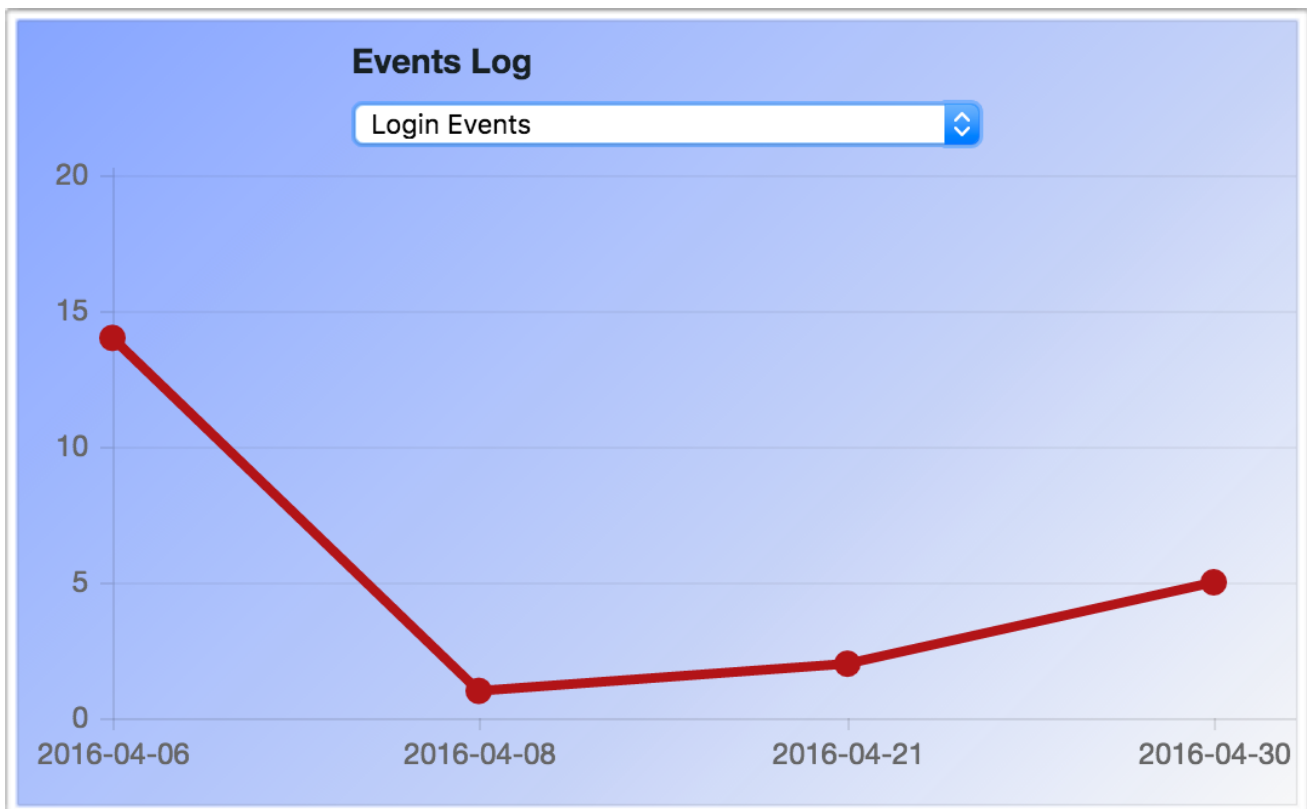


Yoav Nathaniel  
Professor Victor Miller  
Senior Project  
April 30, 2016

## ActMonitor - Real-Time Event Tracking



A cloud-based database portal that utilizes the power of APIs, ActMonitor is a secure and fully customizable tool to create analytics for your software. Mobile applications and SaaS (Software as a Service) are recent examples the direction of which technology is heading. They are based on a model where a secure and centralized control center can update endpoints (user devices) in real time while also providing accessibility from virtually anywhere. This is major step from desktop applications, which heavily rely on users to constantly update their software.

**“‘Does it better’ will always beat ‘did it first.’”**

*-Aaron Levie (CEO at Box)*

Cloud technology not only sends information to users in real time, but it also receives information as well. Depending on the specific application, endpoints may perform logins, button clicks, scrolls, open/close the app, submit forms, and more. For a developer to come up with ideas on how to improve the UX (user experience), he/she must acknowledge what the user likes (or dislikes) about the application. This led to several inquiry methods like surveys, ‘Ask a question’ buttons and live chats; these methods have pros and cons, as displayed in Exhibit 1.

