Documentation

Dev Team Documentation (‘25-’26)

1. Overview

This document serves as the central technical reference for ACM Development Team projects, specifically the website and membership portal. It explains the architecture, setup, deployment, and maintenance workflows to help new and current members onboard quickly.

1.1 Quick Links

- Website: https://www.uclaacm.com/
    - Repo: https://github.com/uclaacm/website
- Subdomain: http://acm.cs.ucla.edu/
- Membership Portal: https://members.uclaacm.com/login
- Password Manager: vault.bitwarden.com/

1.2 General Contribution Guidelines

Work on feature branches

Follow commit message conventions.

Submit pull requests for review before merging into master or any branch

1.3 Other

See passwork.me vault for access to the following

- Jenkins
- EC2, AWS account (note: instances will cost ~$35/month)
- Namecheap
- Netlify

2. ACM Website

The ACM Website is the central hub for our chapter, displaying committees, initiatives, events, about, and many more. We use Google sheets as our content management system (CMS), allowing non-developers to easily update website content without touching code. Please read the README file in ACM Website to get started. It has all the key information you need, including how to run the project locally and how to contribute to the project.

2.1 Tech Stack

- React → Most widely-known Javascript library to build the website
- Next.js → React framework for better performance, SEO, and backend functionalities
- Netlify → Hosts and deploys the website. Automatically updates the live site whenever new commits are pushed to the `master` branch of repo
- ESLint + Stylelint → Tools that automatically check the code quality and formatting
- Design → All visual designs are created by the ACM Design team

2.2 Running the Project Locally

```
git clone https://github.com/uclaacm/website.git
npm install
npm run dev
```

2.3 Key Folders

`/components` → Reusable UI componens

`/data` → pre-generated JSON and small content data used by pages/components

`/pages` → Next.js routes + pages; controls what content is rendered and how it's fetched (SSG/SSR))

`/public` → Static assets (images, files, favicon) served directly at the site root.

`/scripts` → Build-time helpers that fetch/transform content (e.g., Google Sheets => JSON).

`/styles` → SCSS/CSS modules and global styles used by components and pages

3. Membership Portal

The membership portal is split into three repositories:

1. Frontend Repo (membership-portal-ui)
2. Backend Repo (membership-portal)
3. Deployment Repo (membership-portal-deployment)
    a. This repo is private, you have to be added to UCLA ACM group. Ask Dev Directory for access

Why? Partially, in industry you will often see separate frontend/backend repos, especially when frontend & backend teams are separate. But more importantly, because it's been that way since 2017. Trying to merge or restructure them is technically possible, but you're probably better off not touching it. If it works then don't touch it!

3.1 Tech Stack:

- React → Most widely-known Javascript library to build the website
- Node.JS → Runtime environment utilizing JavaScript as its programming language
- Express → Popular web framework known for simplifying server-side development
- PostgreSQL → Database
- AWS → Acts as the storage and deployment backbone
- Docker → Containerization
- ESLint + Stylelint → Tools that automatically check the code quality and formatting

3.2 Running Project Locally:

Open Docker Desktop and wait until it's fully running. Ask officer for .env file info if you don't currently have one.

1. git clone https://github.com/uclaacm/membership-portal-deployment.git
2. git clone https://github.com/uclaacm/membership-portal.git
3. git clone https://github.com/uclaacm/membership-portal-ui.git
4. cd membership-portal-deployment
5. cd dev
6. docker compose build
7. docker compose up -d

3.3 Frontend

The frontend is responsible for handling what users see and interact with.

- Manages the app's visual components and user interactions.
- Connects the app state to UI components via containers.

