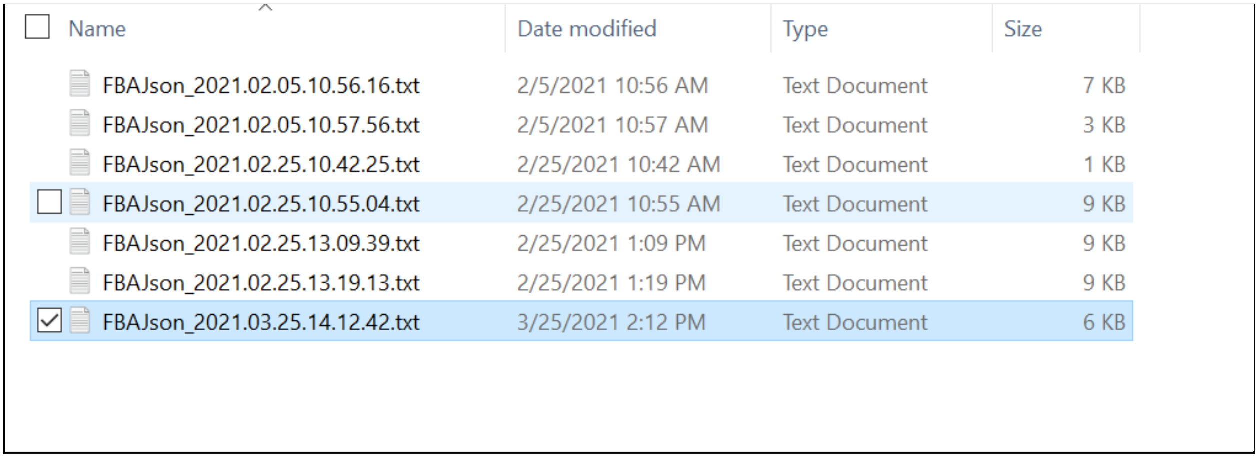
Look in: This PC

3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
Windows (C:)
RECOVERY (D:)
jdveatch (\\myfiles.iastate.edu\Users) (U:)

File name:

Files of type: All Files

The application will run according to the configuration given to the job, and output the results from the application to a file.



Name	Date modified	Type	Size
FBAJson_2021.02.05.10.56.16.txt	2/5/2021 10:56 AM	Text Document	7 KB
FBAJson_2021.02.05.10.57.56.txt	2/5/2021 10:57 AM	Text Document	3 KB
FBAJson_2021.02.25.10.42.25.txt	2/25/2021 10:42 AM	Text Document	1 KB
□ FBAJson_2021.02.25.10.55.04.txt	2/25/2021 10:55 AM	Text Document	9 KB
FBAJson_2021.02.25.13.09.39.txt	2/25/2021 1:09 PM	Text Document	9 KB
FBAJson_2021.02.25.13.19.13.txt	2/25/2021 1:19 PM	Text Document	9 KB
☑ FBAJson_2021.03.25.14.12.42.txt	3/25/2021 2:12 PM	Text Document	6 KB

- Running the application from the command line

Alternatively, the application may be run from the command line. This gives the user greater configurability, opening up the option to run Selenium drivers from Docker containers. The benefit of running the application from the command line is that it can be scripted, allowing tests to be run in an automated environment. In addition, utilizing a Docker container allows the tests to be run without being coupled to the browser and version on the user's machine. Docker containers containing specific browser versions can be configured for use via the command line.

Pull the docker container that holds the desired browser and version.

```
docker pull selenium/standalone-chrome:88.0
```

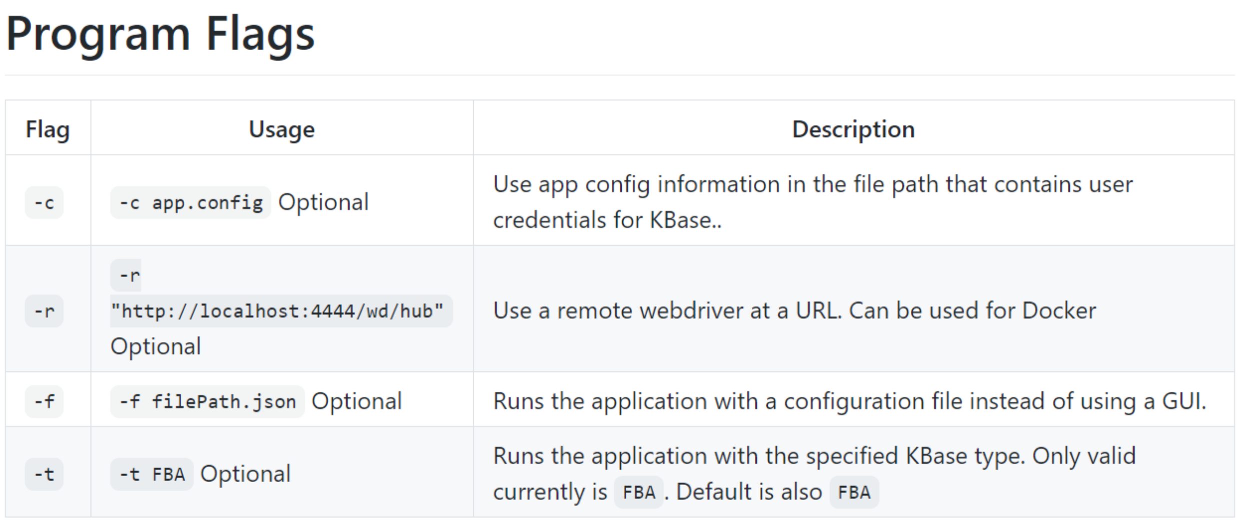
See SeleniumHQ/docker-selenium for available browser images.

