THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

A Non-Invasive Productivity Tracker for Programmers

Supervising Professor
Dr. Sonia HAIDUC

Author
CONNOR J. CHRISTIAN

Committee Member
Dr. Xin YUAN

Committee Member
Dr. Shayok CHAKRABORTY

March 27, 2019

A Non-Invasive Productivity Tracker for Programmers

Connor J. Christian
Department of Computer Science
Florida State University
Tallahassee, Florida 32304
Email: cochrist@cs.fsu.edu

***Abstract*—Time is a valuable resource that should not be wasted from being distracted during work hours. As a programmer, our work is almost entirely done on a computer and online. Distractions are everywhere from the applications installed on your machine to the websites you visit in your browser. When working remotely or freelance, without the team settings or the restrictions from a company computer, it is easy to fall down the rabbit hole when research leads to distracting websites.**

This project, Monty, offers a lightweight full-stack solution that prioritizes simplicity and non-invasive data gathering to achieve results comparable to many open-source and proprietary projects. Currently, the source code requires approximately 15MB of hard drive space and the SQLite database is currently 9MB in size containing 22,671 logged minutes and 270 ranked applications and websites. When running idle, the project utilizes less than 1% CPU time and approximately 50MB of memory. When computing a month's worth of data, CPU usage is around 12% and approximately 73MB of memory. Non-invasiveness is achieved through limited data gathering and not distributing any logged information.

Current limitations of the project are the use of a Linux Operating System utilizing Upstart and the Firefox browser. The project was developed on Ubuntu 16.04 OS using Firefox Quantum.

***Index Terms*—Software engineering, non-invasive, lightweight, full-stack, productivity tracker**

1. Introduction

Properly utilizing time working on the computer is vital for a software developer as the majority of our work is done online. Avoiding distractions is often harder than it sounds like. Distractions come in many forms: posts and threads in discussion boards, notifications, installed applications, even emails can unknowingly be costing you time. An obvious approach is to not install any distracting programs, a blacklist of websites, or work solely on a designated work machine to keep you from these common distractions. However, this is not always ideal if you work remotely or do not have a designated work machine and you need to use your computer for more than just work. This project, Monty, aims to be a lightweight and non-invasive solution that gives you the freedom to use your computer as you like while keeping you informed on how you are utilizing your time.

When designing Monty, three things were kept in mind:

- simplicity
- lightweight
- non-invasiveness

Simplicity is showcased in the modular design of the source code, allowing for future modifications to be easy to implement; as well as an intuitive web user interface (UI). The user should be presented with an easy to digest analysis of the logged data without having to navigate through many button presses or pages. For being lightweight, only lightweight tools were used, minimal data is collected to achieve informative graphs, and multiprocessing was utilized to keep performance fast. Lastly, achieving a non-invasive solution requires retrieving data on how the user is utilizing their time without storing the details of the content. For example, it is important to know that the user is watching *a video* but not *what* they are watching. As it turns out, each factor influences the others. A simplistic design leads to less code and a smaller deliverable. Non-invasive data gathering reduces the amount of storage space necessary as well as reducing the computation time.

This project aims to develop a productivity tool that's target audience are programmers. Many developers are nervous about how their data is being used online, therefore Monty keeps all data local. By doing so not only is the collected data safe, but it is also readily available for further experimentation. For example, the SQLite database can be queried to have the output feed to machine learning algorithms or fed to other applications. As Monty is targeted towards programmers, it is developed on and designed for the Ubuntu 16 operating system and utilizing the Firefox browser. Future versions will expand on the scope of operating systems and browsers supported as well as increase functionality.

The rest of this paper is organized as the follows. Section 2 provides information on related work. Section 3 goes in depth on the approach I took to developing Monty. Section 4 discusses testing. How to use Monty is described in Section 5. Section 6 speaks of overall performance of the project.

Section 7 discusses threats to validity. Section 8 speaks of my conclusions and Section 9 talks of future work.

2. Related Works

While there exist many productivity tracking software products, often they are not free, intended for businesses or teams, and store personal information on remote servers. A very popular product on the market is ”Saent” [1]. Saent’s offers a button which acts as a timer for starting and stopping ”sessions”. During these sessions, Saent actively blocks distractions, such as notifications, banned websites, and discourages multitasking to promote productivity. Benefits of this product come in the form of ”smart breaks”, based on the Pomodoro Technique, and the ability to save data locally [2]. TimeTracker is another proprietary product targeted towards teams and developing spreadsheets for reports at a faster rate [3]. TimeTracker does support a free version of the software; however, it is limited to only one week’s worth of data, whereas the paid versions come with more features and at different price points. Similar to TimeTracker is an open-source project called Fluxday, designed to also track the productivity of teams but for start-ups [4]. Fluxday sports a simpler user interface than other products having an uncluttered design. On a more personal scope, like Saent, RescueTime offers analytics about the user’s daily computer usage with an intuitive design [5]. Unlike Saent, RescueTime does not have the option to store data entirely locally, but it does boast a free version that can store data up to a history of three months. Tools that are open-source and commonly used are Ubuntu’s Zeitgeist and related Activity Log Manager [6] [7]. Zeitgeist is installed by default in Ubuntu distributions, running in the background logging user’s activity to be used by other applications. The data that Zeitgeist logs are very invasive, logging files opened and conversations had with people. Although the logged data is kept internally on the machine, it is meant to be used by other applications installed. Activity Log Manager is a separate program that acts as a setting user interface to Zeitgeist, allowing the user to set restrictions to what can and cannot be logged.

