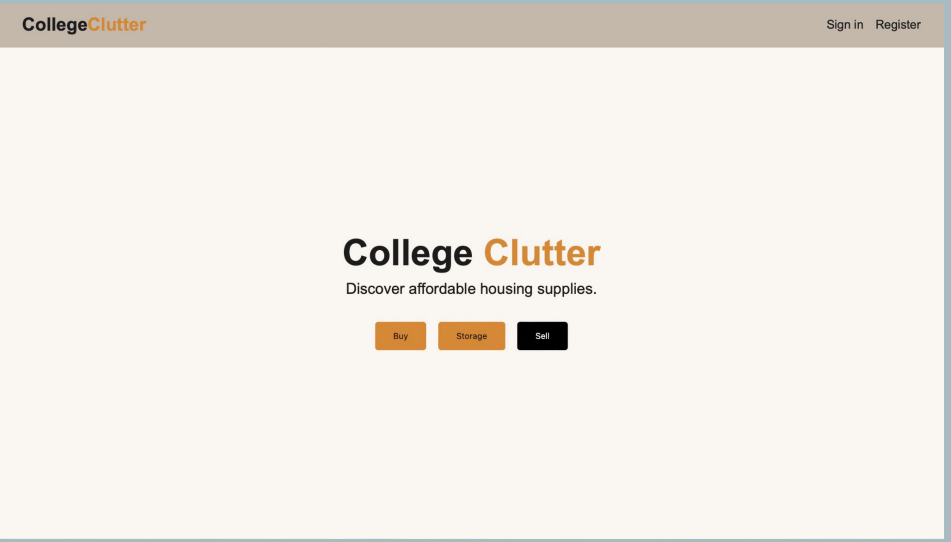
College Clutter



By:
Brandon Byrne
Sargam Nohria
Simran Lekhwani
(GROUP 36)

CS 326 SPRING 2025

Milestone #7

The project's vision.

Project's name: College Clutter
Description: Every year students buy new furniture every year to offset costs of storage and leading to tons of waste at the end of every semester. College Clutter is a website designed to connect students to other students and local storage facilities to collectively create optimal storage/selling opportunities. Waste and cost of living are both increasing at remarkable rate for students at Amherst. By offering a reasonable storage option we can give students an option to reduce costs between semesters and save on massive waste while improving local business outcomes during the down season.
Key functionalities: The website will be a lightweight platform that will use collective data from user inputs to generate optimal storage options with other users and local storage facilities. The user will have a simple drop-down interface to use commonplace data on furniture sets for ease of use and fast adoption. Users can list housing items for sale, buy items, or find storage for their items using the simple interface.

Tagged repository: https://github.com/umass-byrneb/CS326-Group-36
Milestone: #7
Issues on GitHub: https://github.com/umass-byrneb/CS326-Group-36/milestone/6

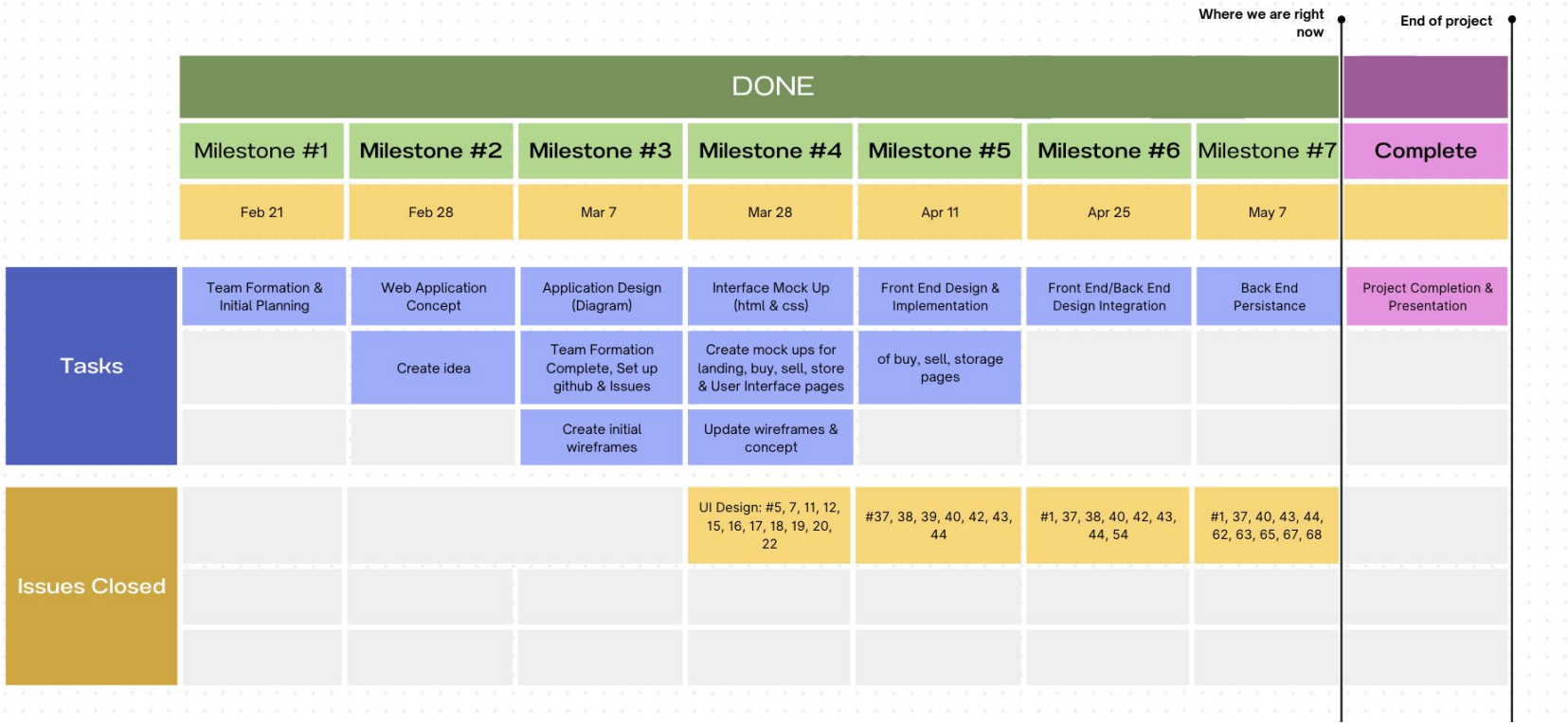
The builders.