Start the docker container hosting the webdriver

```
docker run -d -p 4444:4444 --shm-size 2g selenium/standalone-chrome:88.0
```

Run the application pointing at the docker container as a remote web driver.

```
java -jar senior-design-sdmay21-26.jar -f "path/to/inputFile" -c
"path/to/configFile" -r "http://localhost:4444/wd/hub"
```



Program Flags

Flag	Usage	Description
`-c`	`-c app.config` Optional	Use app config information in the file path that contains user credentials for KBase..
`-r`	`-r "http://localhost:4444/wd/hub"` Optional	Use a remote webdriver at a URL. Can be used for Docker
`-f`	`-f filePath.json` Optional	Runs the application with a configuration file instead of using a GUI.
`-t`	`-t FBA` Optional	Runs the application with the specified KBase type. Only valid currently is `FBA`. Default is also `FBA`

Appendix II - Alternative Versions of the Design

Our team largely benefited from the fact that our advisor was also our client. We met weekly to hammer out any design details that we may not have interpreted fully. We also benefited from the scope and idea of our project being largely fleshed out before we had to undertake it. Our advisor suggested that we use selenium to operate on top of the KBase platform. We were given a bit of freedom with the tool we ended up using, but once we saw how easy it was to interact with KBase using Selenium it was a clear cut choice. Thankfully, for the team we decided on a job/runner structure for running input values from our application ran through Selenium to the KBase interface and then eventually back again. We never wavered from this design decision simply because it worked and it was efficient. Our application was simple in theory. Create a program that interfaces with an external GUI/File reader and send that input data to Selenium which interacts with the KBase interface. Then get the return data and highlight values of importance. The job/runner structure was perfect for this and allowed us to easily implement concurrency with a job queue. Due to this, we never had to change our initial design; we simply had to change some features where necessary. Initially, we were unsure what direction the project would take design wise, but we benefited a lot from our advisor doubling as our client. Running an agile development methodology we got to demo our progress every two weeks causing constant and consistent feedback from our client/advisor. I believe this also played into us never making any large design changes. Another reason our design didn’t change was due to our project being well defined and the core specifications stayed concrete. At its core the project did not change at all, we needed to craft a third party tool to automatically interface with KBase to run tests concurrently while gathering and flagging appropriate output. We only added onto this concept and added additional helpful features such as: command line interactions, reading from files, completely randomized samplings, manually configured base samplings, and self garbage collection to name a few. The implementation of some of these features may have changed due to a better understanding of the benefits. For example, we wanted a way to automatically perform base samplings. The feature was implemented and demonstrated, but was not a hundred percent representative of base sampling. In the end we had to trade the feature for a manual base sampling tool, but this is a good example of a time one of our features changed. To conclude, our design never changed from the initial version. Thanks to our advisor/clients' good direction and well fleshed out idea. Our weekly meetings and biweekly demonstrations gave us plenty of feedback and allowed us to continue to pursue the job/runner design style. Our clients specifications for the project were immutable and gave us a good idea of our objective. We benefitted largely from the project's overall specifications not changing throughout the year. New feature ideas were presented by our advisor/client, but our initial design allowed these to be seamlessly built on top of existing code without affecting our systems design. We did not consider any other designs that are significantly different than our current/initial design. Our advisor had the idea, we chose a design style that fit that narrative, and we executed that plan while maintaining and making additions as new features were presented. We did not make any design decisions until we fully understood the project and our requirements. Our design met all the requirements and specifications for the project resulting in not needing to change from our initial design. As a result of all the points previously discussed, our initial design proved to be our best design. Resulting in it becoming our final design.

Appendix III - Other Considerations

Our project being built on top of an existing interface presented some slight risks. The KBase interface is a very stable one that has not undergone many changes since its inception. The interface started on version 1 and per our advisor this interface has not changed in a long time. This gave us some confidence that our selenium driver would be fine to be hardcoded. Little did we know that the day before our advisor sprint demo KBase rolled out a version two of the interface changing most of the interface. Our selenium was no longer running correctly as it was not grabbing the correct web elements. This shut down our development for about ten days as we tried to get back online. Our fix made everything a lot more modularized this way if the KBase interface changes again we only need to make slight changes to our end to successfully transition. From this we learned that relying on interfaces or anything in general to not change is not a dependable strategy. We must make a plan and craft flexible code in order to better endure these changes. If our client wants a specific software that is built on top of something we have no control over then this would be the way to go. Modularize everything you can to make changes smoother.

The entire team learned a lot throughout this year-long endeavor. For starters, we kept up with an agile two week sprint cycle. This taught the team how to work together using one of the most popular development methodologies. This gave us all good experience to take into the workforce. This project also gave everyone a chance to learn new API's and work on facets that they wanted to. The core implementation was done in the first semester giving us a full semester to add additional features. The team got to explore facets that interested them while adding features to the project. Whether the members wanted to work on UI, dockerizing, working with Selenium there were plenty of opportunities to explore paths and improve specific skill sets. The team also gained valuable experience working with a real client and speaking with real users. We had an hour long meeting with KBase developers/users and highlighted the benefits of our application to them. This meeting was very beneficial to the team as we spoke to and got feedback from other users other than our main client. While we learned many things this past year, I think the most important thing was how to successfully work within a small team. We had to work closely together to ensure we were prepared for our biweekly sprints. This experience will be very useful as we all move forward professionally.