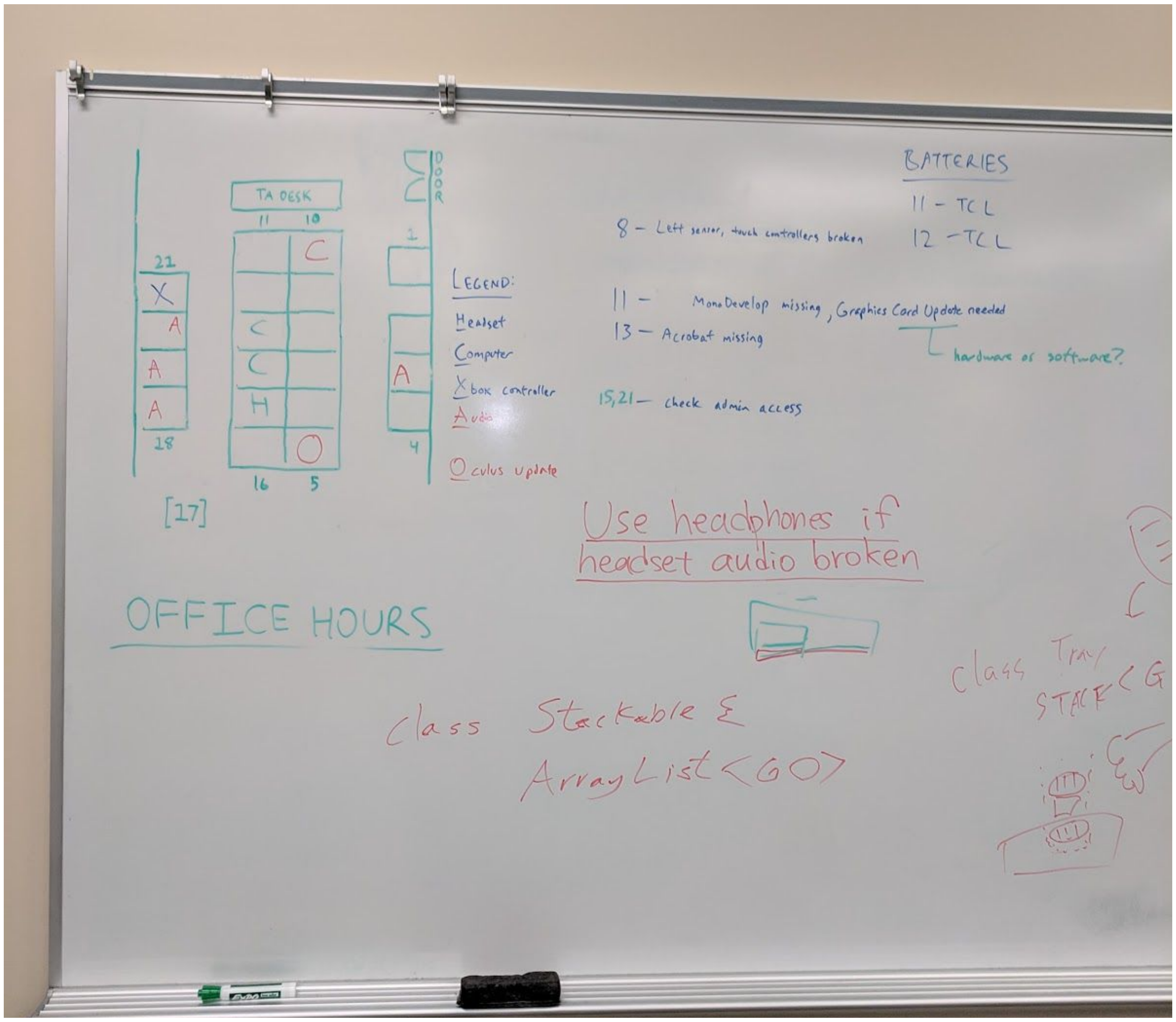
CS 411 Project Track 1: VR Lab Monitoring Website

Team Rocket (Group 27)

Michael Rechenberg (rchnbrg2)
Adam Burkett (aburket2)
Thomas Varghese (tvarghe2)
Tianyu Liang (tliang7)

What Problem Our Project Solves



This is a picture taken from the front whiteboard in the VR lab in Seibel this year. There are a lot of moving parts required for the VR lab to function: powerful desktop computers to run the software, Oculus headsets, Oculus Touch controllers, XBOX controllers, and complex software like Oculus calibration software and Unity (a game development package). With many moving parts means that many of those parts break over time from regular use. To try and keep as many workstations in the lab as operational as possible, the course staff and students decided to use the front whiteboard and ask students/TAs to record if any of the workstations had issues.

This whiteboard guided the design of our project as we aimed to offer the same information and functionality but in a more convenient web application. A couple of takeaways from this picture:

- In the top left of the whiteboard we see a top-down diagram of the workstations in the lab so students/TAs can map the location of the workstation in the lab to its numerical id.
- In the top-middle we see that students and TAs have recorded the current issues with machines in the lab. For example, Machine 8's left sensor and Oculus Touch controllers are broken and for Machine 13, Adobe Acrobat is missing. When an issue is resolved, a TA erases the issue from the whiteboard.