Brandon Byrne
Senior Computer Science Major | bbyrne@umass.edu
Background knowledge: C,C++, Python, Java, js and React
Other Interests: Hiking(46er) with my dogs 🚶🐕🐕
Reflection: I believe in this website as I personally have had these problems in the past and think it would improve student life at the end/beginning of the semester and decrease waste on campus
Role: backend development, front end development

Sargam Nohria
Senior Computer Science & Anthropology Major | snohria@umass.edu
Background knowledge: C, Python, Java, HTML, CSS
Other Interests: running, rock climbing, baking
Reflection: I am interested in increasing students' accessibility to affordable housing items and a more efficient exchange of existing resources. I believe this website will help people live well.
Role: wireframes/UI, front end developer

Simran Lekhwani
Senior Computer Science Major | slekhwani@umass.edu
Background Knowledge: C, C++, Python, Java, Go, JS, SQL
Other Interests: Dancing, Rock climbing, Reading
Reflection: As a student who has lived off campus and is graduating as well, this application would be very helpful to reduce some stress associated with moving and relocating by introducing a platform that allows one to communicate with other individuals who are interested in selling or buying furniture.
Role: Backend development, front-end development

Historical timeline.



Brandon Byrne

Assigned Work Summary

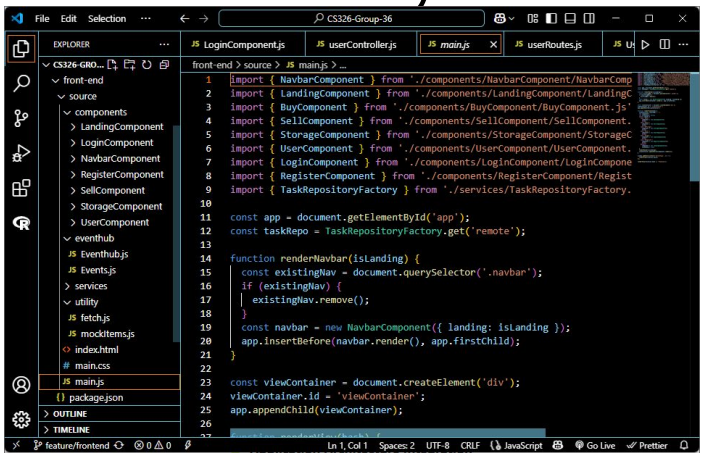
Assigned Issues: #63, #62
Commits: allcomponment files, user task sequelize models, usercontroller, taskcontroller

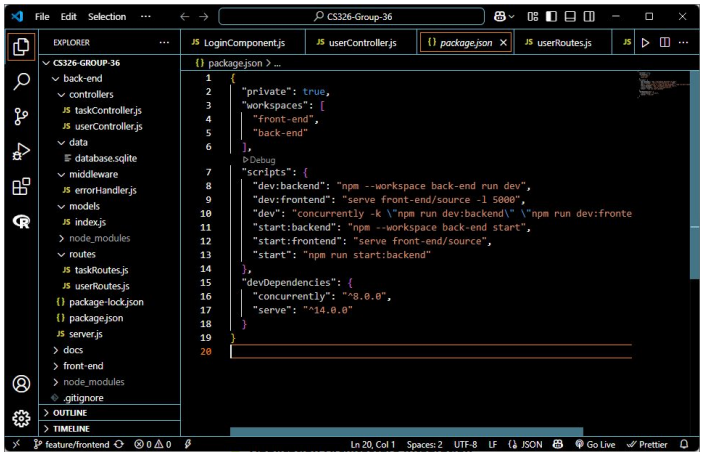
Tasks completed: Throughout this milestone, my primary responsibility was transitioning our application's backend from using flat-file storage (JSON files) to a robust, persistent storage solution using SQLite and Sequelize ORM. This included defining and implementing Sequelize models, integrating database migrations, and refactoring existing controller logic to leverage Sequelize methods. Specifically, I completed several key tasks such as removing legacy file-based models (fileTaskModel.js, fileUserModel.js) and model factories (modelFactory.js), replacing them with clear, maintainable Sequelize models (User and Task). I ensured that our database schema closely matched the original data structure, maintaining data integrity and application functionality. Additionally, I updated our server.js to include automatic database synchronization at startup, ensuring tables are created and updated seamlessly when the server launches.