**Exhibit 1 - Pros and Cons of User Inquiries to Improve Software**

The Goods	The Bads
Human to human interactions can lead to direct understanding of ways software can be improved	Not many users are willing to spend time to improve the software they use
Users may enjoy when the software adjusts to their requests	Asking users the wrong questions can leads to ineffective campaigns
	Aggressive popups can lead to user decline
	Until artificial intelligence is fully ready, this requires a human to understand what’s missing

ActMonitor steps in as the next-generation method aimed at improving software. While traditional methods depend on a direct approach of Q&A, ActMonitor seamlessly tracks user activity to analyze which parts of an application appeal to the users, and which parts don’t. Using a dynamic API structure, each type of event collects specific-event-related data each time an endpoint performs such event. For example, a developer adds a button to an application with 100 users. Each time the button is clicked, an API request is sent to ActMonitor to record that event. After a month, the developer sees on ActMonitor that the button was only clicked 5 times, thus the new button was ineffective. Or in a different scenario, the developer may find 5,000 button clicks, which turns the new button into a critical part of the software.

Technology empowers companies so that they rely less on human resources. With ActMonitor, less people need to work at a help desk, more insights about features can be managed at a time, and the effects new features have on a product become measurable. Software companies can finally understand how their products are being used over time and which developer is more effective at creating a healthy product. For example, ActMonitor can be set up to track bugs per developer to calculate how many bugs are caused by some developer. Such insights can lead to an improved workflow so production can increase at a faster rate.

# Table of Contents

<b>Introduction</b>	<b>1</b>
<b>User Manual</b>	<b>5</b>
<b>Design Structure</b>	<b>15</b>
<b>Test Plan</b>	<b>20</b>
<b>Final Notes</b>	<b>26</b>

# User Manual

## Features

Dashboard	6
Overview	6
Navigation Menu	6
Trackers	7
What is a Tracker?	7
All Trackers	7
Tracker Profile	8
Create a New Tracker	9
Alerts	10
What is an Alert?	10
Existing Alerts	11
Create a New Alert	12
User Management	12
Managing System Access	12
Login and Logout	13

# Dashboard

## Overview

The URL path for the Dashboard is “/” (root). This page can be accessed through the navigation menu under “Dashboard”.

The Dashboard should provide a quick glance of the system. You can find the 15 most recent events, the total number of alerts in the system, and a dynamic graph displaying the number of events per date\*. The graph has a select button to adjust which tracker should be displayed.

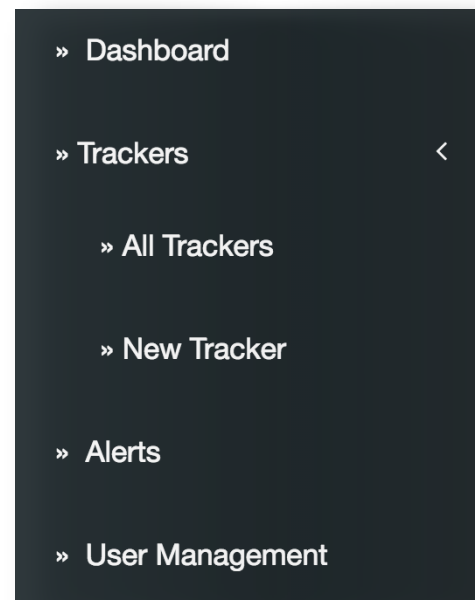
\* The graph only shows dates where the event count is greater than 0

## Navigation Menu

The navigation menu is available on the left side of every page in the application. It has 2 parts to it: menu pages and submenu pages.

Menu pages contain all of the information and features you need regarding their content. For example, the User Management page has all of the features one would need to handle which users have access to the ActMonitor machine.

Submenu pages have a submenu title and are marked with the arrow to the right. They were created to reduce the need of navigating inside pages. For example, “Trackers” was divided into two sections: “All Trackers” for easy access to existing trackers and “New Tracker” to quickly create a new tracker. In such way, creating a new tracker is a 1 click away from any page. Easier access typically translates to more usability.



