PennCloud

Peter Akioyamen, Bilal Ali, Akanksha Ashok, Milan Filo

CIS 5050 - Software Systems (Spring 2024), University of Pennsylvania

Abstract

This final report details the design, architecture, and features of our distributed cloud platform PennCloud.

Introduction

Approach

PennCloud is a cloud-based system designed to provide web services, including: file storage, mail, and administrative functions. As an overview of its functionalities, PennCloud handles user requests through a load-balanced network of frontend servers that is connected to a backend system where data is stored through key-value store abstraction.

Division of Labour

- Peter Akioyamen - Coordinator, UI, Authentication, Utils
- Bilal Ali - HTTP Server, Backend Servers (standard operations [get, put, etc], replication, checkpointing, recovery)
- Akanksha Ashok - Admin Console, Drive, Utils
- Milan Filo - Load balancer, Mailbox, SMTP Relay

Architecture

Overview

The major components of this project include:

1. Key-value store (backend)
2. HTTP Server
3. Mailbox & SMTP Relay (front end)
4. Drive (front end)
5. User accounts
6. Admin console

At a high level, users connect to the system via their web-browser, and their initial requests are intercepted by the Load Balancer (LB). The LB evaluates the current load and availability of the frontend servers and routes the user's request to an appropriate Frontend Server (FE). This decision is informed by regular heartbeat messages from the frontend servers that include an FB's workload and status. The frontend server runs on an HTTP protocol and processes the user's request. This can involve multiple types of operations, such as file uploads/downloads, email access, or administrative functions.

For operations requiring data retrieval or storage, the frontend server interacts with the backend Key-Value Storage (KVS), executing operations like data fetching, storage, or updates. A central Coordinator oversees the backend servers, which are organized into clusters and maintains a map of which backend servers (primary and secondary) are responsible for specific key ranges of the KVS. Primary servers in each backend cluster handle write operations, while secondary servers handle reads and serve as backups for data replication and recovery. An admin console manages and provides an overview of the system.

Components

In the following section, we detail design choices for each component, and any relevant implementation details.

KVS

Coordinator We have a coordinator running on its own server. The coordinator plays a critical role in the following situations:

When a client requires data for some row key, the frontend server will query the coordinator. The coordinator will select a server within the group assigned to that row key, and all subsequent requests that are within that key range will be directed to that server. This removes the coordinator from the critical path, ensuring that it is not involved in read and write requests.

The coordinator also plays an important role in backend server initialization, detecting liveness, and assisting with recovery. When a backend server is initialized for the first time, it sends *INIT* to the coordinator. The coordinator communicates the server's key range, whether it is a primary or secondary, and information about its group (address of primary and secondaries in the group). The coordinator maintains up-to-date knowledge of each backend server group via heartbeats. Each backend server sends *PING* at a 1 second interval. If 5 seconds pass and the coordinator does not receive a heartbeat from a backend server, it will mark that server as dead and broadcast a message to all live servers in that group. In the case that a primary fails, the broadcast message will contain an updated primary, which it will select at random from the list of live secondaries in the group. In the case that a secondary fails, the primary will remain as the group's primary, and all servers in the group will receive an updated list of secondaries. When a previously dead server starts back up, it will send *RECO*

message to the coordinator. The coordinator will respond with the address of the primary server for that group. More details about the recovery process can be found below in the recovery section.

Operations The KVS server can handle the following read and write operations:

- **GETA()** - Retrieve all rows in a tablet. This is used exclusively by the admin console to retrieve a server's data.
- **GETR(r)** - Retrieve all columns in row r.
- **GETV(r,c)** - Retrieve a value in column c of row r.
- **PUTV(r,c,v)** - Store value v in column c of row r. If r does not exist, it will create it.
- **CPUT(r,c,v1,v2)** - Store value v2 in column c of row r if the current value in column c of row r is v1.
- **DELR(r)** - Delete row r.
- **DELV(r,c)** - Delete column c of row r.
- **RNMR(r1,r2)** - Rename row r1 to r2.
- **RNMC(r,c1,c2)** - Rename column c1 of row r to c2.

Backend Server Groups A backend server group consists of 1 primary, and 3 secondaries (although our system is designed to scale up to any number of secondaries). The entire key range is divided evenly across backend server groups. For example, if the key range is A-Z, group 1 would handle A-I, group 2 J-R, and group 3 S-Z. This ensures that request loads are distributed relatively evenly across backend server groups. The coordinator can select any backend server (primary or secondary) to handle requests from a frontend server. Each server group further maintains a set of tablets. For example, if group 1 manages the key range A-I, it might have individual tablets for A-B, C-D, E-F, G-H, and I-I. This allows a server group to keep some tablets in memory to handle requests, while evicting others to disk, thus ensuring that a server group's data capacity is not limited by the size of its memory, but instead by the size of disk. Each tablet maintains its own log.

Replication and Consistency Data is replicated within a server group. We use primary-based replication, and follow the remote-writes protocol. Read requests can be handled by any server (primary or secondary), since the same replica exists on each server. Write requests can be received by a secondary, but they are immediately forwarded to the primary. When the primary receives the request, it assigns a sequence number to the request. Therefore, the primary acts as a centralized sequencer for all requests, and ultimately decides the sequence in which write requests are performed. This guarantees sequential consistency of updates. To perform the write request, the primary will initiate the two-phase commit (2PC) protocol to ensure data is replicated across all secondaries in the server group. The steps are detailed below:

1. Primary opens connection with all secondary servers, then writes *BEGN* in its log. It then acquires a row lock for the update and sends *PREP* (prepare) to all secondaries. It will now wait with a timeout of 3 seconds for all secondaries to respond to the PREP command with their votes.
2. Secondaries receive the *PREP* command and attempt to acquire a row lock for the update. If successful, they log *PREP* with the row, and send back *SECY* (indicating that they've voted yes). However, if they're unable to acquire the row lock, they will log *ABRT*, and send back *SECN* (indicating that they've voted no).
3. The primary receives the votes. If any secondary sent back *SECN* (or if the primary itself wasn't able to acquire the row lock), it will log *ABRT*, and send *ABRT* to all secondaries. However, if all secondaries sent back *SECY* and the primary was also able to acquire the row lock, it will log *CMMT*, and send *CMMT* to all secondaries, with information about the update that it wants it to perform. It will then perform the update on its tablets.
4. Secondaries receive either the *ABRT*/*CMMT* command depending on the decision made by the primary. If a secondary receives *ABRT* and it had previously voted *SECY*, it will release the row lock it had acquired previously, log *ABRT*, and send back *ACKN*. If a secondary receives *CMMT*, it will perform the update on the tablet, and then release the lock it acquired and send back *ACKN*.
5. The primary will receive *ACKN* from all secondaries. The result of the operation will be communicated back to either the secondary that forwarded the write request, or the frontend server if the request came directly to the primary.

The above procedure ensures that any write request is replicated across all servers in a server group, ensuring that all servers in a group maintain the same state.

Checkpointing Every 60 seconds, the primary will initiate checkpointing on all tablets. While checkpointing, all write requests are rejected, but read requests can still be processed. We use centralized checkpointing to ensure that each server in a group is checkpointing at the same time. When a checkpoint is initiated, the primary will increment the checkpoint version number and send *CKPT* to all servers with the checkpoint version number. When a server receives *CKPT*, it will checkpoint each of its tablets and then clear the corresponding log. We optimize checkpointing by checking the log first. If the log is empty, a checkpoint is not necessary for that tablet, since no updates have been made since. The server can retain the old checkpoint and just update the checkpoint file's number to reflect the new checkpoint version number. When the server has checkpointed all of its tablets, it will send back *ACKN*. When the primary receives *ACKN* from all servers, checkpointing is complete.

Recovery Our backend servers are designed to be killed and resurrected via ONLY the admin console. We do not currently support the servers being killed via the terminal, since this requirement was not communicated in time for us to implement this feature.

When a server is killed, it will stop all normal operation, including sending heartbeats, receiving requests, checkpointing, etc. It will also clear all of its in-memory state. When a server is resurrected via the admin's *WAKE* command, it will first

send a message to the coordinator asking it for the address of the primary. There are two situations here:

If there are no servers in the group (all group servers were previously dead), the coordinator will respond with the recovering server's address, indicating that this server should now take over as the primary. In this case, the server will use its saved checkpoints and log files (both of which are on disk) to recover its state. When it has completed recovery, it will begin sending heartbeats to the coordinator, alerting the coordinator that this server is now ready receive requests.