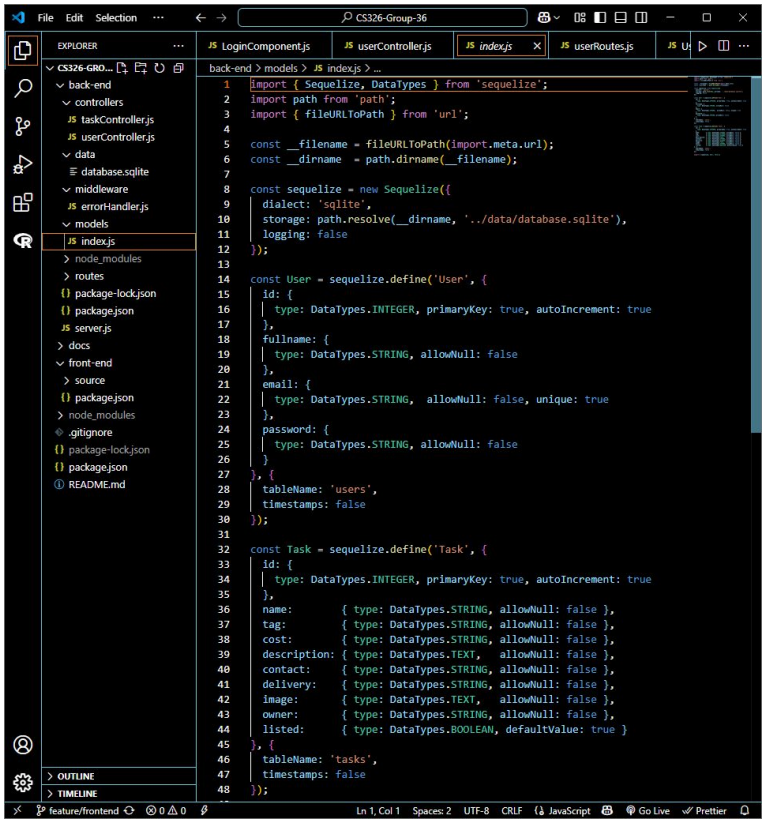
- Links to PRs closed on GitHub:
- https://github.com/umass-byrneb/CS326-Group-36/pull/71

Brandon Byrne

Screenshots and demonstration







File Structure and sqlite/sequelize integration

Brandon Byrne

Feature demonstration and code explanation

The primary feature implemented was persistent backend storage using SQLite and Sequelize ORM. This allowed our application to handle data reliably and consistently, providing a professional-grade backend infrastructure. Both the backend and frontend components of this feature were successfully completed and are currently fully functional. On the backend, I defined Sequelize models for User and Task, established routes for interacting with these models, and refactored controllers to use Sequelize queries instead of direct file operations. On the frontend, I integrated API endpoints effectively so the application can perform full CRUD (Create, Read, Update, Delete) operations seamlessly.

https://github.com/umass-byrneb/CS326-Group-36/pull/71/commits

Brandon Byrne

Code Structure & Organization

The project's directory structure clearly delineates between frontend and backend components, adhering strictly to best practices and guidelines provided in class. The backend now resides in a well-organized directory (/back-end) containing clearly labeled subfolders for controllers, routes, models, middleware, and database configurations. For example, Sequelize models (User.js and Task.js) are located within the /models directory, clearly separated from controller logic (taskController.js and userController.js), found within /controllers. Routes (taskRoutes.js and userRoutes.js) exist distinctly in the /routes folder, providing clear separation of concerns. Our SQLite database file (database.sqlite) resides neatly within the /data directory. The frontend and backend separation is maintained meticulously, with the frontend located in /front-end/source. Communication between these two components strictly occurs through RESTful API endpoints, ensuring that changes in one do not disrupt the other, significantly improving maintainability and scalability.

https://github.com/umass-byrneb/CS326-Group-36/tree/main/back-end

https://github.com/umass-byrneb/CS326-Group-36/tree/main/front-end

Brandon Byrne

Front-End Implementation

On the frontend, a significant contribution was the implementation of the item creation feature within the SellComponent. The JavaScript function responsible for interacting with the backend API (handleSaveItem) utilizes a fetch call to the new backend endpoint /v1/tasks, sending user-submitted data in JSON format. This function captures form data, sends it securely to the backend, handles responses, and provides immediate user feedback via a modal notification upon success. This integration is deeply woven into the application architecture as it directly leverages the new Sequelize-powered backend API. A notable challenge encountered here was synchronizing API changes and ensuring robust error handling, especially transitioning from basic alerts to inline notifications and modals. I resolved this by implementing explicit success and error modals, providing a significantly enhanced user experience through clear, immediate feedback without abrupt interruptions or unclear error messages.

https://github.com/umass-byrneb/CS326-Group-36/blob/main/front-end/source/components/SellComponent/SellComponent.js

Brandon Byrne

Back-End Implementation

On the backend, one of my key contributions was refactoring the task management controller (taskController.js) to interact with the SQLite database using Sequelize ORM. For instance, the implementation of the addTask method now cleanly utilizes Sequelize's Task.create() method, receiving JSON payloads from the frontend and inserting validated data directly into our SQLite database. This seamless integration drastically improves reliability and data consistency. All backend components—models, routes, controllers, and middleware—are clearly structured and labeled in their respective folders. Sequelize models (User, Task) define database schema; controllers (userController.js, taskController.js) encapsulate business logic and Sequelize interactions; routes (userRoutes.js, taskRoutes.js) manage HTTP request routing, and middleware centrally manages error handling and response formatting. Utilizing Sequelize provides clear, declarative syntax for data operations, simplifying CRUD operations and improving developer experience. Initially, a major challenge was the conceptual shift required when moving from flat-file storage to database-driven storage. To overcome this, I closely studied Sequelize documentation and followed best practices to implement robust and maintainable data-handling logic.

https://github.com/umass-byrneb/CS326-Group-36/tree/main/back-end/controllers

Brandon Byrne

Challenges and Insights

Obstacles:

Transitioning from a simple file-based backend to a sophisticated ORM-based solution presented several challenges, primarily in understanding ORM concepts such as associations, constraints, and migrations. Initially, the complexity of Sequelize and SQL-based database schema design required significant research and experimentation. A critical lesson learned was the importance of comprehensive validation and error handling at every stage of data interaction, greatly facilitating debugging and maintenance.

Working collaboratively with the team was essential; clear communication and regular peer reviews significantly improved the quality of the final implementation. This collaborative approach enabled us to quickly identify and resolve issues, thus accelerating overall development and reinforcing strong teamwork and effective problem-solving skills.

