COMS 3090 Dashboard Handbook

Rida Azam, Lottie Peterson, Vinayak Trigunayat, Maria Sanchez, Lev Schaul

This is a guide to using the 3090 dashboard. This handbook includes a getting started guide, breakdown of the code for the front and back ends, and a guide to the application itself.

Table of Contents

Table of Contents **2**
Quick Links **3**
Getting Started **3**
Frontend 3
Backend 3
PostgresSQL 3
Helpful things: 4
pgAdmin Set Up 4
Setting Up the Virtual Environment 4
Running the backend server 5
Environment Variables 6
Authentication Setup (Clerk) 7
Create a Clerk Application 7
Environment Variables 7
ISU Email Restriction 7
Navigating Pages and Features **8**
Home Page 8
Group Page 8
Individual Student Page 10
Student Overview Page 11
Status Page 12
Staff Management Page 13
Import Page 15
Local Photo Upload and Storage 15
Code Structure **16**
Frontend 16
Backend 17
CI/CD Pipeline **19**

Quick Links

Gitlab Repository: https://git.las.iastate.edu/SeniorDesignComS/2026spr/402c/ug_fs_4
Tamagui Documentation: https://tamagui.dev/
Django REST Framework Documentation: https://www.django-rest-framework.org/
Backend Server: http://coms-4020-024.class.las.iastate.edu:8080/admin/

Getting Started

Frontend

1. Move into the Expo project.
   ```
   cd 3090_Dashboard
   ```
2. Install dependencies
   ```
   npm install
   ```
3. Start the app.
   ```
   npx expo start
   ```

If you need to clear the cache, start the project with the `-c` flag.

Backend

PostgresSQL

Any SQL database should be compatible with the dashboard. We used PostgresSQL, so our instructions will be for that technology. The same setup instructions can be used for the remote server and for local installation.

1. Download the Database from here
2. After it is downloaded, create your own user
   Mac: `sudo -u postgres psql`
   Powershell: `psql -U postgres`
3. Create a user in PostgresSQL
   ```
   CREATE USER <NET_ID> WITH PASSWORD 'mypassword';
   ```
4. Create the PostgresSQL database
   ```
   CREATE DATABASE backend402 OWNER <NET_ID>;
   ```
   For other users who need permissions, they can be granted with:
   ```
   GRANT ALL PRIVILEGES ON DATABASE backend402 TO <NET_ID>;
   ```

If you want to access the DB command line via this new user:

```
psql -U <NET_ID> -d backend402
```

Helpful things:

- To start the DB process: `sudo systemctl start postgresql`
- To configure DB to boot on restart: `sudo systemctl enable postgresql`
- To stop the DB process: `sudo systemctl stop postgresql`
- To reload the DB configs: `sudo systemctl reload postgresql`

pgAdmin Set Up

To access the DB similarly to how you would with MySQL Workbench, I downloaded pgAdmin on my Mac. To connect:

1. Be sure you're in the VPN if you're accessing an ISU server, and in pgAdmin
2. Click: Add server
3. *General Tab*:
    a. Name: `<db name>`
4. *Connection*:
    a. Host name/address: `<address of your server>`
    b. Port: `<server's port>`
    c. Maintenance database: `<db name>`
    d. Username: `<NET_ID>` (the user you made before)
    e. You shouldn't need the password.

Setting Up the Virtual Environment

The backend requires a Python virtual environment. Create and activate the virtual environment:

```
cd backend
python3 -m venv .venv
source .venv/bin/activate
```

Note that the location may differ depending on where your python installation lives!

Install dependencies:

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

Deactivate the virtual environment when done:

```
deactivate
```

The virtual environment only needs to be created once. On subsequent sessions, just run:

```
source .venv/bin/activate
```

from the backend directory before starting the server.

Running the backend server

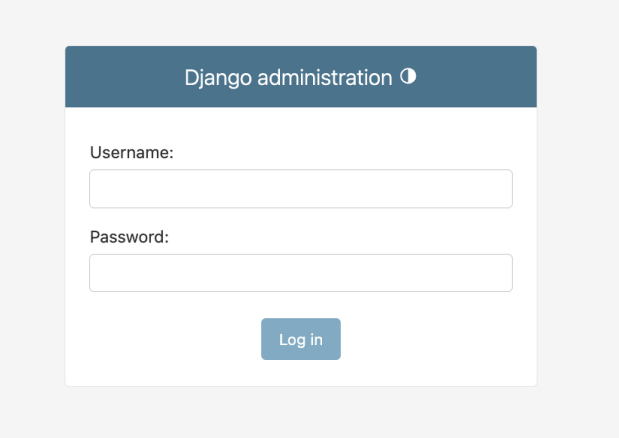
1. Move into the Django project.
   ```
   cd backend
   ```
2. Activate a virtual environment.
   ```
   source .venv/bin/activate
   ```
3. Run database migrations.
   ```
   python manage.py migrate
   ```
5. Start the backend server.
   ```
   python manage.py runserver 0.0.0.0:8000
   ```
6. Professor only: Create an admin account.
   ```
   python manage.py createsuperuser
   ```

You will use this to set up the first user in the backend.