**IMPORTANT NOTE:** Submenu pages are hidden unless you hover over the the Submenu title. For example, you'd have to hover over "Trackers" to see the submenu pages in that category.

## Trackers

### What is a Tracker?

A tracker is a single type of event, and this allows flexibility track any application, regardless to how extensive it is. A single tracker may track all login events in some applications, while more in-depth applications may have the following trackers:

- All Logins
- Successful Logins
- Failed Logins

The level of depth trackers should have depends on the application's ability to recognize such events. If an application can detect 5 consecutive failed login attempts, then a "Suspicious Logins" tracker may be created to match the security features of the application.

Each tracker has a unique database table and so it holds its own records with its own properties. While one tracker may contain the username of the event actor, another tracker may contain the age of the event actor. The key concept to trackers is **customizability**.

### All Trackers

The URL path for the All Trackers page is "/trackers/". This page can be accessed through the navigation menu as a submenu page under "Trackers".

The dynamic content of the All Trackers page displays all of the existing trackers. A brief overview of each existing tracker allows the user to proceed to a tracker's profile - where more information about the tracker can be found - or delete the tracker from the system\*. In addition, the "New Tracker" button leads to the New Tracker page (can also be accessed through the "Trackers" submenu).

**All Trackers** Yoav - Master [Logout](#)

[Create a new tracker](#)

[Go to a tracker's profile](#)

Name	API Name	Events Tracked	Delete
<a href="#">Miller</a>	miller	8	x
<a href="#">Gavin</a>	gavin	4	x
<a href="#">Login Events</a>	login_events	22	x

[New Tracker](#)

[delete a tracker](#)

\* Deleting a tracker removes deletes the tracker, its unique API, and all of its collected data. This action is irreversible.

## Tracker Profile

The URL path for the Tracker Profile page is “/trackers/profile/<tracker api name>”. This page can be accessed from multiple locations in ActMonitor, but mainly through the All Trackers page.

Since a tracker has dynamic properties, the Tracker Profile page must display dynamic content. This page provides the following functionalities:

- Example request snippet - specifies how a request to insert an event to the tracker looks like.
- Example request - generates a request to insert a sample event to the tracker. Can be used for debugging purposes.
- Create alert rule - create a new rule to receive an alert each time some column is some value.
- View tracker events - check out all of the events inserted to this tracker.
- Delete tracker events - delete a specific event in this tracker.



## Login Events

### Example Snippet

#### Request Structure

```
var url = "https://customer.actmonitor.com/api/tracking/insert/login_events";
var method = "POST";

var insert_data = {
  "username": "VARCHAR",
  "age": "INTEGER"
};
```

#### Create a custom alert

[Create Alert](#)
[Sample Request](#)

#### Test tracker usage

#### Pagination

[Previous Page](#)

### Tracker's inserted data

[Next Page](#)

Username	Age	Timestamp Created	Delete
Example String	12345	2016-04-06 18:31:42.379345	×
Example String	12345	2016-04-06 18:34:06.527630	×
Example String	12345	2016-04-06 18:36:15.791088	×
Example String	12345	2016-04-06 18:39:05.006248	×

Delete  
an event

## Create a New Tracker

The URL path for the “New Tracker” page is “/trackers/new/”. This page can be accessed through the button in the “All Trackers” page and in the navigation menu.

The “New Tracker” page allows you to create a customizable tracker to fit the specifics of the tracked properties. You must follow the these steps to create a tracker:

1. Enter a unique tracker name. **Note:** This name should be unique not a duplicate of another tracker in the system.
2. For each property that needs to be tracked:
  1. Click “Add Property” to add a new field for a property.
  2. Enter a property name. **Note:** a property name should be unique inside the tracker. A tracker cannot have two properties of the same name.
  3. Select a property type.

4. Select the first checkbox if the property is nullable (can be equal to NULL).
5. Select the second checkbox if the property must be unique (no 2 events can have the same value for this property).

The screenshot shows a web form for adding a property to a tracker. At the top is a 'Tracker Name' input field. Below it is a table-like structure with two rows. The first row has a 'Property Name' input field, a 'Boolean' dropdown menu, and two checkboxes labeled 'Can be null:' and 'Is Unique:'. The second row is identical but is highlighted with a red border. A red arrow points from the text 'A single tracker property' to the second row. At the bottom of the form are two buttons: 'Add Property' and 'Create Tracker'.