Monty aims to find a common ground between the aforementioned works. I propose a non-invasive solution that only tracks running processes owned by the user and the base URLs of websites visited, unlike Zeitgeist. The data collected will only be stored locally in a SQLite database unlike most of the products mentioned; with exception to Saent if most users utilize this feature. Monty will be created as an open-source productivity tracker that offers an intuitive user interface and informative graphs allowing the user easy access to the code base for modifications or importing the data into other works. Monty is meant to showcase how the user is utilizing their time such that they can fix potential problem areas to better themselves. Monty will never block computer functionality from the user, unlike Saent, giving the user full control over what they do.

3. Method

To develop a lightweight productivity tracker, only a small data set needs to be actively logged. The most important pieces of information is *what* is running and *when* it is running. In this case, that information is the name of the application (or website visited) and the time-stamp at which the process was detected. This is the very least amount of data that can be logged that has meaning; however, many running processes are not directly related to the user, e.g. system processes and service daemons. By only logging processes owned by the user, and not those owned by root or other users, we can eliminate a vast majority of the noise. After filtering out all the processes not owned by the user, there are often still a large number of processes owned by the user but are still OS related or daemon process used to keep programs alive and running. Examples of these are *check-new-release, compiz, desktop-launch, ibus*, etc. While these are owned by the user, they are not informative as they are nearly all prompted by the OS and hidden. To avoid being buried in commonly running system processes, I propose a ranking system to filter the logged data. There are five possible ranks enumerated as follows:

Unknown (0)
: Newly logged application that has yet to be classified.

System (1)
: Operating system processes/daemons/etc not indicative of what you are running.

Distracting (2)
: Applications that do not improve your levels of productivity.

Neutral (3)
: Applications that can equally be used for productive work or as a distraction.

Productive (4)
: Applications that help you work or you utilize to be productive.

It should be noted that all detected process are initially classified as ”Unknown” until manually classified otherwise. To accommodate the users from having to manually sort hundreds of processes from initial setup, the repository of the project contains a pre-classified database intended to be used. All items found in the database can be reclassified depending on the user’s interpretation of the effect an application has on themselves. For example, I may find that listening to music as I work encourages my productivity; however, the user may disagree and finds music distracts from productivity. If that is the case, the user may simply reclassify the application under distracting. If there is no database found or it is removed, the project will create an empty database and fill it as new processes are discovered classifying them as ”Unknown”.

Lastly, it is possible for the user to have different levels of activity when working. In other words, is the user *actively* using the computer to work, is the user *passively* working, or has the user walked away or locked the computer. This ultimately leads to three possible working states:

High
: High activity is detected meaning that the user is actively typing or moving the mouse.

Low
: Low activity meaning there is no input from keyboard or mouse and the computer is unlocked.

Away
: The computer's lock-screen is activated.

To account for websites a simple boolean flag indicating whether or not the name that was logged corresponds to a website or an installed program will suffice. With the above-mentioned ranks, activity levels, and website flag, we can intelligently log processes encountered by name and time-stamp detected. When determining the frequency of how often data should be collected, one minute was selected as it was a small enough unit to accumulate accuracy over long periods of time, while simultaneously being large enough as to not bloat the database with unneeded duplicated logs. Intuitively, timeframes less than a minute is not long enough to realistically work on anything and although five-minute increments were considered, one minute gave more accuracy as being able to detect when a user has walked away briefly.

3.1. Tools

When developing the software to gather and process data efficiently the following tools and frameworks were utilized:

- Python3
- Flask
- setproctitle
- SQLite
- Upstart
- top
- Firefox
- lz4json [8]
- Bootstrap
- AngularJS
- ChartJS
- DataTables

Python3 was chosen as the language to write productivity tracker due to its library ecosystem and ease of portability. Utilizing python's standard library features, very little was needed to be installed via third-party (Flask, Flask-API, setproctitle). Flask is a lightweight python micro-framework for web development. SQLite is chosen as the database used to store the logged data and rankings as it is a lightweight disk-based database. It is also shipped with the installation of python itself.

Upstart is Ubuntu's system and user init script daemon allowing scripts to be stopped and started via events. Upstart is utilized for stopping and starting Monty on user login, logout, and shutdown. Top is utilized for detecting running system processes in real-time, allowing for quick parsing, taking advantage of command-line arguments to remove excess information unneeded for tracking purposes.

Firefox is shipped with all Linux operating systems, therefore my focus for logging surfing habits started here. Firefox stores backup data of user sessions in an lz4 compressed JSON object in the user's home directory. Using the lz4json tool to decompress and read the session object, I have access to the web addresses actively opened in the browser. Lastly, Bootstrap, AngularJS, ChartJS, and DataTables are JavaScript and CSS frameworks that allow for mobile-friendly web development and dynamic graphs to display datasets, respectively.

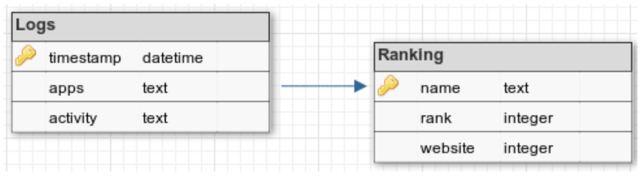


Figure 1. SQLite schema used to relate two tables. Illustrating that Logs contains a list of application names which are used as keys to entries of Ranking.

3.2. Database

Utilizing a SQLite database for storing the mined information allows for a light impact on storage space while maintaining the efficiency of a relational database all without installation headaches of server management on the user's end, as SQLite is file-based. This allows for cross compatibility and easy manipulation of the stored data.

