Galvanize

Content Hub

Design Document

March 30th , 2020
Version 1.0

Team Galvatron

David Guo	57734162
Pengwei Zhou	73569758
Peter Han	14912166
Tony Kong	47078150
Tyler Nee	22705157
Vickie Yen	13358156
Yuting Wen	44625168

Revision History

Date	Version	Status	Prepared by	Comments
Jan 27, 2020	0.1	Start	Team Galvatron	Initial design document submission
Mar 30, 2020	1.0	Final submission	Team Galvatron	Final design document submission

Stakeholder Sign-off List

Role	Name	Signature	Date
Project Sponsor	Peter Smith		

Introduction

Overview

The product will consist of a website that consolidates content and points to the content's original location from other websites and repositories. The site will have admins from contributing organizations monitoring content published to the site via an admin approval system. The main interface of the website will contain a portal to access linked articles. Any individual will be able to create an account and submit content to the site subject to admin approval.

Goals

- The scope of the Content Hub is to aggregate user submitted blog posts and blog streams on a single platform. More specifically:
  - Users can submit posts and blog streams.
  - Users can search and filter posts.
  - Users can view posts via an RSS feed.
  - Users can receive notifications of newly published posts.
  - Approvers can approve user submitted posts.
  - Admins can add other admins, add/remove approvers, add blog categories,
  - A scheduler runs to scan approved blogs for content to publish and send emails to subscribed users.
- Multi-browser support (Chrome, Microsoft Edge, Safari, Firefox)

Assumptions

- There is a maximum of 30 approvers
- Maximum articles is 5000

Programming Environment

Languages: Javascript ES9
Libraries/Frameworks: React 16.13.1, Express 4.17, Passport 0.4.1, Material-UI 4.9.8, Sequelize 5.25.5, cron 1.8.2
Tools: Babel 7.9.0, npm 6.13.4, Swagger
Database: MySQL Community Server 8.0.19
Environment: Node.js 12.16.1
Operating System: Ubuntu 18.04 LTS
IDE: Visual Studio Code 1.43
Source control: GitLab

Production and Test Environments

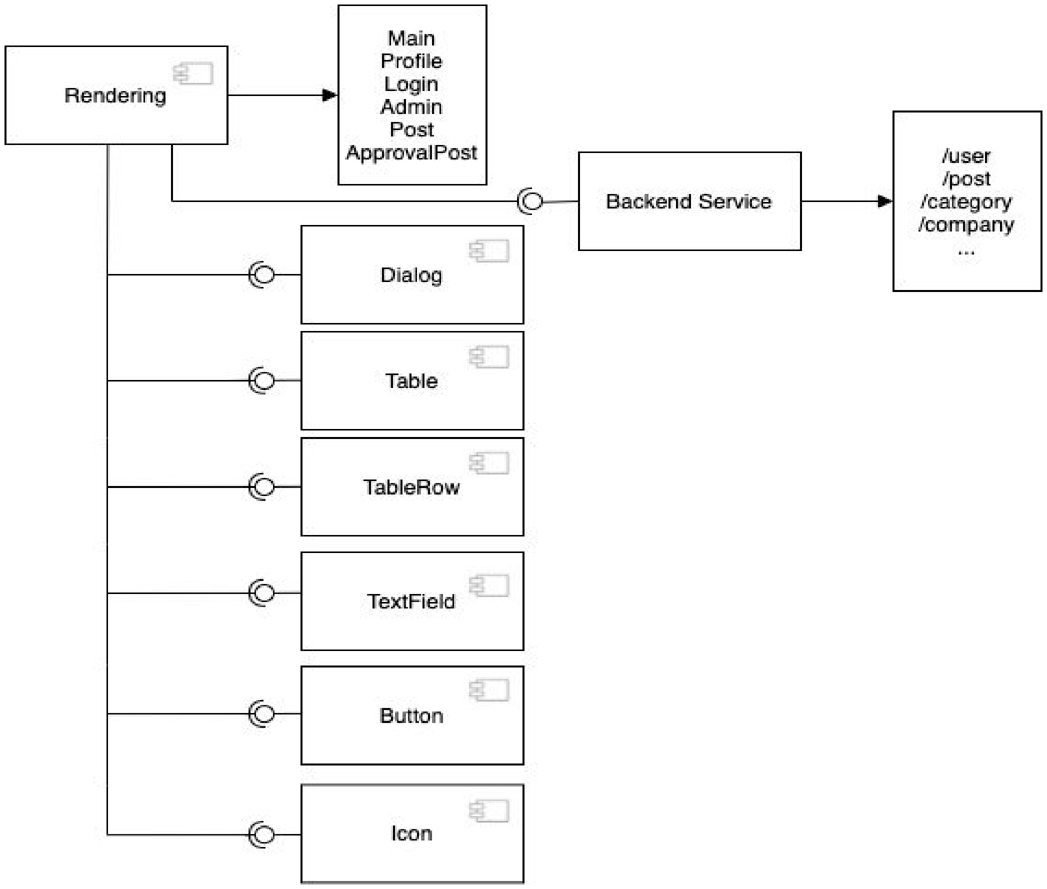
Our production environment will be deployed on a free tier Amazon EC2 t2.micro instance. We will have scripts that will be run to pull the latest code and rebuild the environment.