- Updates state using reducers, actions, and dispatch

3.3.1 Key Terms

- Containers: Connect the app's data to UI components to always display the intended information and handle user logic
- State Management: Controls how data changes and updates across the app
- Redux: JavaScript library primarily used for managing and centralizing application state
    - Store: A central object that holds the entire global state of your application. It acts as the single source of truth for all application data, making it accessible to any component that needs it
- Jenkins: A CI/CD tool used for automating tasks like building, testing, and deploying code

3.3.2 General Logic

- Redux enforces a strict unidirectional data flow, ensuring state changes are predictable and traceable. This flow involves:
    - Actions: Plain JavaScript objects that describe what happened (e.g., user clicked a button, data fetched).
    - Reducers: Functions that take the current state and an action, then return a new state based on the action. Reducers never directly modify the existing state.
    - Dispatch: Functions used to send actions to the store, triggering the reducer to update the state.
    - In short, When something happens in the app (like a user clicks a button), an **action** is sent to **dispatch**, which calls the **reducer** to create a **new version of the state**
- index.js combines all the reducers into one main reducer and this combined reducer is what dispatch communicates with when updating state.
- config.js tells the frontend how to send requests to backend services and what environment settings to use (dev, prod, etc)
- Jenkins
    - See passwork.me vault for access
    - If stuff takes forever to build, try making another temp commit to branch

3.4 Backend

The backend is responsible for handling requests, managing data, and connecting the frontend to the database.

- Processes requests from the frontend.
- Manages database interactions and protects against common attacks.
- Provides API endpoints for the frontend to fetch or update data.

Backend endpoints and documentation at wiki

3.4.1 Key Terms

- Middleware: A software that acts as a bridge, connecting disparate applications, databases, and operating systems to enable them to communicate and exchange data.
- PostgreSQL: A relational database with an ORM (Object-Relational-Mapper). This lets developers write JavaScript-like syntax to interact with the database instead of raw SQL.

3.4.2 General Logic

- Index.js handles most of the server setup
    - Enables local requests: Lets your computer (localhost) talk to the server.
    - Adds unique IDs: Every incoming request gets its own ID so it's easy to track.
    - Set up logging middleware: A function that runs in the middle of each request to log what's happening.
    - Defines routes: Uses server.use('/app/api') to direct traffic to the main API routes.
    - Handles errors: If something breaks, the error middleware catches it and passes it to the next handler.
    - All the /app/api routes are well-documented for easy reference.
- Database
    - Security: Protected against common database attacks.
    - Transactions: The transactionNamespace handles multi-step operations safely.
    - Prototype methods: Functions like User.findUser are defined on the model's prototype, meaning every user instance automatically has access to them.

3.5 Database Structure

- There are 7 tables within the Postgres Database

```
postgres=# \dt
           List of relations
 Schema |    Name     | Type  |  Owner
--------+-------------+-------+----------
 public | activity    | table | postgres
 public | attendances | table | postgres
 public | events      | table | postgres
 public | images      | table | postgres
 public | rsvps       | table | postgres
 public | secrets     | table | postgres
 public | users       | table | postgres
(7 rows)
```

To view the contents of any table:

```
docker exec -it dev-postgres-1 /bin/bash
# This enters the postgres docker container
psql -U postgres
# This brings you into the postgres instance
```

```
\dt
# This lists all of the tables in the database
SELECT * FROM users;
# This lists everything in the users table
/d users
# This lists the structure of the users table
```

Users:

ID	UUID	Email	Access Type	State	First Name	Last Name	Picture	Year	Major	Points	Last Login (UTC)	Created At (UTC)	Updated At (UTC)
1	c272daba-7353-4fff-9afb-50c81720d1be	dylon@g.ucla.edu	STANDARD	ACTIVE	Dylon	Tjanaka	—	3	Computer Science	9001	2025-11-07 00:49:37	2025-11-07 00:49:37	2025-11-07 00:49:37
2	b4b5590e-2b5b-4866-a8ba-a3491ff2fc31	acm@g.ucla.edu	SUPERADMIN	ACTIVE	ACM	chapter at UCLA	—	5	Computer Science	0	2025-11-07 00:49:37	2025-11-07 00:49:37	2025-11-07 00:49:37
3	3c40326c-01bc-470f-8ff1-4b9975f37db1	admin@g.ucla.edu	ADMIN	ACTIVE	Nikhil	Kansal	—	2	Computer Science	0	2025-11-07 00:49:37	2025-11-07 00:49:37	2025-11-07 00:49:37