If there are servers in the group, then the coordinator will respond to the recovering server with the address of the primary in the group. The recovering server will contact the primary, sending *RECO* along with its last checkpoint number and last sequence number. When the primary receives this message, it will use the checkpoint number and sequence number sent by the recovering server to determine what information it needs to send the recovering server for it to fully recover. If the checkpoint number is out of date (checkpoint number sent by the recovering server is less than the primary's current checkpoint number), the primary will stream the recovering server its entire checkpoint file, along with its entire logs for each tablet. Otherwise, if the checkpoint number is the same, the primary will look through its logs and stream logs with a sequence number GREATER than the sequence number sent by the recovering server. When the recovering server receives this data from the primary, it will build a tablet in memory, using either a combination of its checkpoint and logs on disk and the logs sent from the primary (if the checkpoint numbers were the same), or using only the checkpoint and logs sent by the primary (if the checkpoint numbers differed).

We designed our backend servers to be able to handle write requests even while a server is in recovery. When a recovering server contacts the primary, the primary will add the recovering server's address to a list of recovering servers. Then, when a write request occurs, the primary will forward the write request to the recovering server. The recovering server will place any write requests in a special log. This log will be replayed at the very end once its done recovering to ensure that it has updated state from any writes that occurred while it was recovering.

Frontend/Backend Communication

When a user makes a request for some data, the frontend server utilizes several utility functions. These utility functions implement commands such as *PUT*, *GET*, *CPUT*, etc., along with the row, column, and value key, to communicate with the KVS. For example, a *GET* request handler can use the *kv_get* utility function that fetches a specific value from a KVS server given a row and column by sending a GET command to the backend. On the other hand, for a *PUT* request handler, the function *kv_put* sends a *PUT* command to store or update a value at a specific row and column in the KVS. Additional functions encompass functionalities such as deleting specific columns/rows/values, checking if responses from a KVS operation are succesful, retrieivng columns for a particular row, etc.

Furthermore, these operations allow the coordinator to assign a backend server to the frontend. Once a backend server has been assigned, all subsequent communication will occur directly between the frontend and backend server via a thread that spins off of the backend server's server loop. This ensures that the coordinator is NOT involved in any actual KVS operations and no data is passed through it, which avoids a potential bottleneck in our system.

Frontend

HTTP Server

1. HTTP Protocol: our system supports the HTTP/1.1 protocol as detailed in RFC2616, which includes handling multiple HTTP requests over a single connection and sending appropriate responses.
2. Request Handling: our system implements multiple handler functions that are able to process HTTP requests like *GET* and *POST*, and can handle additional header information. The system responds with appropriate status codes (e.g., *200* for successful requests), headers (e.g., Content-type, Content-length), and the response content itself as an array of bytes
3. Cookie and Session Management: our server checks for cookies in the request headers, crucial for identifying and managing user sessions
4. Encoding and Decoding URLs: the system encodes and decode URLs or parts thereof (paths, queries such as unique identifiers of emails, etc.) to ensure safe and effective communication with our servers and transmission over HTTP

Mailbox

Users are able to view their email inbox, view/delete particular emails, send new emails, respond to emails, as well as forward emails. The Mailbox is designed to handle both receiving and sending emails, integrating closely with the system's backend storage by storing emails directly into the distributed key-value storage system. A user's mailbox occupies a single row in the KVS. A user's mailbox rowkey is represented with their username appended by the *-mailbox* suffix (e.g. *user-mailbox*) with the column key being the UIDL of the email. The contents of the email are stored in the row-column cell. In addition to being able to send and accept mail to/from users within the PennCloud system that is handled by our HTTP server implementation, we incorporate an SMTP protocol that handles mail service external to our PennCloud system using two additional components:

- An SMTP Server adaptation for accepting incoming emails from external clients, such as email applications like Thunderbird.
- An SMTP Client for sending emails outside of our system that uses a DNS lookup to find the Mail Exchange (MX) records for the recipient's domain to determine where emails should be sent.

Drive
Files uploaded to the system are stored in the KVS. Users can view, download, delete, and manage their files and folders, including creating, renaming, and moving them between different locations. The system accommodates a nested folder structures, supports files up to 25MB in size, and handles various file types including text, PDFs, images, and video. In the KVS, this is represented as follows: A user can have multiple rows representing folders in their drive. For a particular folder, files inside this folder are stored in the parent folder. This allows us to recurse through the user's directory, which is especially useful when deleting folders for files. Any leaf folders have no column values. For example, a root home folder can be represented as *user/*. To illustrate this, imagine a user having a folder called "folder" and a "subfolder" stored inside "folder" in their drive, with a file called "file.txt" stored in this "subfolder". As a result, the user would have the following two rows in the KVS:
user/folder
user/folder/subfolder

The contents of the file would be stored in the parent folder directory (for the aforementioned reason), with the column key being the name of the file, i.e. *file.txt*. The user's root folder contains a shadow password and sid file (inaccessible to a user through the front end) to help with authentication. This resembles security practices in most UNIX systems to prevent read/write access beyond a superuser.

Load Balancer
The Load Balancer, whose core runs on a modification of an HTTP Server, handles initial client connections by redirecting clients to one of several frontend servers for load balancing and fault tolerance. The Load Balancer keeps an updated list of active front end servers that is periodically updated based on the last heartbeat of a frontend server.
Clients only need to connect to the Load Balancer's address localhost:7500 without knowing the details of the frontend servers. The Load Balancer monitors the status and workload of these servers, redirecting clients to an active and possibly less busy server using a scheduling protocol. After the first redirect, clients communicate directly with the selected frontend server, unless the server that the client is connected to crashes. Upon server failure, the client is redirected to another active server. The Load Balancer only assumes load balancing of frontend servers. Backend load balancing is handled independently on the backend.

Admin Console
Information of System The console is accessible via localhost:8082/admin/dashboard. The console displays both frontend and backend servers and shows whether they are up or down, enabling the functionality to disable and restart individual servers using a toggle button. The list of frontend and backend servers are communicated initially upon start up by the Load Balancer and the Coordinator respectfully. Additionally, the console also makes it possible to view the stored data in the KVS by showing a table of key-value pairs.

Status Control of Frontend/Backend Servers Servers can be killed via the Admin dashboard. The system kills components of our server through HTTP POST requests that determine the operational status of the system's backend servers or the frontend servers. Based on the status extracted from the form: if the status is 0, it constructs a message *KILL* to send to the server, indicating that the server should be shut down or stopped. If status is 1, it constructs a message *WAKE* to send to the server, indicating that the server should start or resume operation. The admin then checks whether the server identified by component ID is a frontend or backend server using the respective maps, opening communication with that server and sending the constructed message.

User Accounts: Authentication and Session Management
The system manages user accounts and sessions through the KVS. A user on sign-up has their username and password saved across servers within the relevant KVS cluster. User passwords are not stored in plaintext but instead are hashed using the SHA256 algorithm, and then stored. This is to ensure that in the case that any data leaks or breaches were to occur, user data would still remain protected. User's, on login at a later time, are then authenticated using a challenge-response protocol, wherein a random challenge is sent to the client and hashed with the user password, and then checked against the stored password retrieved from the KVS hashed with the random challenge. If the two hashes match, then the user is authenticated and redirected to the Home page, otherwise they are sent a 401 unauthorized error for incorrect credentials. When a user is logged into PennCloud, they are given a new session ID which persists in the KVS. Every subsequent request by this user for a page that requires authentication first checks the session ID cookie provided in the browser against the session ID stored in the KVS, if the two match, the user's request is processed, otherwise the user is logged out immediately as this indicates their session is no longer valid and has become stale. Session cookies have a lifetime of 10 minutes, after which, their session cookies are automatically invalidated. The lifetime of a session cookie is extended at each new request within the 10 minute allotment. If a user is logged in, the IP and port of their assigned KVS server is cached making requests more efficient. If a KVS server fails, and a user was assigned to it, a retry protocol ensures this user's request is routed to a different KVS server within the cluster group, as long as at least one server within the group is still alive.

Summary of Major Design Decisions

To manage **consistency** amid concurrent operations requested by multiple clients, the use of locks was essential. They are implemented at various levels such as on assignment of servers by the Load Balancer or reading from tablets on the back end. We approached **fault tolerance and backend server failure** with regular checkpointing and an append-only log, both of which were saved to disk to ensure the backend servers survived crashes. Both were integral in helping to restore state. The log helped recover the last known state in the event of a server crash by "replaying" logged operations (which included all write operations). To manage the size of the log, periodic checkpoints coincided with the logs being cleared.
When it comes to **data robustness and replication**, for resilience against data loss during server failures, copies of each data tablet were stored across multiple running backend servers. We used primary/secondary **replication techniques** and a **sequential consistency** model due to its intuitiveness and suitability for environments requiring concurrency. The system incorporates mechanisms to ensure these copies remain consistent and synchronized, especially after failures. Aforementioned recovery protocols are in place, such as allowing a server to resynchronize after a crash.