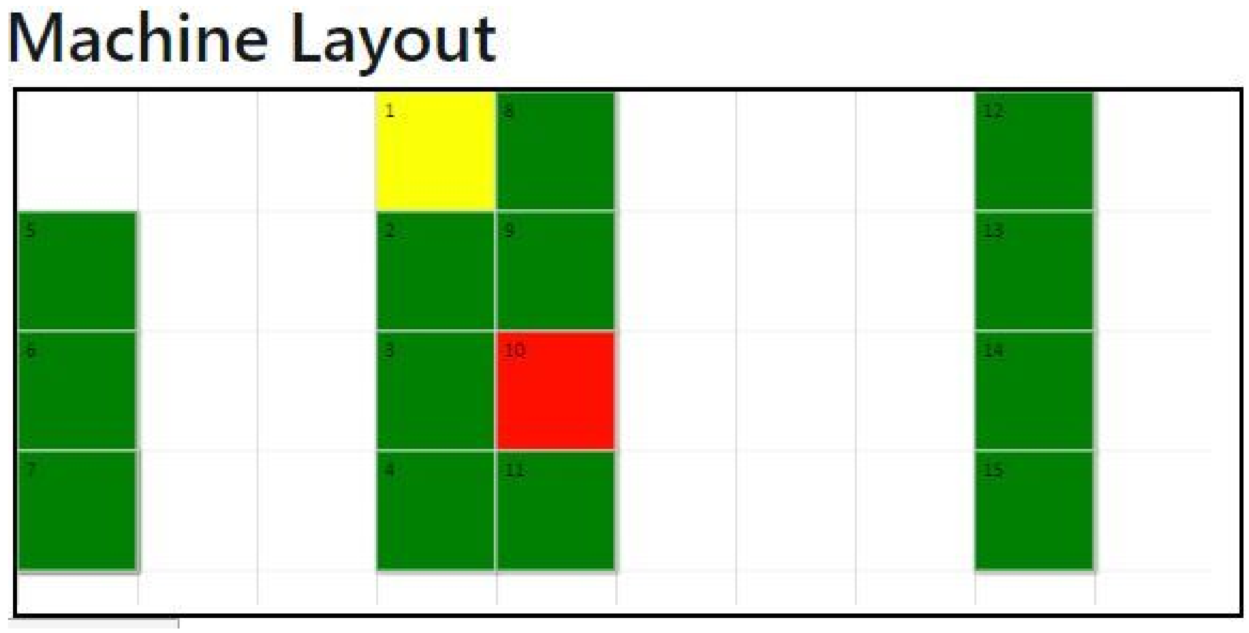
Our website aims to improve on this whiteboard-based issue tracking. With the whiteboard, there is no lasting record of what issues have happened for machines before or who was using the machine after a TA erases the whiteboard. And if a student wants to work at a fully functioning workstation, he has to 1) pick a workstation that isn't in the list of machines with issues and 2) physically look around the lab to see if someone is using the machine he chose. Particularly close to MP deadlines, the lab can be at capacity and students can arrive to the lab only to find that there are no workstations they can use, wasting their trip to Siebel.

Our website solves the first problem of issue storage and monitoring by storing a database of comments related to issues found with workstations that TAs can resolve at a later time. By making the barrier to entering an issue easier (just fill out a small comment box on the computer you're already sitting at instead of hunting down a marker and writing on the whiteboard and remembering which machine you were working on), we hope to encourage more students to report issues so TAs can resolve them and keep the uptime of the lab as high as possible.

And we solve the second problem of machine availability by showing users that a machine is either ALIVE (There are no current issues with the machine AND no one is currently using the machine), IN-USE (There are no current issues with the machine BUT someone is currently using the machine), or BROKEN (There is currently an issue with the machine so no-one can use it) via colored indicators of workstations on a 2D grid.

What Our Project Accomplished

Our website is hosted on cPanel and can be visited here:
http://teamrocket.web.illinois.edu/home

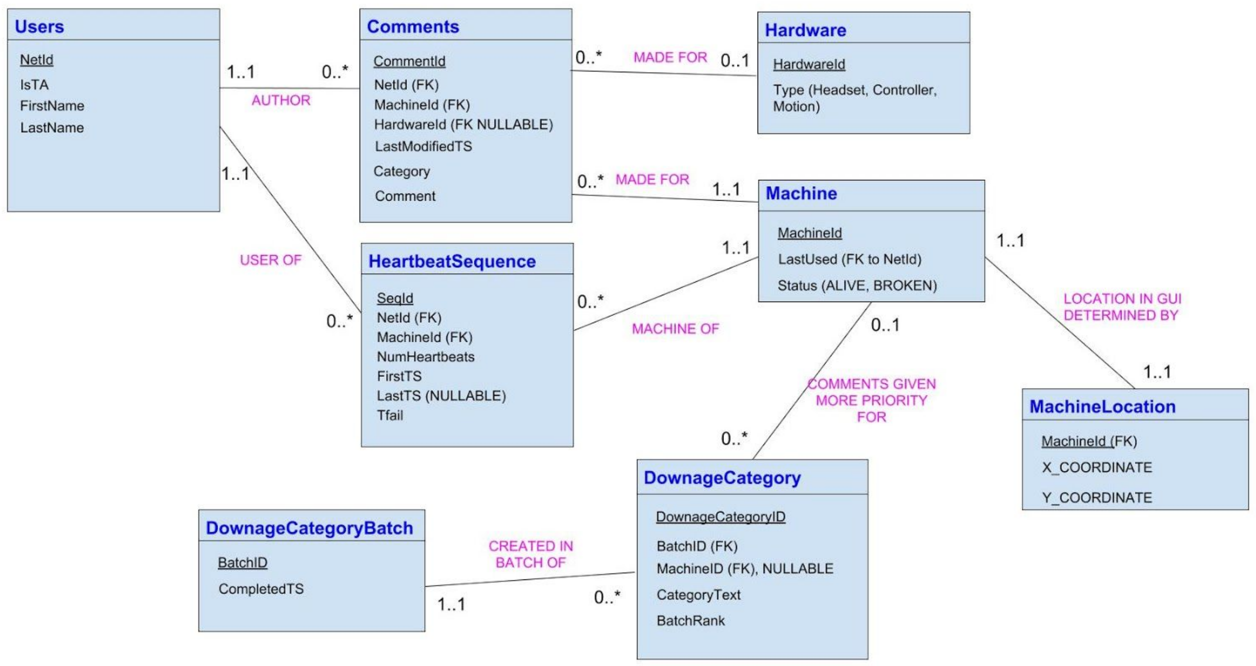


This is a screenshot from our website showing the grid of machines in the lab and their availability. The layout is similar to the diagram of the lab in the whiteboard picture in the previous section. All machines that are green (machines 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, and 15) are AVAILABLE, all machines that are yellow are IN-USE (machine 1), and all machines that are red are BROKEN (machine 10). So a student could log into our website and figure out that she should work on machine 6 because it is operational and no one is currently using it...all without leaving her apartment.