Column	Type	Modifiers / Default	Description
id	integer	NOT NULL DEFAULT nextval('users_id_seq'::regclass)	Primary key (auto-incremented).
uuid	uuid	—	Universally unique user identifier.
email	varchar(255)	NOT NULL, UNIQUE	User's email address.
accessType	enum_users_accessType	DEFAULT 'STANDARD'	User access level — e.g., STANDARD, ADMIN, or SUPERADMIN.
state	enum_users_state	DEFAULT 'PENDING'	Account state — e.g., PENDING, ACTIVE, SUSPENDED.
firstName	varchar(255)	NOT NULL	User's first name.
lastName	varchar(255)	NOT NULL	User's last name.
picture	varchar(255)	—	URL or file path to user's profile picture.
year	integer	NOT NULL	User's academic year (e.g., 1, 2, 3, 4, 5).
major	varchar(255)	NOT NULL	User's declared major.

points	integer	DEFAULT 0	Gamified or participation points.
lastLogin	timestamp with time zone	—	Most recent login timestamp.
createdAt	timestamp with time zone	NOT NULL	Record creation timestamp.
updatedAt	timestamp with time zone	NOT NULL	Record last updated timestamp.

Indexes:
"users_pkey" PRIMARY KEY, btree (id)
"users_email" UNIQUE, btree (email)
"users_email_key" UNIQUE CONSTRAINT, btree (email)
"users_uuid" UNIQUE, btree (uuid)
"user_points_btree_index" btree (points, points DESC)

Activity

id	uuid	user	type	description	pointsEarned	date	public	createdAt	updatedAt
1	93baae59-49de-4c4f-bc25-737cbcd5e097	2506a817-7d2f-4b81-bf26-c30b3d3353ef	ACCOUNT_CREATE		0	2025-11-07 00:49:53.946+00	t	2025-11-07 00:49:53.946+00	2025-11-07 00:49:53.946+00
2	9965137d-3a7d-45b6-a514-0218504aa691	2506a817-7d2f-4b81-bf26-c30b3d3353ef	ACCOUNT_LOGIN		0	2025-11-07 00:49:53.954+00		2025-11-07 00:49:53.954+00	2025-11-07 00:49:53.954+00
3	60052aae-6418-4117-9de3-38a290697b88	2506a817-7d2f-4b81-bf26-c30b3d3353ef	ACCOUNT_ACTIVATE	Registered - added year and major	0	2025-11-07 00:52:13.18+00		2025-11-07 00:52:13.18+00	2025-11-07 00:52:13.18+00

3.5 DevOps

DevOps is responsible for building, deploying, and maintaining the application environment.

- Compiles frontend code into browser-ready files using Webpack.
- Manages server setup, Docker containers, and Nginx configuration.

3.5.1 Docker Logic

Key Concepts:

- Dockerfile: A script that defines how to build a Docker image.
- Docker Image: A packaged version of an app with all its dependencies.
- Docker Container: A running instance of a Docker image in an isolated environment.
- DevOps: The practice of automating and streamlining software build, test, and deployment.

- Jenkins: A tool that automates CI/CD pipelines, often using Docker.
- Registry (ECR, Docker Hub): A storage place for Docker images, so they can be deployed anywhere

Docker Flow

- Developers write a **Dockerfile** to define how their app should be built. This file creates a **Docker image**, which contains everything the app needs to run. When executed, the image becomes a **Docker container**, an isolated environment running the app. In a DevOps workflow, tools like **Jenkins** automate this process—building the image, running tests in containers, and deploying updated versions to production.

Development-to-Deployment Flow

- Code push triggers Jenkins: Jenkins pulls the latest code, runs linting/tests, and executes build scripts
- Build & publish images: Docker images are built, tagged, and pushed to a registry (ECR, Docker Hub, etc.) for later deployment.
- Deployment / orchestration:
    - Local dev → Docker Compose spins up multiple containers together
    - Staging/production → Deployment scripts pull images from the registry and run containers with environment variables and secrets.
- Runtime & operations: Containers expose ports (e.g., 8080/8000), connect to databases (like PostgreSQL), and are monitored through logs, health checks, and metrics. Rollbacks or redeployments are triggered automatically if issues occur

In short:
Developer pushes code → Jenkins checks & builds → Docker images are published → Deployment scripts start containers → Monitoring ensures the app runs smoothly

3.5.2 General Logic

Docker is the core of our containerization, but General Logic shows how all tools and commands work together from development to deployment.