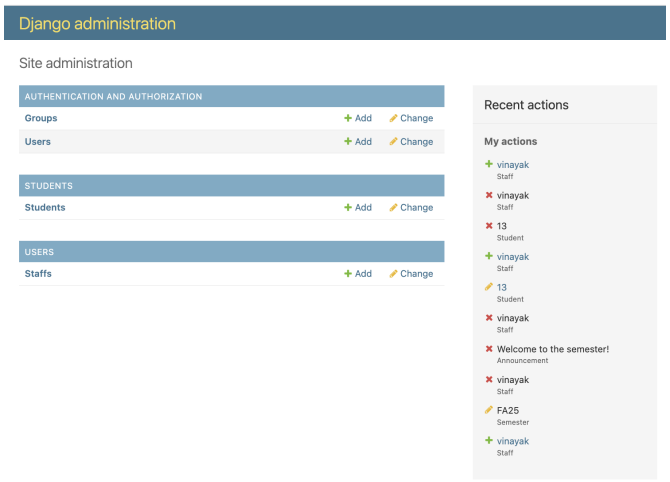
First User Setup

1. Go to `/admin/` and sign in with your Django superuser account.

Ex: `http://coms-4020-024.class.las.iastate.edu:8080/admin/`



2. Under **Users** → **Staffs**, create a record with your name, your ISU netID, and role `PROFESSOR`. Leave Clerk ID blank — it is auto-populated on first sign-in.
3. Sign into the app with your ISU Google account. The backend links the Clerk account automatically by matching your netID from your email.



After this, the Professor can add additional staff through the app's Staff Management page.

Environment Variables

The backend reads configuration from environment variables when available.

- `SECRET_KEY`
- `DB_NAME`
- `DB_USER`
- `DB_PASSWORD`
- `DB_HOST`
- `DB_PORT`
- `CLERK_SECRET_KEY`
- `CLERK_ISSUER`
- `DEBUG`

If you run the backend in Docker, make sure the container can reach the Postgres service and that the database variables are set consistently with your Compose file.

Authentication Setup (Clerk)

Clerk handles all user authentication. The frontend acquires a signed JWT after sign-in and attaches it to every API request. The backend validates it and maps the identity to a `Staff` or `Student` record. Only `@iastate.edu` emails are accepted.

Create a Clerk Application

1. Go to clerk.com and create an account.
2. Create a new application (e.g. "3090 Dashboard").
3. Enable **Google** as a sign-in provider — the app only supports Google SSO.
4. From the Clerk dashboard → **API Keys**, copy the **Publishable Key**, **Secret Key**, and **Issuer URL**.

Environment Variables

Backend — set in the server environment or `backend/.env`:

```
CLERK_SECRET_KEY=sk_...        # Clerk dashboard → API Keys
CLERK_ISSUER=https://<instance>.clerk.accounts.dev
SECRET_KEY=<random 50+ char string>
DB_NAME=<db name>
DB_USER=<postgres user>
DB_PASSWORD=<postgres password>
DB_HOST= 127.0.0.1 for localhost
DB_PORT=5432 for localhost
```

Frontend — set in `3090_Dashboard/.env`:

```
EXPO_PUBLIC_CLERK_PUBLISHABLE_KEY=pk_...  # Clerk dashboard → API Keys
EXPO_PUBLIC_API_BASE_URL=https://<your-server-domain>
```

ISU Email Restriction

The backend (`backend/accounts/authentication.py`) rejects any Clerk token whose email does not end in `@iastate.edu`.

Navigating Pages and Features

Home Page

Here, you'll see the following components:

Current groups - TAs only see groups that are assigned to them, and Head TAs and Professors can view ALL the available groups

Create group button - press the button that says "+ New Group," a pop-up will appear. Fill out all the necessary fields and click "Add Group" once finished. Do note that TAs and students must already be created to add them.

Filter groups button - located next to the Create group button, once clicked on, it will display the different ways to filter the groups. This is done either by selecting a TA from the dropdown or by status color.

My Tasks bar - the user is able to see any of their assigned tasks and create more by clicking the "+" button on the top right corner

Group Page

This page displays information related to a specific group. To access the page, click on a group's card from the homepage or click on the name of a group from the status page.

Here, you will see the following components:

Individual group information - Displays information related to a group including:

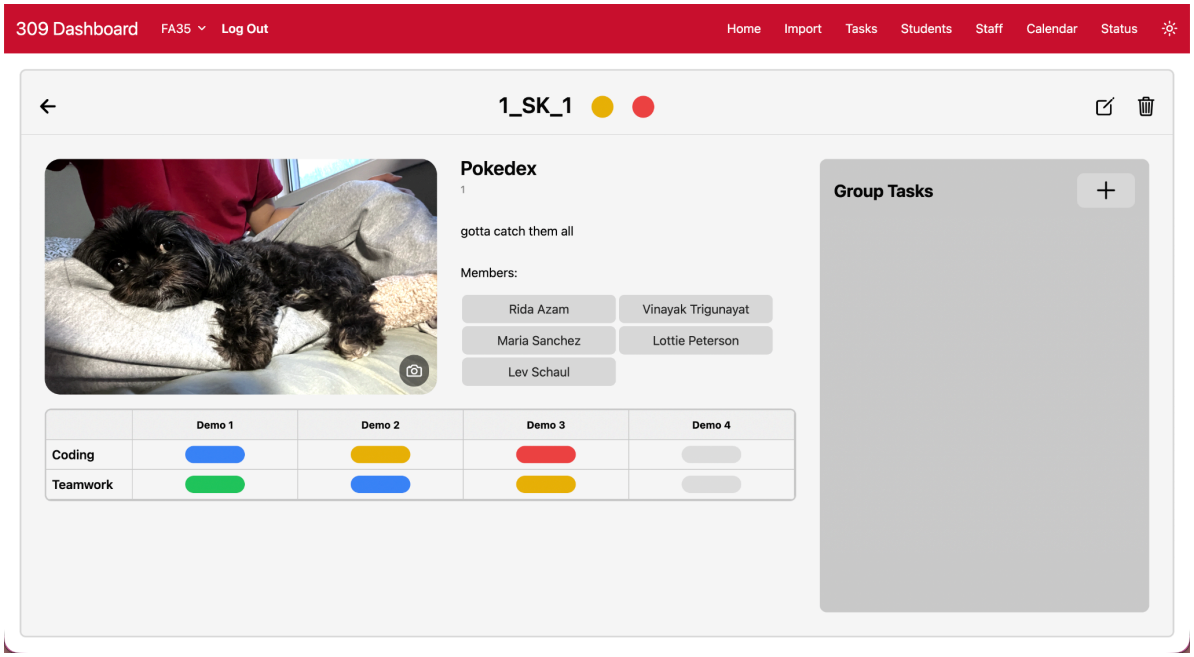
- group name
- group project
- Section
- student members
- TA
- tasks related to that group
- demo status
- group image

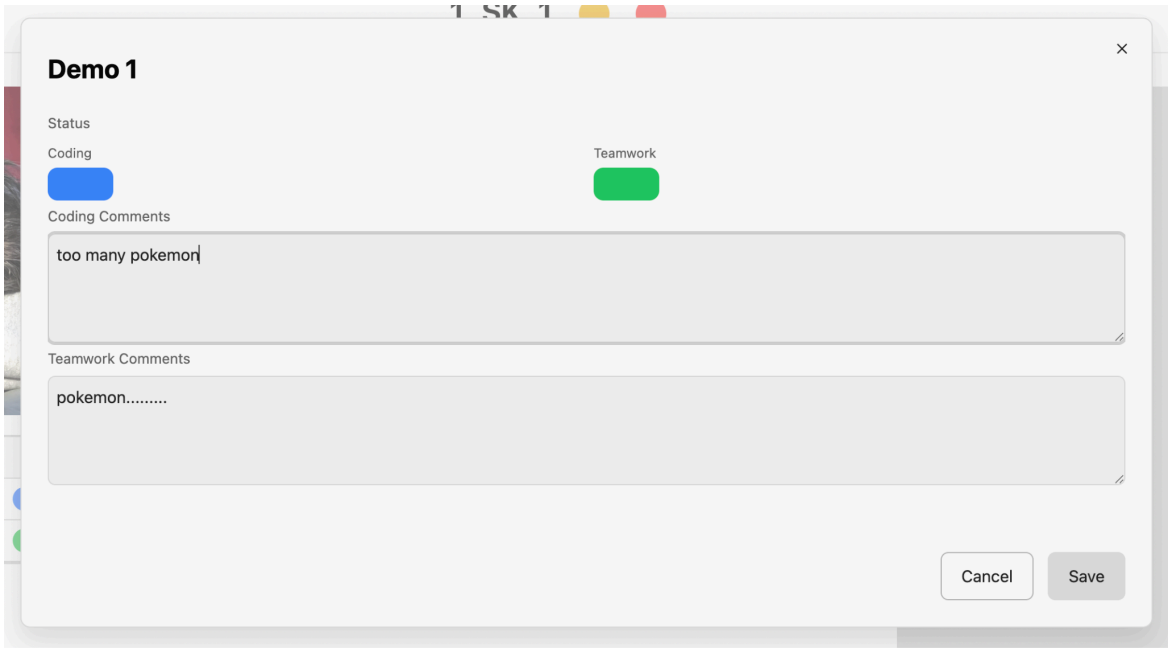
Edit group information button - Clicking this button allows users to edit group information such as group name, project, and demo statuses.

Delete group button - Clicking this button allows users to delete a group.

Individual student button - Clicking this button will show an individual student's page with their information. Check the Individual Student Page section for more information.

Demo status chart - Chart displaying a group's status for their demos. Clicking the name of the demo in the column header will display a modal where users can add comments specific to code and/or teamwork.





Individual Student Page

Here, you will see the following components:

Basic student information that includes their:

- Name
- Group
- ISU netID
- Picture - must be chosen manually by clicking on it and finding the picture you wish to display
- Toggle bar next to name - can be switched on or off to signal whether the student is “at risk.”

Student Demo Statuses - shows how well the student has been performing in the demos. The colors each signify the state they are for a category:

- red (“bad”)
- yellow (“poor”)
- green (“good”)
- blue (“excellent”)

Edit Student button - located next to the flag toggle, once clicked on, it will allow you to edit the various components above, except for their netID. To complete a change, click the green checkmark, or if you choose to cancel, click the red ‘x’.