The schema of the database needs to easily relate the identified information as shown in figure 1. A list of all detected processes and websites (represented as a comma-separated string) are stored in the ”Logs” table, identified by the UNIX time-stamp, at which it was generated, as the primary key. For each individual name logged, there is also a rank associated with it, as well as a flag indicating whether or not the name indicates a website. This information is stored in the ”Ranking” table where the unique name is the primary key to an entry storing the associated rank and flag. The relation between Logs and Ranking is many-to-many as Logs contains a string of many application names, each of which is assigned an entry in Rankings; so one log points to many Rankings. Likewise, one rank is assigned to a unique application name, which can be found in many logs.

Although this is an intuitively simple database design, it is all that is necessary to maintain the relationships between the mined data.

3.3. Back-end

With the database designed, a tool used to acquire the information and insert into the database is needed. Constraints of the program are that it should work behind the scenes without the need for user input, accumulate data over one-minute increments, and gracefully save its state when the user logs off or sends a signal to stop. The structure of the source code of the back-end is as follows:

__main__.py
: The main driver for the back-end of the productivity tracker.

src/activity.py
: A module created to handle determining levels of computer activity.

src/firefox.py
: A module created to handle retrieving and parsing Firefox session data.

src/log.py
: A module created for opening a connection to the database as well as inserting and updating entries into.

In __main__, the first things that are done are to rename the title of the process to "Monty", identify the username of the computer user, and creating a signal handler class called "KillHandler". The reasoning behind renaming the process is to be able to differentiate processes that are run by the user and those that are spawned from "Monty"; since Monty will be running continuously running in the background as a process it should not be included in the generated graphs. Identifying the name of the user is important for filtering out unwanted processes. This is done by simply accessing the environment variable $USER. The KillHandler class simply listens for the SIGINT and SIGTERM, setting a boolean attribute "killed" to true, indicating that a signal has been caught, allowing Monty to gracefully exit. SIGKILL cannot be handled, so it is not possible to gracefully exit from; however, this is not an issue as SIGKILL is not a very common occurrence and the potential loss of data is in units of minutes. Next, utilizing python's multiprocessing module an Event, Queue, and Process class are all instantiated such that data processing can occur in parallel. Since child processes inherit the signal handler of the parent, SIGINT and SIGTERM are *temporarily* ignored. When defining Process we assign it a function *process_loop*, which will be discussed below, and arguments containing an instance of Event, *exit_event*, and an instance of Queue, *work_queue*. After the defined Process is started, the signals SIGINT and SIGTERM are restored, a delay of sixty seconds is defined and a loop is entered that terminates on detection of a signal.

Within the main loop, three things are accomplished within a try-except block:

1) Call the top command and retrieve its output.
2) Queue a job containing the current time-stamp and the previously retrieved output.
3) Check if a signal has been caught.

First, in a try, function *call_top()* is called. The purpose of this function is to construct the appropriate top command to retrieve the needed data and return it. Top command-line arguments used are

-b : Batch-mode
: Prints output ideally for piping output to files or other programs.

-i : Idle-process
: Toggles idle-processes, in this case removing them from output.

-d : Delay-time
: Delays top from printing until one minute has passed; containing one minutes worth of logged data.

-u : User-filter-mode
: Only includes processes owned by the user.

-n : Number-of-iterations
: Used to auto-stop top. Monty sets n to 2 so after top refreshes from the minute delay, top exits.

-w : Output-width-override
: Forces top to set width to a specified amount, in this case 512 columns, which is the maximum possible by top.

We then strip the returned data from top using grep matching against all lines that include the username. This gets rid of any superfluous system data that is often at the beginning of top's output. Then the sub-process is renamed as to not be included in the generated graphs later using "bash -c" and "exec -a". The constructed command is the following:

```
bash -c "exec -a Monty_Top top -bid\
 60 -u <username> -n 2 -w 512"
```

Where

bash -c
: Allows for commands to be passed as an argument string.

exec -a
: Names the process.

Since the function call is within a try block, if an interrupt were to occur during the execution of top, it will be caught with:

```
subprocess.CalledProcessError
```

Although the exception block itself is empty, we desire this as Monty should not do anything in this case but skip directly to clean up. This happens because adding jobs to the work queue is found in the "else" block of the try, meaning job creation only occurs if there was no previous exception. Therefore, the next code that is reached is the "finally" block that is always executed in a try block.

If there are no exceptions caught during the execution of top, then we put the generated output from top and the time-stamp of the previous minute onto the work queue. This then gets removed from the queue and the data is processed in parallel.

Lastly, there is a "finally" block that is used for cleanup purposes. Here an instantiating of the KillHandler class is checked for any caught signals. If a signal has been caught, the exit_event is set and the process running in parallel finish up. Next, Monty waits for the process to finish clean up, then breaks out of the main loop, and closes the connection to the database that was opened when importing the src/log.py module

Within the process running in parallel there is a loop that runs until the exit_event is set; forcing the process alive in times when there is nothing in the work_queue. If there is data in the queue, it is popped off and the arguments are passed into the *parse_data()* function to be parsed and stored in the database. After popping off the queue, the loop sleeps for one second as to not consume 100% CPU power. In the event of clean up, the initial loop exits (no longer being forced alive), if there is any remaining data in the queue it is immediately processed and allowed to gracefully exit.

In *parse_data()*, the output string generated from top is transformed into a set of unique process names by splitting the string on the newline character, creating one line per process, then using regular expressions to isolate the name. If "ibus" is one of the names detected then input has been received via mouse or keyboard indicating that the activity level should be set to "High". Once all lines from top have been parsed, the activity levels are determined by calling the *check_activity()* function imported from the activity module. This is done in case high activity was not detected, then the computer may have been locked, or the user could have not been actively using the computer. To acquire recent web history the *query()* function, imported from the firefox module, is called. The output is then added to the set of all processes. Lastly, *insert_logs()* and *insert_ranking()*, from the log module are called, inserting the acquired data into the database.