Challenges and Future Improvements

1. **Replica maintenance**: As the size of our KVS grows, replica maintenance becomes a heavier and slower task, especially on larger data.
2. **Query Caching**: Implementing a cache system on the frontend that caches key-value pairs in order to make frontend/backend communication more efficient.

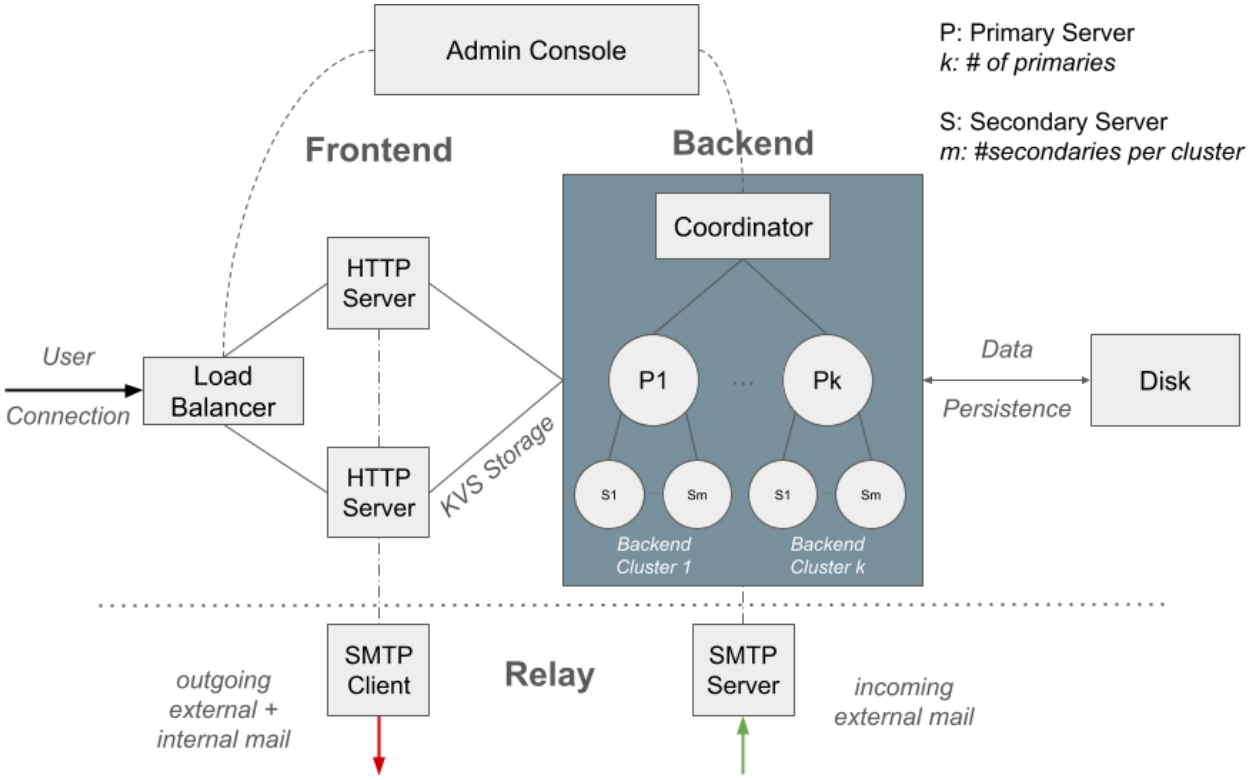

Figure 1. PennCloud Architecture Diagram

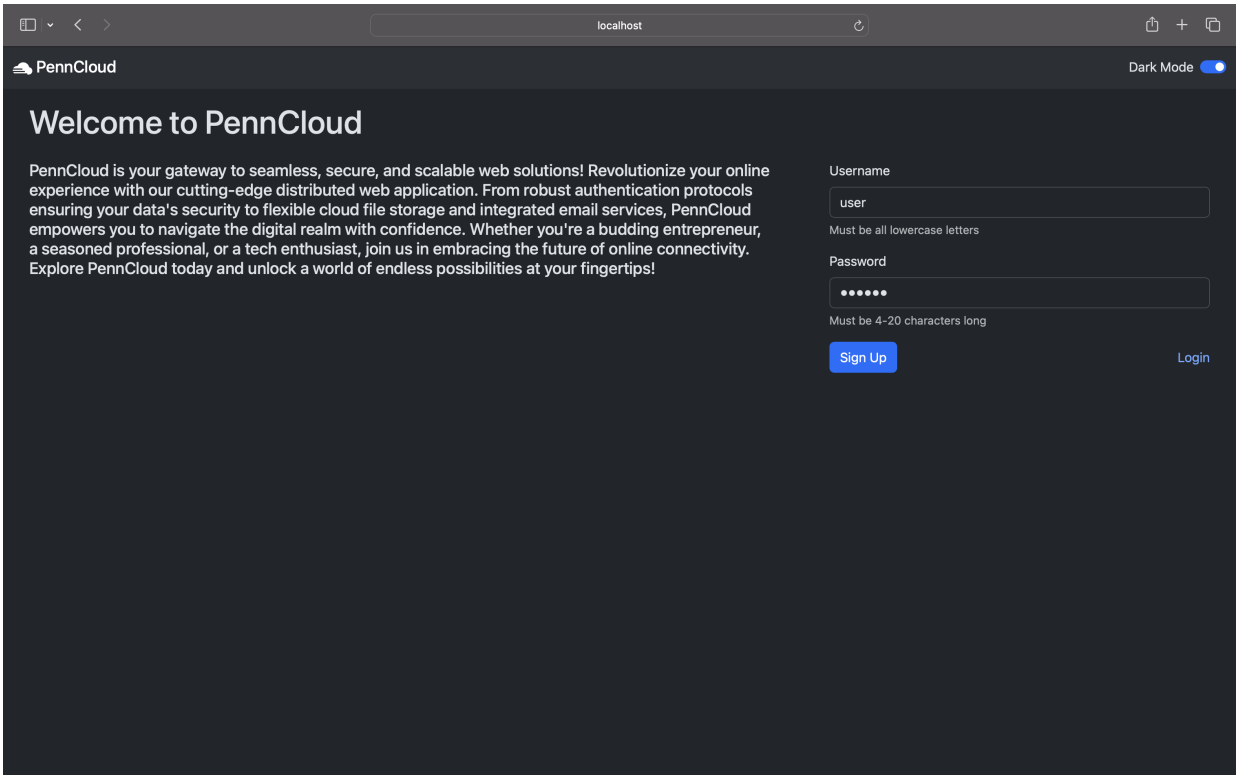

Figure 2. PennCloud signup/login page

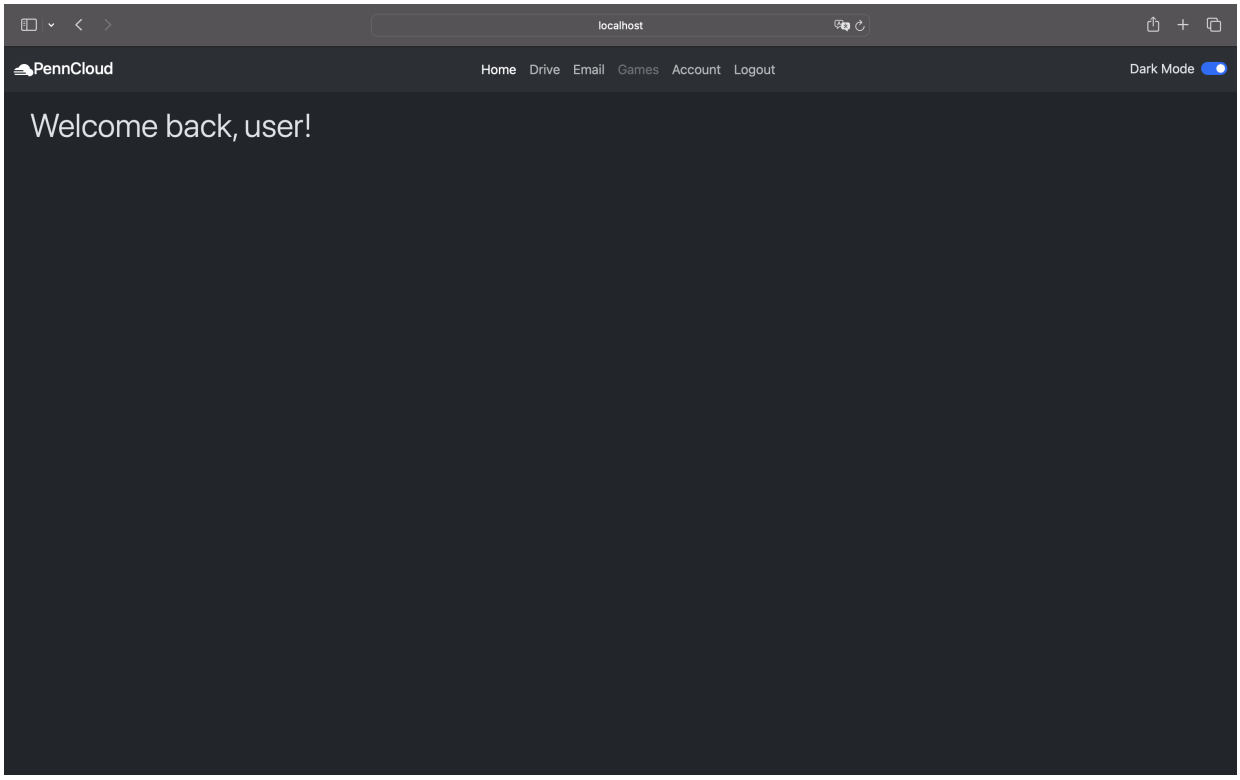