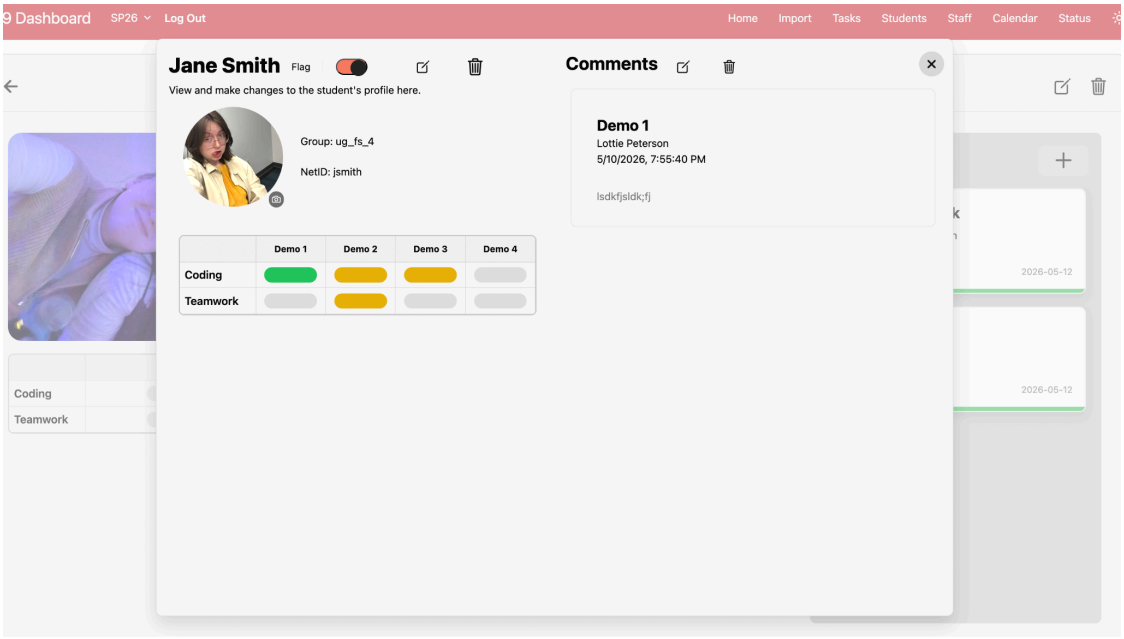
Delete button - located next to the edit button, a user can delete a student from the database.

Comments Section:

Current comments - on the right-hand side, the user can view any current comments that are on the student. Each comment has a title, the commenter's name, a timestamp upon creation, and a description.

Create a comment button - click the creation button next to the Comments header. It will bring you to a pop-up, where you'll fill out the title and comment description.

Delete comment button - located next to the Create comment button; clicking it allows professors to delete any comments by selecting the undesired comment. Once finished, press the 'x' button located next to the Comments header



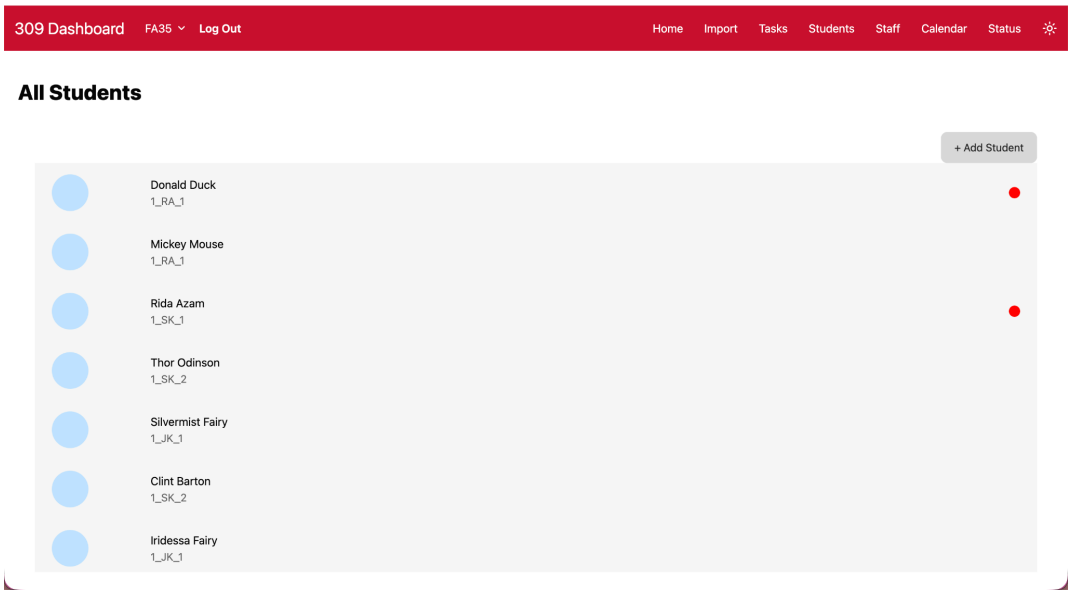
Student Overview Page

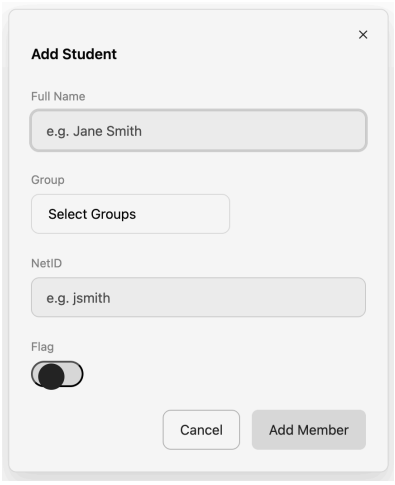
This page displays information related to the current student and their assigned group. To access the page, navigate to it from the main dashboard or sidebar navigation.

Here, you will see the following components:

List of students - Users can scroll through students and quickly see their name, profile picture, their group, and whether or not a student is flagged. Clicking on a student will pop up the Individual Student Page for that student. Check the Individual Student Page for more information.

Add Student Button - Clicking this button allows users to add a student manually by inputting their name, group, and netID.





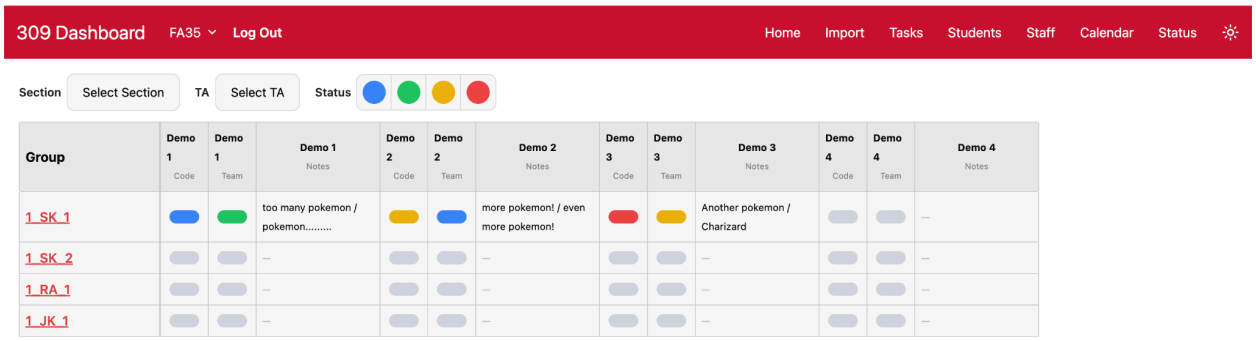
Status Page

The Status Page displays a list of all groups and their statuses as an easy overview for users.

Here, you will see the following components:

Status chart - Displays a group's name, their demo statuses, and comments for each demo.

Filtering - Options to filter the groups based on semester, TA, and/or demo status.

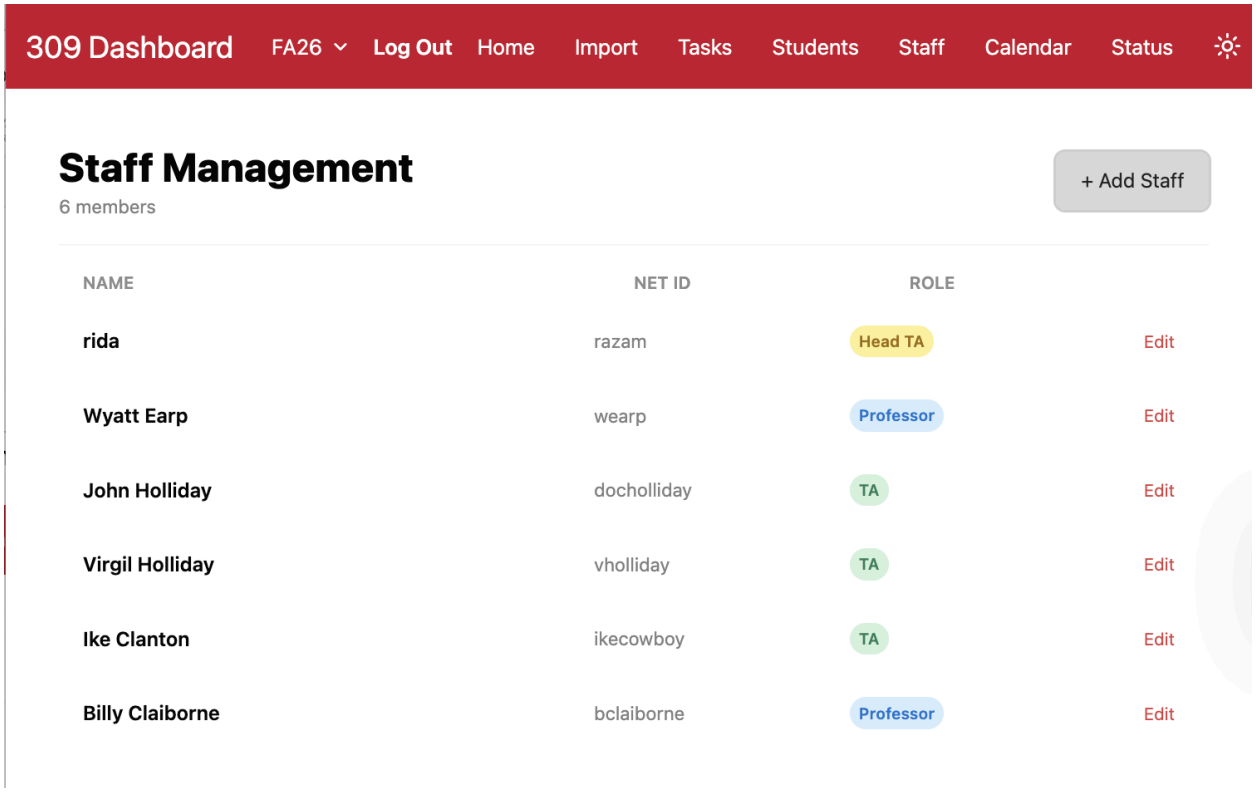


Staff Management Page

The Staff Management page allows Professors and Head TAs to view and manage the staff members associated with the current semester.

Accessing the page

Navigate to Staff in the top navigation bar. The page displays all staff members for the currently selected semester, showing each person's name, NetID, and role.



Roles

There are three staff roles:

- Professor — full access; can add and edit staff of any role
- Head TA — can add and edit TAs only
- TA — read-only access to the staff list; cannot add or edit members

Adding a staff member

Professors and Head TAs will see an + Add Staff button in the top right. Click it and fill in:

- Full Name — the staff member's display name
- NetID — their ISU netID (e.g. jsmith). This must match exactly what is on file, as it is used to link their account on first login.
- Role — select from the roles available to you

Click Add Member to save. The new staff member will appear in the list immediately.

Editing a role

Click the Edit button next to any staff member (you cannot edit your own role). Select a new role from the dropdown and click Save.

Important notes

- Adding a staff member here creates their record in the database but does not send them any notification. They simply need to sign in with their ISU Google account and the system will link their Clerk session automatically.
- The staff list is filtered by the currently selected semester. If a staff member does not appear, check that the correct semester is selected in the dropdown.
- A NetID cannot be changed after a record is created. If an incorrect NetID was entered, the record will need to be deleted and recreated via the Django admin panel.

Import Page

The Import page allows staff to bulk-load students and groups for a semester from a single CSV file, rather than creating them one by one.

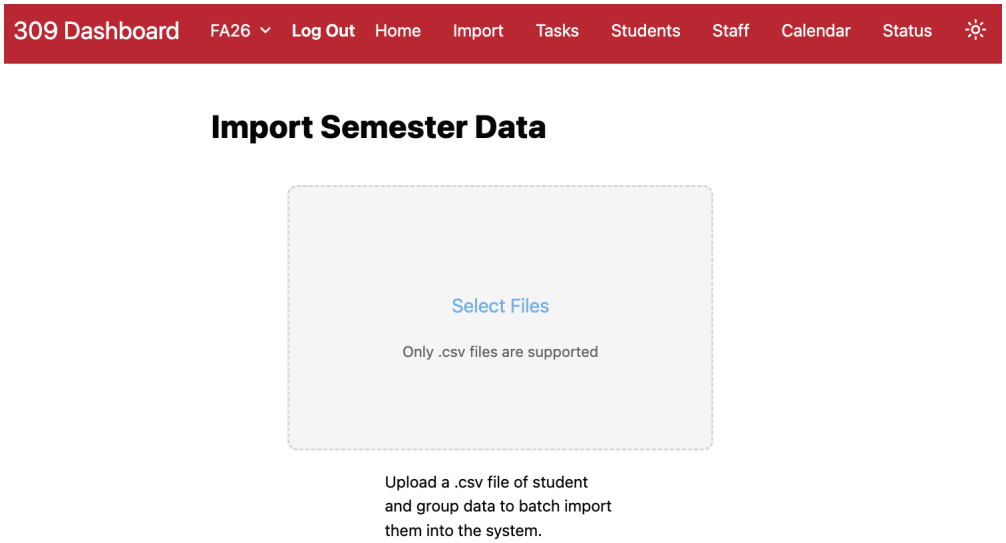
Accessing the Page

Navigate to Import in the top navigation bar. Before uploading, ensure the correct semester is selected in the semester dropdown at the top of the screen — the import is scoped to whichever semester is currently active.

How to Upload

1. Go to the Import page
2. Either drag and drop a .csv file onto the upload area, or click Select Files to browse your device
3. Once a file is selected, its name and size will appear in the upload area
4. Click Upload File to submit

Only .csv files are accepted. If the upload succeeds, a confirmation message will appear and the file selection will be cleared.



CSV Format

The file must contain the following columns:

- First Name — Student's first name
- Last Name — Student's last name
- netid — Student's ISU netID (e.g. jsmith)
- Group — Group identifier string (see Group Format below)

Example:

First Name,Last Name,netid,Group
Jane,Smith,jsmith,1_SK_1
John,Doe,jdoe,1_SK_1
Alice,Jones,ajones,2_RT_2

Group Identifier Format

The Group column drives most of the import logic. It must follow this format:

<section>_<TA initials>_<group number>

For example, 1_SK_1 means:

- Section: 1
- TA initials: SK (e.g. a staff member named "Sarah Kim")
- Group number: 1

The system uses this string to automatically create the section, find the TA, create the group, and link everyone together.

How TAs Are Assigned

The TA is matched by computing initials from each Staff member's name and comparing against the initials in the group string. For example, a Staff member named "Sarah Kim" produces initials SK.

Important: The TA must already exist as a Staff record in the database before importing. If no match is found, the group is still created and the student is still added — but the TA assignment is silently skipped. Always verify all TAs have been added on the Staff Management page before running an import.

What Gets Created

For each row in the CSV, the import will:

1. Create or update the Student — matched by netID. If the student already exists, their name is updated to match the CSV.
2. Create the Section — if a section with that number does not already exist for the selected semester, it is created automatically.
3. Create the Group — if a group with that name does not already exist in that section, it is created automatically.
4. Assign the TA to the Group — if the initials match a Staff record.
5. Add the Student to the Group.

The entire import runs as a single database transaction. If anything fails partway through, no partial data is written and the database is left unchanged.

Recommended Workflow for a New Semester

1. Create the new semester via the Django admin panel (or ensure it already exists)
2. Select the new semester in the semester dropdown
3. Add all TAs for the semester via the Staff Management page
4. Prepare the CSV from your roster (e.g. exported from Canvas or a spreadsheet), ensuring the Group column follows the correct format
5. Upload the CSV via the Import page
6. Verify that groups and students appear correctly on the home page

Local Photo Upload and Storage

Photos for students, staff, and groups are stored locally on the device running the app and are not uploaded to or synced through the backend. This means photos persist across sessions on a single device or browser, but are not shared between users and will be lost if browser storage is cleared.

All photo logic lives in two files. `src/lib/photoService.ts` is the low-level storage layer, and `src/hooks/use-local-photo.ts` is the React hook that components use directly. Each photo is identified by a key of the form `{type}_{id}` (Example: student_14 or group_3). The three valid types are student, staff, and group. `photoService.ts` exposes three functions built around this key scheme: `getPhotoUri`, `setPhoto`, and `removePhoto`.

Because the app runs on both web and native, the storage mechanism differs by platform:

- Web: The raw image blob is saved in the browser's IndexedDB (database LocalPhotos, store photos). A small "idb" marker is written to `AsyncStorage` so the app knows a photo exists for that key. On retrieval, a temporary `URL.createObjectURL()` is returned for the current session.
- Native: The image file is copied into the app's document directory under a photos/ subfolder and saved as `{type}_{id}.jpg`. The file URI is stored in AsyncStorage and read back directly on subsequent loads.

In both cases, AsyncStorage acts as the index with a JSON map from photo keys to either a file URI (native) or the "idb" marker (web).

The useLocalPhoto(id, type) hook is what screens actually call. It loads the current photo URI on mount, and returns a pickPhoto function that opens the device's image library via `expo-image-picker` and saves the result through `setPhoto`.

Code Structure

Frontend

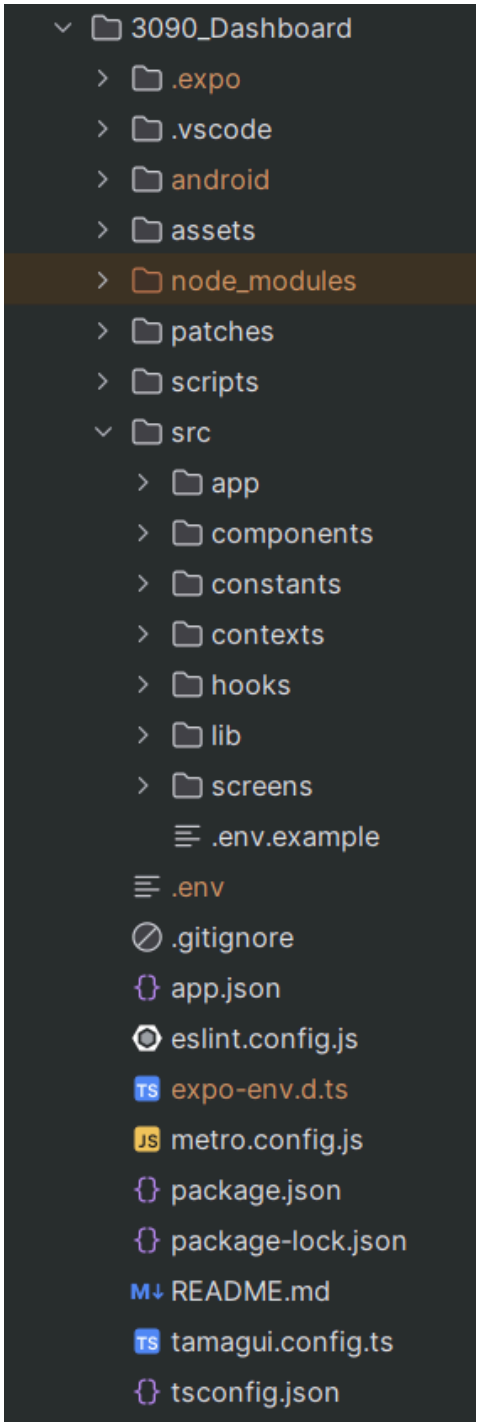
The frontend code for the project lives under the `3090_Dashboard/` folder. This contains our entire React Native project, including the specific environment variables, node modules, and TypeScript/Tamagui code. All of the latter lives under the `src/`

folder. From there, code is organized into several folders, the most important of which are `app/`, `components/`, and `screens/`.