The activity module works by maintaining a variable that holds the last registered activity level (High, Low, and Away) and getter/setter functions for updating said variable. Every time *parse_data()* calls *check_activity()*, we need to determine whether or not the screen is locked. This is because regardless of input being detected, if the screen is locked then the user is not actively working on the computer; therefore the activity level is set to "AWAY". If the screen is unlocked, then either there was activity detected, via "ibus" found in process logs, or not. If so then the variable is set to "HIGH", otherwise, "LOW". It is possible to leave the computer unlocked yet walk away from the computer, or simply never lock the computer, in this case, there is no difference between "AWAY" vs "LOW" activity. The levels are meant to help the user deduce what they were doing in that time based on their habits as the only important level is "HIGH" activity or not.

The firefox module navigates to the "recovery.jsonlz4" session backup file located within the user's hidden ".mozilla" directory, opens the file using the lz4jsoncat tool and parses through the resulting JSON object. Within the JSON object, a set of unique URLs visited in the last minute is constructed. This is done by checking all windows that are open and iterating over every tab, checking the logged time-stamp under the "lastAccessed" attribute. For each valid URL found, only the base URL is stored. In other words, sub-directories and ports are not logged as this is potentially sensitive information. To ensure we maintain our non-invasive stance for productivity tracking, the URLs are trimmed.

Lastly, the log module opens a connection to the SQLite database if it is found, if not it creates the database and any missing tables according to the schema described in Figure 1. It also provides three helper functions to be used in *parse_data()*:

insert_logs()
: Inserts into the Logs table a new entry containing the current time-stamp (accounting for local timezone), the set of all detected processes/websites (as comma-separated string), and the current activity level.

insert_ranking()
: Inserts into the Ranking table a *newly detected* process/website, assigning it the default rank 0 for "Unknown", and set the website flag.

close_connection()
: Closes connection initially opened.

These components form the back-end of the application that collects data on the user's computer usage in minute increments. All data is stored in the SQLite database described in the previous section. Utilizing command-line tools and multiprocessing allows for efficient data processing that is not resource intensive.

Monty contains a second independent application that runs the web UI. The files responsible for hosting the UI are:

__init__.py
: The main driver for the Flask web UI that dynamically servers the html files.

db.py
: A module that handles opening and closing the database for Flask when the UI is activated and closed.

In __init__, located in the "web" sub directory, is the code responsible for configuring a Flask web page as well as the functions used to deliver content to create dynamic web pages. The aspects within this file that is personally related to Monty, and not common to all Flask applications, are the functions defined nested within the *create_app()* function. Most notably are the *day()*, *week()*, and *month()* functions which prepare data over a days/weeks/months time frame to be displayed in graphs, and the *crud()* and *export()* functions which both showcase the contents of the database. It should be noted that *crud()* does so in the form of an interactive table for reclassifying and removing entries in the Ranking table, whereas the *export()* function returns contents of the database as a JSON object. The *getRanks()* and *updateRanks()* are helper functions for *crud()* of which the former retrieves the contents of the Ranking table to display in the UI, and the latter updates the Ranking table according to input from the UI. The remaining functions, *dayStats()*, *getData()*, *classifyData()*, and *colorWeek()* are refactored functions of shared functionality between the day/week/month methods.

The statistics needed for graphs produced by *day()* are retrieved from *getData()* which returns the following datasets over the time-frame of an hour: a list of all programs detected, a list of all programs detected during high/low/away activity, the number of minutes logged during high/low/away activity. An hourly breakdown where each hour is labeled as having high/low/away activity levels, based on which appears the majority of the time, is also returned. This is used to quickly signify, as a whole, which hours was the user actively at the computer or not. The *classifyData()* further organizes the previously acquired data into tuples containing the data's name, rank, and frequency of appearance during the hour, and if they appeared during high/low/away activity time. These tuples are grouped by names that belong to program, websites, or those of which that have an "Unknown" rank. This allows for a quick break down of how a day has been spent, as well as an in-depth breakdown for a particular hour containing graphs and a log of programs and websites accessed.

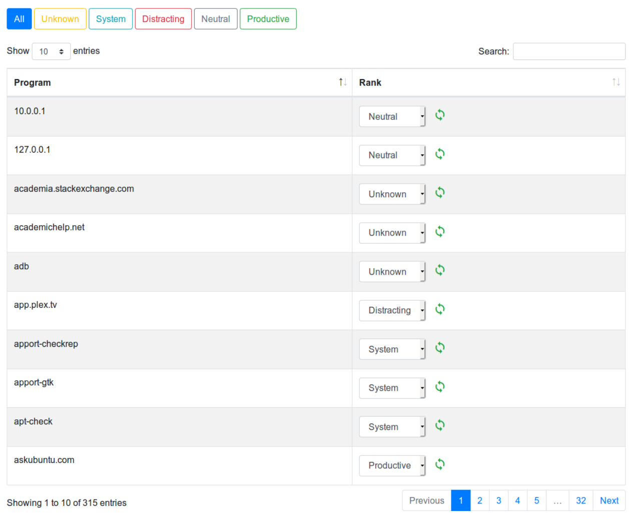
To construct the data for a week's view, *dayStats()* was utilized to collect the hourly activity and productivity levels (number of hours productive/distracted/neutral hours) as well as the sum of each logged program and website's frequency. By running this function for each day a given week, a daily breakdown is constructed in units of hours. The running total of frequencies is used to construct the top five programs and websites used for the given week. The *colorWeek()* returns the daily activity level depending on which level appears the most often over the course of the day, which in turn is used to assign the day a color representation.