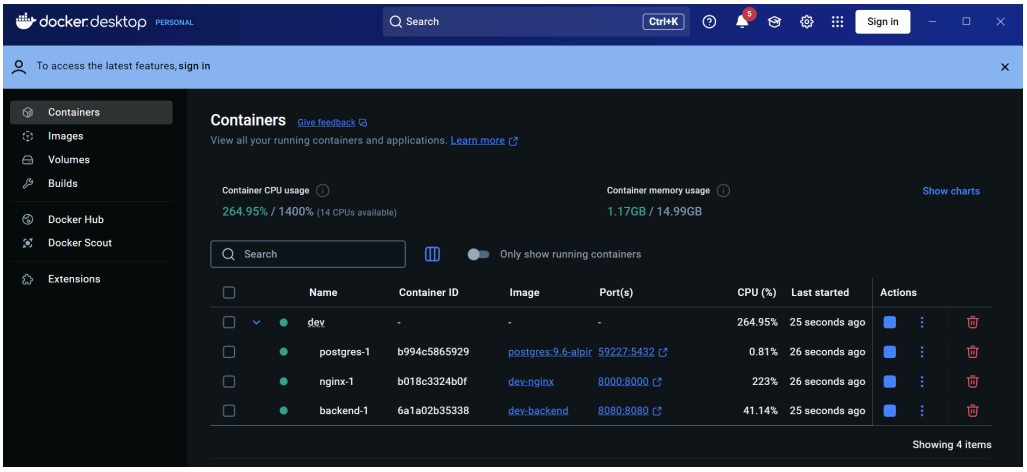
Key Concepts

- Webpack: A build tool that compiles all the React source code into files that browsers can understand. It bundles JavaScript, CSS, and other assets together for efficiency.
- Nginx: Powerful, open-source web server, reverse proxy, load balancer, and HTTP cache. It also handles routing, like sending /app requests to the backend.
    - Reverse Proxy: A server or "middle layer between users and internal server" that receives client requests and forwards them to one or more backend servers. This setup allows you to load balance traffic, increase security by hiding the origin servers, and serve multiple applications from a single IP address
- Alpine image: Lightweight Linux base image used for final Docker container to reduce size

- Nginx.conf: Configuration file that routes API requests (e.g., /app) to the backend while serving frontend files.
- Environment Variables / Secrets: Configuration and credentials provided to containers so they know how to connect to databases or APIs.
- Make Commands: Shortcuts for running common tasks (like building, deploying, or logging into servers)
    - **make dev** → Runs the app in development mode (keeps files in memory).
    - **make ssh** → Opens a secure terminal connection to the AWS EC2 server.
    - **make ecr-login** → Logs into AWS ECR (Elastic Container Registry) to push/pull Docker images
    - **make build** → Compiles both frontend and backend files and preps them for deployment
    - **make build-static** → Compiles frontend code into static files and starts Nginx to serve them

How Everything Ties Together:

1. **Local Development**
    - Run all services locally with Docker Compose:
        - Backend: port 8080
        - Frontend / Nginx: port 8000
        - PostgreSQL: database
    - Webpack compiles the React frontend so the browser can run it, and make dev keeps files in memory for instant reloads.
    - This allows you to test the full stack locally in a production-like environment.



2. Building & Packaging

- Dockerfile + make build package the app (frontend + backend) into a Docker container with all dependencies.

- Make build-static compiles production-ready frontend files into /var/www/membership/static and starts Nginx to serve them.
- The Alpine image keeps the container small by including only the compiled files.

3. Container Registry & Deployment

- Docker images are tagged and pushed to AWS ECR via make ecr-login and make build.
- Deployment scripts or CI/CD pipelines pull the images from ECR and run containers in staging or production with proper environment variables and secrets.
- Nginx ensures frontend requests are served directly, while backend API calls (/app) are routed correctly.

4. Runtime & Operations

- Containers expose necessary ports (e.g., 8080 for backend, 8000 for frontend).
- They connect to services like PostgreSQL.
- Logs, health checks, and metrics are monitored to detect issues, and CI or infra scripts can trigger rollbacks or redeployments if needed.

In short:

Docker Compose → test locally → Make builds & packages → Docker images stored in ECR → Deployment starts containers → Nginx serves frontend, backend runs API → Monitoring keeps everything healthy.

4. The Nitty Gritty (Production & Maintenance)

Overview
Everything happens via Docker and the Makefile inside the membership-portal-deployment repo. There are 3 main containers:

1. **Postgres** (database)
2. **Backend** (API)
3. **Nginx** (frontend/UI)

Local development uses Docker Compose; production uses Makefile commands, ECR, and Jenkins pipelines.

4.1 SSH, Server Access, Important Commands

- make ssh → Logs into the EC2 instance.
    - On mac, will need: chmod 400 id_rsa_west beforehand
- cd into membership-portal-deployment/prod
- Once connected, you can run Docker commands, manage the database, or deploy updates.
- make ecr-login
    - Steps to generate credentials if needed:
    - Go to AWS Console → Services → IAM → Users → nkansal → Security Credentials.
    - Create an access key (Access Key ID + Secret Access Key).
    - On your machine, run: aws configure
    - Enter the Access Key ID, Secret Key, region (us-west-1b), and output format (json, text, or table).
    - Purpose: This configures your local AWS CLI so make ecr-login can authenticate to ECR.
- sudo docker system prune -a (removes all old images)
    - Removes **all unused Docker objects**: stopped containers, unused networks, dangling images, and build caches.
    - **Effect:** Frees up disk space and ensures you don't have old or unused Docker images hanging around that could cause confusion or conflicts.
    - ⚠️ **Warning:** This deletes all unused images and containers, so make sure you don't need them before running it.

4.2 Updating Production

Updating prod" means bringing the production environment (the live app that users access) up to date with the latest code from the deploy branch. Steps to update prod to match deploy branch:

1. In both repos:
   git pull
   git checkout <branch>
   make build
   make push

2. In membership-portal-deploy/prod:
   make deploy

4.3 Deploying

- Merge master into deploy and get PR approvals
- This triggers the Jenkins pipeline for deployment automatically.

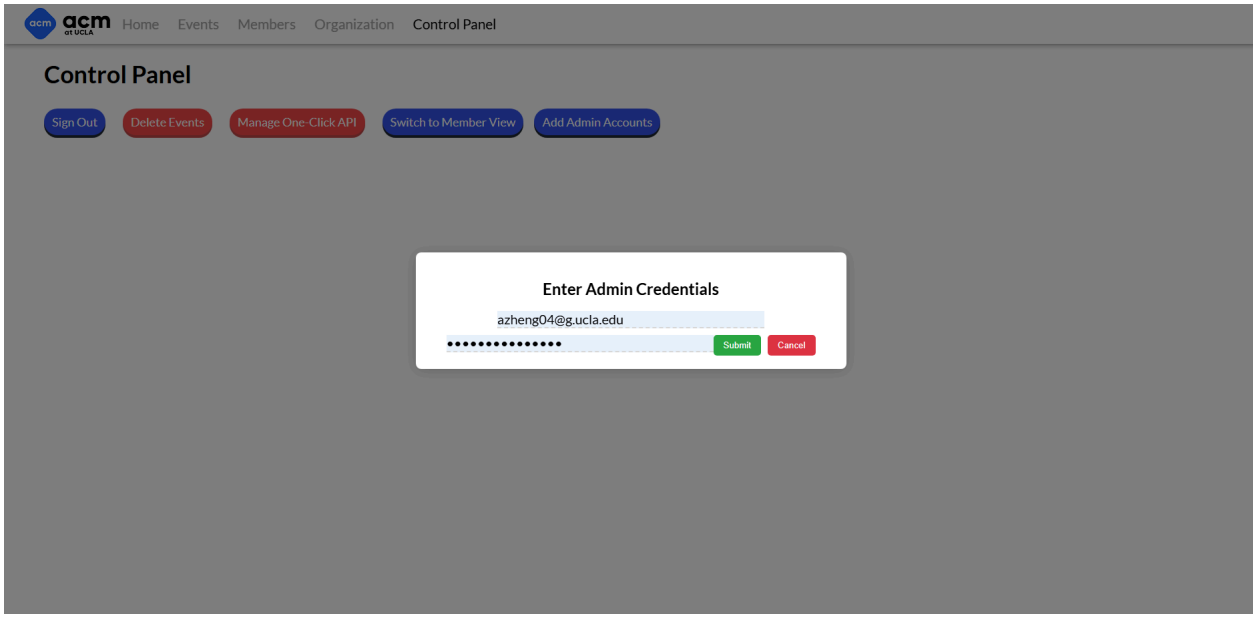
4.4 Creating a New Instance

Creating a new EC2 instance gives you a clean, secure, and reliable environment for deploying the membership portal—whether it's for production, testing, or scaling.

1. **Get a new SSH key**
   Generate a key and update it in the membership-portal-deployment repository so deployment scripts can access the server.
2. **Install Docker and Docker Compose**
   Required to run containers for the backend, frontend, and database.
3. **Decrypt Certificates:** make certs
   Prepares SSL/TLS certificates for Nginx so HTTPS works on the new instance
4. **Decrypt Environment Variables:** make env
   Loads secret variables (like database credentials or API keys) needed for the app.
5. **Deploy the App:** make deploy
   Builds Docker images and starts all containers on the new instance. Ensures the membership portal is live and functioning.

4.5 Adding a New Event as Admin User

1. Login with your @g.ucla.edu account to create an account if one doesn't already exist
2. Two ways (New & Legacy)
3. New:
   a. Sign into membership portal at members.uclaacm.com
   b. Go to settings
   c. Add Admin Accounts button



   d. Add your email (and any other officers to add as admin)
   e. Password is `UCLA_Admin_Pass`
      i. Hard coded for now, will be updated in the future
4. Legacy
   a. cd into membership-portal-deployment/prod
   b. SSH into prod: make ssh
   c. Find Postgres container: docker ps → <container_id>
   d. Enter container: docker exec -it <container_id> /bin/bash
   e. Update admin user in Postgres:

```
psql -U postgres
SELECT * FROM users WHERE email = 'your-email@g.ucla.edu';
UPDATE users SET "accessType" = 'ADMIN' WHERE email =
'your-email@g.ucla.edu';
```

   f. Log out and log back in using your @g.ucla.edu email → now you can add events.

4.6 Certificates

- **THESE STEPS WILL DROP THE DB. MAKE SURE YOU TAKE A BACKUP**
- Certificates expire every 3 months.
  - Last renewed: 2025-11-19
  - Next renew: 2026-02-19
- Steps to renew:
  1. SSH into prod: make ssh
  2. Backup DB: make pg_bkup
  3. Stop services: sudo make stop
  4. Renew certificates: sudo make renew-certs
     a. Option 1

    b. Domain Name(s): members.uclaacm.com
5. Package certs: sudo make package-certs
    a. Passphrase: uclaacm
    b. Overwrite: yes
6. Git add, commit, and push it (If push doesn't work, just proceed to deploy)
7. Deploy: make deploy

- Verify by checking the cert expiration (https certs) on the membership portal page.

4.7 Restoring/Backing Up Database

- To save a backup:
    - make pg_bkup (want to take this off the instance)
- To restore backup in SSH:
    1. Run docker ps to find <container_id> for postgres container
    2. Enter container: docker exec -it <container_id> /bin/bash
    3. Recreate database:

```
psql -U postgres -d template1
DROP DATABASE IF EXISTS postgres;
CREATE DATABASE postgres;
\q
psql -U postgres -d postgres -f /backup/<backup_file_name>
```

4.8 Extracting Database Data to Local Machine

1. ssh into EC2 server because database is hosted there
2. make psql will connect you to production database
3. run \copy command in postgres to convert to csv
    a. Example Query:

```
\copy (
  SELECT concat("firstName", ' ', "lastName")
  FROM (
    SELECT attendances.user AS user
    FROM attendances
    WHERE event=<EVENT UUID>
  ) AS s
  INNER JOIN users ON users.uuid = s.user
) TO '<PATH TO FILE>'
```

    This example gets a list of users who attended a specific event and saves it as a CSV
4. Move file from Postgres container to EC2 host (ssh into postgres container to copy files)
   docker cp <container_id>:/<PATH TO FILE> .
5. Move files from EC2 to local machine
   sudo scp -i id_rsa_acm_west ec2-user@members.uclaacm.com:<PATH TO FILE> .

4.9 Resetting the Membership Portal

- Create a milestone (after each quarter)
- Yearly, delete all the users (not done yet)
    - Another option to build, something that if they haven't logged in in a year, send email saying account will be deleted

4.10 Reset Portal for the Year

1. In the membership-portal-deployment repository, cd into prod and run make ssh
    a. On mac, will need: chmod 400 id_rsa_west beforehand
2. cd into membership-portal-deployment/prod
3. run docker ps to find <container_id> for postgres container
4. run the following commands
    a. docker exec -it <container_id> /bin/bash
    b. psql -U postgres
    c. UPDATE users SET points = 0;

5. Discord Bot:

To add the bot to the server, use the discord permissions calculator (https://discordapi.com/permissions.html), select the necessary permissions, put the client id from the dev portal, and use "bot application.commands" as the scope to generate the required join link

Verification Bot Github

Discord Bot Github

6. ACM Design Requests

Add a ticket to the following notion: ACM Design Notion
You may have to reach out to an ACM Design CPrez to get access to the notion

Problems/Tips

Entries should be formatted in the following order:

1. **Title / Short Description**
   E.g., “Not pulling the latest changes before committing.”
2. **Symptoms / How it Shows Up**
   E.g., “Merge conflicts appear when you try to push, or your changes are overwritten.”
3. **Cause / Why it Happens**
   E.g., “Working on an outdated branch without syncing from main/master.”
4. **Solution / Best Practice**
   E.g., “Always run git pull origin main before starting work and before pushing commits.”
5. **Optional Tips / Notes**
   E.g., “Consider enabling auto-fetch in your IDE to stay updated.”
6. **Categorization**
   - Git / Version Control
   - Frontend / UI Issues
   - Backend / API Mistakes
   - Deployment / Hosting Errors
   - Data / Spreadsheet Integration Errors

Website

Short Description	How it shows up	Cause	Solution/Tips	Categorization

MP - Deployment

Description	Issue	Solution/Tips/Notes

Membership Portal Debugging	`docker build -t 527059199351.dkr.ecr.us-west-1.amazonaws.com/membership-portal:deploy /home/ec2-user/membership-portal`
	`aws ecr get-login-password --region us-west-1 | docker login --username AWS --password-stdin 527059199351.dkr.ecr.us-west-1.amazonaws.com`
	`docker push 527059199351.dkr.ecr.us-west-1.amazonaws.com/membership-portal:deploy`
	`cd ~/membership-portal-deployment/prod`
	`make deploy`
	
	`docker build -t 527059199351.dkr.ecr.us-west-1.amazonaws.com/membership-portal-ui:deploy /home/ec2-user/membership-portal-ui`
	`aws ecr get-login-password --region us-west-1 | docker login --username AWS --password-stdin 527059199351.dkr.ecr.us-west-1.amazonaws.com`
	`docker push 527059199351.dkr.ecr.us-west-1.amazonaws.com/membership-portal-ui:deploy`
	`cd ~/membership-portal-deployment/prod`
	`make deploy`	

MP - Frontend

Short Description	How it shows up	Cause	Solution/Tips	Categorization

MP - Backend

Short Description	How it shows up	Cause	Solution/Tips	Categorization