`app/` holds the code for the login/auth components and routing. `_layout.tsx` creates the nav bar that allows you to access each page.

`components/` holds several Tamagui components that are common between pages.

`screens/` holds the code and components for each individual screen in the project. The `index.tsx` file is the main file that gets displayed.



Backend

Concepts in our project are separated into apps in Django. Mostly we used the models, serializers, urls and views. The models are the models of the object, so the Demos app would have multiple models for Group Demo statuses and Student Demo statuses. This

is where the relationships between models happen, so a Group may have multiple Students. Serializers are where we make endpoints,

The backend code for the project lives under the `backend/` folder. This contains our entire Django project, including the environment variables, virtual environment, and all Python application code. The project uses **Django REST Framework (DRF)** to expose a REST API consumed by the frontend.

The top-level structure has a few key pieces. `config/` is the Django project root — `settings.py` holds environment config (database, CORS, Clerk credentials, installed apps), and `urls.py` is where all API routes are registered. All endpoints are served under /api/ using DRF's DefaultRouter, which automatically generates the standard CRUD routes for each registered ViewSet.

The rest of the backend is organized into Django **apps**, one per feature area:

- `accounts/` handles authentication. It contains the Clerk JWT verification logic (authentication.py) and a dev bypass (DevNetIDAuthentication) that lets you authenticate via a `Dev-User: <netID>` header during local development.
- `permissions.py` defines the IsStaff permission class used across most ViewSets.
- `users/` manages Staff records (TAs, Head TAs, Professors).
- `students/` manages Student records.
- `groups/` manages project groups and their membership.
- `sections/` manages course sections and semesters.
- `tasks/` manages the task board items.
- `comments/` manages student comments left by staff.
- `demos/` manages demo status tracking (the colored status indicators per demo per group/student).
- `events/` manages calendar events.

Every app has a few key files with the majority of important information:
`models.py` defines the database schema. This is where we create objects like Groups and Students that can connect to each other with foreign keys. Many models can live under one app, so `demos/` holds multiple models related to group and student demo status.
`serializers.py` converts model instances to/from JSON. This is what will be sent out from the API endpoints.

`views.py` contains the ViewSet with any custom filtering logic. This is where Django was really helpful, because we could define new endpoints to return specific information with very little overhead.
`urls.py` (mostly unused now that the router handles registration in config/urls.py) contains legacy route definitions.

CI/CD Pipeline

The project uses a GitLab CI pipeline that runs automatically on every push to main and every merge request. Its purpose is to catch two categories of problems before they reach the server: broken code and missing migrations.

Stages

The pipeline runs three stages in sequence, each depending on the previous passing:

1. **Build:** verifies the Django settings are valid (manage.py check) and checks that all model changes have a corresponding migration file (makemigrations --check --dry-run). No database is needed here. This catches the common mistake of editing a model without running makemigrations.
2. **Unit Tests** — runs model-level tests against a throwaway PostgreSQL 15 container. These test ORM behavior in isolation, without going through the HTTP or auth layer. Covers all core apps: comments, demos, groups, sections, students, tasks, and users.
3. **Integration Tests** — runs full request/response tests against a fresh PostgreSQL instance. These test the actual API endpoints, including permission enforcement (verifying that unauthenticated or non-staff requests are correctly rejected).

The pipeline is defined in a single file at the root of the repo: `.gitlab-ci.yml`. This is the standard location GitLab looks for automatically, no additional configuration is needed to activate it.

The tests it runs live in each backend app's tests.py file:

- backend/api/tests.py
- backend/comments/tests.py
- backend/demos/tests.py

- backend/groups/tests.py
- backend/sections/tests.py
- backend/students/tests.py
- backend/tasks/tests.py
- backend/users/tests.py

Test Suite

Each tests.py file contains two classes:

- <App>ModelTests — run in the unit test stage. These create model instances directly via the ORM and assert that relationships, constraints, and defaults behave correctly. No HTTP requests are made.
- <App>APITests — run in the integration test stage. These use Django's test client to make real HTTP requests against the API endpoints and check responses. They test that creates, reads, updates, and deletes work correctly end to end.
- <App>PermissionTests — also in the integration stage. These specifically verify that unauthenticated requests and non-staff requests are rejected with the correct status codes. This is what ensures the IsStaff permission class is actually wired up on every endpoint.

The api/tests.py file is the exception — it only contains a HealthCheckTests class that verifies the /api/test/ endpoint is reachable.

To run the full test suite locally:

```
cd backend
python manage.py test --verbosity=2
```

The pipeline intentionally does not test the frontend. All CI coverage is on the backend, where data integrity and access control matter most. The split between unit and integration tests is deliberate; unit tests run faster and catch model-level regressions; integration tests catch auth and permission issues that only surface at the HTTP layer.

Pip dependencies are cached and keyed to a hash of requirements.txt, so the cache is only invalidated when dependencies actually change.

Important note

If the build stage fails, it almost always means either a settings misconfiguration or a model was changed without running makemigrations locally first. Run `python manage.py makemigrations` and commit the resulting file. The pipeline does not

include a deployment stage; there is no CD. Passing CI does not automatically update the server. Deployments are manual: after merging to main, someone needs to SSH into the remote server, pull the latest changes, and restart the backend process.