.

Brandon Byrne

Future improvements & next steps

Looking beyond the current milestone, several improvements have been identified to enhance the application further. One crucial improvement would be implementing robust authentication mechanisms using JSON Web Tokens (JWT) stored in HTTP-only cookies, providing secure and scalable session management beyond our current simplistic localStorage solution. Additionally, introducing advanced API features such as pagination, filtering, and search capabilities could significantly improve user experience and application usability when handling larger datasets. Improved error handling and detailed logging mechanisms could enhance application robustness and simplify future debugging efforts. To clearly document and track these improvements, I have created new issues in our GitHub repository under the "Future Tasks" milestone. These include Implement JWT Authentication and enhance error reporting. Addressing these issues will reduce technical debt and significantly optimize application performance and maintainability in future iterations.

https://github.com/umass-byrneb/CS326-Group-36/issues/76

https://github.com/umass-byrneb/CS326-Group-36/issues/77

.

Sargam Nohria

Assigned Work Summary

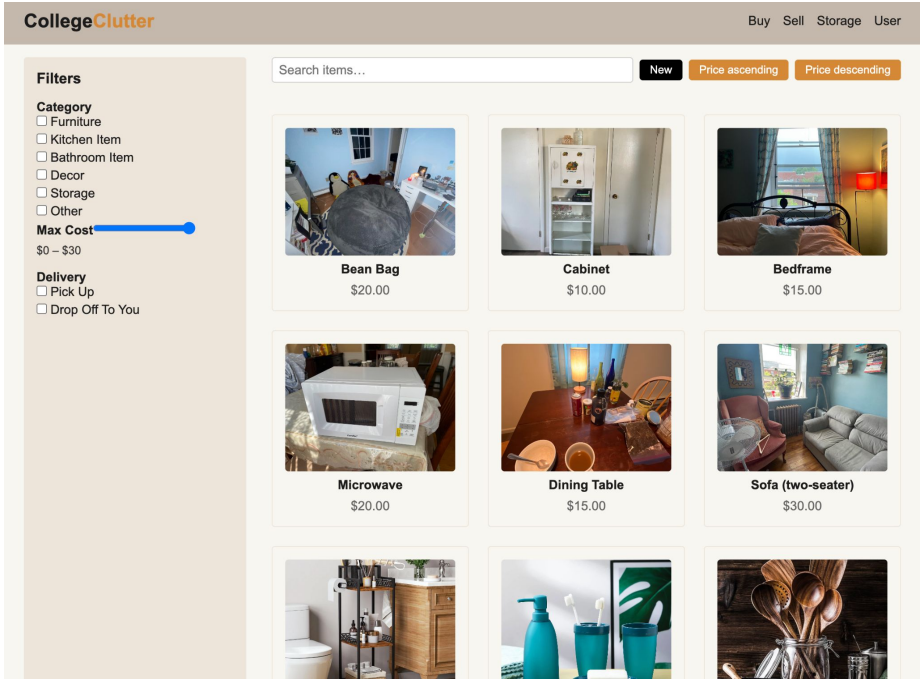
Assigned Issues: #65
Commits: Database propagation

Tasks completed: I successfully propagated the database to support backend functionality of the website. I did in-depth testing of the application's features to make sure they are functional, including updating the express dependency version to incorporate the latest features/security improvements.

- Links to PRs closed on GitHub:
- https://github.com/umass-byrneb/CS326-Group-36/pull/72