The month's data representation is comprised of the data returned from *dayStats()*, however, instead of iterating over a given week, it is down over a given month. For this dataset, the number of productive hours vs distracted is desired. This gives an intuitive breakdown over a month's worth of data, without overwhelming the user.

3.4. Front-end

The goal of the front end is to deliver informative graphs representing the data that has been collected in an intuitive manner. The constraints were to have a simple design that allowed the user to access the tracked data in as few clicks as possible while still easily accessing all of the data available, all while not sending any collected data remotely. Flask serves the dynamic webpages on localhost via port 5000 (localhost:5000), this ensures that only you can view your data on your machine. With that in mind, the features of the website would contain only a few buttons to navigate the site, those are the "Day", "Week", "Month", "Update", and "Export" buttons which redirect to the appropriate URL. There is also a day picker which allows for quick navigation to a particular day.

The export page is intended to function as an API call to the database, returning a JSON object of the contents of the database. This data could be used as a simple means of input to avoid using SQL calls if a user wanted to do so. The update page, as shown in Figure 2, similarly gives access to the contents of the database in the form of a dynamic table. This allows the user to easily view paginated entries into the database with the options to reclassify or delete specific entries. Since all graphs are generated in real-time, any updates to the database entries have immediate effects.

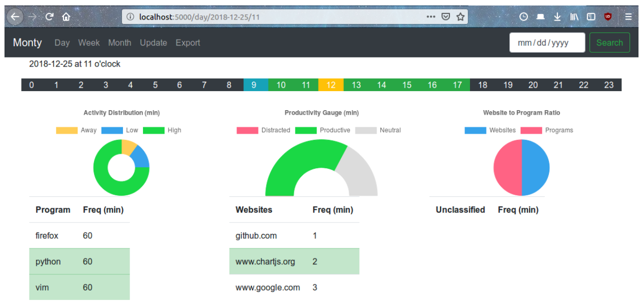


Show 10 entries   Search:

Program	Rank
10.0.0.1	Neutral
127.0.0.1	Neutral
academia.stackexchange.com	Unknown
academichelp.net	Unknown
adb	Unknown
app.plex.tv	Distracting
apport-checkrep	System
apport-gtk	System
apt-check	System
askubuntu.com	Productive

Showing 1 to 10 of 315 entries

Figure 2. A screenshot of what the update page typically looks like. Featuring a paginated table of ranked programs and websites.



2018-12-25 at 11 o'clock

Program	Freq (min)
firefox	60
python	60
vim	60

Websites	Freq (min)
github.com	1
www.chartjs.org	2
www.google.com	3

Unclassified	Freq (min)

Figure 3. A screenshot of what the day page typically looks like. Featuring highlighted hours of the day, informative charts indicating how the hour was spent, and what programs and websites were detected.

The day view acts as the splash page and automatically redirects to the current date and time. This is because the most relevant data when a person is wondering how they are spending their time is also the most recent. Following this notion, the rest of the site is designed where the more distant data require more site navigation. For example, to view the current hour's productivity, navigating to the splash page will present you with the relevant graphs. To view a different hour requires either changing the hour represented in the URL or selecting the hour in the colored row at the top of the page. For a different day, change the date in the URL or use the date-picker in the search bar. Similar levels of abstraction can be found in the week and month views. The day page, as illustrated in Figure 3, showcases the days overall hourly activity levels using a colored row, representing each hour of the day, starting from 0 (midnight) to 23 (11 pm), with the possibility of being colored: yellow for away from the computer, red for distracted, blue for low activity and not distracted, and green for high activity and not distracted. The reasoning for the latter to be described as high activity and not distracted (instead of productive) is because only you can determine whether or not you were productive or not.

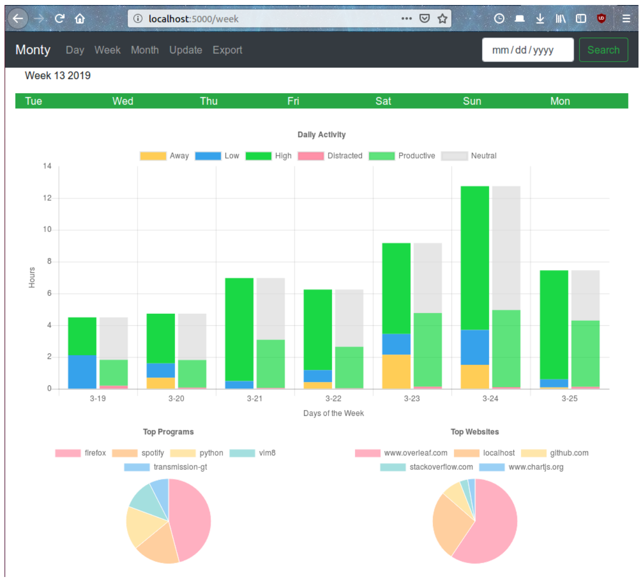


Figure 4. A screenshot of what the day week page typically looks like. Featuring highlighted days of the week, informative bar charts indicating how the week was spent, and the top five programs and websites that were detected.

The reasoning behind both high activity and productivity sharing the color green is due to the fact that if you are not distracted and there is high activity, then you are probably productive but not guaranteed. This bar also doubles as quick navigation to different hours of the selected day. For helping the user determine whether or not they were productive, a table breakdown of logged websites and programs for that time are presented with the number of minutes actively using them and highlighted easily identifying their respective ranks. Doughnut and pie charts are present in the day page, representing the number of minutes attributed to each activity level (Activity Distribution), the number of collective minutes of each productivity level based on active programs (Productivity Gauge), and a website to program ratio.