When a student has decided to come to the lab and work on workstation M, they can log into our website (users are loaded by TAs) and specify that they are working at machine M. We use the concept of heartbeating https://en.wikipedia.org/wiki/Heartbeat_(computing) so that as long as that student has an internet browser open to the main page of our website after logging in, the browser will continually send heartbeats to the server. Using these heartbeats, all other users can see that machine M is taken because the original student's browser is still sending heartbeats. When the user leaves the lab, the heartbeats will time out and the workstation will be marked as available again (assuming no unresolved comments exist for machine M).

If there is a problem with a workstation, students can create comments detailing what is wrong with their workstation, and later TAs can start resolving those issues by searching for unresolved comments using various search criteria.

ER/UML Diagram and Schema



Meaning of Entities

- A **User** represent students and TAs of the VR class.
- A **Machine** represents a single workstation/PC within the VR lab.
- A **Hardware** is a piece of additional hardware used in the VR lab (like an Oculus headset, XBOX controller, or Motion controllers). We deliberately do not have a relation between Hardware and Machine because the peripherals tend to move around the lab as students move around the lab...so XBOX Controller with Hardware id 7 may be at Machine 18 today but at Machine 2 tomorrow. We instead use Comments to track the last known location of a specific piece of Hardware to a Machine.
- A **Comment** is a text comment describing a current issue with a machine (and optionally, Hardware)...such as “The headset at Machine 13 fails to calibrate”. Comments can be changed to resolved or unresolved at a later date (e.g. when the TA updates the software and the headset can now be calibrated again) and can be

categorized based on “downage categories”, which we talk more about later in the report.

- A **HeartbeatSequence** represents an unbroken sequence of heartbeats made for a given User at a given Machine (https://en.wikipedia.org/wiki/Heartbeat_(computing), see CS 425). When a user logs in to our website, their browser will periodically send POST requests acting as heartbeats so we can know if a user is still using a machine or not. If the User of a HeartbeatSequence sends a heartbeat within Tfail amount of time since the last heartbeat was received, we increment the NumHeartbeats counter by one for the most recently updated in HeartbeatSequence corresponding to that machine. Otherwise, we start a new heartbeat sequence by inserting a new row into the HeartbeatSquence table. This strategy allows us to track machine usage in a stateless manner.

  We can check if anyone is using a particular Machine M at time t.now by looking at the most recently updated heartbeat sequence for M and seeing when it was last updated at t.hb and its timeout value of Tfail. If t.hb + Tfail < t.now, then we conclude that machine M is available because the timeout for heartbeats (Tfail) has been surpassed. Otherwise, we received a heartbeat within the timeout so someone is still using the machine.

- A **MachineLocation** describes where a given Machine should be rendered on the Konva canvas in our webpage. Using this relation, we allow TAs to modify where machines are located to best map to where the workstations are in the actual VR lab and persist these changes on webpage reload.

- For one of our advanced functions, we wanted to generate useful categories of comments that could be represented by unigrams/bigrams and dynamically present them to the user when they go to enter/edit a comment. These categories would allow users to quickly categorize their comments and TAs could use these groupings of comments to see what they might need to change (If a lot of comments are talking about “XBOX”, then we might need to order new controllers). These generated categories are represented by a **DowndageCategory**. Later in this report when we talk about Advanced Functions, we will explain more in detail how these downage categories are computed.

- A **DownageCategoryBatch** represents when the batch job to compute downage categories completed. In this batch job, we compute downage categories for the whole lab by a corpus of comments from all machines (in a simple batch), and also computing downage categories on a per-machine basis by partitioning the corpus based on which comments were made for machine M and all other comments (in a mixture batch). This will be discussed in more detail later in the report.

Discuss Data in DB and How It Was Obtained

In the Users table, we have 39 users with their LastName, FirstName, NetID and whether he is a TA. For this part of data, to make things “real”, we obtained all the TA data from the this semester’s VR course website. And for all the students data (a user is either a TA or a student), we lookup the current Piazza page of the VR course, find some students’ names, look up their NetID and then inserted into our table.

For data inside the Machine table and the Machine location table, we have all the data inserted by ourselves. The machine table contains the MachineID, the last user and whether it is alive or not. There are 20 rows inside the machine table, with machineID from 1 to 20, and all last user set to null and status set to ALIVE. Similar to the Machine location table, we have MachineID with range from 1 to 20, together with their X and Y coordinate on the Konva Canvas, we set the coordinates the way that they appear an approximate machine layout of the VR lab on our Konva Canvas.

Besides the Machine table, we also have a Hardware table with all the hardware data of all the machines. Basically, each machine has 2 hardware, a headset and a controller. Hence, there are 40 rows inside this table (2 per machine, and we have 20 machines in the VR lab). We inserted the hardware data manually.

The HeartbeatSequence table contains data that shows usage time window of all the machines with a sequence ID, Tfail, start timestamp, end timestamp, number of heartbeats, the user NetID and the corresponding Machine ID. The mechanism of heartbeat sequence is already discussed in above section. For each data, we set the default Tfail to be 5 minutes, and the default start timestamp (i.e. the FirstTS) to be the current timestamp. The seq ID is set to be auto incremented by 1. And for each update, the end timestamp (i.e. the LastTS) is set to be the current timestamp by default. Some of the data inside this table is inserted by ourselves for initial testing. We also ask some friends to manually post data to this table or login to our web page which will automatically send ajax call to our database and update or insert to this table accordingly.

The data inside the Comments table records each of the user made comment with the comment category, the content of the comment, whether it is resolved, the related hardware ID, the NetID of the user who made the comment, and the machineID associated with the comment. In the Comments table, we insert some of the comments ourselves for initial testing. And we ask some friends to post some comments on our website.