3. Click "Create Tracker".
4. If this was successful, you will be redirected to the "All Trackers" page where you should be able to find your new tracker. If you were redirected but do not see the new tracker, simply refresh the page.
5. If this was not successful, follow the error message to fix the new tracker form. **Note:** the tracker name and all property names must be filled (non-empty) for the form to be considered as complete.

## Alerts

### What is an Alert?

An alert is a custom rule to inform the system users of when a new event with certain values is inserted into the system. Each alert has a name, target tracker, target column, and target value.

The following rules apply to alerts:

- Alerts are per tracker - You may have multiple alerts reviewing one tracker, but each alert only checks one tracker.

- The target value is always a string (collection of characters). If you want to target a non-string column, simply target the string version of that column. For example, a Boolean column C would have either True or False. If you wanted to target all events where C is True, you would enter “True” for target value.

An example alert an “All Logins” tracker with “Is Suspicious” Boolean column could be:

Alert Name: Suspicious Logins

Target Tracker: All Logins

Target Column: Is Suspicious

Target: True

## Existing Alerts

The URL path for the “Alerts” page is “/alerts/”. This page may be accessed from the Dashboard (there is an infobox) and from the navigation menu.

Alerts					Yoav - Master <a href="#">Logout</a>
Rules					
Rule Name	Tracker Name	Tracker Column	Tracker Value	Delete	
First Rule	<a href="#">Login Events</a>	age	12345	x	
Sample Rule	<a href="#">Sample</a>	haha	True	x	
Miller is here	<a href="#">Miller</a>	name	Example String	x	
Findings					
<a href="#">Previous Page</a>					<a href="#">Next Page</a>
Rule Name	Tracker Name	Tracker Column	Tracker Value	Delete	
First Rule	<a href="#">Login Events</a>	age	12345	✓	
First Rule	<a href="#">Login Events</a>	age	12345	✓	

The page is split into 2 parts: rules and findings. Alert rules describe the requirements of an event to be considered an alert. If a new event arrives, it will be checked against

each alert rule. For every alert rule that applies to the event and matches rule's column and value, a new Alert Finding is created. Alert findings are actual events that matched a rule. Each alert finding correlates to some event and contains an explanation (using properties) as to why this event was matched.

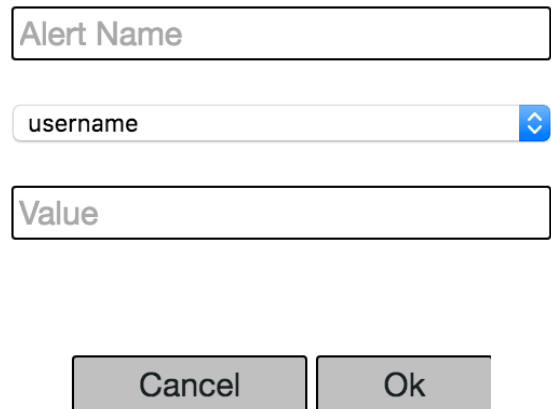
**Note:** one event may contain between 0 to N alert findings. N is the number of alert rules.

## Create A New Alert

Creating a new alert is done from the "Tracker Profile" page (page 8 of this document) by click on the "Create Alert" button. A popup form show up to allow you to specify:

- Alert name - a name relevant to the purpose of the alert.
- Column - any column you set up for this tracker.
- Value - an event with this value in the chosen column will trigger an alert finding.

### Create New Alert



Alert Name

username

Value

Cancel Ok

## User Management

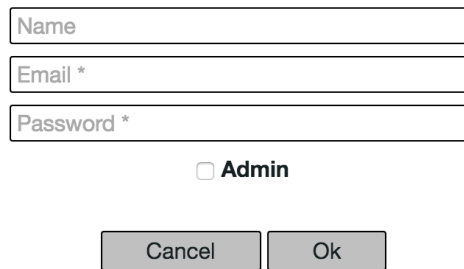
### Managing System Access

Managing what users are authorized to use the system is done in the "User Management" page. The URL path for the "User Management" page is "/user-management/".