The week page, as shown in Figure 4, also contains a colored bar at the top of the page; however, instead of showing an hourly break down, there is a daily break down which can also act as navigation to the appropriate day for a further in-depth breakdown. This page consists of a stacked bar chart, which is grouped into two bars per day. One bar represents the collective number of hours spent with high/low/away activity, the other showcase a ratio of time spent productive/distracted/neutral. The bars are in units of hours and always add up to the same amount. Due to the productivity levels being represented as sums of minutes of all programs executed, there is easily more "time" recorded than there are hours in a day. To sync up the grouped bars, a ratio is calculated for the productivity levels using the following:

$$relative\_hours\_prod = \frac{hours\_prod}{total\_prod} * total\_act$$

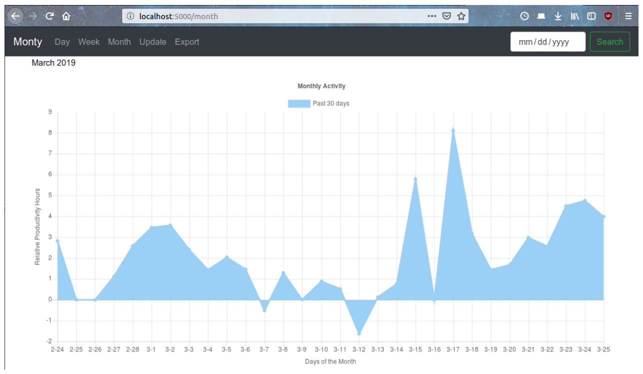


Figure 5. A screenshot of what the month page typically looks like. Featuring a line graph representing the number of *relative* hours spent productive or distracted, indicated by a positive or negative value respectively.

Where $total\_prod$ is the days total sum of every program and websites' frequency over the course of the day, $total\_act$ is the days total amount of logged hours, $hours\_prod$ is the total sum of just the days productive websites and programs' frequency, and $relative\_hours\_prod$ is the resulting ratio. Calculating the ratio for neutral and distracting hours is simply calculated by replacing $hours\_prod$ with the respective value. The week page also contains two pie charts for the top five programs and websites of the week based on minutes accumulated.

Lastly, the month page, as shown in Figure 5, contains a line graph showcasing the user's productivity over the past thirty days or over a specified month. Productivity is indicated via plotting the resulting hours after taking the difference between total hours spent productive minus the total hours spent distracted; therefore it is possible to attain "negative" hours. This is meant to clearly indicate when a person has been distracted for some amount of time and not to show that a user is a time traveler. This page only contains this line graph as processing every minute over the past minute is intensive. To attain more details on the data presented, simply single in one a time frame. In other words, move from a month's view to a particular week or day.

When developing the front-end JavaScript and CSS frameworks such as Bootstrap, AngularJS, ChartJS, and DataTables were used to develop a responsive design with a modern look-and-feel. Bootstrap is utilized to allow for easy re-sizing of the web pages without losing content. AngularJS is used for client-side functionality in the toolbar; namely for redirection dynamically when choosing a new time-frame to jump to. ChartJS is used extensively throughout the dynamic web pages, creating all the graphs present (doughnut and pie charts, stacked bar graphs, and a line graph). DataTables is used to generate a paginated table consisting of all the ranked programs and websites found in the database; allowing for quick searches and updates to the database.

4. Testing

During the development, testing was performed in the form of unit and regression tests. In respect to the back-end, as the submodules (activity, firefox, and log) were being written, before they included in the main driver, __main__, they were each independently run to verify that they perform properly and return the expected results. For example, the firefox submodule is expected to properly return a list of URLs that have been accessed in the past minute. To verify the output is correct I checked what tabs I had recently accessed through Firefox's history menu option. For the activity submodule, when executed it would print the current activity of the computer after a time delay of ten seconds in a loop until manually deactivated; this allows me enough time to open or lock the computer screen, or actively use the input devices (keyboard and mouse) and verify the generated output. The log submodule acts as a wrapper for SQL commands to the SQLite database. For testing purposes, a dummy database was created and the commands were tested with dummy data. The inputs to the SQL commands are cleaned before execution. Once the submodules were complete and introduced into the main driver regression testing was used to ensure previous functionality was not lost. For example, Monty was tested to make sure it could still catch signals and abort properly, as well as running a diverse set of applications and websites over the course of several minutes to see if they were still being detected and logged at the appropriate times. After the completion of the back-end portion of the productivity tracker, it is tested by continuously running in the background while it is being watched for errors, such as aborting due to an uncaught exception or an unaccounted for bug.

While the front-end was being developed, each button was repeatedly tested to ensure the output was consistently and correctly appearing. To ensure the data represented in the front-end was correct, the database was checked for the number of minutes for the time-frame in question and compared to the number of minutes that was reported on the webpage. Knowing that the two ends of the project were matching up meant that the data calculated was also correct. This is due to the fact that the graphs are produced from the number of times application names, website URLs and activity/productivity levels appear in the database for a given time-frame; since data is logged by the minute no value should be greater than the number of minutes logged for that time.

5. Usage

After installing Monty, as according to the README found in the software repository, the program should automatically begin running. Below describes different ways to use Monty. Once Monty is up and running, you can view the collected data via going to the following address in a web browser:

```
http://localhost:5000
```

This by default will navigate you to the present day and hour.