Sargam Nohria

Screenshots & Demonstration

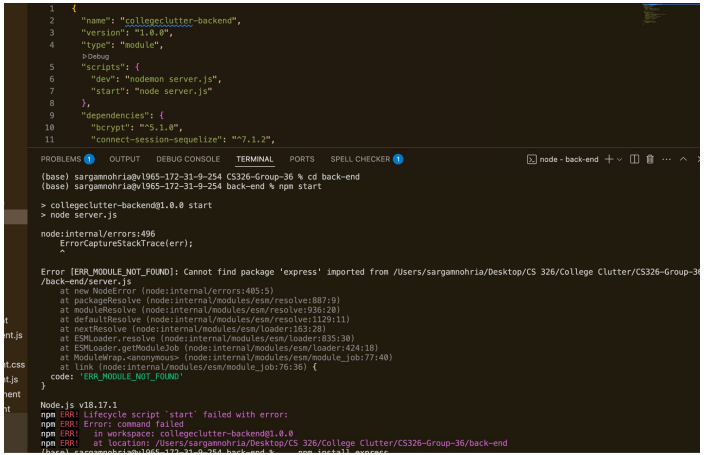


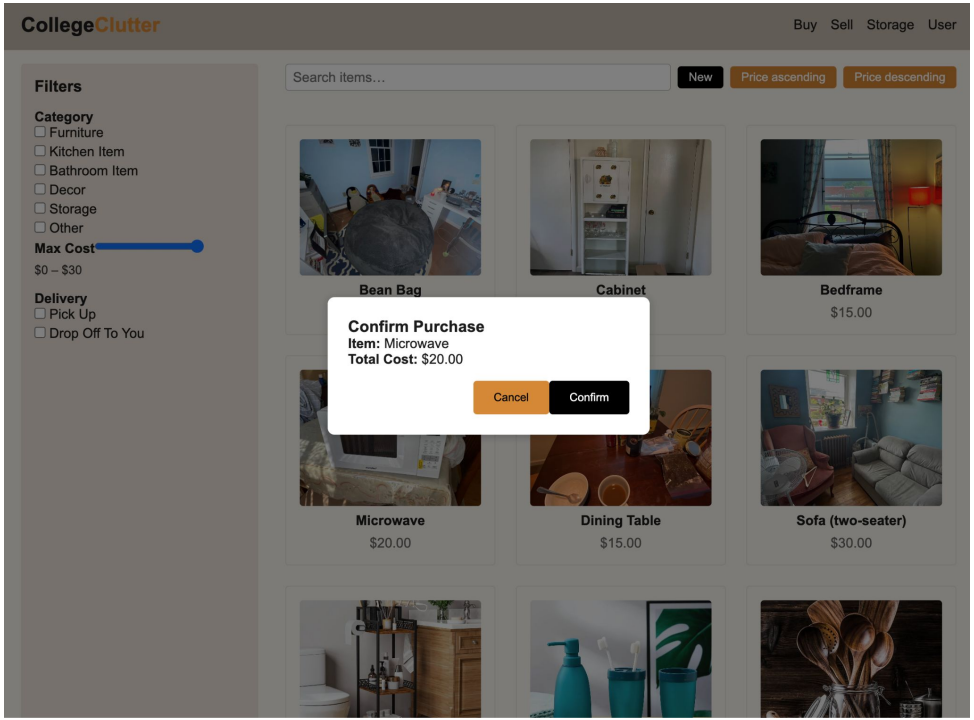
Database propagation

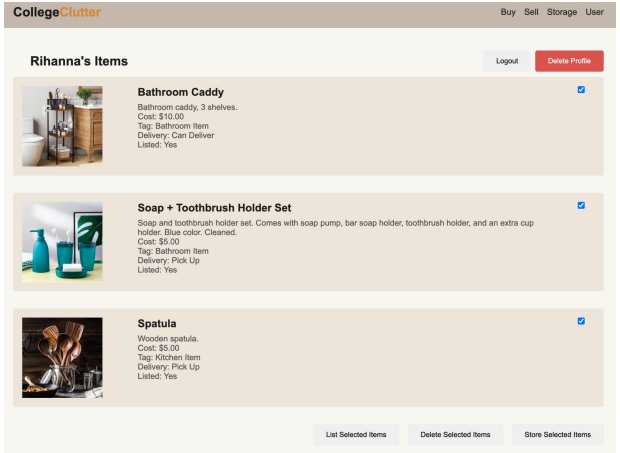
Sargam Nohria

Screenshots & Demonstration

Front-end and back-end testing







Sargam Nohria

Code and UI explanation

Over this milestone I set up and propagated the project's database to support core application functionality, ensuring schema alignment with both front-end and back-end requirements. As part of maintaining up-to-date dependencies, I upgraded Express from version 4.18.2 to 4.21.2 in the package.json file to incorporate the latest performance improvements and security patches. I conducted comprehensive testing across both the client and server sides, validating API endpoints, data flow, and user interactions to ensure seamless integration and consistent functionality throughout the application stack.

Sargam Nohria

Challenges and Insights

During this milestone, I encountered several challenges related to version control and project consistency. Ensuring I was working on the correct Git branch required careful coordination with my team, especially when multiple features were being developed in parallel. I also had to verify that my local environment reflected the latest package versions and migrations, which involved regularly pulling from the main branch, managing package-lock.json, and clearing caches when dependencies behaved unexpectedly. These experiences reinforced the importance of disciplined Git practices, clear branching strategies, and regular communication within a collaborative development workflow.

Sargam Nohria

Future improvements & next steps

- While this is the last milestone and we completed all the functionality we aimed to complete, next steps for this application, were it to be used by real users, would be to complete the buying functionality. Then people would be able to insert their credit card/payment info and be able to actually buy the item and pay the seller, as well as truly use the storage option. It would also be a good idea to have a more robust and comprehensive user profile within the application, housing more information about each user. Another next step could be to add a “history” tab showing the user’s previous actions, such as what they previously sold or bought or stored somewhere. This could help users keep track of their items and their behavior.

Simran Lekhwani

Assigned Work Summary

Assigned Issues: #54 (Storage Page), #68 (Classification of Storage items on user page)
Commits: Back-end migration to SQLite DB, Add Renting storage space functionality, Enhance Sell, Storage and User page UI, front-end and back-end integration using services

Tasks completed:

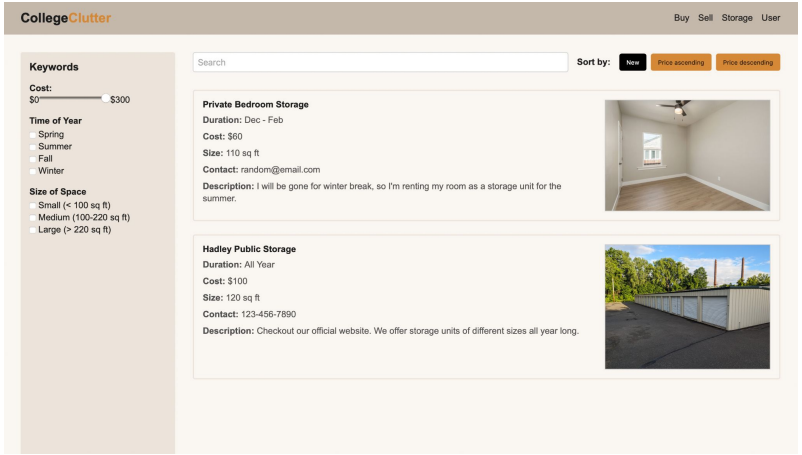
- Completed the back-end migration from in-memory model to SQLite model, where the Storage controller uses Storage Sequelize model to perform CRUD operations.
- Rendered an additional form on Sell page which allows user to making a storage space posting. The positing is added to the DB using POST and is redirected to the user page (multi-UI view).
- Added a Storage tab on the user dashboard to allow users to view their items and storage space postings separately. Added a 'List Selected Item' and 'Delete Selected Item' and consequently a PUT and DELETE (by id) endpoint (note: the PUT endpoint was established in previous milestone).
- When these user actions are performed (i.e. a selected storage item is listed or deleted), using Events and Services, the list displayed on the Storage page is dynamically updated. This same list can be filtered by using the buttons and filters on the side bar
- Enhanced the UI of the Storage, Sell and User page