Our test environment will be the same as our production environment as we do not have the resources to deploy another Amazon EC2 instance. This environment will be used to run manual tests, while our automated tests will be run locally.

Software Architecture

Client side

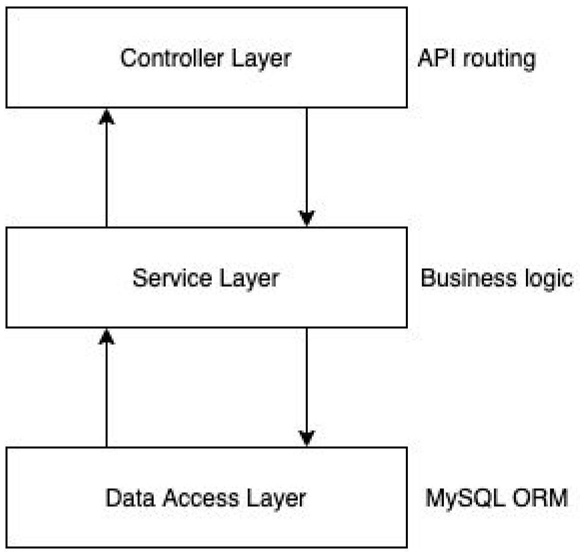
We will be using React along with Material-UI for our UI. We will be having multiple pages that will be built up from multiple components.