5.1. Using Upstart

Upstart is installed in Ubuntu 16 (and earlier) as a service manager. Monty will be installed as a user-level Upstart service. Meaning that it will automatically start and stop when the user logs in and logs off or powers off the machine. To tell if Monty is currently active, simply run:

```
status monty/start
```

This will indicate whether or not Monty is running and show its associated PID. To manually stop Monty:

```
stop monty/start
```

To manually start Monty:

```
start monty/start
```

To manually restart Monty:

```
restart monty/start
```

If you would like to stop the productivity tracking but still view the contents of the database, it is possible to run Flask independently. To run Flask without the tracker running in the background:

```
stop monty/start
start monty/flask
```

If you wanted to run the tracker in the background and turn off Flask (presumably to save 20MB of memory), you can do so like so. If the tracker is *already* running:

```
stop monty/flask
```

Otherwise, start the tracker then disable flask

```
start monty/start
stop monty/flask
```

5.2. UI Navigation

From the menu bar, there are six possible navigation elements:

Day
: This will navigate the user to the current date and hour.

Week
: This will navigate the user to the current week.

Month
: This will navigate the user to the current month.

Update
: This will show the user to a data table consisting of all entries in the Ranking table in the database.

Export
: This will generate a JSON object containing all data in the database.

Date-Picker

After selecting a date and pressing search, the user will be navigated to the selected date. Showing them *only* the hourly breakdown of the entire day. To see more details an hour must now be selected.

From the day page, different hours can be selected for an in-depth view by pressing the white hours (0-23 inclusive). All graphs in the day, week, and month pages have toggle-able data sets.

From the week's page, specific days can be selected for more detail. This redirects the user to the appropriate day's page.

5.3. URL Based Navigation

By utilizing URL parameters you can directly skip using the navigation bar to traverse the web UI by appending parameters to the base URL.

For viewing a specific day, simply append a date in the format of YYYY-MM-DD. For example:

```
/day/2018-12-25
```

For a particular hour, append the hour in military format (0-23):

```
/day/2018-12-25/13
```

A shortcut to the current date and hour (instead of typing out the date and time):

```
/day
```

To directly view the current week:

```
/week
```

To view a specific week of the current year you will need to append the corresponding week number. The first week of the year is denoted 1 and the last week is 52.

```
/week/10
```

For other years:

```
/week/2018/52
/week/2019/10
```

Similarly to week's URL parameters are the month page. To navigate to the current month:

```
/month
```

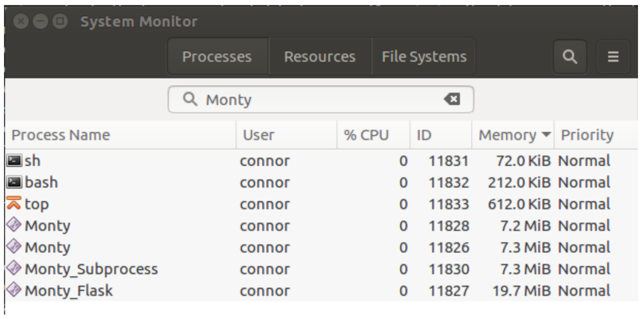
For a specific month, use 1 for January and 12 for December. If a year is not specified, the current year is assumed.

```
/month/3
/month/2018/12
/month/2019/3
```

To navigate to the data table:

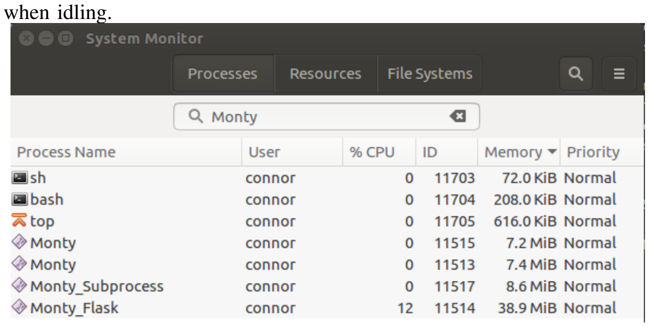
```
/update
```

Lastly, to export the entire contents of the database as a JSON object:



Process Name	User	% CPU	ID	Memory	Priority
sh	connor	0	11831	72.0 KiB	Normal
bash	connor	0	11832	212.0 KiB	Normal
top	connor	0	11833	612.0 KiB	Normal
Monty	connor	0	11828	7.2 MiB	Normal
Monty	connor	0	11826	7.3 MiB	Normal
Monty_Subprocess	connor	0	11830	7.3 MiB	Normal
Monty_Flask	connor	0	11827	19.7 MiB	Normal

Figure 6. System monitor showcasing the CPU and memory usage of Monty when idling.



Process Name	User	% CPU	ID	Memory	Priority
sh	connor	0	11703	72.0 KiB	Normal
bash	connor	0	11704	208.0 KiB	Normal
top	connor	0	11705	616.0 KiB	Normal
Monty	connor	0	11515	7.2 MiB	Normal
Monty	connor	0	11513	7.4 MiB	Normal
Monty_Subprocess	connor	0	11517	8.6 MiB	Normal
Monty_Flask	connor	12	11514	38.9 MiB	Normal

Figure 7. System monitor showcasing the CPU and memory usage of Monty when under heavy load. CPU power has temporarily increased to 12% and memory by about 20MiB.

```
/export
```

For a smaller JSON object only containing the rank data for entries that have names containing a subset of the input used as a parameter, do the following:

```
/month/vim
```

This return an object with entries containing ”vim”, ”vim8”, ”www.vim.org”, etc.