Links to PRs closed:

- https://github.com/umass-byrneb/CS326-Group-36/pull/75

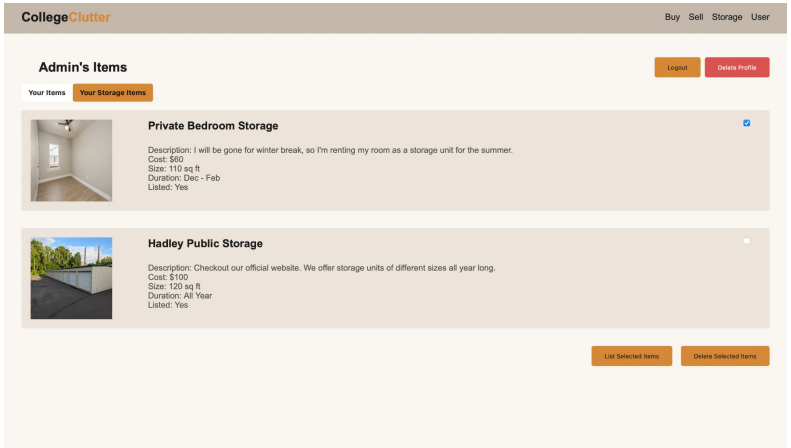
Simran Lekhwani

Screenshots and Demonstration

- 

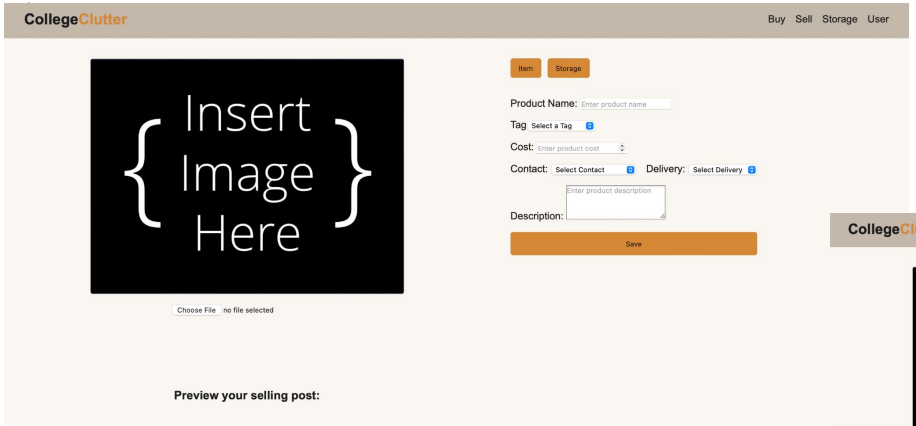
Storage List is populated on the Storage page and only listed items are shown

Storage Items are displayed on the user dashboard. The user can list or delete selected items by using the checkbox

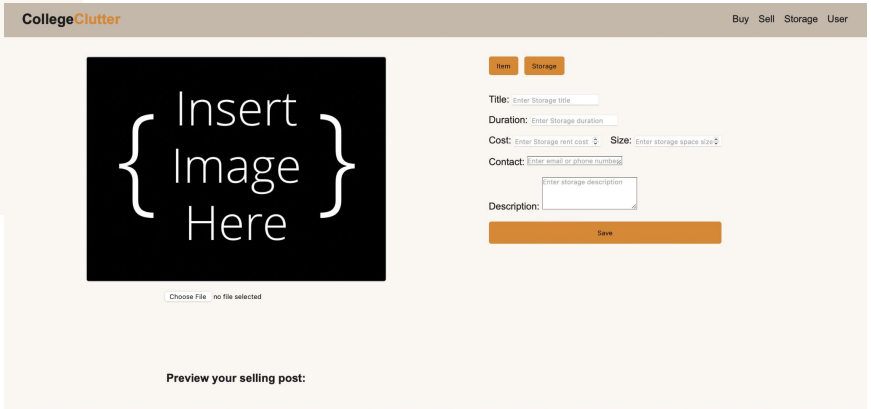


Simran Lekhwani

Screenshots and Demonstration



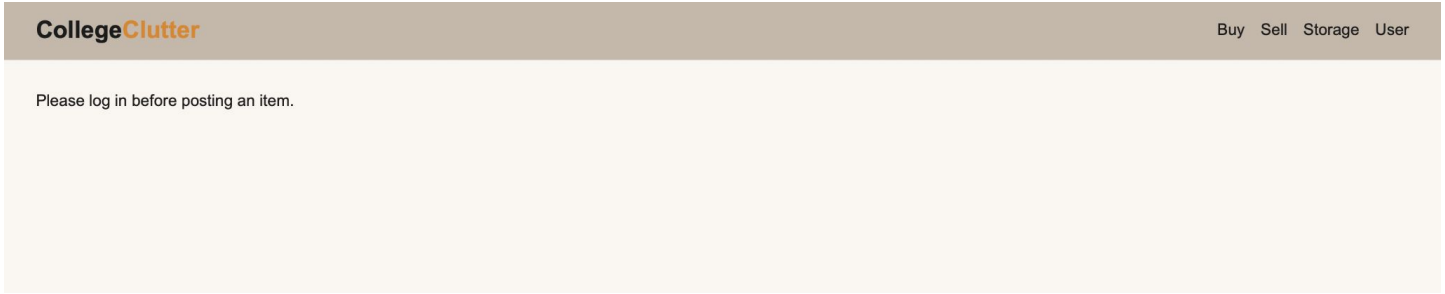
Item Sell form is populated upon clicking the Item button