This page shows a table of all of the visible\* users with access to the system. You are also able to manage authorized users. You may create a new user with the "Create User" button. This will lead to a popup form allowing you to add a new user.

**Note:** the new user's email must not already exist in the system.


## Create New User



A dialog box titled "Create New User" with three input fields: "Name", "Email \*", and "Password \*". Below the fields is a checkbox labeled "Admin". At the bottom are two buttons: "Cancel" and "Ok".

\* There are authorized users hidden from the UI for administration purposes. A default example is "Yoav - Master" user.

You may also delete a visible user by click on the "X" button in the user's row.

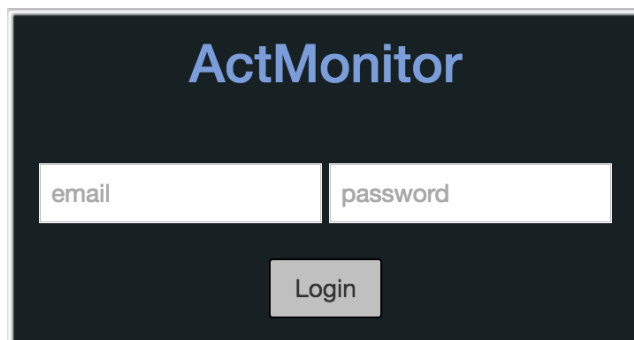
Name	Email	Is Admin	Delete
Sample	a@gmail.com	False	 x
Duck	duck@gmail.com	True	x

## Login and Logout

The system is kept secure by only allowing authorized user access to the collected events and internal APIs. Except for the API to insert new events, all APIs require authentication.

If you were not previously logged into the machine, you will not be allowed to navigate to any page in the system except the "Login" page. The URL of the "Login" page is "/login/". Once you manage to login successfully, you will be redirected to the Dashboard.

If you are logged into the machine, you will be allowed to navigate to all internal pages in the



A login page with a dark background. At the top, the text "ActMonitor" is displayed in a light blue font. Below it are two input fields: "email" and "password". At the bottom center is a "Login" button.

system. You will always have the option to logout by clicking on “Logout” on the top right of any internal page.

# Design Structure

The key concept of ActMonitor is **dynamic**. The program must accommodate an unspecified number of trackers, which translates to an unspecified number of database tables, events (rows in tables), and even unpredictable table columns. The solution - use ORM (Object-Oriented Mapping) to map dynamic classes to tables so it's possible to create and manage dynamic tables on the fly.

## Terminology

### Dynamic Object

ActMonitor relies on **dynamic objects**, classes with customizable properties. Each tracker represents 1 dynamic object which contains 1 customized class and 1 database table with customized columns.

### Dynamic API

APIs (Application Program Interface) are non-webpage URL paths that allow communication of data between a client and a server. APIs are used all over the internet to create non-static and responsive webpages. For example, a browser can request a webpage, the server would return the skeleton page very quickly which will issue additional requests for other data like user name. With APIs, clients and servers can interact with each other without requiring the page to reload, thus the webpage performs actions without the user noticing.