Feature Specs of Our Website

Configurable Locations of Machines via Drag and Drop

- TAs can drag and drop machines to place them so they are laid out according to their configuration in the lab and persist these changes on page reload
- All other users will see the updated locations upon refreshing their page

Track Issues of Machines via Comments

- Students and TAs can write comments describing what is wrong for a given machine without leaving their computer. These comments are stored and can be analyzed by TAs to prevent issues the next semester
- Comments can be searched by any combination of comment author, whether a comment is resolved or not, machine id, and case-insensitive matches of downage category or comment text. Sort comments by last modified time to respond to the most recent issues of machines
- Comments can be updated or deleted by the author of the Comment

Create new users via our login page

Dynamically Determined Downage Categories

- Via a POST route, calculate downage categories (unigrams and bigrams that summarize common problems like “Unity” or “Oculus Update”) based on the most recently updated comments
- Downage Categories can be calculated lab-wide or calculated such that for machine M, we give more weight to words in comments made for Machine M than for other machines
- Downage categories for machine M are presented to users when creating a comment or updating a comment for machine M

Machine Availability at a Glance

- Colored indicators for each user quickly show users which machines are BROKEN, AVAILABLE, or IN-USE
- If there are any unresolved comments for a machine, mark that machine as BROKEN and colored red so TAs know they have to fix that machine and students know that machine is not operational
- Use **heartbeats** to determine if a student is still using a machine: students can check if there are available workstations in the VR lab before leaving their apartment. Students in the lab just have to minimize their web browser on our main page for the heartbeats to keep periodically being sent, and when the student leaves the lab their workstation will be marked as available once the heartbeats timeout.

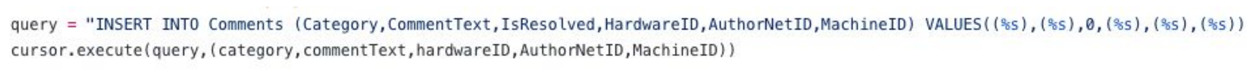
Convenient REST API so other applications can query our data

Basic function & SQL Snippets

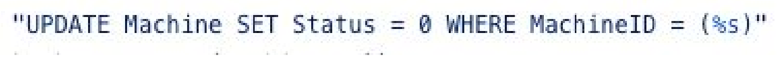
COMMENT CRUD

CREATE

Creating a comment requires sending a payload as a JSON object in a POST request to our API. This object is parsed and a corresponding INSERT SQL query is created which is then executed through our connection to the database that we opened upon receiving the request. This will also cause the machine which the comment was created on to be marked as BROKEN since new comments are unresolved upon creation. The machine is set through an UPDATE SQL query.



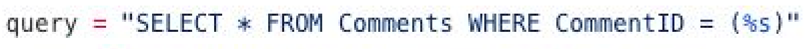
```
query = "INSERT INTO Comments (Category,CommentText,IsResolved,HardwareID,AuthorNetID,MachineID) VALUES((%s),(%s),0,(%s),(%s),(%s))
cursor.execute(query,(category,commentText,hardwareID,AuthorNetID,MachineID))
```



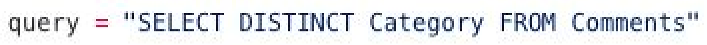
```
"UPDATE Machine SET Status = 0 WHERE MachineID = (%s)"
```

READ

Reading a comment requires just a comment_id. With that comment_id, a GET request is made to our API which will cause a connection to be made with our database and a SELECT SQL query is made based on the comment_id. The found comment will be returned as a JSON object which is then read . Similarly, it also possible to read all the downage categories found in our comments table, but this does not require a comment_id.



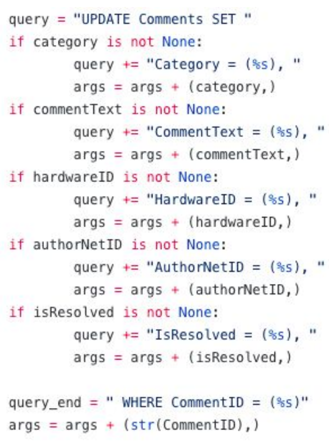
```
query = "SELECT * FROM Comments WHERE CommentID = (%s)"
```



```
query = "SELECT DISTINCT Category FROM Comments"
```

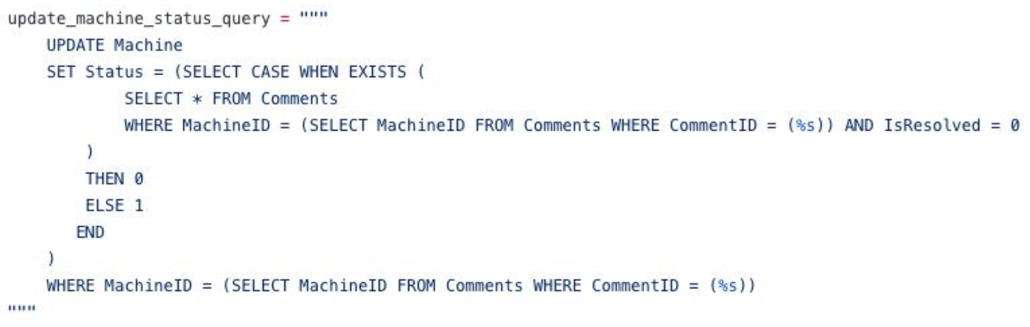
UPDATE

Updating a comment also requires sending a payload as a JSON object as with **CREATE** but this time in a PUT request to our API. Note that a comment_id is also required. The JSON object is parsed and an UPDATE SQL query is created with the fields found in the object. Some fields are optional, so every attribute in the Comments relation does not need to be updated. After creating the query, it is executed through our connection to the database. If upon, updating a comment, it is found that on all comments on the machine that we are currently updating a comment on have been marked resolved, then the machine is marked ALIVE otherwise it is kept BROKEN.