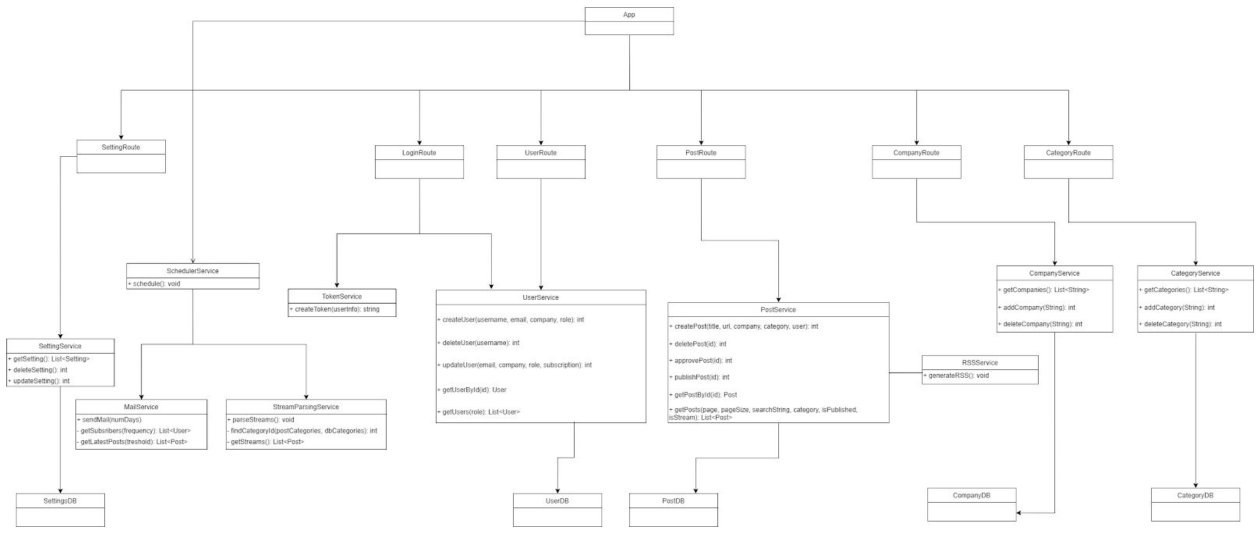
Server side

3 Layer Architecture Model

On the server side we will be separating our logic with a 3 Layer Architecture model. We will be separating our routing logic into a controller layer, our business logic into our service layer and our database access into our data access layer.



UML Diagram
Here’s a UML diagram describing our planned modules and dependencies. We will be using Express for our routing and Passport for our authentication. Our data will be fetched using Sequelize as our MySQL ORM. To look at the diagram in more detail:
https://drive.google.com/file/d/1XdlQfMoRCmLgVX_1BFeMPJebkF_tY1I6/view?usp=sharing


App
SettingRoute
LoginRoute
UserRoute
PostRoute
CompanyRoute
CategoryRoute
SchedulerService
+ schedule(): void
TokenService
+ createToken(userInfo): string
UserService
+ createUser(username, email, company, role): int
+ deleteUser(username): int
+ updateUser(email, company, role, subscription): int
+ getUserById(id): User
+ getUsers(role): List<User>
PostService
+ createPost(title, url, company, category, user): int
+ deletePost(id): int
+ approvePost(id): int
+ publishPost(id): int
+ getPostById(id): Post
+ getPosts(page, pageSize, searchString, category, isPublished, isStream): List<Post>
RSSService
+ generateRSS(): void
CompanyService
+ getCompanies(): List<String>
+ addCompany(String): int
+ deleteCompany(String): int
CategoryService
+ getCategories(): List<String>
+ addCategory(String): int
+ deleteCategory(String): int
SettingService
+ getSetting(): List<Setting>
+ deleteSetting(): int
+ updateSetting(): int
MailService
+ sendMail(numDays)
- getSubsribers(frequency): List<User>
- getLatestPosts(treshold): List<Post>
StreamParsingService
+ parseStreams(): void
- findCategoryId(postCategories, dbCategories): int
- getStreams(): List<Post>
SettingsDB
UserDB
PostDB
CompanyDB
CategoryDB