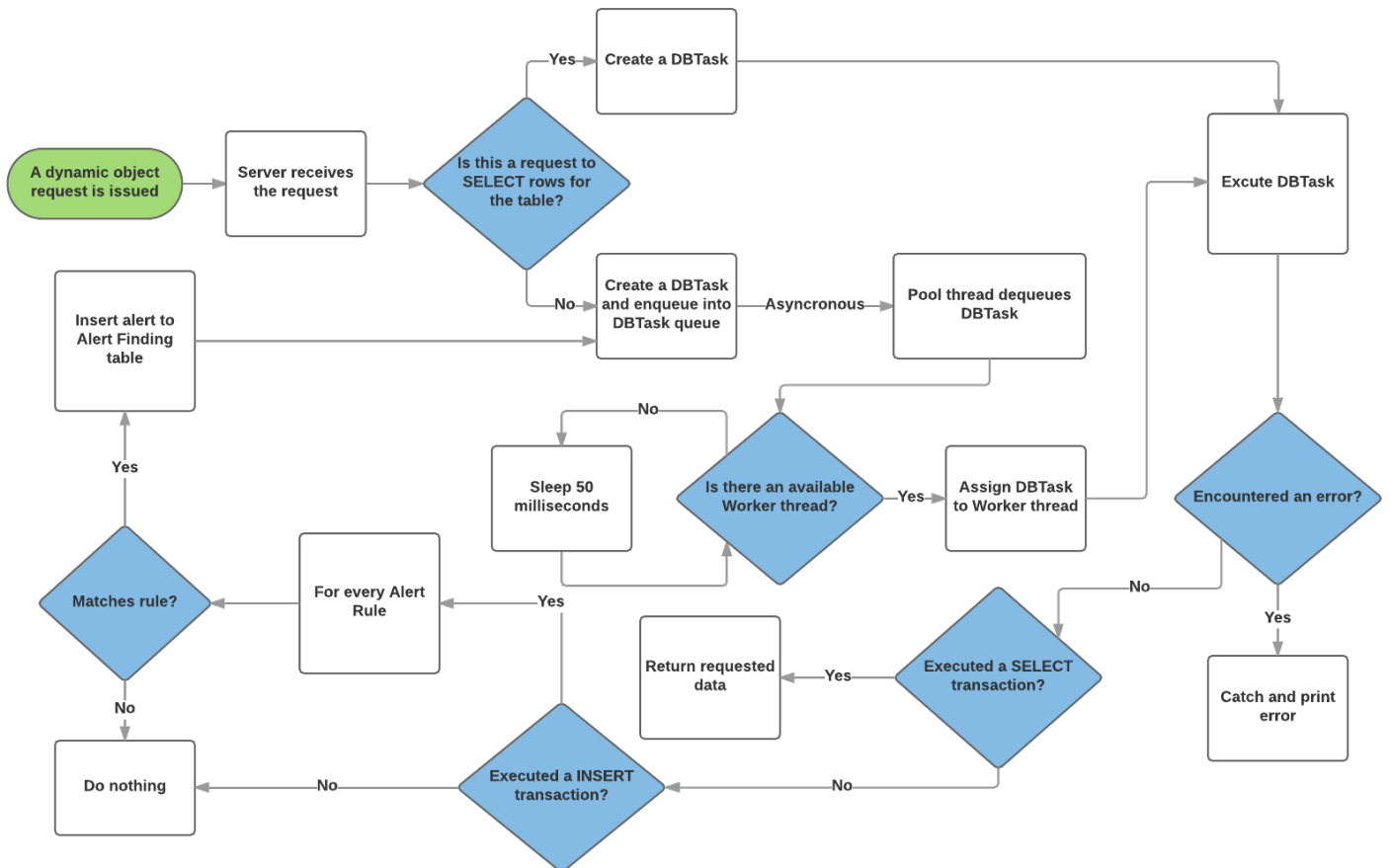
Each dynamic object receives a **dynamic API** name which can be inserted into a URL path. The dynamic URL path creates interactions related to that dynamic object. For example, plugging in the API name into `https://customer.actmonitor.com/api/tracking/insert/[ DYNAMIC API NAME ]` will allow the client to insert into that object.

### Database Task (DBTask)

The ORM model creates fragile functionality which demands a standard method of handling database transactions. The standard requires that only tasks of the correct format be handled. Every request to execute a database transaction is converted into a

standard format of a **database task** so all functions can perform in a standardized manner.

## Dynamic Object Workflow



## Classes and Responsibilities

### DBManager

Sets up the DBUtils and DBPool instances and the database to meet the required standards upon initialization. Required standards include setting up default database schema and tables for dynamic objects. For example, the `dynamic_objects` and the `alert_rules` tables.

Reformats database transaction requests and enqueues DBTask instances into the DBTaskList.



## DBUtils

Can be thought of as the assistant of the DBManager. Allows quick 'global' access to most of the parts data structures. It is 'global' because almost every object has access to this and each object just uses what it needs. Key items this class includes: database engine connection, dynamic objects, DBTaskList, and DBPrinter

Has functionality to save and restore DBTask instance if program is shut down before tasks were executed.

## DBTask

The standard format of a single database transaction. It allows one action to be done to one dynamic object with a set of parameters. For example, CREATE TABLE sample\_table with some set of columns.

## DBTaskList

Thread-safe queue of DBTask instances. The DBManager creates DBTask instances and inserts them into this class. The DBPool (if possible) dequeues from this class.