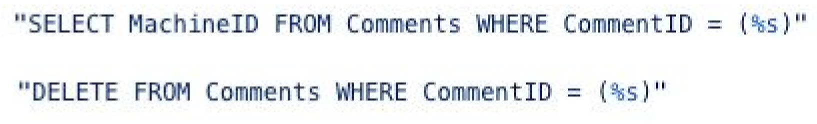
```
query = "UPDATE Comments SET "
if category is not None:
        query += "Category = (%s), "
        args = args + (category,)
if commentText is not None:
        query += "CommentText = (%s), "
        args = args + (commentText,)
if hardwareID is not None:
        query += "HardwareID = (%s), "
        args = args + (hardwareID,)
if authorNetID is not None:
        query += "AuthorNetID = (%s), "
        args = args + (authorNetID,)
if isResolved is not None:
        query += "IsResolved = (%s), "
        args = args + (isResolved,)

query_end = " WHERE CommentID = (%s)"
args = args + (str(CommentID),)
```



```
update_machine_status_query = """
    UPDATE Machine
    SET Status = (SELECT CASE WHEN EXISTS (
            SELECT * FROM Comments
            WHERE MachineID = (SELECT MachineID FROM Comments WHERE CommentID = (%s)) AND IsResolved = 0
        )
        THEN 0
        ELSE 1
       END
    )
    WHERE MachineID = (SELECT MachineID FROM Comments WHERE CommentID = (%s))
"""
```

DELETE
Deleting a comment just requires a comment_id. With that comment_id, a DELETE request is made to our API. Before creating the DELETE SQL query, a SELECT SQL query is first executed through our connection to the database to find the machine associated with this comment. After finding the machine, then the DELETE SQL query is executed in the Comments table where the comment_id is matched. Upon deletion, we also make an UPDATE SQL query to update the status of the machine associated with this comment to check if there are any more unresolved comments and if not, the machine is set to ALIVE otherwise it is kept BROKEN as with **UPDATE**.



```
"SELECT MachineID FROM Comments WHERE CommentID = (%s)"

"DELETE FROM Comments WHERE CommentID = (%s)"
```

Dataflow

Everything starts from the initial login page that the client sees. The client has the option to either login or create an account. Logging in requires a username and the id of the machine that the client is working on. The login process involves checking our database for the username that is entered along with the machine id.If the username entered does not exist, the client is shown a message similar to "User does not exist". If a new account is created, this new user is then inserted into our database assuming it provides all the required fields. After a user is created, they will have to go through the login process.

After the login process is completed, the main page of our website is displayed to them which includes a few things. First, a top-down view of the layout and status of the machines in the lab which is generated through querying our machine availability route which not only provides the status but also the location through a GET request. If the client is a TA, they will have the option to change the layout of the lab using simple drag and drop. After changing the layout to their liking, they should press the "Modify Locations" button which in turn sends a PUT request to our server that will result in communication with the MachineLocations table as needed. If the client is not a TA, the server would deem their changes invalid. This process is described in more depth in the advanced function section of our report.

Second is something that is only visible to a client if he/she chooses to open the developer console. Upon doing so, the client is able to see the Heartbeats that is being sent from the client continuously to our server through POST requests to let our backend know that the machine is occupied. This is used to give real time view of the machine statuses.

Third, there is a view component that allows the client to add a comment on the current machine that they are working on. Adding a comment requires clients to enter text into the form along with a drop down input named category which essentially summarizes what the issue with the currently used machine is. The way these categories are determined are described in more depth in the advanced function section of our report. After adding the comment, a POST request is made to our server that updates the database with the newly created comment. As described in our basic function, this will also lead to the database marking the machine as BROKEN. Now if the client refreshes the page, the machine that they just created on a comment will be displayed in a red color.

Lastly, there is a view component that allows the client to see all the comments made on the machines in the VR lab. Without clicking on the "View All Comments", the client can click on individual machines in the Machine Layout component to see all the comments for that machine. This process involves querying our database for comments on specific machines using a machine id that is found on clicking on the machine. If the user were to click on the "View All Comments" button, then they would be redirected to a new route that displays all the comments made on all the machines in the VR lab. Near the top, there is a form that can be used to filter the comments on specific fields. When the client decides that he/she wants to filter the comments, a POST request is made to our server that uses the provided filters to generate the required HTML elements to show comments using the filters. If the client is a TA, he/she would also be shown a button that allows them to update a comment which makes a PUT request to our server as mentioned in our basic function section.

Advanced Functions

Advanced Function 1: Discovering Downage Categories From Previously Made Comments

Concept/Idea

We defined a *downage category* as a small piece of text categorizing a comment about why a given machine was not functioning correctly. For example, if Unity (the game engine used in CS 498) was crashing on a machine and the comment was “Unity keeps crashing when trying to load MP4”, one downage category could be “Unity“. In our midterm project demo, we gave the user a list of 3 statically defined downage categories to choose from: Machine Problem, Hardware Problem, and Software Problem. Our advanced function uses basic NLP to discover downage categories relevant to each machine, using previously made comments as the input corpus to extract k downage categories.

Implementation

We utilized the gensim Python package https://radimrehurek.com/gensim/ for our text pre-processing and NLP.

We have 2 main methods of computing downage categories using the last MAX_NUM_COMMENTS_FOR_DOWNAGE_BATCH comments: simple and mixture. Both methods use the same pre-processing (see preprocess_corpus.py) where we remove punctuation, numeric characters, and stopwords (stopwords are words that give sentences grammatical structure but to not contribute to semantic meaning, such as “and”, “but”, and “the”). Then, we use gensim’s Phrases class to detect any bigrams that are usually considered as one term (e.g. “hot dog” is consists of two words, but refers to one concept...a hot dog...and Phrases will convert “hot dog” into “hot_dog” to indicate that “hot dog” should be treated as one term). The Phrases class uses mutual information to automatically extract these bigrams: see https://radimrehurek.com/gensim/models/phrases.html and ‘https://svn.spraakdata.gu.se/repos/gerlof/pub/www/Docs/npmi-pfd.pdf for more info on the bigram extraction. Finally, we tokenize the corpus so each comment is represented as a bag-of-words https://en.wikipedia.org/wiki/Bag-of-words_model.

Simple Downage Categories

In the “simple” downage category batches, we grab the last modified MAX_NUM_COMMENTS_FOR_DOWNAGE_BATCH Comments as our input corpus C and perform the preprocessing mentioned earlier. Then we compute the probability of each token (where each token is a bigram or unigram) occuring in the corpus by summing up each token’s occurence and dividing by the total number of tokens in C. We also apply some additive smoothing to prevent zero probabilities. The k downage categories are the k most likely tokens.

Mixture Downage Categories

The “simple” downage category batch provides a good summary of problems with the lab as a whole, but we lose some information about which comments were made for each machine when we aggregate comments into one corpus. To improve upon this, we make the assumption that comments previously made for machine M will slightly influence later comments made for machine M. We base this assumption on that some problems may not be fully fixed for a machine the first time a TA tries to fix it.

This different strategy of discovering downage categories is the “mixture” batch. Like in the simple batch, we grab the MAX_NUM_COMMENTS_FOR_DOWNAGE_BATCH most recently updated Comments as our input corpus C and perform the preprocessing. We also compute the simple lab-wide downage categories as before.

However, we also compute n groups of k downage categories where n is the number of machines that have at least one comment made for them. For each such machine M, we partition C into two corpora: C.M, which contains all the comments made for machine M, and C.rest, which contains all other comments. We then compute token probabilities for C.M and C.rest independently.

Then, we calculate the probability of each token p(w) using the below formula, similar to those used in mixture models:

$$p(w) = p(w, z = M) * p(\alpha) * p(w, z = rest) * p(1 - \alpha)$$

Where p(w, z) is the probability that token w has for occuring in Corpus C.z, and α is the probability of choosing a token from corpus C.M. See downage_category_view.py for our chosen values of α and MAX_NUM_COMMENTS_FOR_DOWNAGE_BATCH.

Via α and by partitioning C into two different corpora and then determining token probabilities, we can give more weight to tokens found in comments made for machine M when determining downage categories for M.

Like before, we use the k tokens with the highest p(w) as the downage categories for M. So combined with the simple lab-wide downage categories that we use as defaults/fallback, the mixture batch produces (n+1)*k downage categories per batch.

Usefulness, Difficulty, and Novelty

The usefulness of this advanced function is that students can easily categorize their comments and TAs can search by categories to see which types of issues are more common. Students and TAs and search comments based on these categories to try and solve similar issues at the same time

This advanced function was technically challenging because of the text preprocessing required and attempting to do NLP on very short documents. We had to spend some time determining how we were going to perform the preprocessing and how we were going to utilize gensim to compute our token probabilities. Also, we had to modify our dataflow to pull downage categories from our database rather than serve up the static list of downage categories.

In addition, most comments for our system will be short sentences like “Headset is broken” or “XBOX controller left button won’t work”, so more traditional NLP techniques for discovering categories such as LDA were less effective. LDA works much better with corpora with long documents because it has more examples to infer topic distributions. We attempted to use LDA to determine downage categories via gensim’s LDA implementation, but found that it gave similar results to the simpler token-probability counting we did before (we tested on mock comment data, as well as on small subsets of roughly 100 tweets from https://www.kaggle.com/kazanova/sentiment140).

The novelty in this advanced function is that the categories are generated dynamically and be recalculated at any time via a POST to /project/downage-category/mixture/startBatch or /project/downage-category/simple/startBatch (say if you wanted to recompute downage categories every day, week). Some forum platforms like Piazza allow you to categorize comments, but those categories must be inputted by the instructors a priori. Our method allows categories to evolve over time as new comments are added/modified.

A nice addition to our project could be a web page to analyze how categories of comments change over time...perhaps comments made in the beginning of the year differ from those at the end of the year, or when MP deadlines are weeks vs days away. This information could help course staff detect patterns and possibly do proactive action to prevent the same issues from occurring the next year.

Advanced Function 2: Configurable Positions of Machines in UI via Konva

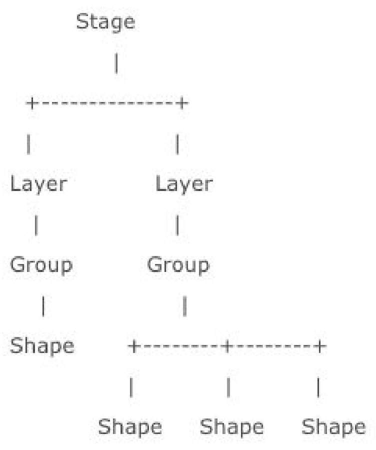
Concept/Idea

The machines in the VR Lab are laid out in a particular way and it is possible that in the future, the layout of the machines could change. For example, currently there are 24 or so machines separated into 4 rows of 6 machines laid out in an adjacent fashion. If the layout were to change to 3 rows of 8 machines, we want this change to be easily configured and displayed in our web application. Essentially this advanced function serves the purpose of adding flexibility to the layout of the machines.

This flexibility was implemented through a drag and drop feature that allows TA's to change the layout of the machines by essentially dragging and dropping the machines into different places.

Implementation

This feature required us to utilize a library called Konva (https://konvajs.github.io/) which we used to create canvases that support interactivity with javascript. The Konva framework requires everything start from a stage (Konva.Stage) that will contain user created layers (Konva.Layer). The layers contain objects defined by the user and these objects can be anything supported by Konva. The objects for our application are the rectangles that were created to represent machines. Each layer will have two renderers, one for the visuals and one for the event handling. Lastly, the stage, layers and shapes are nodes similar to the DOM nodes in HTML. An example of the hierarchy can be found below.



In regards to our usage of Konva, there were three things we needed to implement. The first being a grid structure that was not only visible to the user but also could be used by the canvas to determine placement of defined objects. Then, the actual machine itself as modeled by a rectangle shape as mentioned earlier. After that, creating a shadow of each machine that would show the user where the currently held block would go on the drag-end as a means to improve the user experience. Lastly, implementing the drag and drop feature. Each step in this process is detailed below.

GRID

After creating the initial stage structure required by Konva and defining some variables such as height, width and block size, a grid layer is created. In the grid layer, Line objects are drawn in vertical and horizontal directions in such a way that it mimics a grid and since the Konva canvas is created using a real plane, the lines are drawn in using x and y coordinates. This grid layer will be added as the first layer on our stage.

RECTANGLES

Adding the rectangles to represent machines required us to create an additional layer (rectangle layer). After this layer is created, there is a GET request made to our API that returns the set of machines that must be modeled by our rectangles. This API call will return an array of JSON objects where each element is a machine along with its location (modeled by (x,y) coordinates), the status of the machine (used to color the rectangles and show users machines that are available, in use or broken) and an identifier. For each of these machines, a new rectangle object is created with the required attributes. Displaying the machine identifier on each machine required us to use to toolTips which is an object that displays text on screen. Also every machine is inserted into a data structure that is used to keep track of the machine locations when we change the layout using the drag and drop feature. Then after creating all the machines, they will be placed onto the newly created layer. The rectangle layer will be placed on top of the grid layer.

SHADOW

Creating a shadow behind each machine as it is dragged involves creating a new rectangle with attributes that represent a “shadow” that is shown under each machine when it is dragged and hidden as the machine is placed on a drag-end event. This was also placed into the rectangle layer.

DRAG AND DROP

Konva allows drag and drop by defining the functionality on the shapes itself through attributes and object methods. This meant adding an attribute called ‘draggable’ to each rectangle and then defining the drag-start and drag-end functions as needed. In both of these functions, there needed to be some manipulation of the structure of our canvas which included adjusting the shadow, creating toolTips to display text, calculating new location and updating the data structures used to hold location.

After some modification of the layout of the machines, a user would press a submit button that would send a POST request to our API to update the backend as needed. Note here that the back end does some verification make sure the person trying to make the modification is a TA.

Usefulness, Difficulty, and Novelty

The reason that this feature is useful other than adding flexibility to the web application is that it creates a natural mapping between the UI and the layout of the machines. This feature will also conform to our natural tendency to want to drag and drop things in our everyday usage of technology. It also allows for simplification because creating a fixed grid in the application would require the users to actually “write code” if the layout were to ever change whereas this feature requires much less of the user.

This advanced function was technically challenging because not only did it force us to learn a whole new library but also required us to bring together a lot of individual pieces that needed to work together. This can be seen in almost every aspect of the drag and drop feature from having a grid that would “snap” moved machines into place to deciding when to show/hide each layer in our canvas through the renderers.

This feature is novel when compared to other systems in our university that involve a layout of some sort in that it allows the layout to change very easily. For example, consider the computer based testing facility in Grainger, the layout of the machines have been “hard-coded” into the view that the proctor looks at and it is not possible for proctors to easily rearrange the machines if a layout change needed to happen.

Technical Challenge We Encountered

The most disruptive technical challenge was trying to deploy our code onto cPanel. The documentation for installing a Python web framework (like Flask or Django) is incomplete on the Illinois wiki space:
https://wiki.illinois.edu/wiki/display/cpanelsoft/Advanced%3A+Programming%2C+databases%2C+and+more

We had to figure out for ourselves how to perform these 3 tasks:

- Run a Python web framework while it was hosted on cPanel, with configurable environment variables. On other hosting platforms you could use a Dockerfile, but the entry point to code hosted on cPanel is not straightforward.
- Python package/virtual environment management on cPanel
- How to apply code changes to the live website

I ended up making a wiki for our team a couple of months back to document the process in case we needed to start from scratch https://wiki.illinois.edu/wiki/display/cs411sfa18/cPanel+Setup . We summarize those steps here.

Running a Python Web Framework on cPanel

First, you'll want to need to create a "Python App". This will create a directory and Python virtual environment that cPanel uses for your code.

1. **Log in to your cPanel account dashboard** at https://web.illinois.edu/.
2. In the **Software** section of the dashboard, click on the **Setup Python App** utility.
3. **Choose the Python version** you want from those available.
4. Enter a **directory name** in your home directory where you would like your application files to be stored.
5. Enter the **name your application will be referred to** in the browser. The domain is already provided for you.
6. Click **Setup** and wait for the page to update.

When it is done with the configuration, your new application will be listed under the "Existing applications" section.

Now go to the cPanel account dashboard and click **Terminal**. Inside of the terminal, cd to the directory you created in step 4 and clone your git repository from there.

Now we need to do some configuration so cPanel knows where to hook up your WSGI application (Flask or Django's app). In the file where you define your Flask app (say gitrepo/myapp.py), save that Flask() object under the variable name **application**

```
from flask import Flask

app = Flask(__name__)

# Define routes and Flask related stuff here ...

@app.route("/")

def hello():

      return "Hello World!"


# WSGI of cPanel needs the callable Python object to in a variable called
application

application = app
```

Now edit passenger_wsgi.py (located in the project directory cPanel generated for your Python App) to import your Flask application

```
from gitrepo.app import application
```