Figure 3. PennCloud home page upon successful login

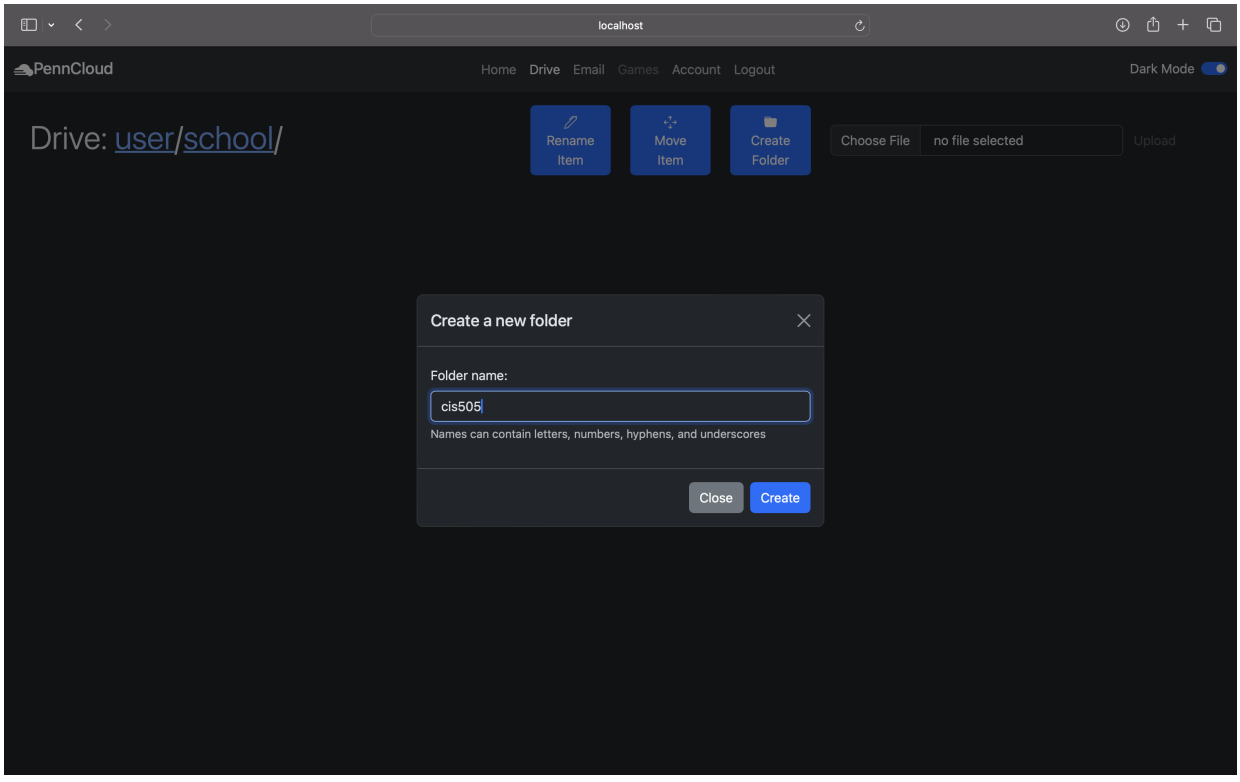

Figure 4. Creating a folder in the user's drive

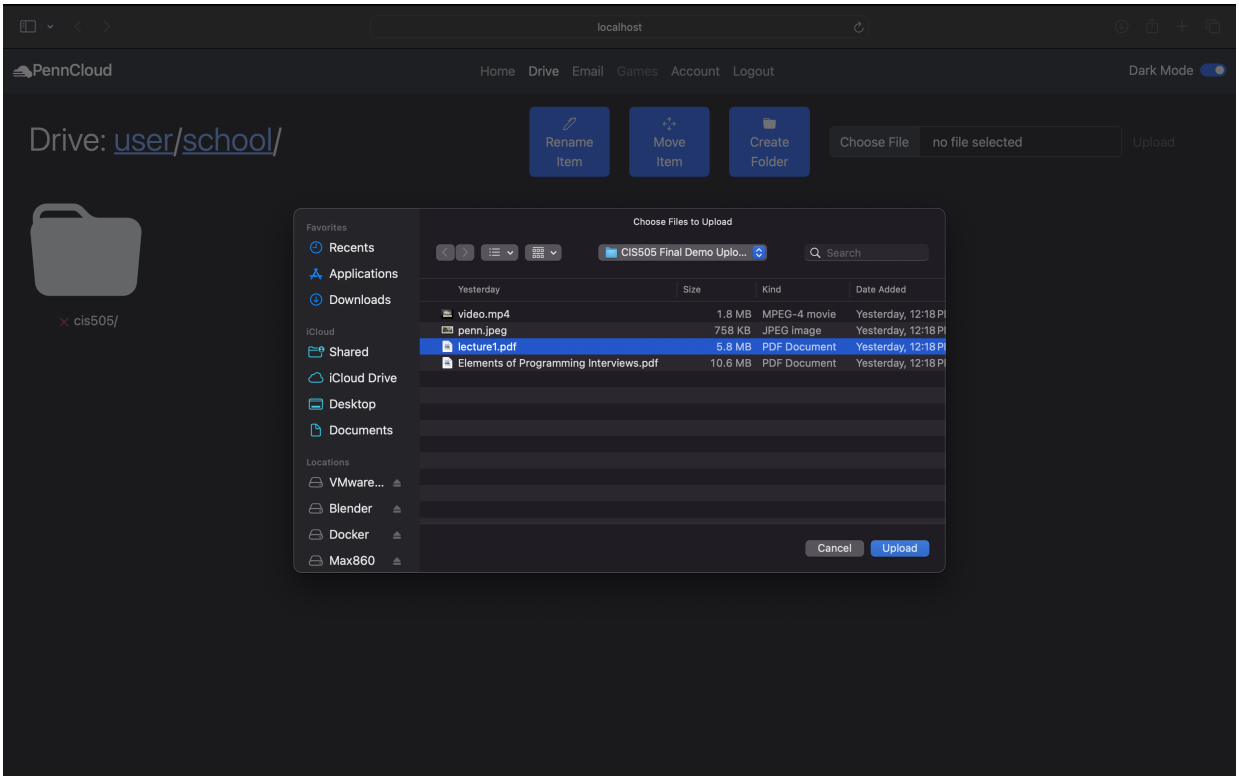

Figure 5. Uploading a file to the user's drive

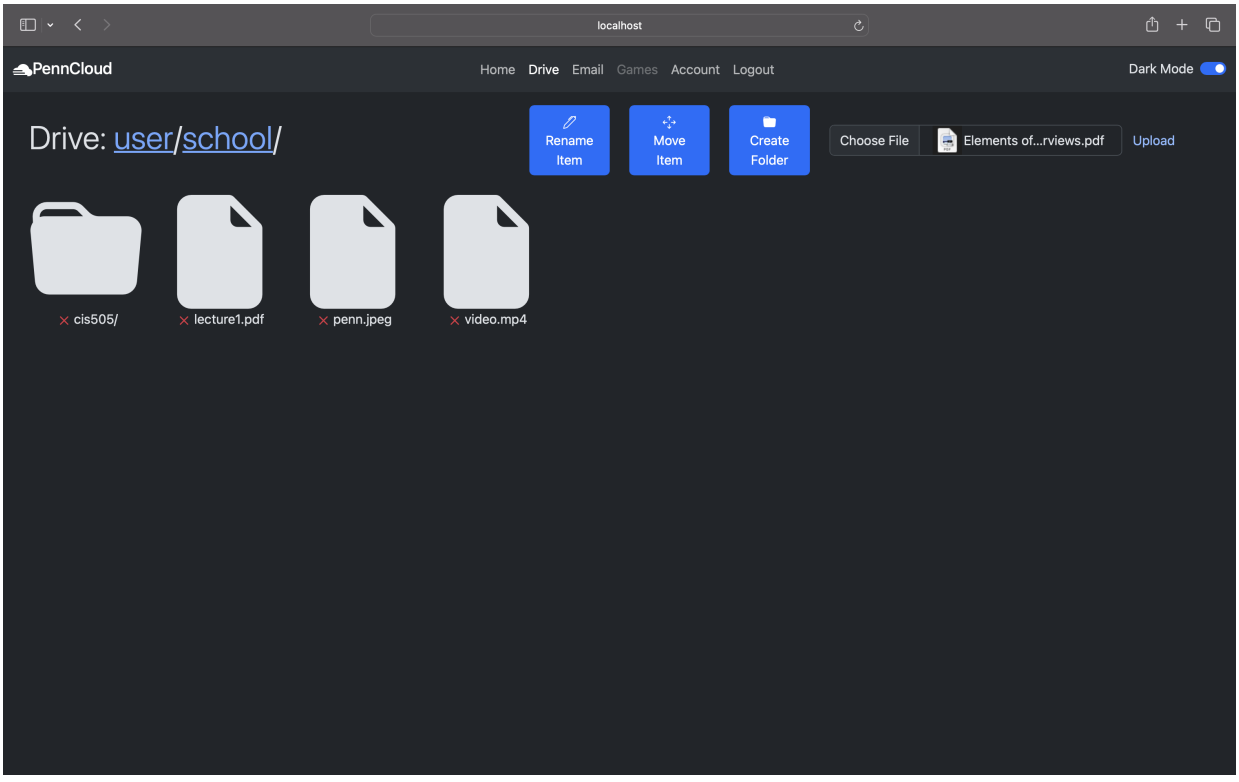