Authentication

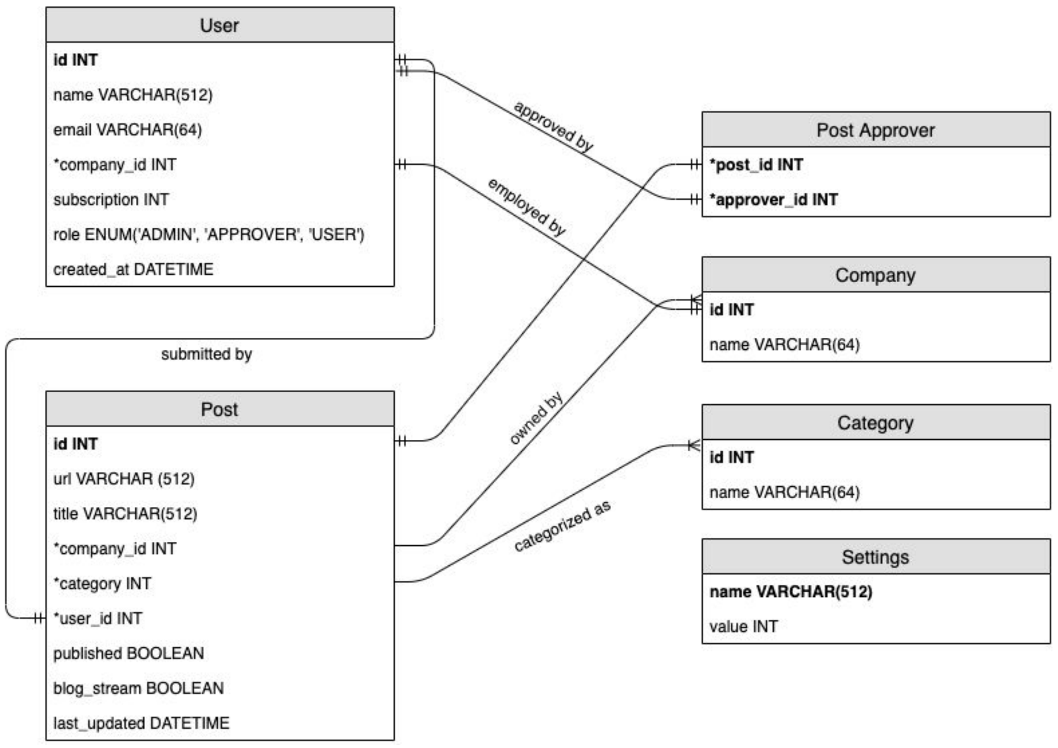
We will be using Google OAuth2.0 for our authentication workflow. After going through our authentication workflow we will be managing our user session using cookies. The benefit of cookie authentication is that the user session will be persisted for a certain amount of time, so the user will not have to re-authenticate on every page visit. Our authentication workflow is as follows:

1. **Client**: Goes through OAuth2.0 process and fetches back the google user including google access token. Sends google access token to server.
2. **Server**: Verifies google access token and creates a new user if one doesn't already exist. Generate a JWT by signing the user id + role. Add to cookies and send to client
3. **Client**: Receives response with user, and cookie is set.