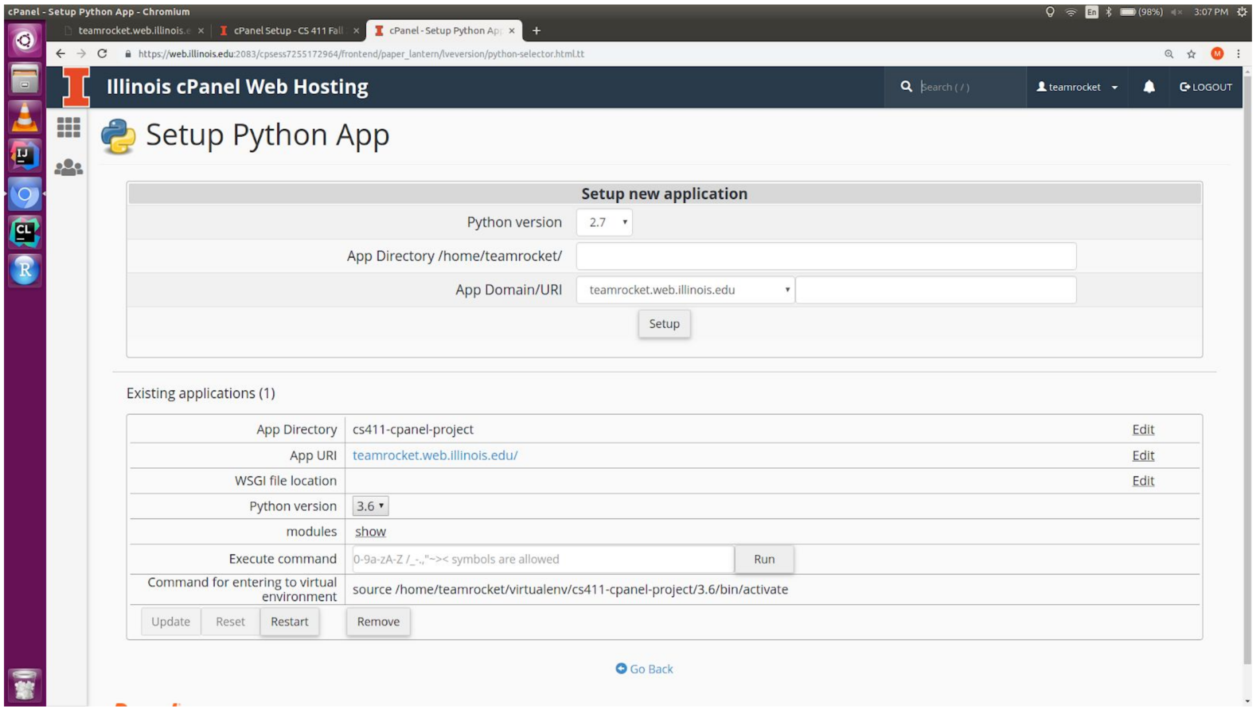
If you restart your Python app you should see that your application is live at the URL given for your Python App by cPanel. We couldn't figure out how to run a web framework at a different port other than the default HTTP port (80).

Note that If you need to have any environment variables in your application Python code (like DB user credentials), modify passenger_wsgi.py such that before the `from gitrepo.app import application` line of code, you use os.environ[] to export any environment variables you need in your application code.

Python Package/Virtual Environment Management in cPanel

cPanel uses venv for its virtual environment and pip for package management. In order to have the Python interpreter that cPanel will invoke to run your code have access to custom python libraries like Flask, Dango, or gensim, you'll need to define a requirements.txt file in your git repository containing those package definitions.

To install your custom list of Python packages, you'll have to activate the virtual environment for your Python App and then install your packages. To find out which virtual environment you have in your cPanel Python App, go to "Setup Python App" and find the list entry for the Python App you set up earlier. Copy the path to the *activate* binary given for your python app (in the below picture, we want to save the path /home/teamrocket/virtualenv/cs411-cpanel-project/3.6/bin/activate)


cPanel - Setup Python App - Chromium
teamrocket.web.illinois.e
cPanel Setup - CS 411 Fall
cPanel - Setup Python App
https://web.illinois.edu:2083/cpsess7255172964/frontend/paper_lantern/lveversion/python-selector.html.tt
Illinois cPanel Web Hosting
Search (/)
teamrocket
LOGOUT

Setup Python App

Setup new application

Python version	2.7
App Directory /home/teamrocket/	
App Domain/URI	teamrocket.web.illinois.edu

Setup

Existing applications (1)

App Directory	cs411-cpanel-project	Edit
App URI	teamrocket.web.illinois.edu/	Edit
WSGI file location		Edit
Python version	3.6	
modules	show	
Execute command	0-9a-zA-Z /_-.,"~>< symbols are allowed Run	
Command for entering to virtual environment	source /home/teamrocket/virtualenv/cs411-cpanel-project/3.6/bin/activate	

Update Reset Restart Remove

Go Back

Now open up a **Terminal** from the cPanel account page and cd to the directory of your Python App. Run

```
source path/to/activate/from/the/previous/step
```

From the terminal to activate the virtual environment. Then cd to the directory containing your git repo's requirements.txt and run

```
pip install -r requirements.txt
```

Now restart your Python app from the "Setup Python App" option in the cPanel account dashboard and your code will now be running with the packages defined in requirements.txt. If you ever update your requirements.txt, make sure to repeat these steps to update the Python interpreter cPanel is using.

Deploying Changes To Your Code to cPanel

To apply changes to your application code, go to the **Terminal** from the cPanel account dashboard website and cd to the directory for your cPanel Python App. Then pull/checkout whatever Git branch of code you want to deploy.

IMPORTANT: If you want to have your changes to code go live, you'll have to restart the cPanel Python App from the "Setup Python App" dashboard. Otherwise your changes won't be applied.

Talk About If Everything Went According To The Initial Plan...If Not, Why?

Our project came along fine during development and went mostly according to the initial plan. The only hiccup we had was that sometimes Adam would want to test if the front-end code was working correctly but it was dependent on the backend API routes being finished, so sometimes Adam was blocked by that. A solution for next time would to check in code that returned a mock version of whatever real data would be taken from the database, so that even if the backend wasn’t fully ready yet, the frontend code wouldn’t be blocked by that fact.

Describe Division of Labor

- Mike
    - Python/SQL implementation of Downage Categories
    - cPanel configuration of MySQL and python connector
    - Have Comment CRUD update Machine.Status appropriately
    - JOIN Machine with HeartbeatSequence to determine the availability of all machines
    - Go to person for development questions
- Adam
    - All Python flask front-end development
    - ER Design
- Thomas
    - Drag and Drop (Konva)
    - Initial CRUD of basic entities (Comments, Users, Machines)
- Hop
    - Heartbeat implementation
    - Machine location backend
    - Obtaining data from VR website and Piazza page

Teamwork Logistics

We used Github to code collaboratively ->
https://github.com/MichaelRechenberg/cs411webproject

We used Google Docs to compose our final project report and Facebook Messenger to communicate throughout the semester