Has boolean lock that can disable enqueueing to the queue. Enqueueing is disabled when the program is shutting down and then the queue extracts pending DBTask instances into a dictionary format (basically JSON).

## DBPool

Thread class that manages DBWorker threads and creates the DBAction instance. This class dequeues DBTask instances from DBTaskList. In a loop, it checks if the DBTaskList queue size is greater than 0, dequeue one DBTask and set it as a pending-to-execute task. Then it checks to find the first available DBWorker to assign the dequeued task to. Once the task is assigned, restart the loop.

The DBPool has a safety mechanism in case a DBWorker remains stuck working on a DBTask. Typically, a single DBTask is expected to take less than 0.5 of a second to execute, if not less. If the DBWorker is working on a single DBTask for over 3

seconds, stop using this DBWorker and create another DBWorker that will work on the same DBTask.

## DBWorker

Thread class that receives up to one DBTask instance at a time and oversees the execution of the DBTask. Uses the DBAction instance it's assigned by DBPool.

Has a boolean lock tells the DBPool if this DBWorker is working on a DBTask.

## DBAction

The class that directly engages with the database and creates/deletes dynamic objects. On startup, it's able to extract all existing dynamic objects from the database.

Each dynamic object this class creates has the following properties:

- Customized class (customizes an instance of a class template)
- SQLAlchemy table that directly correlates with the database table
- Name (Can contain spaces, dashes, and capital letters)
- API Name (lower case with no spaces or dashes)
- RLock to make sure only one DBWorker thread touches this dynamic object at one time
- Count (quick access to table rows count)

When creating an existing dynamic object during startup, this class enqueues a DBTask to update the dynamic object count (count the rows in the existing table). If a dynamic object is created after the startup (anytime the program is simply running), the object will have a count of 0 by default and it will be registered into the dynamic\_objects database table. If a dynamic object is deleted, delete its table, its record in the dynamic\_objects table and the dynamic object stored in DBUtils.

The DBAction instance executes all DBTask instances by routing them based on their action type and converting them into database queries that affect the respective dynamic object (this information is all found in the DBTask).

## DBPrinter

Thread queue of statements to print. This is accessible through DBUtils and so it's 'globally' available for all classes to enqueue a statement to print. Every 400 milliseconds, the queue will be locked and the statements in the queue will be printed in the order of which they were received.

This class prevents weird printouts where there can be an extra blank line or printed statements being cut off.

An external function may enqueue 1 statement or a list of statements. If a list of statements is enqueued, the class will enqueue the statements one by one.

## TemplateDynamicObject

All dynamic objects inherit from this class. The constructor takes an unexpected number of named arguments and inserts them to the object dictionary (built in for all python classes). The override of the default getter function grabs properties directly from the object dictionary, instead of just the object.

# Test Plan

ActMonitor can be tested a variety of ways because of its multiple parts. This includes:

- Database transactions
- Dynamic objects
- Web interface
- Front-to-back integration

The back end is made to work using APIs, which are also used by the front end. To take full advantage of the APIs, it is essential to use a browser that is authenticated into ActMonitor.

## Insert API

Send a request to the authenticated machine using the following details:

Request URL path: /api/tracking/insert/[ Dynamic Object API ]

Request method: POST

Request body:

```
{  
    property1: value1,  
    property2: value2  
}
```

properties are the names of the columns in the table for this dynamic object.

Expected results:

If you insert 2 rows with the same value in a unique column, the server will encounter an error and the second row will NOT be inserted.

If you insert an empty/NULL value into a non-nullable column, the server will encounter an error and the row will NOT be inserted.

If the previous statements do not apply, the new row will be inserted.

## Update API

Send a request to the authenticated machine using the following details:

Request URL path: /api/tracking/update/[ Dynamic Object API ]

Request method: POST

Request body:

```
{
  "where": {
    property_for_filter: value_for_filter
  },
  "update": {
    property_to_update: updated_value
  },
  "limit": integer max number of affected rows. 0 means all rows (default),
  "offset": integer row number to begin with. default is 0
}
```

properties are the names of the columns in the table for this dynamic object.

Expected results:

If you have a limit of 1 and 2 or more rows match the WHERE filter, only the first of those rows will be updated.

If you have a limit of 0 and 1 or more rows match the WHERE filter, all of these rows will be updated.

If no rows match the WHERE filter, no rows will be updated.