Data Design

Entity-Relationship Diagram

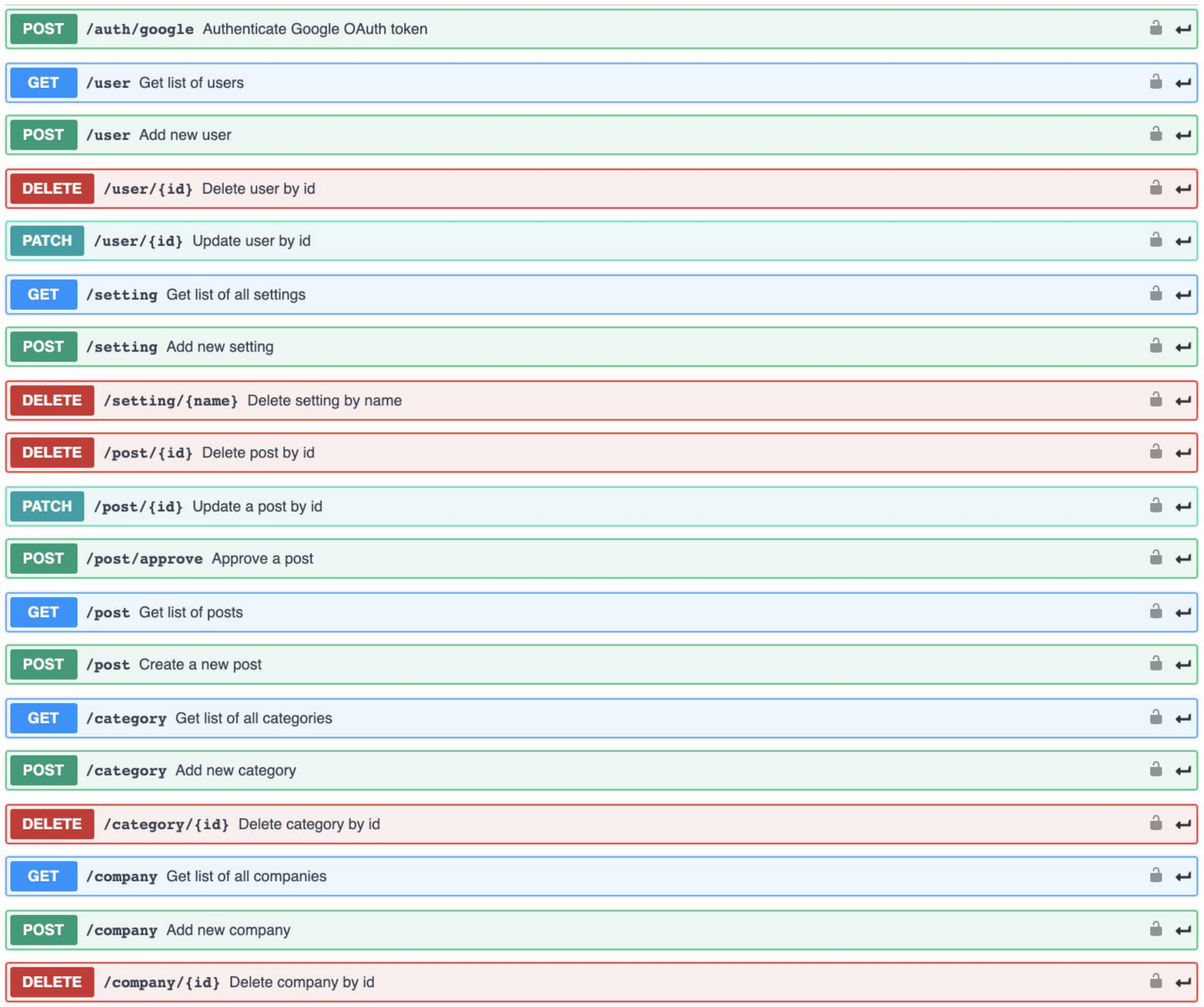
Our database will have 6 tables. Each table User, Post, Company, Category, and Settings represents an entity and holds information to do with that entity. The Post Approver table holds information regarding the relationship between posts and approvers - who has approved what post. This will allow us to show the names of the approvers who have approved a post.



API Design

We have used swagger to create our API documentation. Follow this link for more detailed information: https://app.swaggerhub.com/apis/vickieyen/content_aggregator/1.0.0-oas3

We will be creating a RESTful API, that supports create, retrieve, update, and delete operations on each of our database entities as appropriate. When retrieving posts, we decided to use pagination in order to reduce load times. For instance, in GET /post, two additional parameters are provided, page and pageSize. These parameters will ensure that only pageSize entries are returned for the corresponding page provided. Most of our endpoints require authentication, in order to be authenticated, the request will need to include a session cookie.



POST /auth/google Authenticate Google OAuth token
GET /user Get list of users
POST /user Add new user
DELETE /user/{id} Delete user by id
PATCH /user/{id} Update user by id
GET /setting Get list of all settings
POST /setting Add new setting
DELETE /setting/{name} Delete setting by name
DELETE /post/{id} Delete post by id
PATCH /post/{id} Update a post by id
POST /post/approve Approve a post
GET /post Get list of posts
POST /post Create a new post
GET /category Get list of all categories
POST /category Add new category
DELETE /category/{id} Delete category by id
GET /company Get list of all companies
POST /company Add new company
DELETE /company/{id} Delete company by id