Figure 6. Drive populated with folders and files

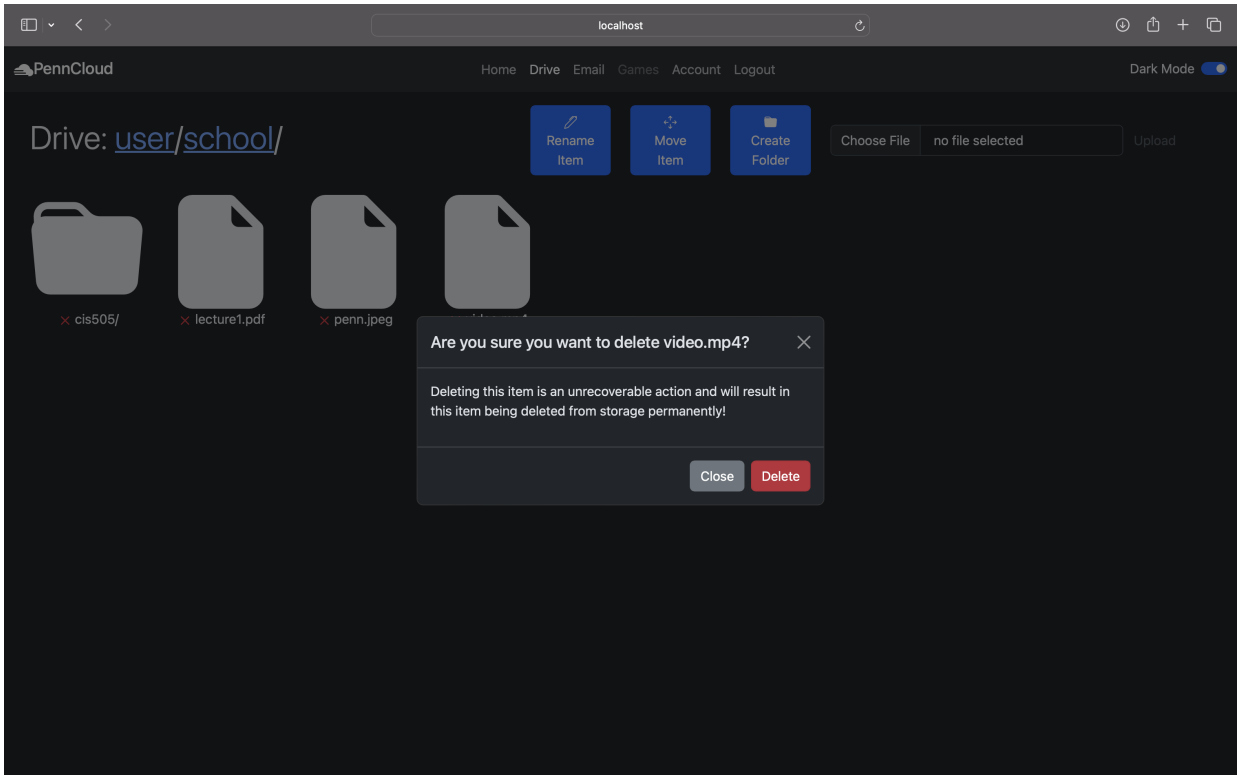

Figure 7. Deleting a file from the drive

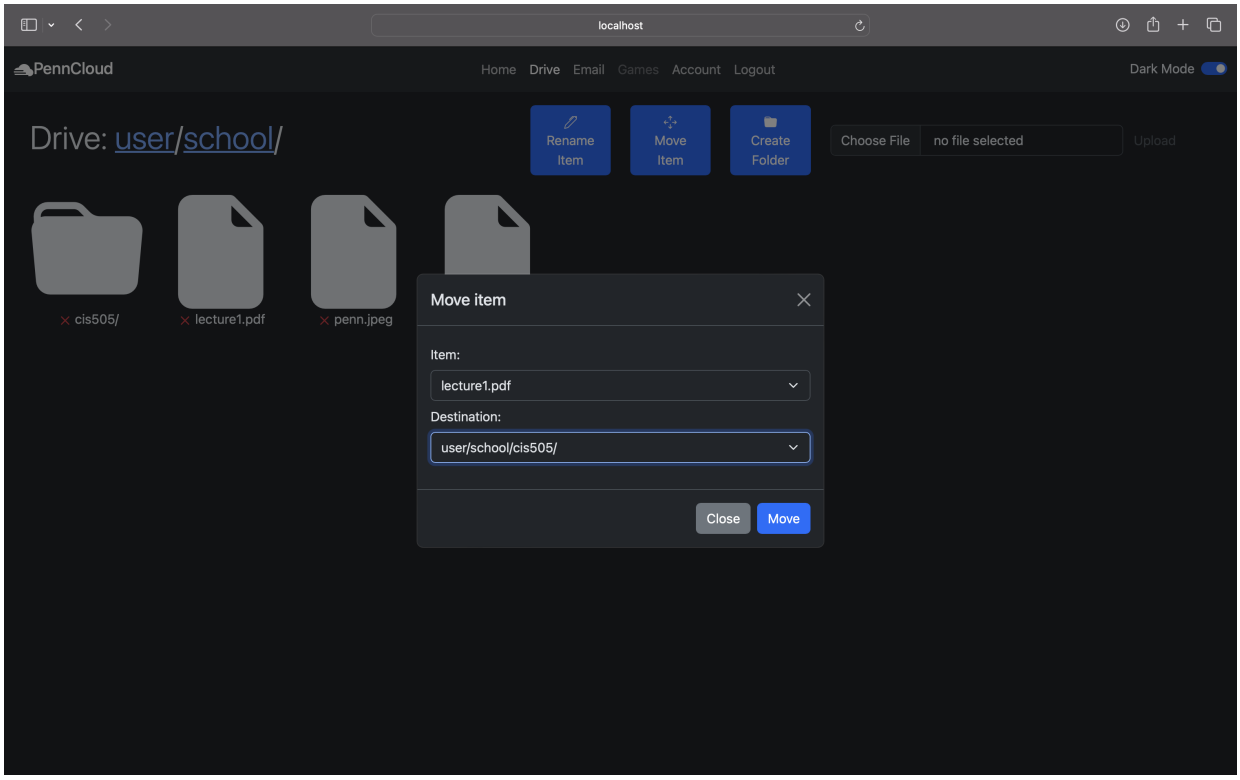

Figure 8. Moving a file to a sub-folder

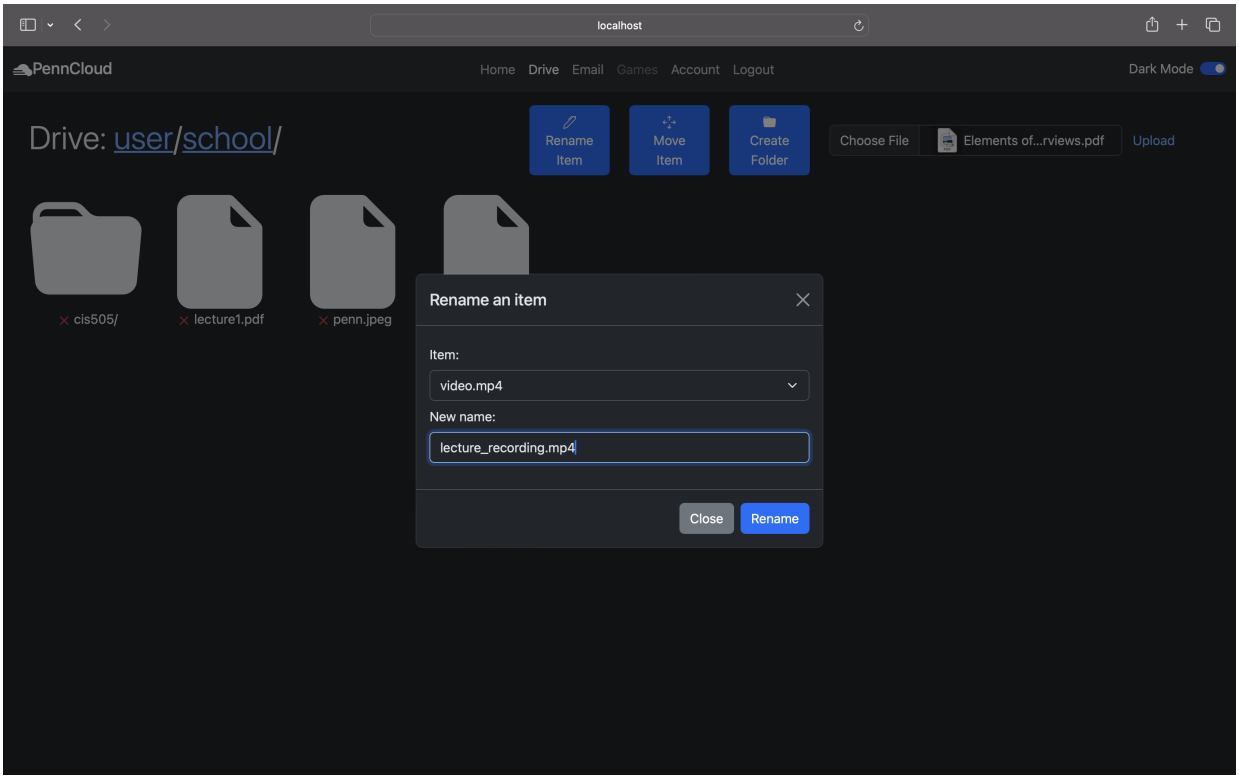