6. Performance Analysis

When Monty first starts up there is a delay between having started and initially logging data; however, the delay is due to top waiting one minute to produce output and this is intended functionality. Having the output generated from top processed in parallel allows for virtually no delay between calling top and the next iteration of calling top. The only slowdowns noticeable is when signaling Monty to quit with a SIGINT or SIGTERM as the time it takes nearly one second after having been running for over a minute, instead of a fraction of a second when aborting immediately after having started the program. This is due to the loop in the subprocess, tasked with processing the output, containing code to finish any leftover jobs in the queue when a signal is detected. This allows for a graceful exit without a loss of data (unless SIGKILL is sent).

When navigating to the day webpage of the front-end, all data is loaded in a fraction of a second. For the week webpage, anywhere from six seconds to less than one is necessary to generate the charts for display. This is due to having to iterate over potentially 10,080 logged minutes in a week and analyze the data contained in each minute. Similarly, the month webpage has a similar issue due to the large amounts of data to cover. When viewing the data over a month, load times can take anywhere from less than a second to approximately sixteen seconds depending on just how much data was logged for that month. The more data that is stored for a particular time-frame, the longer it will take to load. To improve load times would require preprocessing data after a unit of time has passed (e.g. a month) which would require more storage space on disk. It would also require updating the preprocessed files every single time an entry in the database is updated. This may cause significantly more time processing a time-frame instead of calculating the data as needed. Viewing the dynamic table, under the update page, takes less than a second to load, and data dumping the contents of the database as a JSON object requires approximately three seconds.

The back-end application that logs user data never reaches 1% CPU usage and consistently requires 25MB of memory. When idling the Flask server uses less than 1% CPU usage and approximately 20MB of ram; when under heavy loads, such as viewing week and month pages, CPU usage peaks at 12% and memory usage increases to approximately 40MB. This is shown in Figures 6 and 7.

7. Threats to Validity

The main threats to validity are the means of which I present the analyzed data in the web UI. The reasoning behind selecting the charts contained in Figures 3-5 is to best showcase the different types of data that is being logged without overwhelming the viewer. Overwhelming, in this case, means to avoid over cluttering the web page with too many graphics. Through trial and error, it seemed that four charts are too many; however, this is ultimately arbitrary. Using that limit, I thought of what potential patterns could be found when graphing over several days, and what metrics would be insightful for a user wondering how their day was spent. Again the decisions behind choosing how and what to chart was more or less of what I found useful. The level of importance of the presented metrics may differ between users. To account for this, the export page generates a JSON file that can be used to find other patterns may be more insightful. Also, users could make SQL calls directly to the underlying SQLite database. Another current issue of the software is that it is tailored to Ubuntu 16 and Firefox. To handle the use of other browsers would require further development in how such browsers store their session data. For enabling Monty to easily install to other Linux distributions would require to replace/supplement the current support for Upstart, which has been deprecated, with SystemD. Both issues are currently a top priority in future work as the software is stable in Ubuntu 16.

8. Conclusion

This paper has discussed the pros and cons of several currently existing productivity trackers on the market and aims to show that a useful productivity tool can be developed without the downsides of existing ones. Most popular productivity trackers come at a price of being non-free proprietary software with the uncertainty of how your data is being used remotely, such as Saent, TimeTracker, and RescueTime. This project shows that productivity trackers do not have to send your personal data remotely to offer a service that is fast, simple to use, and not resource intensive. By limiting logged data to time-stamps, names of user run programs, the base URLs of websites visited, whether or not input was detected and if the lock screen was activated, informative graphs can be produced to help deduce where users are becoming distracted. The small amount of data being tracked allows for a small SQLite database and quick lookups. Saving all of the data locally ensures that the collected data stays on your machine. This also allows the user to perform their own experiments on their own data. With a minimal web user interface, you can interact with the collected data or simply view time-frames in question for an in-depth analysis.

Currently, Monty requires 15MB of available hard drive space for the source code with additional room for the database. With over 270 ranked applications and websites and 22,671 logged minutes, over the span of approximately four months, only produces a 9MB database. Ultimately Monty serves as an alternative to existing productivity trackers as it is convenient to use, non-invasive and lightweight.

9. Future Work

Future modifications to this project would first and foremost to support SystemD in place of Upstart, for enabling Monty as a service. This is due to most Linux distributions are now supporting SystemD and Upstart is no longer shipped with Ubuntu after version 16. Enabling multi-selections in the dynamic data table for fast database updates via the web UI is another future feature. Currently, only one row can be modified at a time. Automatically refreshing the day webpage to show the most up-to-date data is also a future work. As of now, if you would like to view new data of the current date and hour the user would need to periodically refresh the page. Lastly, another feature I look forward to implementing in the future is the ability to use Monty on different work stations and being able to combine exported databases into one merged database, showcasing the work patterns on different machines environments.

References

[1] (2019) Saent homepage. [Online]. Available: https://www.saent.com/

[2] F. Cirllo, ”The pomodoro technique (the pomodoro)”, *Agile Processes in Software Engineering and*. Harlow, England: Addison-Wesley, 2006, vol 54.

[3] (2017) The TimeTracker homepage. [Online]. Available: https://www.openhour.com/timetracker/

[4] (2016) The Fluxday homepage. [Online]. Available: https://fluxday.io/

[5] (2017) The RescueTime homepage. [Online]. Available: https://www.rescuetime.com/

[6] (2009) The Zeitgeist Project. [Online]. Available: https://launchpad.net/zeitgeist-project/

[7] (2011) Activity Log Manager. [Online]. Available: https://launchpad.net/activity-log-manager/

[8] (2019) lz4json software. [Online]. Available: https://github.com/andikleen/lz4json/