Notable Trade-Offs

- Using **pagination** as opposed to **infinite scroll** may reduce loading time.
    - As mentioned under the API Design Section, we will be using pagination to retrieve posts, as this will speed up home page load time. As there are more and more posts published overtime, using infinite scroll would hinder performance time, thus using pagination is a better choice.
- Having **separate pages for approving unpublished posts** as opposed to **aggregating all posts (unpublished/published) on the same page**
    - We decided to keep unpublished posts and published posts on separate pages, as this was more organized and easier for admins and approvers to approve posts.
- **No access to deleted posts** as opposed to **access to deleted posts** allows us to not keep track of previously deleted posts.
    - We will not keep track of deleted posts, as there is no requirement for admins to be able to access deleted posts. Keeping track of deleted posts will require database changes and more work and we would like to focus on the goals outlined under the Goals Section.

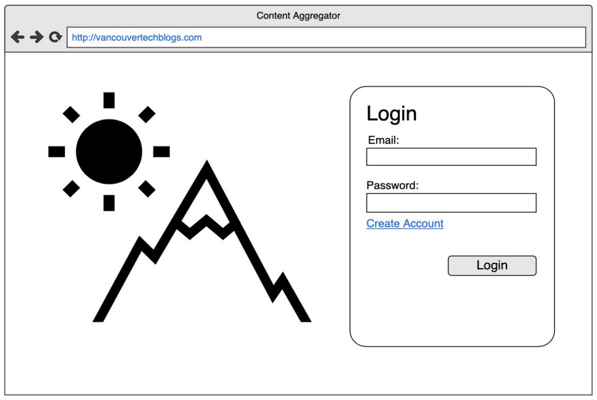
Notable Risks

- Deploying to AWS could prove to be difficult and burdensome, as no team members in the group have experience in this. Therefore, this might take more time than required.
- Security risks with user session management, as no team members know the proper security protocols for properly managing user tokens.

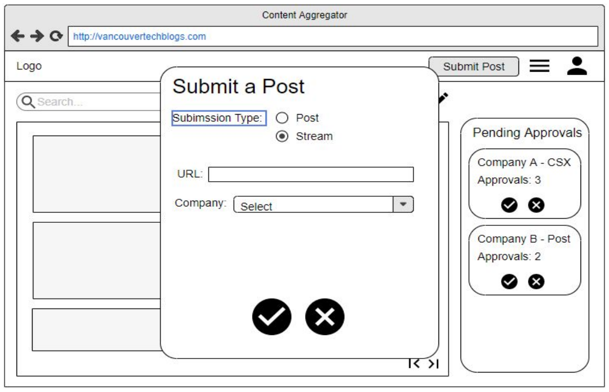
UI Mockups

Our initial UI Mockups include ideas for our login page, user profile page, home page, and admin panel.

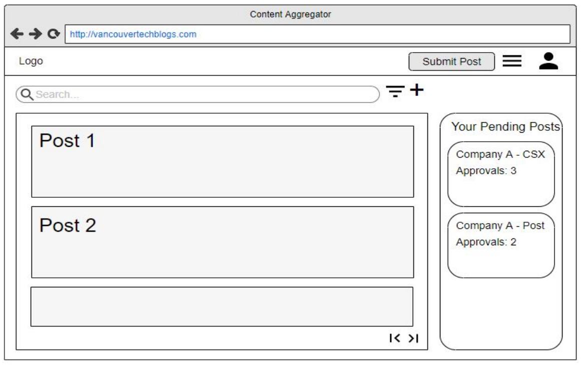
Login Page:



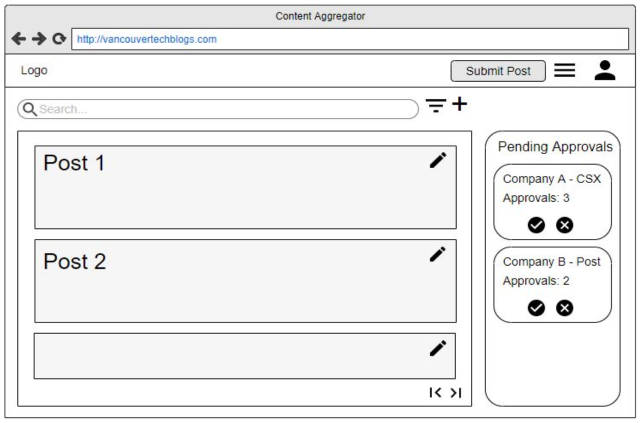
‘Submit Post’ Popup (Post):



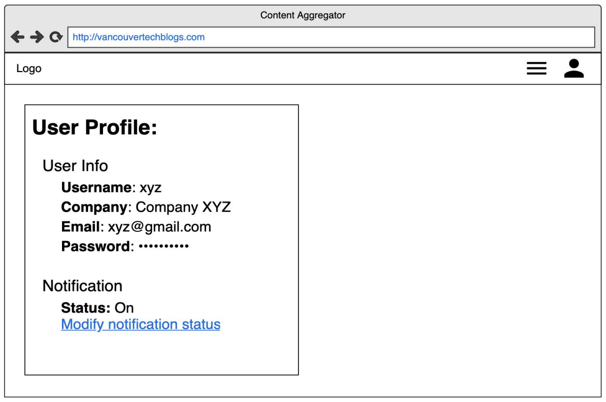
Home Page (User View):



Home Page (Admin View):



User Profile:

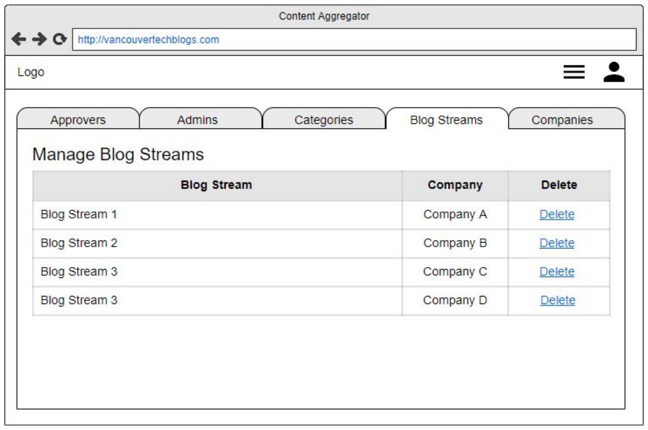


User Profile:

User Info
- **Username**: xyz
- **Company**: Company XYZ
- **Email**: xyz@gmail.com
- **Password**: ••••••••••

Notification
- **Status:** On
- Modify notification status

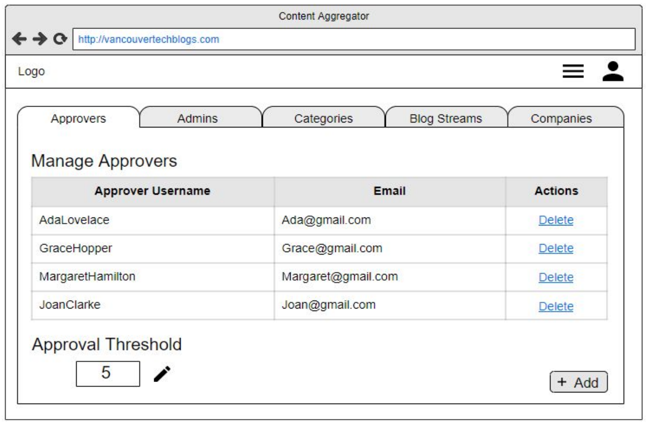
Admin Panel (Blog Stream Tab):



Manage Blog Streams

Blog Stream	Company	Delete
Blog Stream 1	Company A	Delete
Blog Stream 2	Company B	Delete
Blog Stream 3	Company C	Delete
Blog Stream 3	Company D	Delete

Admin Panel (Approvers Tab):