Only rows at or after the offset can be updated.

## Delete API

Send a request to the authenticated machine using the following details:

Request URL path: /api/tracking/delete/[ Dynamic Object API ]

Request method: POST

Request body:

```
{
  "where": {
    property_for_filter: value_for_filter
  }
}
```

properties are the names of the columns in the table for this dynamic object.

Expected results:

All rows that match the WHERE filter will be deleted.

## Select API

Send a request to the authenticated machine using the following details:

Request URL path: /api/tracking/select

Request method: GET

Request body:

```
{
  "object_name": name of dynamic object
  "where_data": {
    property_for_filter: value_for_filter
  },
  "column_data": [
    "property_to_select"
  ]
  "limit": integer max number of affected rows. 0 means all rows (default),
  "offset": integer row number to begin with. default is 0
}
```

properties are the names of the columns in the table for this dynamic object.

Expected results:

If you have a limit of 1 and 2 or more rows match the WHERE filter, only the first of those rows will be selected.

If you have a limit of 0 and 1 or more rows match the WHERE filter, all of these rows will be selected.

If no rows match the WHERE filter, no rows will be selected.

Only rows at or after the offset can be updated.

Only the chosen columns will be returned.

## Create API

Send a request to the authenticated machine using the following details:

Request URL path: /api/tracking/create

Request method: GET

Request body:

```
{
  "name": name of dynamic object to create (cannot contain underscores)
  "properties": [
    {
      "name": property name,
      "type": property type,
      "unique": true/false,
      "nullable": true/false
    }
  ]
}
```

properties are the names of the columns in the table for this dynamic object.

property types are any of the following: "Boolean", "DateTime", "Integer", "Float", "String", "Unicode"

Expected results:

If a dynamic object with this name already exists, the api returns an error with a description.

If the name of the dynamic object contains an underscore, the api returns an error with a description

If any of the properties were not of the right format, an error will be printed in the server and the dynamic object will **not** be created.

If all of the rules were followed correctly, a dynamic object (server memory) will be created, a table (database) will be created, and a new row in dynamic\_objects table (database) will be inserted. Another way to check for this is to go to the All Trackers page to find the newly created tracker.

## Web UI

Play around in the web ui and click on the buttons to see that data is always shown where needed. You may follow the user manual (in this document) to find the expected results from the user interface. This includes what buttons do and what each page represents.

## Authentication - Login

If you are not authenticated with the browser you're using, attempt to login with fake credentials, and you'll receive an error popup.

If you are not authenticated with the browser you're using, attempt to go to a restricted URL, such as "/" (the main dashboard page), and you'll be redirected to the login page.

If you are not authenticated with the browser you're using, login with working credentials and you'll be redirected to the main dashboard page.

Master admin credentials:

login email: [ynathaniel@gmail.com](mailto:ynathaniel@gmail.com)

login password: 1234

## Authentication - Logout



If you are authenticated with the browser you're using, attempt to go to a working URL path, and you'll be directed to that page, not the login page.

If you are authenticated with the browser you're using, attempt to go to a fake URL path, and you'll be directed to the 404 (not found) page, not the login page.

If you are authenticated with the browser you're using, click on the 'logout' button in the top right (found on every non-login page) and you'll be redirected to the login page.

## Final Notes

The senior project gave me the opportunity to create something cool and meaningful. I came up with a challenging project with requirements I had not much experience in, like working directly with a database, creating APIs, and authenticating my software. On top of that, ActMonitor is a very basic utility used by every tracking software. Huge companies like Facebook and Google are masters tracking in a large scale while smaller companies track less but crucial data. Especially in 2016, where companies are competing to sell tracked user data, ActMonitor creates a portal for full customizability of anything that should be tracked.

Asynchronous functionality plays a huge role in this application as tracked products do not need to wait on tracker inserts. Instead, they can send a simple request of “track this data” and keep on with their functionality with unnoticed disruption. At the same time, administrators that track this data, do not have to worry much about processing data as the server contains a queue of tasks that lets many insert requests be handled quickly but the server will take its time accomplishing all of the tasks; a major aspect that saves the day during peak hours of product use.

I will definitely enhance the functionality of ActMonitor with better alert and perhaps even creating relations between two or more trackers so in-depth analysis can be made.