Figure 9. Renaming a file

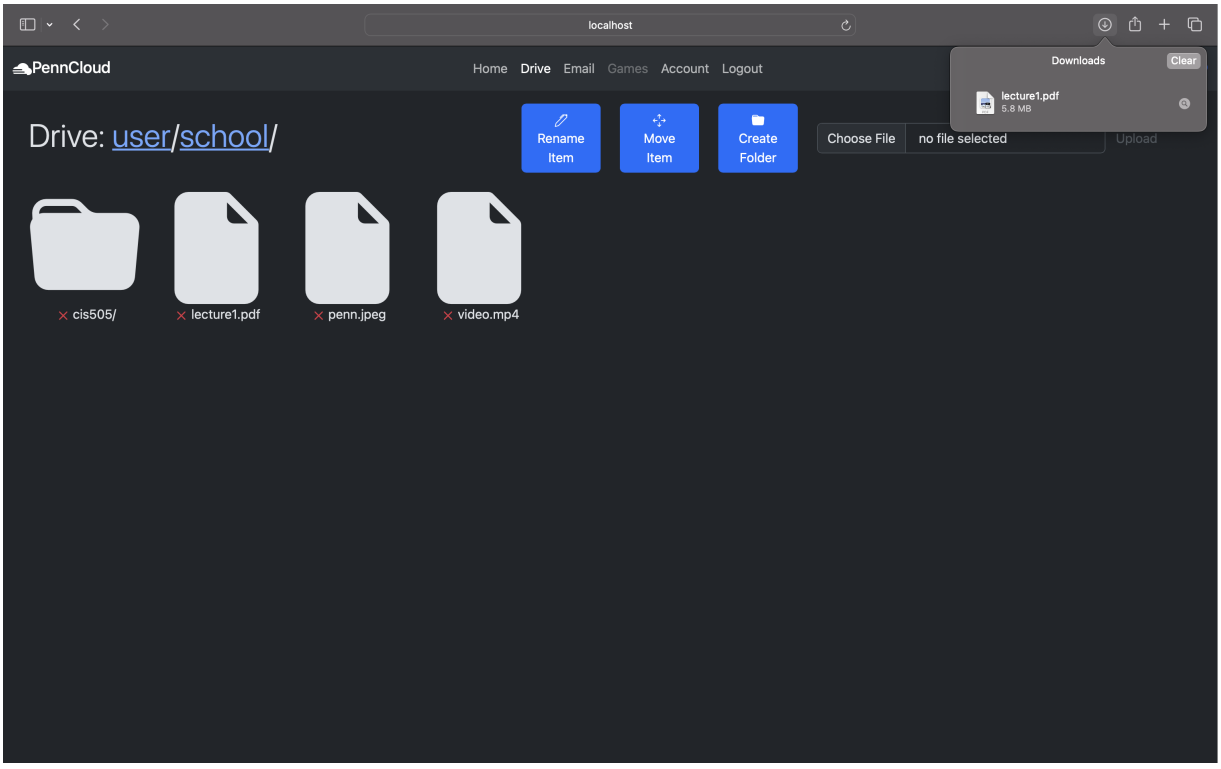

Figure 10. Downloading a file

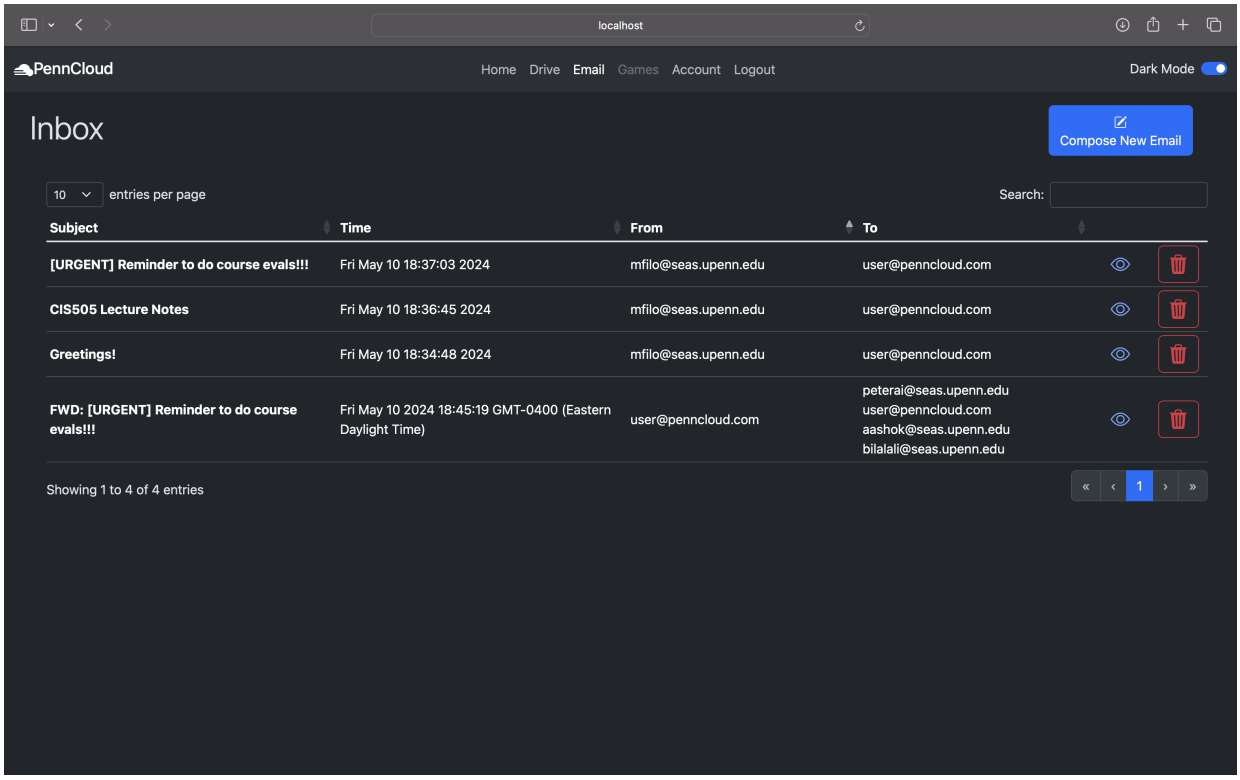

Figure 11. Mailbox

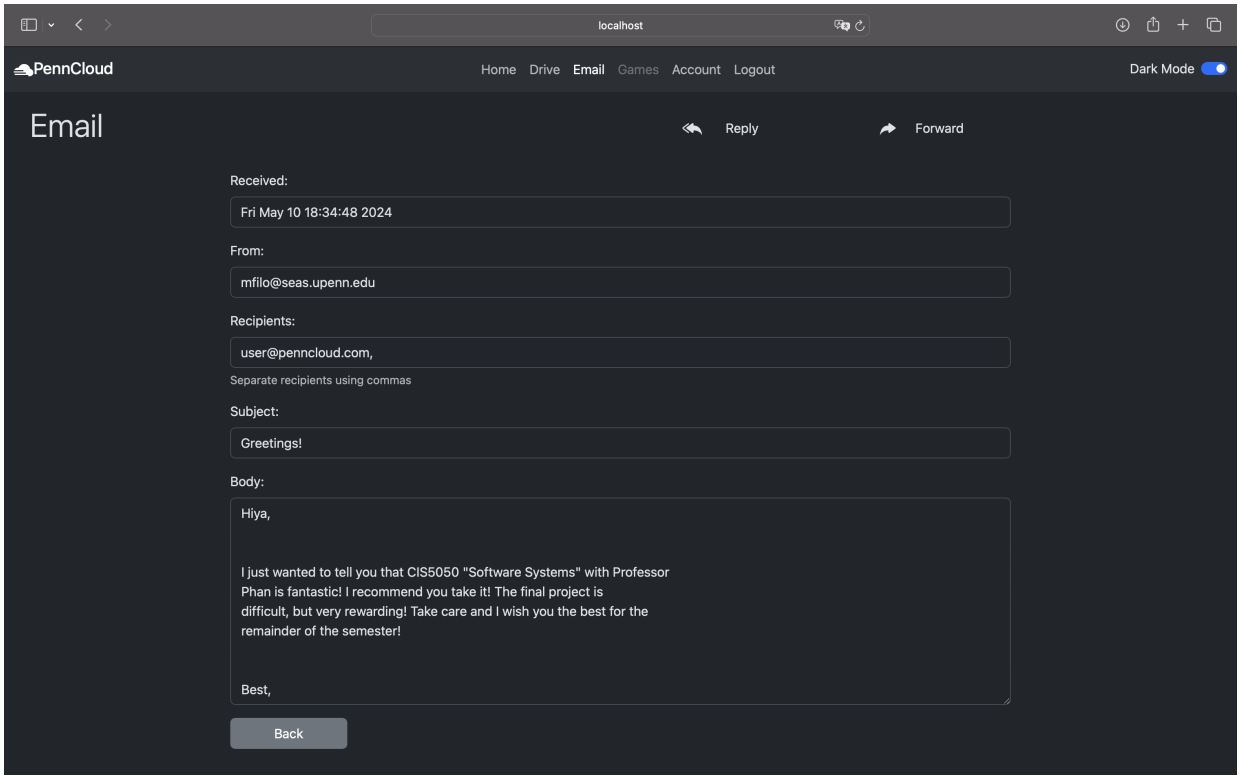

Figure 12. Composing an email

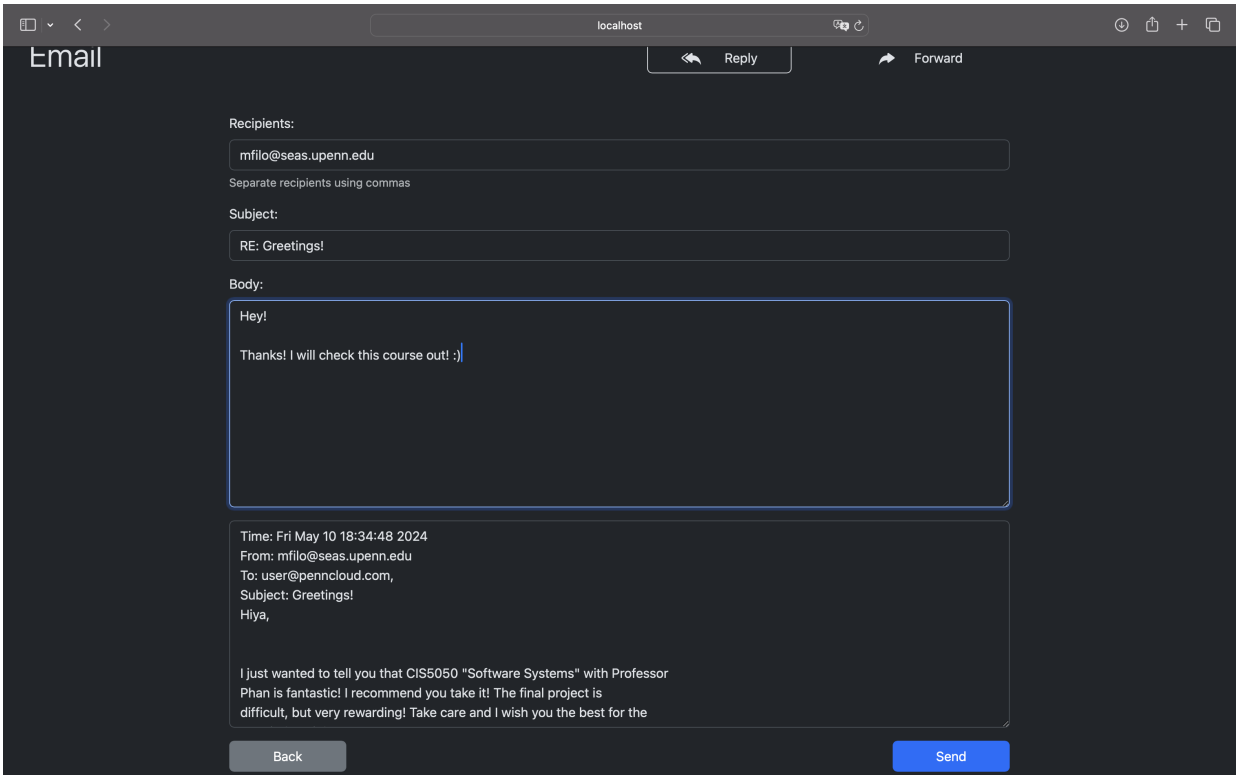

Figure 13. Responding to an email

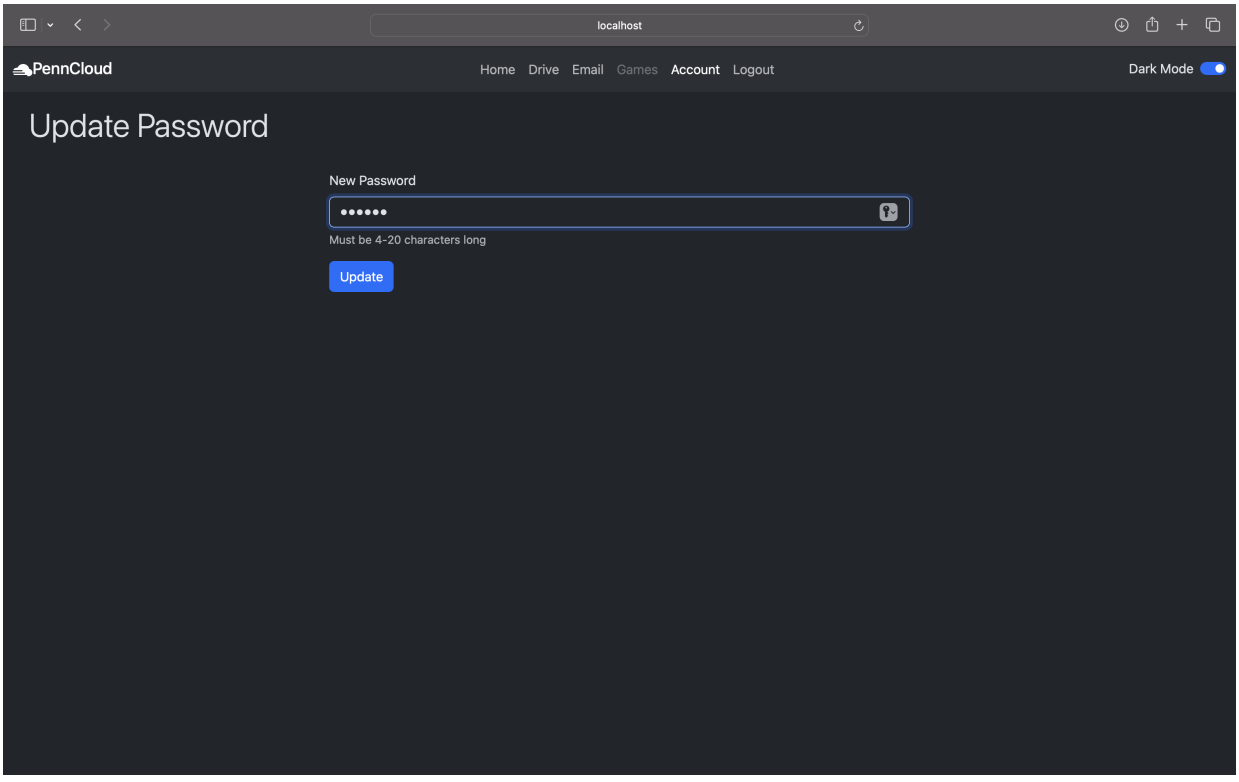

Figure 14. Changing account password

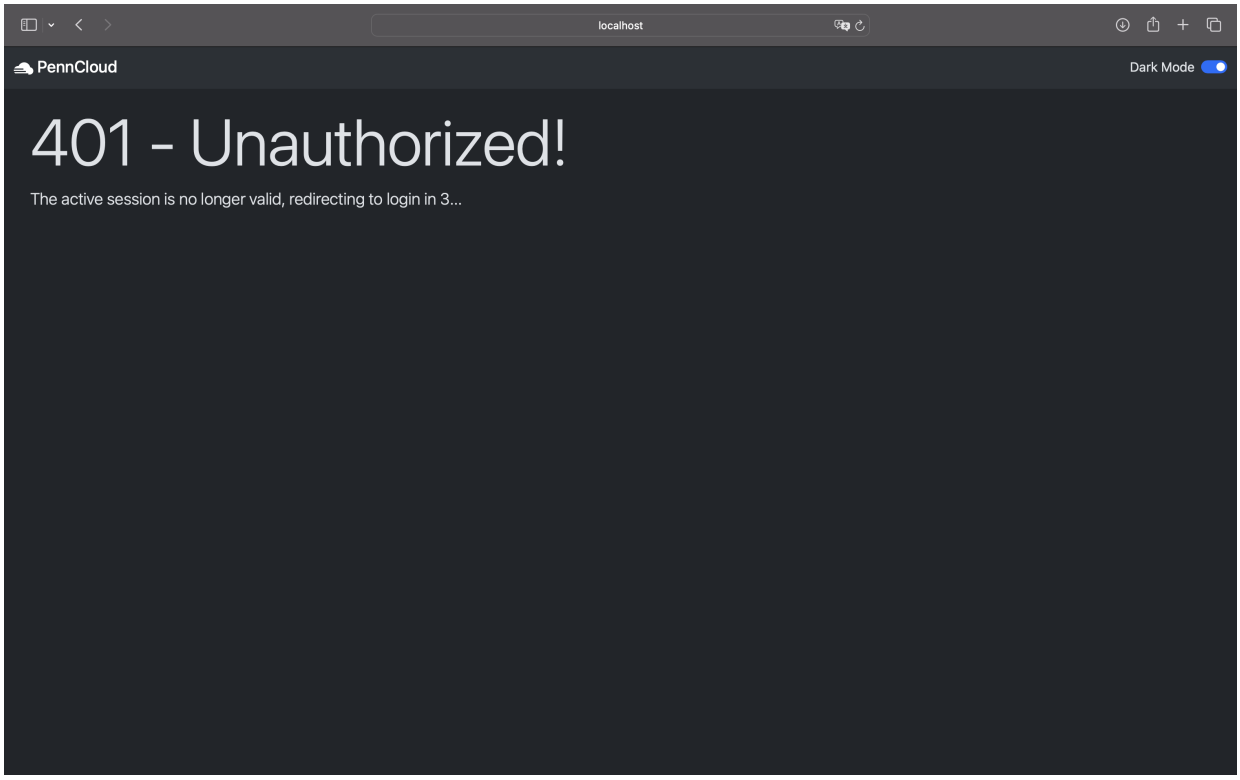

Figure 15. Session timeout

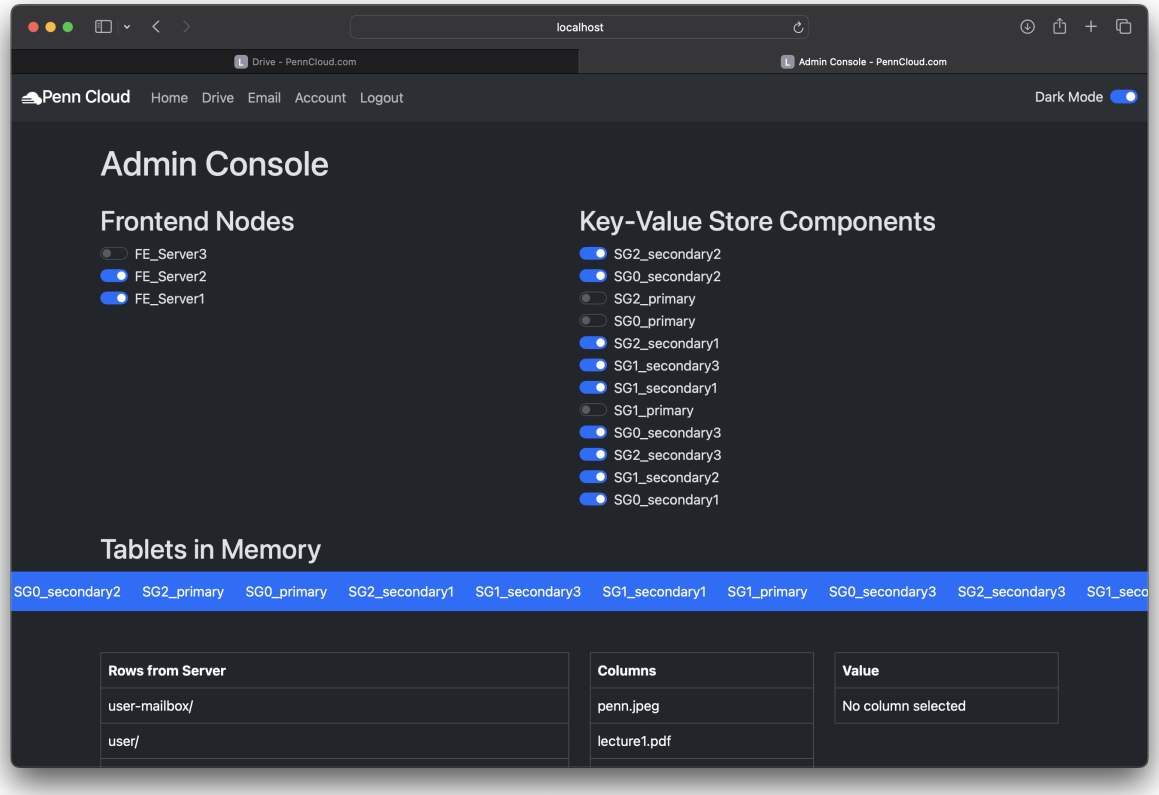

Figure 16. Admin Console dashboard