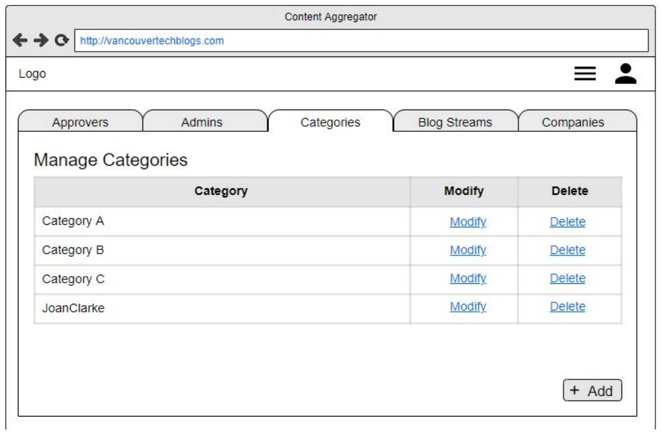
Manage Approvers

Approver Username	Email	Actions
AdaLovelace	Ada@gmail.com	Delete
GraceHopper	Grace@gmail.com	Delete
MargaretHamilton	Margaret@gmail.com	Delete
JoanClarke	Joan@gmail.com	Delete

Approval Threshold

5

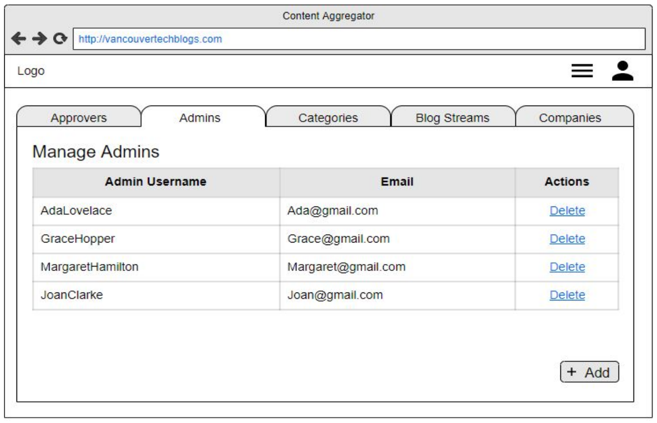
Admin Panel (Categories Tab):



Manage Categories

Category	Modify	Delete
Category A	Modify	Delete
Category B	Modify	Delete
Category C	Modify	Delete
JoanClarke	Modify	Delete

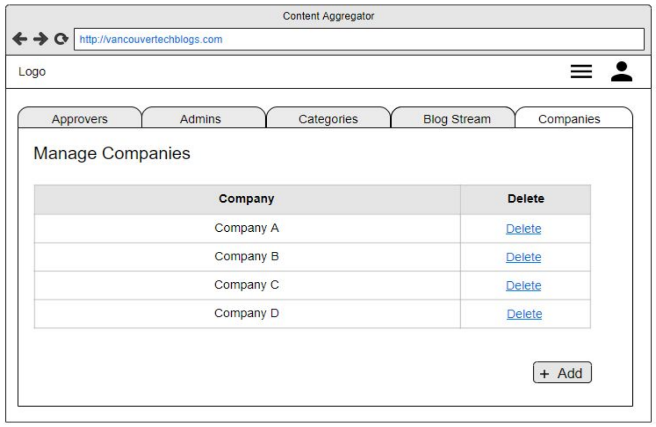
Admin Panel (Admins Tab):



Manage Admins

Admin Username	Email	Actions
AdaLovelace	Ada@gmail.com	Delete
GraceHopper	Grace@gmail.com	Delete
MargaretHamilton	Margaret@gmail.com	Delete
JoanClarke	Joan@gmail.com	Delete

Admin Panel (Companies Tab):



Manage Companies

Company	Delete
Company A	Delete
Company B	Delete
Company C	Delete
Company D	Delete