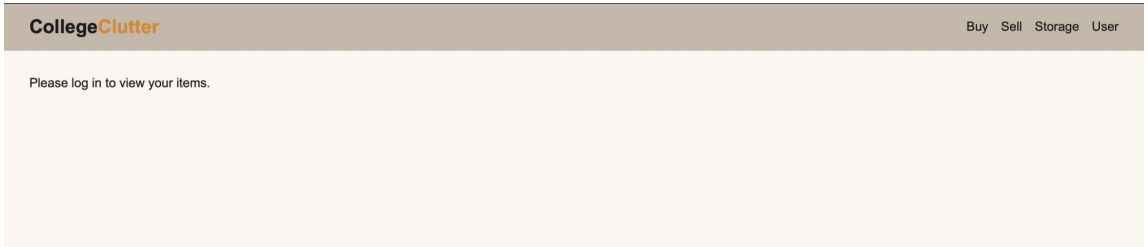
Storage form is populated upon clicking the Storage button

Simran Lekhwani

Screenshots and Demonstration



Users cannot sell storage spaces or view their storage listings without being authenticated i..e they need to log in to view them



Simran Lekhwani

Screenshots and Demonstration

```
  tableName: 'users',
  timestamps: false
});

const Task = sequelize.define('Task', {
  id: {
    type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true
  },
  name:        { type: DataTypes.STRING, allowNull: false },
  tag:         { type: DataTypes.STRING, allowNull: false },
  cost:        { type: DataTypes.STRING, allowNull: false },
  description: { type: DataTypes.TEXT,   allowNull: false },
  contact:     { type: DataTypes.STRING, allowNull: false },
  delivery:    { type: DataTypes.STRING, allowNull: false },
  image:       { type: DataTypes.TEXT,   allowNull: false },
  owner:       { type: DataTypes.STRING, allowNull: false },
  listed:      { type: DataTypes.BOOLEAN, defaultValue: true }
}, {
  tableName: 'tasks',
  timestamps: false
});

const Storage = sequelize.define('Storage', {
  id: {type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true},
  title: {type: DataTypes.STRING, allowNull: false},
  duration: {type: DataTypes.STRING, allowNull: false},
  cost: {type: DataTypes.FLOAT, allowNull: true},
  size: {type: DataTypes.FLOAT, allowNull: false},
  contact: {type: DataTypes.STRING, allowNull: true},
  description: {type: DataTypes.STRING, allowNull: true},
  image: {type: DataTypes.TEXT, allowNull: false},
  owner: {type: DataTypes.STRING, allowNull: false},
  listed: {type: DataTypes.BOOLEAN, defaultValue: false}
}, {
  tableName: 'storage',
  timestamps: true,
})

export { sequelize, User, Task, Storage };
```

Sequelize Storage Model

```
import { Storage } from "../models/index.js";


export async function getAllListings(_, res, next) {
    try {
        const listings = await Storage.findAll();
        return res.status(200).json({listings});
    } catch (err) {
        console.log("error received: ", err);
        next(err)
    }
}

export async function addStorageItem(req, res, next) {
    try {
        const {title, duration, cost, size, contact, description, image, owner, listed} = req.body
        if (!title || !duration || !image || !size) {
            return res.status(400).json({error: "All required fields weren't received."});
        }
        const newitem = await Storage.create({title, duration, cost, size, contact, description, image, owner, listed});
        return res.status(200).json(newitem);
    } catch (err) {
        console.log("Err in adding storage item: ", err);
        next(err);
    }
}

export async function updateStorageItem(req, res, next) {
    try {
        const itemID = parseInt(req.params.id);
        const { listed } = req.body;
        if (isNaN(itemID) || typeof listed !== 'boolean') {
            return res.status(400).json({ error: 'Invalid request.' });
        }
        const [count] = await Storage.update({ listed }, { where: { id:itemID } });
        if (!count) {
            return res.status(404).json({ error: 'Storage item not found.' });
        }
        const updated = await Storage.findByPk(itemID);
```

Storage Controller relies on the Storage Sequelize model to perform CRUD operations

Simran Lekhwani

Code structure and Organization:

For modularity, the back-end is structured in different “modules” or directories: controllers, data, middleware, model, routes, and server.js

Middleware does error handling.

Under routes, there is a specific file dedicated for Storage page routes, which perform actions based on asynchronous functions imported from the Storage Controller in the controllers dir. These functions rely on the **Storage Sequelize model** to perform CRUD operations and all the data is saved in database.sqlite

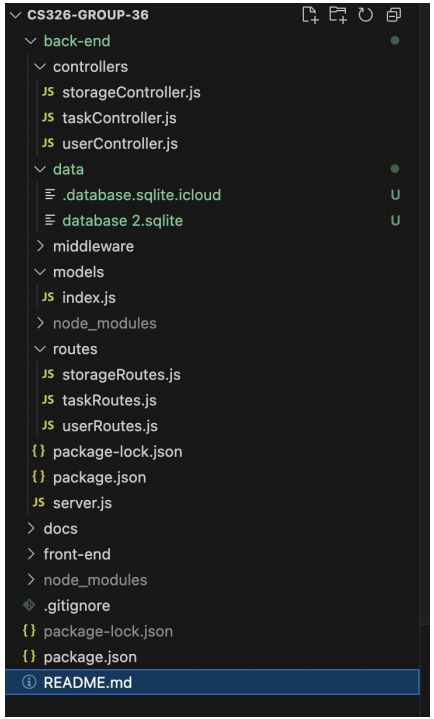
For isolation and modularity, continuing from the previous milestones, the front-end is broken down into source, eventhub and services.

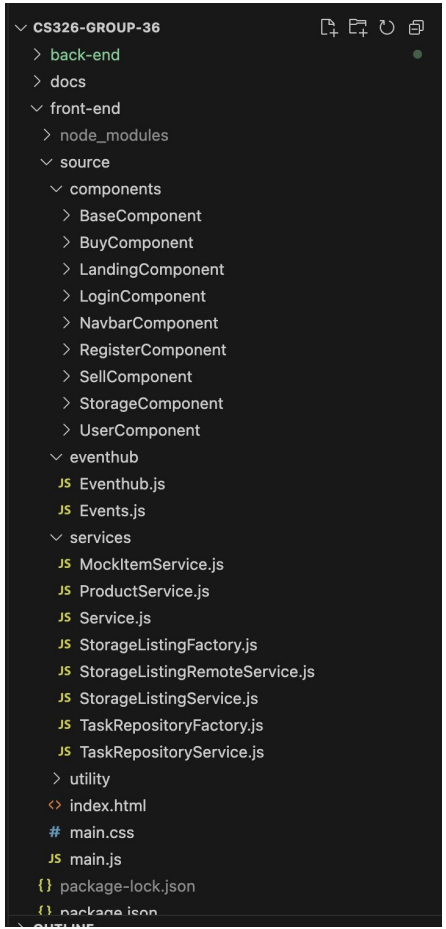
Updates were made to the Sell page to allow users to post/sell/rent storage spaces. Consequently, the user page was reflected with the new storage spaces.

StorageListingFactory was integrated to allow one point access to the remote (fetch calls to DB) and local (indexedDB) data. Using StorageListingService, StorageListingRemoteService and EventHub, whenever a new post was added or the user updated it by listing it, an appropriate HTTP request (GET, POST, and PUT) was made to the back-end. Similarly, a DELETE request was made when the user deleted a posting from their dashboard.

Simran Lekhwani

Code structure and Organization:





Simran Lekhwani

Front-end Implementation:

There were significant contributions on the front-end to allow users to post/rent storage spaces. On the sell page, two buttons were added - one for items and one for storage spaces. Clicking on them renders an item form and a storage space form respectively. When a storage form is submitted, an AddStorageItem Event is triggered, which leads to POST request to the DB through StorageListingRemoteService while the user is redirected to the user dashboard (multi-UI) after a 'successfully saved' confirmation.
On the user dashboard/page, two tabs/buttons were added again - one for displaying all the items the user has posted and one for all the storage spaces. The storage postings by the current user (which is retrieved using local storage) is displayed on the page using a GET request. The user can select upto multiple items, and once at least one posting is selected, it ables the 'List Selected Items' and 'Delete Selected Items' button. When the selected items are listed, a PUT request is sent with the id of the posting as the parameter to mark it as listed. Once it is marked as listed, it publishes a message which allows the display list on the Storage page to dynamically load. Similarly, deleting the selected item(s) sends a DELETE request and when it is successfully performed, a message is published updating the storage page view.
In addition, UI of the filters on the side bar were enhanced, and functionality was added to them (such as the cost filter). The update on the list depends on data stored in SQLite DB as well as indexedDB. Whenever a filter or keyword is added, it uses the data from indexedDB to avoid multiple API calls (using StorageListingService), but whenever the DB is updated (new item, deleted item, updated item), it fetches from it and updates the list accordingly (using StorageListingRemoteService).

Simran Lekhwani

Back-end Implementation:

My main contribution on the backend was to handle all the requests for storage items using the SQLite DB and Sequelize model. I created a sequelize model for Storage which captured all the data related to the posting (title, description, size, etc..) but also included columns for id - which was the primary key - and the owner. I used functions like findAll(), destroy({where}), update({where}), and create(obj) to perform CRUD operations and add data to the SQLite DB. In addition, the storageController would validate the data received from the requests, and once it was validated using these functions provided through the Storage Sequelize model to create a storage listing (route: GET /v1/storage/listings), add a storage posting (route: POST /v1/storage/listings), (where the primary key was automatically incremented with the help of Sequelize), update a storage posting (route: PUT /v1/storage/listings/:id), and delete a posting based on user or id.

Simran Lekhwani

Front-end and Back-end integration:

The communication between the front-end and back-end was done through Events, EventHub and Services (StorageListingService, StorageListingRemoteService, StorageListingFactory).

StorageListingFactory was instantiated in main.js and would return either StorageListingRemoteService or StorageListingService.

StorageListingRemoteService handled fetch API calls to the SQLite DB and was triggered only when an add, update or delete storage item event was published. It was also triggered during the first load of the web page to retrieve listings from the DB and store in indexedDB.

Storage

Simran Lekhwani

Challenges and Insights

- One of the key challenges I faced was dealing with different project versions. While GitHub makes it easier to collaborate, not following common practices can lead to a lot of conflicts. For instance, during this time, my changes from the previous milestone were completely overwritten and performing git fetch or git pull would also rewrite the changes in my dev branch. Therefore, I had to figure out a workaround by pushing additional changes to my files (which were deleted on the main branch aka overwritten by another PR). While trying to figure out a solution, I learned more about possible Github actions and how to reverse them.
- Another challenge was making sure all endpoints were met. To further explain, I added a functionality which allows users to rent storage spaces as well, but to do this, I had to change a lot of working components, leading to breaks. Finding every loose endpoint/end to ensure all previous functions and the new one work was tedious and required understanding another member’s code and logical breakdown of it.

Simran Lekhwani

Future improvements & next steps

While this is the last milestone and we have added several functionalities to this application, there is definitely room for improvement in terms of existing features or adding new ones. Some possible improvements or next steps are below:

- Payment page when an item is bought

Link:

- Users can interact with the people who posted items or storage spaces to know more details

Link:

- Editing posts on the user dashboard

Link: