

Entwicklung einer Social-Media-Plattform zur Veröffentlichung von GIFs und Integration moderner Datenbanktechnologien

Marko Livajusic

28. Juni 2024

Inhaltsverzeichnis

1 Einführung	2
2 Funktionen	2
2.1 GIFs hochladen, suchen	2
2.2 Follow-System	2
2.3 Liken	2
2.4 Kommentieren	3
2.5 Individualisierung der Benutzerprofile	3
2.6 Suchfunktion von Nutzern bzw. von GIFs durch Kategorien	3
2.7 Sortierung von GIFs nach bestimmten Kategorien	3
2.8 Benachrichtigungen-System bzw. Message-System	3
2.9 Tech-Stack	3
2.10 Projekt-Struktur	3
2.11 Schwierigkeiten / Design-Entscheidungen	4
3 Datenbankdesign	4
3.1 Entitätstypen (Tabellen)	4
3.1.1 AuroraUsers	4
3.1.2 Roles	5
3.1.3 AuroraGifs	5
3.1.4 GifCategory	5
3.1.5 BelongsTo	6
3.1.6 Follows	6
3.1.7 Comments	6
3.1.8 Likes	7
3.1.9 ProfilePictures	7
3.1.10 Settings	7
3.1.11 Notifications	8
3.2 Beziehungen	8

3.2.1	Relationenmodell (optimiert)	9
4	Ein Tieftauchgang durch die Codebase: Ausführungen von SQL-Code	9
4.1	NavigationBar.java	10
4.2	HomeView.java	13
4.3	LoginView.java	18
4.4	UploadView.java	18
4.5	MyProfileView.java und UserProfileView.java	19
4.6	Settings.java	22
5	Screenshots	24

1 Einführung

Aurora ist eine Plattform, auf der Nutzer GIFs hochladen und mit anderen teilen können. Auf Aurora können Benutzer ihre GIFs präsentieren, durch verschiedene Kategorien stöbern und mit der Community interagieren. Die Anwendung ist darauf ausgelegt, eine unterhaltsame Möglichkeit zu bieten, visuelle Inhalte mit Freunden und anderen Interessierten zu teilen.

2 Funktionen

2.1 GIFs hochladen, suchen

Auf Aurora lassen sich eigene GIFs hochladen, löschen und mit Beschreibungen versehen. Diese Funktionen sorgen dafür, dass jedes GIF leicht auffindbar und verständlich ist. Durch die Suchfunktion können Benutzer gezielt nach bestimmten GIFs suchen, was die Benutzerfreundlichkeit erheblich steigert.

2.2 Follow-System

Das Follow-System von Aurora ermöglicht es Benutzern, anderen Nutzern zu folgen. Dies schafft eine dynamische und interaktive Community, in der Benutzer über die Aktivitäten ihrer Lieblingskreatoren informiert bleiben und deren neue GIFs leicht finden können.

2.3 Liken

Mit der Like-Funktion können Benutzer ihre Wertschätzung für GIFs ausdrücken. Diese Interaktionsmöglichkeit fördert die Benutzerbindung und ermöglicht es den Erstellern von Inhalten zu sehen, welche ihrer GIFs besonders gut ankommen.

2.4 Kommentieren

Die Kommentarfunktion von Aurora erlaubt es den Benutzern, ihre Gedanken und Meinungen zu den GIFs zu teilen. Dies fördert die Interaktion und den Austausch zwischen den Benutzern, wodurch eine lebendige und engagierte Community entsteht.

2.5 Individualisierung der Benutzerprofile

Benutzerprofile auf Aurora sind individuell anpassbar. Benutzer können ihre Profile mit einer Biografie und einem Profilbild personalisieren, was ihnen hilft, sich in der Community zu präsentieren und mit anderen Nutzern zu verbinden.

2.6 Suchfunktion von Nutzern bzw. von GIFs durch Kategorien

Aurora verfügt über eine leistungsstarke Suchfunktion, mit der Benutzer nach anderen Nutzern suchen können. Durch Vaadins **Notification** erhält man außerdem quasi ein „Auto-Complete“, sodass das Suchen nach Nutzern leichter fällt. Insgesamt erleichtert die Suchleiste das Finden von Nutzer und das Vernetzen.

2.7 Sortierung von GIFs nach bestimmten Kategorien

Es lassen sich GIFs nach bestimmten Kategorien suchen und sortieren. Dies erleichtert die Navigation und das Auffinden von GIFs, die den Interessen der Benutzer entsprechen.

2.8 Benachrichtigungen-System bzw. Message-System

Zuletzt verfügt Aurora über ein Benachrichtigungssystem, das Benutzer über Aktivitäten informiert, die sie betreffen, wie z.B. neue Beiträge von den Personen, denen man folgt. Dieses System kann auch als ein Nachrichtensystem fungieren, das die direkte Kommunikation zwischen den Benutzern ermöglicht.

2.9 Tech-Stack

Für die Umsetzung des Projekts habe ich die Programmiersprache Java verwendet, das Spring Boot Framework für die App selbst und Vaadin für die Benutzeroberfläche und die grafische Darstellung. Für das Datenbankmanagementsystem habe ich PostgreSQL verwendet und das gesamte Projekt wird mit Maven kompiliert.

2.10 Projekt-Struktur

Aurora setzt sich aus den folgenden Ordnern zusammen (`/src/main/java/com/livajusic/marko`):

1. **/configuration:** In diesem Ordner wird Spring Security modifiziert, um Anfragen zuzulassen und die von den Benutzern hochgeladenen Inhalte so zu regulieren, wie sie vom Server bereitgestellt werden sollen.
2. **/db_repos:** Interfaces aus diesem Ordner erweitern JpaRepository und ermöglichen so den Zugriff auf die jeweiligen Repositories.
3. **/services:** Klassen aus diesem Ordner interagieren mit den Repositories aus (**/db_repos**) und aktualisieren die Werte in den Datenbanktabellen.
4. **/tables:** Klassen aus diesem Ordner verwenden JPA, um die von Spring erstellten Tabellen zu modellieren.
5. **/views:** Klassen aus diesem Ordner verwenden Vaadin, um die App im Web anzuzeigen.

2.11 Schwierigkeiten / Design-Entscheidungen

Bisher bereitete mir die Anzeige jeglicher Bilder (sei es die Anzeige von GIFs, die von Nutzern hochgeladen wurden, oder von Profilbildern) Probleme, da ich sie zunächst als statische Dateien im Spring-Ordner **/resource**/ speicherte und nur ihre Pfade in der Datenbank hinterlegte.

Trotz verschiedener Konfigurationen, die versuchten, die Erzeugung statischer Inhalte zu regeln, wurden nur die alt-Elemente von HTML angezeigt, da die Bilder nicht verfügbar waren. Erst nach einem Neustart der App konnten alle Bilder korrekt angezeigt werden. Da ich dies aber nicht wollte, habe ich die Datenbank so geändert, dass die Bilder als BLOBS gespeichert werden, was zwar zu einem Leistungsverlust führt, aber für den Benutzer viel angenehmer ist.

3 Datenbankdesign

In **index.html** ist das ER-Diagramm von Aurora zu sehen. Dabei sind folgende Entitätstypen vorhanden:

3.1 Entitätstypen (Tabellen)

3.1.1 AuroraUsers

Die Tabelle AuroraUsers speichert Informationen über die Benutzer der Plattform, die sich auf **/register** registrieren und damit ein Konto anlegen können. Jede Zeile in dieser Tabelle repräsentiert einen einzelnen Benutzer und enthält die folgenden Spalten:

1. **user_id** (Primärschlüssel): Eindeutige Identifikationsnummer für jeden Benutzer. Typ: **bigint**
2. **username:** Der Benutzername des Benutzers. Typ: **character varying(255)**

3. **email**: Die E-Mail-Adresse des Benutzers. Während bei der Anmeldung (`/login`) der Nutzernname gebraucht wird, soll die E-Mail-Adresse für das Zurücksetzen des Passworts dienen. Typ: `character varying(255)`
4. **password**: Das Passwort des Benutzers, welches gehasht gespeichert wird. Typ: `character varying(255)`
5. **bio**: Die Biografie jedes Nutzers, wird bei der Registrierung auf leer () gesetzt. Typ: `character varying(255)`

3.1.2 Roles

Die Tabelle Roles speichert die verschiedenen Rollen, die ein Benutzer haben kann. Jede Zeile in dieser Tabelle repräsentiert eine Rolle und enthält die folgenden Spalten:

- **role**: Ein Teil des Primärschlüssel (also als Composite-Key), ein String, welches eine Rolle repräsentiert. Typ: `character varying(255)`
- **user_id**: Der andere Teil des Primärschlüssel, referenziert `user_id` aus AuroraUsers. Typ: `bigint`

3.1.3 AuroraGifs

Die Tabelle AuroraGifs speichert die GIFs, die von Benutzern erstellt bzw. veröffentlicht wurden. Jede Zeile in dieser Tabelle repräsentiert ein einzelnes GIF und enthält die folgenden Spalten:

1. **gif_id** (Primärschlüssel): Eindeutige Identifikationsnummer für jedes GIF. Typ: `bigint`
2. **user_id** (Fremdschlüssel zu AuroraUsers): Die ID des Benutzers, der das GIF erstellt bzw. veröffentlicht hat. Typ: `bigint`
3. **description**: Die Beschreibung, die bei der Anzeige dargestellt wird. Typ: `character varying(255)`
4. **publish_date**: Das Veröffentlichungsdatum des GIFs. Typ: `timestamp(6) without time zone`
5. **image_data**: Die Bilddaten des GIFs, gespeichert als `bytea`.

3.1.4 GifCategory

Die Tabelle GifCategory speichert die verschiedenen Kategorien, in die GIFs eingeordnet werden können. Jede Zeile in dieser Tabelle repräsentiert eine Kategorie und enthält die folgenden Spalten:

- **category_id**: Primärschlüssel, eindeutige Identifikationsnummer für jede Kategorie. Typ: `bigint`
- **category**: Der Name der Kategorie, also die Kategorie selbst. Typ: `character varying(255)`

3.1.5 BelongsTo

Die Tabelle BelongsTo verknüpft GIFs mit Kategorien. Jede Zeile in dieser Tabelle repräsentiert eine Zuordnung von einem GIF zu einer Kategorie und enthält die folgenden Spalten:

- **id**: Die eindeutige Identifikationsnummer des Eintrags. Typ: **bigint**
- **gif_id**: Fremdschlüssel zu AuroraGifs, d.h. die ID des GIFs. Typ: **bigint**
- **category_id**: Fremdschlüssel zu GifCategory, die ID der Kategorie. Typ: **bigint**

Die Tabellen für das Speichern von Kategorien (bzw. Hashtags) wurden durch folgende Argumentation modelliert: Wenn der Nutzer einen Inhalt hochlädt und die Kategorien angibt, so sollen die Kategorien agnostisch vom GIF selbst in die Tabelle **GifCategory** gespeichert werden. Dem jeweiligen GIF wird dann in der Tabelle **BelongsTo** die Kategorie durch ihren Primärschlüssel (ein ID) gespeichert. Dadurch, dass die Kategorien separat (also in eigener Tabelle) gespeichert werden, werden Dopplungen vermieden.

3.1.6 Follows

Die Tabelle Follows repräsentiert die FFolgenBeziehungen zwischen Benutzern. Jede Zeile in dieser Tabelle stellt eine Folgebeziehung dar und enthält die folgenden Spalten:

- **user_id**: Fremdschlüssel zu AuroraUsers, die ID des Benutzers, der einem anderen Benutzer folgt. Typ: **bigint**
- **follows_user_id**: Fremdschlüssel zu AuroraUsers, die ID des Benutzers, der gefolgt wird. Typ: **bigint**
- **followed_at**: Das Veröffentlichungsdatum, an welchem ein Benutzer einem anderen Benutzer gefolgt ist. Typ: **timestamp(6) without time zone**

3.1.7 Comments

Die Tabelle Comments speichert Kommentare, die von Benutzern zu GIFs abgegeben werden. Jede Zeile in dieser Tabelle repräsentiert einen einzelnen Kommentar und enthält die folgenden Spalten:

- **comment_id**: Primärschlüssel, eindeutige Identifikationsnummer für jeden Kommentar. Typ: **bigint**
- **gif_id**: Fremdschlüssel zu AuroraGifs, die ID des GIFs, zu dem der Kommentar abgegeben wurde. Typ: **bigint**
- **user_id**: Fremdschlüssel zu AuroraUsers, die ID des Benutzers, der den Kommentar abgegeben hat. Typ: **bigint**

- **comment_text:** Der Text des Kommentars. Typ: `character varying(255)`
- **created_at:** Das Erstellungsdatum des Kommentars. Typ: `timestampt(6) without time zone`

3.1.8 Likes

Die Tabelle Likes speichert die „Gefällt mir“-Angaben von Benutzern zu GIFs. Jede Zeile in dieser Tabelle repräsentiert eine „Gefällt mir“-Angabe und enthält die folgenden Spalten:

- **like_id:** Primärschlüssel, eindeutige Identifikationsnummer für jede „Gefällt mir“-Angabe. Typ: `bigint`
- **user_id:** Fremdschlüssel zu AuroraUsers, die ID des Benutzers, der das GIF geliked hat. Typ: `bigint`
- **gif_id:** Fremdschlüssel zu AuroraGifs, die ID des GIFs, das geliked wurde. Typ: `bigint`

3.1.9 ProfilePictures

Die Tabelle ProfilePictures speichert die Profilbilder der Benutzer. Jede Zeile in dieser Tabelle repräsentiert ein Profilbild und enthält die folgenden Spalten:

- **picture_id:** Primärschlüssel, eindeutige Identifikationsnummer für jedes Profilbild. Typ: `bigint`
- **user_id:** Primärschlüssel und Fremdschlüssel zu AuroraUsers, die ID des Benutzers, zu dem das Profilbild gehört. Typ: `bigint`
- **image_data:** Das Profilbild des Benutzers, gespeichert als `bytea`.

3.1.10 Settings

Die Tabelle Settings speichert benutzerdefinierte Einstellungen. Jede Zeile in dieser Tabelle repräsentiert eine Einstellung und enthält die folgenden Spalten:

- **settings_id:** Primärschlüssel und Fremdschlüssel zu AuroraUsers, die ID des Benutzers, zu dem die Einstellung gehört. Typ: `bigint`
- **user_id:** Der Schlüssel der Einstellung. Typ: `bigint`
- **language:** Der Wert der Einstellung. Typ: `character varying(255)`
- **theme:** Der Wert der Einstellung. Typ: `character(1)`
- **others_can_see_my_followers:** Ein Boolean, welches repräsentiert, ob andere Benutzer sehen können, welche Benutzer dem Benutzer folgen. Typ: `boolean`

- **others_can_see_whoiam_following**: Ein Boolean, welches repräsentiert, ob andere Benutzer sehen können, welche Benutzern der Benutzer folgt. Typ: **boolean**
- **others_can_see_whatiliked**: Ein Boolean, welches repräsentiert, ob andere Benutzer sehen können, welche Beiträge dem Benutzer gefallen. Typ: **boolean**

3.1.11 Notifications

Die Tabelle Notifications speichert Benachrichtigungen, die die Benutzer erhalten. Jede Zeile in dieser Tabelle repräsentiert eine Benachrichtigung und enthält die folgenden Spalten:

- **notification_id**: Primärschlüssel, eindeutige Identifikationsnummer für jede Benachrichtigung. Typ: **bigint**
- **created_at**: Das Erstellungsdatum der Benachrichtigung. Typ: **date**
- **message**: Die Nachricht, die dem Nutzer beim Öffnen der Benachrichtigung gezeigt werden soll. Typ: **character varying(255)**
- **user_id**: Referenziert den **user_id** aus **AuroraUsers** (Fremdschlüssel). Typ: **bigint**

3.2 Beziehungen

1. AuroraUsers erstellt AuroraGifs

- Ein Benutzer **kann n** GIFs erstellen.
- Ein GIF **muss** von genau einem Benutzer erstellt werden (1:n).

2. AuroraUsers folgt AuroraUsers

- Ein Benutzer **kann n** anderen Benutzern folgen.
- Ein anderer Benutzer kann **m** Benutzern folgen (m:n).

3. AuroraUsers kommentiert AuroraGifs

- Ein Benutzer **kann n** Kommentare zu verschiedenen GIFs abgeben.
- Ein Kommentar **muss** von **1** Benutzer geschrieben werden.
- Ein GIF **kann m** Kommentare von verschiedenen Benutzern erhalten (m:n).

4. AuroraUsers gefällt AuroraGifs

- Ein Benutzer **kann n** GIFs liken.
- Ein GIF **kann von m** Benutzern geliked werden (n:m).

5. AuroraUsers hat ProfilePictures

- Ein Benutzer **muss 1** Profilbild haben.
- Ein Profilbild **muss 1** Benutzer gehören (1:1).

6. AuroraUsers verwaltet Settings

- Ein Benutzer **muss 1** Instanz von Einstellungen haben.
- Eine Einstellung **muss 1** Benutzer gehören (1:1).

7. AuroraGifs gehört zu GifCategory

- Ein GIF **kann n** Kategorien zugeordnet sein.
- Eine Kategorie **kann m** GIFs zugeordnet sein (n:m).

8. AuroraUsers hat Roles

- Ein Benutzer **muss n**, wobei m bei meisten Nutzern **1** ist, Rolle(n) haben.
- Eine Rolle **muss 1** Benutzer zugeordnet sein (1:n).

3.2.1 Relationenmodell (optimiert)

```
AuroraUser(user_id, username, email, password, bio)
folgt(user_id, follows_user_id, followed_at)
likes(user_id, gif_id)
AuroraGIF(gif_id, description, publish_date, image_data, user_id)
belongs_to(entry_id, gif_id, category_id)
Kategorie(category_id, category)
Comment(comment_id, comment_text, created_at, gif_id, user_id)
Notification(notification_id, created_at, message, user_id)
Role(role, user_id)
ProfilePicture(picture_id, image_data, user_id)
Settings(settings_id, language, theme,
others_can_see_my_followers,
others_can_see_whoiam_following,
others_can_see_whatiliked, user_id)
```

4 Ein Tieftauchgang durch die Codebase: Ausführen von SQL-Code

Aktuell wird Auroras Webinterface (im Ordner `/views`) durch folgende Dateien bzw. Komponenten umgesetzt:

- `NavigationBar.java`
- `HomeView.java`

- LoginView.java
- RegisterView.java
- UploadView.java
- UserProfileView.java
- MyProfileView.java
- SettingsView.java
- AdminDashboardView.java
- dialogs/BaseDialog.java (Basisklasse)
 - ChangePasswordDialog.java
 - CommentsDialog.java
 - CRUDDialog.java
 - FollowersDialog.java
 - FurtherInformationDialog.java
 - Notifications.java

Im folgenden gehe ich darauf ein, wie und welcher SQL-Code in diesen Dateien bzw. Views ausgeführt wird.

4.1 NavigationBar.java

Nach der Feststellung, dass ein Nutzer eingeloggt ist, wird der ID und das Profilbild des eingeloggten Nutzers angefragt. Danach wird in der Datenbank gesucht, ob der Nutzer in den Einstellungen den sogenannten „Darkmode“ aktiviert hat. Falls dies der Fall sein soll, wird das Farbschema entsprechend angepasst:

```
// NavigationBar.java
if (userService.isLoggedIn()) {
    final var userId = userService.getCurrentUserId();
    final var pfpOptional = profilePictureService.getProfilePictureByUserId(userId);

    Image profileImage = getImage(profilePictureService, pfpOptional);
    profileImage.getStyle().set("border-radius", "50%");
    // ...

    dark = settingsService.getUsersTheme(userId).equals("Dark");
    changeTheme();
    // ...
}
```

Schaut man sich den Quellcode der Methoden, die hier verwendet wurden an, so sieht man, dass folgender SQL-Code dahinter steckt:

```
// UserService.java: Eine Einfache Abfrage, die den Nutzernamen anhand
// seines IDs zurueckgibt.
public String getUsernameById(Long userId) {
    Query query = entityManager.createQuery("SELECT u.username FROM
        AuroraUser u WHERE u.userId = :userId");
    query.setParameter("userId", userId);
    return (String) query.getSingleResult();
}
```

Die Methode `profilePictureService.getPfpByUserId(userId)` verwendet hingegen Springs JPA, mit denen sich SQL-Code generieren lässt. So kann man in der Definition dieser Methode sehen, dass lediglich die Methode `profilePictureRepo.findById(userId)` aufgerufen wird, die ich nicht selbst geschrieben habe - sondern generiert habe.

Beim Erstellen der Anzeige für die Benachrichtigungen werden die Benachrichtigungen für den aktuell eingeloggten Nutzer rausgelesen:

```
private Optional<Button> getNotificationButton() {
    if (!userService.isLoggedIn()) {
        return Optional.empty();
    }

    Icon bellIcon = VaadinIcon.BELL.create();
    Button notificationButton = new Button(bellIcon);
    notificationButton.addThemeVariants(ButtonVariant.LUMO_ICON,
        ButtonVariant.LUMO_TERTIARY);

    final var userId = userService.getCurrentUserId();
    // Erhalte die Benachrichtigungen fuer den Nutzer
    final List<NotificationModel> notifications =
        notificationService.getNotificationsForUser(userId);
    if (!notifications.isEmpty()) {
        bellIcon.setColor("red");
        notificationButton.addClickListener(e ->
            createNotificationLayout(notifications, userId));
    } else {
        bellIcon.setColor("badge error pill");
    }

    return Optional.of(notificationButton);
}
```

Um diese Funktionalität zu implementieren, wird lediglich folgende SQL-Abfrage in JPA implementiert:

```
public List<NotificationModel> getNotificationsForUser(Long
    userId) {
```

```

        Query query = entityManager.createQuery("SELECT n FROM
            NotificationModel n WHERE intendedUser.userId = :userId");
        query.setParameter("userId", userId);

        return (List<NotificationModel>)query.getResultList();
    }

```

Markiert der Nutzer eine Benachrichtigung als gelesen (Bild 1), so wird diese aus der Datenbank gelöscht:

```

// NavigationBar.java (getMarkAsReadButton)
markAsReadButton.addClickListener(event -> {
    notificationsDialog.removeComponentFromDialog(component);
    notificationService.delete(notification.getId());
    // ...
});

```

Im Endeffekt wird dieser SQL-Code ausgeführt:

```

@Transactional
public void delete(Long notificationId) {
    Query query = entityManager.createQuery("DELETE FROM
        NotificationModel n WHERE n.id = :notificationId");
    query.setParameter("notificationId", notificationId);
    query.executeUpdate();
}

```

Außerdem wird in NavigationBar.java das Farbschema mithilfe der Datenbank realisiert:

```

private Button getThemeTogglingButton() {
    Button themeToggleButton = new Button(VaadinIcon.MOON.create());
    dark = false;
    changeTheme();

    themeToggleButton.addClickListener(event -> {
        // ...
        if (userService.isLoggedIn()) {
            settingsService.updateUsersTheme(userService.getCurrentUserId(),
                theme);
        }
        changeTheme();
    });
    return themeToggleButton;
}

```

wobei zur Aktualisierung des Farbschemas in der Datenbank folgende Abfrage ausgeführt wird:

```

@Transactional
public void updateUsersTheme(Long userId, char theme) {
    Query query = entityManager.createQuery("UPDATE Settings s SET
        s.theme = :theme WHERE s.user.userId = :userId");
    query.setParameter("userId", userId);
    query.setParameter("theme", theme);

    query.executeUpdate();
    entityManager.flush();
}

```

Zuletzt kann man in der Navigationsleiste nach Nutzern suchen. Dabei kriegt man während des Schreibens mögliche Nutzernamen, nach denen man suchen möchte. Dafür habe ich die Methode `getSimilarUsernames` implementiert und die Anzahl der Spalten auf 10 begrenzt:

```

public List<String> getSimilarUsernames(String baseUsername) {
    Query query = entityManager.createQuery("SELECT u.username " +
        "FROM AuroraUser u " +
        "WHERE u.username LIKE :baseUsername ");
    /* "LIMIT 10"); */
    query.setParameter("baseUsername", baseUsername + "%");
    query.setMaxResults(10);

    final var list = query.getResultList();
    return list.isEmpty() ? null : (List<String>)list;
}

```

4.2 HomeView.java

Aktuell kann man GIFs nach 3 Kriterien filtern:

1. Neuste
2. Top-Likes
3. Älteste

Dafür dienen folgende Methoden mit folgenden Abfragen (nach Reihenfolge sortiert):

```

// LikeService.java
public List<Object> getMostRecentGIFs() {
    Query query = entityManager.createQuery(
        "SELECT COUNT(*), u.username, g, pfp.imageData " +
        "FROM AuroraGIF g " +
        "JOIN g.user u " +
        "JOIN ProfilePicture pfp ON pfp.user.userId =
            u.userId " +

```

```

        "GROUP BY u.username, g.gifId, g.imageData,
        g.description, g.publishDate, g.user.userId,
        pfp.imageData " +
        "ORDER BY g.publishDate DESC"
    );
    final var list = query.getResultList();
    if (list.isEmpty()) return null;
    return list;
}

public List<Object> getMostLikedGIFs() {
    Query query = entityManager.createQuery(
        "SELECT COUNT(l), u.username, g, pfp.imageData " +
        "FROM Like l " +
        "JOIN l.gif g " +
        "JOIN g.user u " +
        "JOIN ProfilePicture pfp " +
        "ON pfp.user.userId = u.userId " +
        "GROUP BY u.username, g, pfp.imageData " +
        "ORDER BY COUNT(l) DESC"
    );
    final var list = query.getResultList();
    if (list.isEmpty()) return null;
    return list;
}

// GIFService.java
public List<Object[]> getAllGifsWithPfp() {
    Query query = entityManager.createQuery("SELECT ag, pfp.imageData
        " +
        "FROM AuroraGIF ag " +
        "JOIN ProfilePicture pfp " +
        "ON ag.user.userId = pfp.user.userId");
    final var list = query.getResultList();
    if (list.isEmpty()) return null;

    return list;
}

```

Da die GIFs jedoch auch durch Kategorien bzw. Hashtags durchsuchbar sind, muss ebenfalls eine Methode zur Verfügung gestellt werden, die die GIFs nach den eingegeben Kategorien filtert und zurückgibt:

```

@Transactional
public List<Object[]> filterGifsByCategory(String categoryCsv) {
    List<String> categoriesList =
        Arrays.asList(categoryCsv.split(","));
    Query query = entityManager.createQuery("SELECT ag,

```

```

    pfp.imageData, bt.category.id " +
    "FROM AuroraGIF ag " +
    "JOIN BelongsTo bt " +
    "ON ag.gifId = bt.gif.gifId " +
    "JOIN ProfilePicture pfp " +
    "ON ag.user.userId = pfp.user.userId " +
    "JOIN GifCategory gc " +
    "ON bt.category.categoryId = gc.categoryId " +
    "WHERE gc.category IN :categories");

query.setParameter("categories", categoriesList);
List<Object[]> results = (List<Object[]>) query.getResultList();
if (results.isEmpty()) return null;

return results;
}

```

Da man aber sowohl durch Kriterien als auch durch Kategorien filtern kann, muss auch eine Methode her, die die gegebenen GIFs nach Kategorien filtert:

```

public List<Object[]> filterGivenGifsByCategory(List<Long> gifIds,
List<String> categories) {
Query query = entityManager.createQuery("SELECT DISTINCT ag,
    pfp.imageData " +
    "FROM AuroraGIF ag " +
    "JOIN BelongsTo bt ON ag.gifId = bt.gif.gifId " +
    "JOIN ProfilePicture pfp ON pfp.user.userId =
        ag.user.userId " +
    "JOIN GifCategory gc ON bt.category.categoryId =
        gc.categoryId " +
    "WHERE gc.category IN (:categories) " +
    "AND ag.gifId IN (:gifIds)");

query.setParameter("categories", categories);
query.setParameter("gifIds", gifIds);

final var filteredGifs = query.getResultList();
if (filteredGifs.isEmpty()) return null;

return filteredGifs;
}

```

Wie vorhin erwähnt, habe ich ein Follow-System implementiert. Deshalb werden auf der Homepage normalerweise die Beiträge von den Nutzern, denen man folgt, angezeigt. Dies lässt sich durch folgende Methode realisieren:

```

public List<Object[]> getGifsFromFollowingUsers(Long userId) {
Query query = entityManager.createQuery(
    "SELECT ag, pfp.imageData " +

```

```

        "FROM AuroraGIF ag" +
        " JOIN ag.user au" +
        " JOIN Follows f ON au.userId =
            f.followsUser.userId" +
        " JOIN ProfilePicture pfp " +
        " ON ag.user.userId = pfp.user.userId " +
        " WHERE f.user.userId = :userId"
    );
    query.setParameter("userId", userId);

    List<Object[]> gifs = (List<Object[]>)query.getResultList();
    return gifs;
}

// RegisterView.java
AuroraUser newUser = new AuroraUser(
    user,
    mail,
    passwordEncoder.encode(pass)
);
userRepository.save(newUser);

Settings settings = new Settings(newUser, true, true, true,
    "English", 'd');
settingsRepository.save(settings);
Role standardRole = new Role(newUser.getId(), "user");
roleRepository.save(standardRole);

try {
    profilePictureService.savePfp(profilePictureService.getDefaultPfpAsInputStream(),
        newUser);
} catch (IOException e) {
    e.printStackTrace();
}

```

Dabei ist solch eine Tabelle wie **AuroraUser** so modelliert, dass sie Attribute mit Gettern, Settern und einem geeigneten Konstruktor enthält. Durch Java Annotationen wird dies dann in SQL umgesetzt. Auch werden initiale Einstellungen und das übliche Profilbild gespeichert.

```

/tables/AuroraUser.java
@Entity
@Table(name = "AuroraUsers")
public class AuroraUser {
    @Id
    @GeneratedValue
    @Column(name = "user_id")
    private Long userId;

```

```

@Column(unique = true)
private String username;
@Column(unique = true)
private String email;
@Column
private String password;
@Column
private String bio;

@OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
private List<AuroraGIF> gifs;

@OneToMany(mappedBy = "user")
private Set<Like> likes;

@OneToMany(mappedBy = "user")
private Set<Role> roles;

@OneToMany(mappedBy = "user")
private Set<Comment> comments;

public AuroraUser() {}

public AuroraUser(String username, String email, String password) {
    this.username = username;
    this.email = email;
    this.password = password;
    // Die Biografie soll spaeter in den Einstellungen geaendert
    // werden.
    this.bio = "I have no biography (yet).";
}

public Long getId() {
    return userId;
}

public void setId(Long userId) {
    this.userId = userId;
}

public String getUsername() {
    return username;
}

public void setUsername(String username) {
    this.username = username;
}

public String getEmail() {
    return email;
}

```

```

    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getBio() {
        return bio;
    }

    public void setBio(String bio) {
        this.bio = bio;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}

// /repo/UserRepo.java
public interface UserRepo extends JpaRepository<AuroraUser, Long> {
    // Diese Funktionen werden selbst erzeugt.
    Optional<AuroraUser> findByUsername(String username);
    Optional<AuroraUser> findByEmail(String email);
}

```

4.3 LoginView.java

Dies übernimmt Spring bzw. Vaadin gänzlich.

4.4 UploadView.java

Beim Hochladen eines GIFs muss

1. Das Programm die Follower des Nutzers erfahren und ihnen eine Benachrichtigung schicken
2. Das GIF in der Datenbank gespeichert werden
3. Das Programm die vom Nutzer angegebenen Kategorien iterieren und schauen, ob sie in der Tabelle GifCategory existieren. Falls nicht, sollen diese in der Tabelle gespeichert werden
4. Das GIF den jeweiligen Kategorien zugeordnet werden (mithilfe der Tabelle **BelongsTo**).

Abgesehen von den Speicher-Operationen in die Datenbank (`.save()`, dessen Definitionen JPA übernimmt und generiert), müssen für die oben genannten Punkte folgende SQL-Abfragen ausgeführt werden (die in Methoden verpackt sind, in Reihenfolge):

```

public List<Object[]> getUsersFollowers(Long userId) {
    Query query = entityManager.createQuery(
        "SELECT au, pfp.imageUrl " +
        "FROM AuroraUser au " +
        "JOIN Follows f ON au.userId = f.user.userId " +
        "JOIN ProfilePicture pfp ON au.userId = pfp.user.userId " +
        "WHERE f.followsUser.userId = :userId"
    );
    query.setParameter("userId", userId);
    return query.getResultList();
}

// UploadView.java
notificationService.save(intendedUser, msg);
// 
public boolean categoryAlreadyExists(String category) {
    Query query = entityManager.createQuery("SELECT count(c) " +
        "FROM GifCategory c " +
        "WHERE c.category = :category", Long.class);

    query.setParameter("category", category);
    Long count = (Long) query.getSingleResult();
    System.out.println("Count: " + count);

    return count > 0;
}

public GifCategory getCategory(String category) {
    Query query = entityManager.createQuery("SELECT gc " +
        "FROM GifCategory gc " +
        "WHERE gc.category = :category");
    query.setParameter("category", category);
    return (GifCategory)query.getSingleResult();
}
// UploadView.java
BelongsTo belongsToEntry = new BelongsTo(gif, gifCategory);
belongsToRepo.save(belongsToEntry);

```

4.5 MyProfileView.java und UserProfileView.java

Auf dieser Unterseite werden die Informationen des Nutzers angezeigt, sowie seine Follower und Nutzer, denen er selbst folgt. Da der Code für die Abfrage von eigenen Followern oben schon dargestellt wurde, wird jetzt der Code für die

Nutzer, denen der Nutzer folgt, dargestellt:

```
public List<Object[]> getFollowingUsers(Long userId) {
    Query query = entityManager.createQuery(
        "SELECT au, pfp.imageUrl FROM AuroraUser au" +
        " JOIN Follows f ON au.userId =
            f.followsUser.userId" +
        " JOIN ProfilePicture pfp on au.userId =
            pfp.user.userId" +
        " WHERE f.user.userId = :userId"
    );
    query.setParameter("userId", userId);
    return query.getResultList();
}
```

Für die Anzahl von Followern, Nutzern, denen man selbst folgt wird folgender SQL-Code mit COUNT(*) verwendet:

```
public Long getFollowersCount(Long userId) {
    Query query = entityManager.createQuery("SELECT COUNT(*) " +
        "FROM Follows " +
        "WHERE followsUser.userId = :user_id");
    query.setParameter("user_id", userId);
    return (Long)query.getSingleResult();
}

public Long getFollowingCount(Long userId) {
    Query query = entityManager.createQuery("SELECT COUNT(*) " +
        "FROM Follows " +
        "WHERE user.userId = :userId");
    query.setParameter("userId", userId);
    return (Long)query.getSingleResult();
}
```

Für die Anzahl von Beiträgen entschied ich mich für das Generieren dieser Funktionalität von JPA:

```
public interface GifRepo extends JpaRepository<AuroraGIF, Long> {
    Long countByUserId(Long userId);
    // ...
}
```

Jedoch muss bei `UserProfileView.java` gesagt werden, dass der Nutzer selber entscheiden kann, ob er möchte, dass die anderen Nutzer sehen, wem er folgt bzw. wer ihm folgt. Diese Einstellung wird in einer separaten Tabelle gespeichert, auf die ich gleich eingehen werde. Anbei der Code für das Auslesen dieser Einstellungen:

```
public boolean canOthersSeeFollowing(Long userId) {
```

```

        Query query = entityManager.createQuery("SELECT
            othersCanSeeWhoIAmFollowing " +
            "FROM Settings " +
            "WHERE user.userId = :userId");
        query.setParameter("userId", userId);

        return (boolean)query.getSingleResult();
    }

    public boolean canOthersSeeWhatILiked(Long userId) {
        Query query = entityManager.createQuery("SELECT
            othersCanSeeWhatILiked " +
            "FROM Settings " +
            "WHERE user.userId = :userId");
        query.setParameter("userId", userId);

        return (boolean)query.getSingleResult();
    }

```

Außerdem werden an diesen Unterseiten die eigenen Beiträge des Nutzers gerendert, sowie die Beiträge, die dem Nutzer gefallen.

Für das Rendern eigener Beiträge wird eine Selektion durchgeführt und mit dem ID des Users eingegrenzt:

```

public List<Object[]> findAllByUserIdAndPfp(Long userId) {
    Query query = entityManager.createQuery("SELECT ag, pfp.imageUrl
        " +
        "FROM AuroraGIF ag " +
        "JOIN AuroraUser au " +
        "ON ag.userId = au.userId " +
        "JOIN ProfilePicture pfp " +
        "ON au.userId = pfp.user.userId " +
        "WHERE au.userId = :userId");
    query.setParameter("userId", userId);

    return query.getResultList();
}

```

Bei der Selektion der Beiträge, die einem Nutzer gefallen, muss die GIF-Tabelle mit der Likes-Tabelle verbunden werden:

```

public List<AuroraGIF> getPostsUserLiked(Long userId) {
    Query query = entityManager.createQuery(
        "SELECT ag " +
        "FROM AuroraGIF ag " +
        "JOIN Like l " +
        "ON ag.gifId = l.gif.gifId " +
        "WHERE l.user.userId = :userId")
    .setParameter("userId", userId);

```

```

        List<AuroraGIF> resultList =
            (List<AuroraGIF>)query.getResultList();
        return resultList.isEmpty() ? null : resultList;
    }

```

4.6 Settings.java

Im ersten Reiter „Account“ (oder Konto) wird die eigene E-Mail-Adresse mithilfe folgender Methode angezeigt:

```

public String getEmail(Long userId) {
    Query query;
    try {
        query = entityManager.createQuery("SELECT u.email " +
            "FROM AuroraUser u " +
            "WHERE u.id = :userId");
        query.setParameter("userId", userId);
    } catch (NoResultException e) {
        e.printStackTrace();
        return null;
    }

    return (String) query.getSingleResult();
}

```

Außerdem hat der Nutzer die Möglichkeit, sein Passwort zu ändern: Im Reiter „Profile“ hat der Nutzer die Möglichkeit, seine Biografie zu aktualisieren, welche mithilfe folgender Methode realisiert wird:

```

@Transactional
public int updateBio(Long userId, String newBio) {
    Query query = entityManager.createQuery("UPDATE AuroraUser " +
        "SET bio = :newBio " +
        "WHERE userId = :userId");
    query.setParameter("userId", userId);
    query.setParameter("newBio", newBio);
    return query.executeUpdate();
}

```

Im Reiter „Privacy“ verwaltet der Nutzer seine Einstellungen bezüglich der Privatsphäre. Dabei kann er entscheiden, ob er möchte, dass:

1. die anderen Nutzer seine Follower sehen
2. die anderen Nutzer sehen, wem er selbst folgt
3. die anderen Nutzer sehen, was ihm gefällt

Dies wird durch folgende Abfragen realisiert:

```
// Zu 1.
@Transactional
public void updateCanOthersSeeFollowers(Long userId, boolean yes) {
    int value = yes ? 1 : 0;
    Query query = entityManager.createQuery("UPDATE Settings s SET
        s.othersCanSeeMyFollowers = :value WHERE s.user.userId =
        :userId");
    query.setParameter("userId", userId);
    query.setParameter("value", value);

    query.executeUpdate();
    entityManager.flush();
}

// Zu 2.
@Transactional
public void updateCanOthersSeeFollowing(Long userId, boolean yes) {
    int value = yes ? 1 : 0;
    Query query = entityManager.createQuery("UPDATE Settings s SET
        s.othersCanSeeWhoIAmFollowing = :value WHERE s.user.userId =
        :userId");
    query.setParameter("userId", userId);
    query.setParameter("value", value);

    query.executeUpdate();
    entityManager.flush();
}

// Zu 3.
@Transactional
public void updateCanOthersSeeWhatILiked(Long userId, boolean yes) {
    Query query = entityManager.createQuery("UPDATE Settings s " +
        "SET s.othersCanSeeWhatILiked = :yes " +
        "WHERE user.userId = :userId");
    query.setParameter("userId", userId);
    query.setParameter("yes", yes);

    query.executeUpdate();
    entityManager.flush();
}
```

Im letzten Reiter „Preferences“ kann der Nutzer aktuell seine bevorzugte Sprache auswählen. Diese wird in der Datenbank wie folgt aktualisiert:

```
@Transactional
public void updateUsersLanguage(Long userId, String language) {
    Query query = entityManager.createQuery("UPDATE Settings s SET
        s.language = :language WHERE s.user.userId = :userId");
    query.setParameter("userId", userId);
    query.setParameter("language", language);
```

```
        query.executeUpdate();
        entityManager.flush();
    }
```

Um die Werte der UI-Elemente zu setzen, bspw. ob der Checkbox bei den Privatsphäreinstellungen angekreuzt werden soll oder nicht, werden die Werte aus der Datenbank angefragt und entsprechend werden die UI-Elemente damit aktualisiert. Diese Methoden sind in der Datei `/services/SettingsService.java` vorzufinden.

5 Screenshots

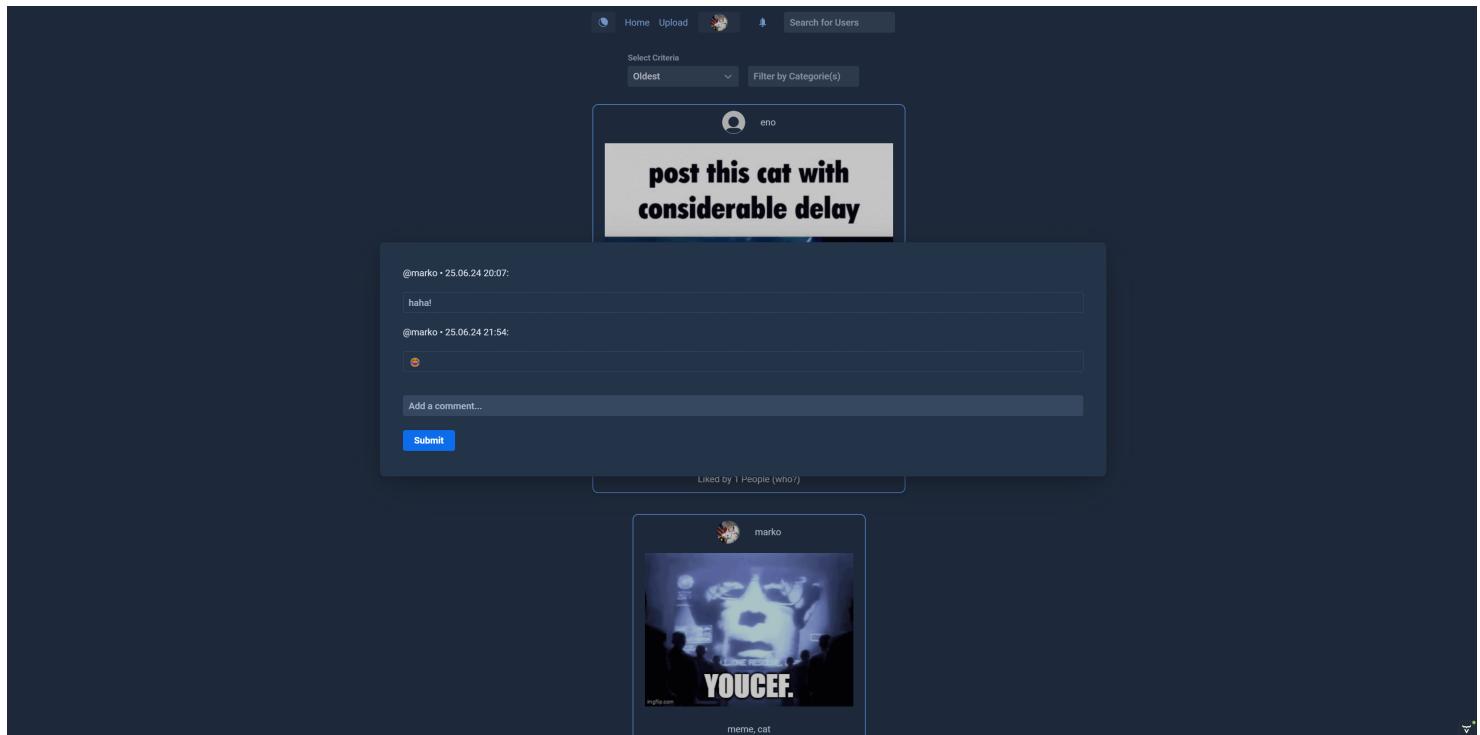


Abbildung 1: Die Kommentarspalte.

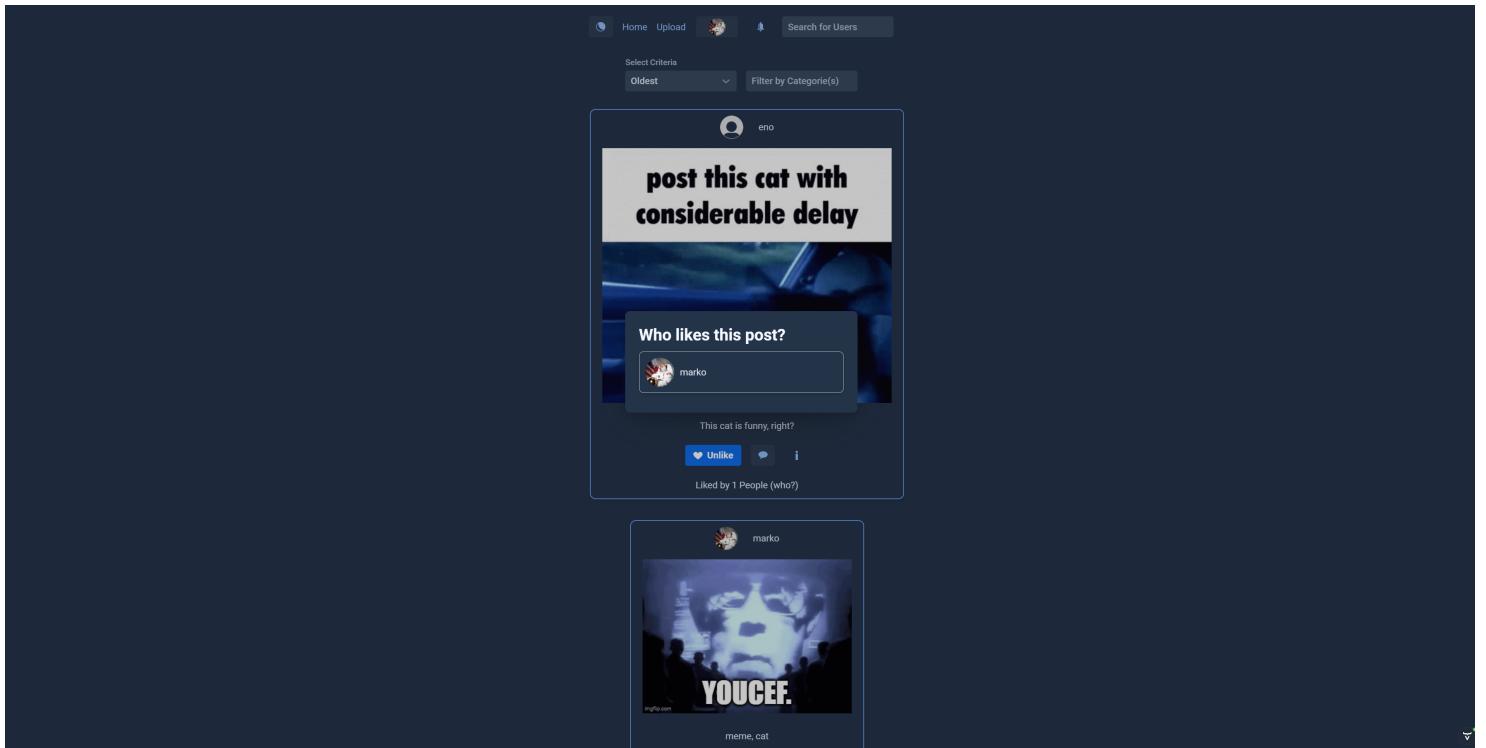


Abbildung 2: Likes / Gefällt mir.

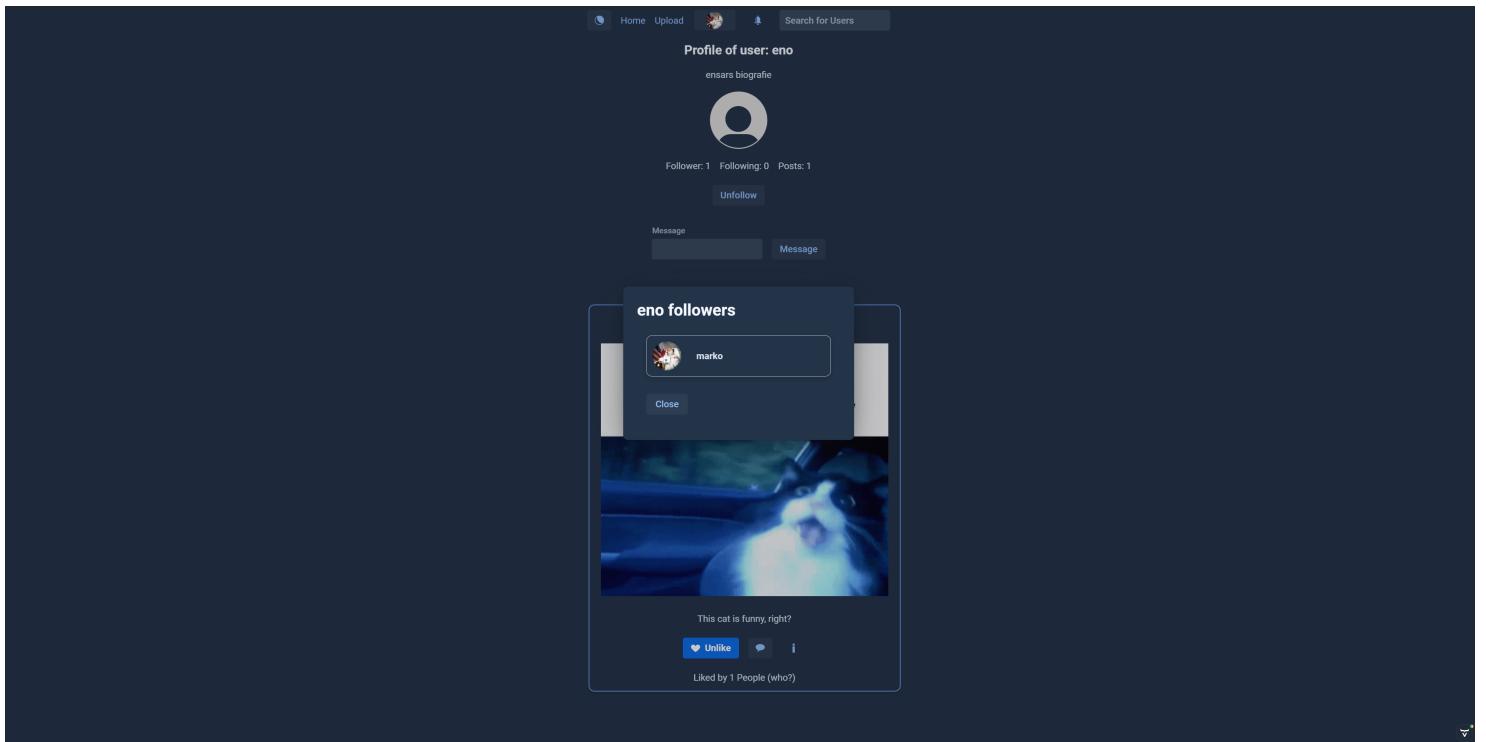


Abbildung 3: Meine Follower.

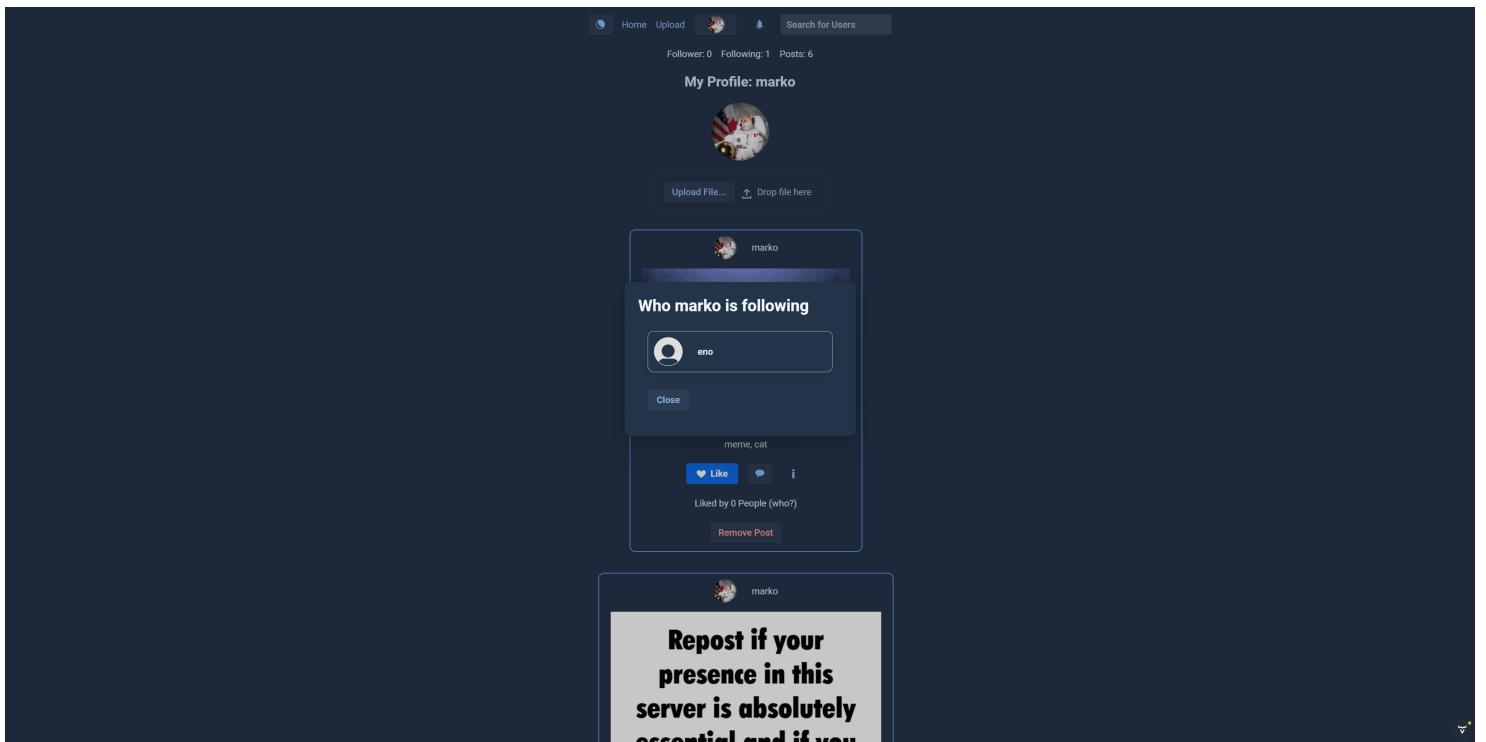


Abbildung 4: Wem folge ich?

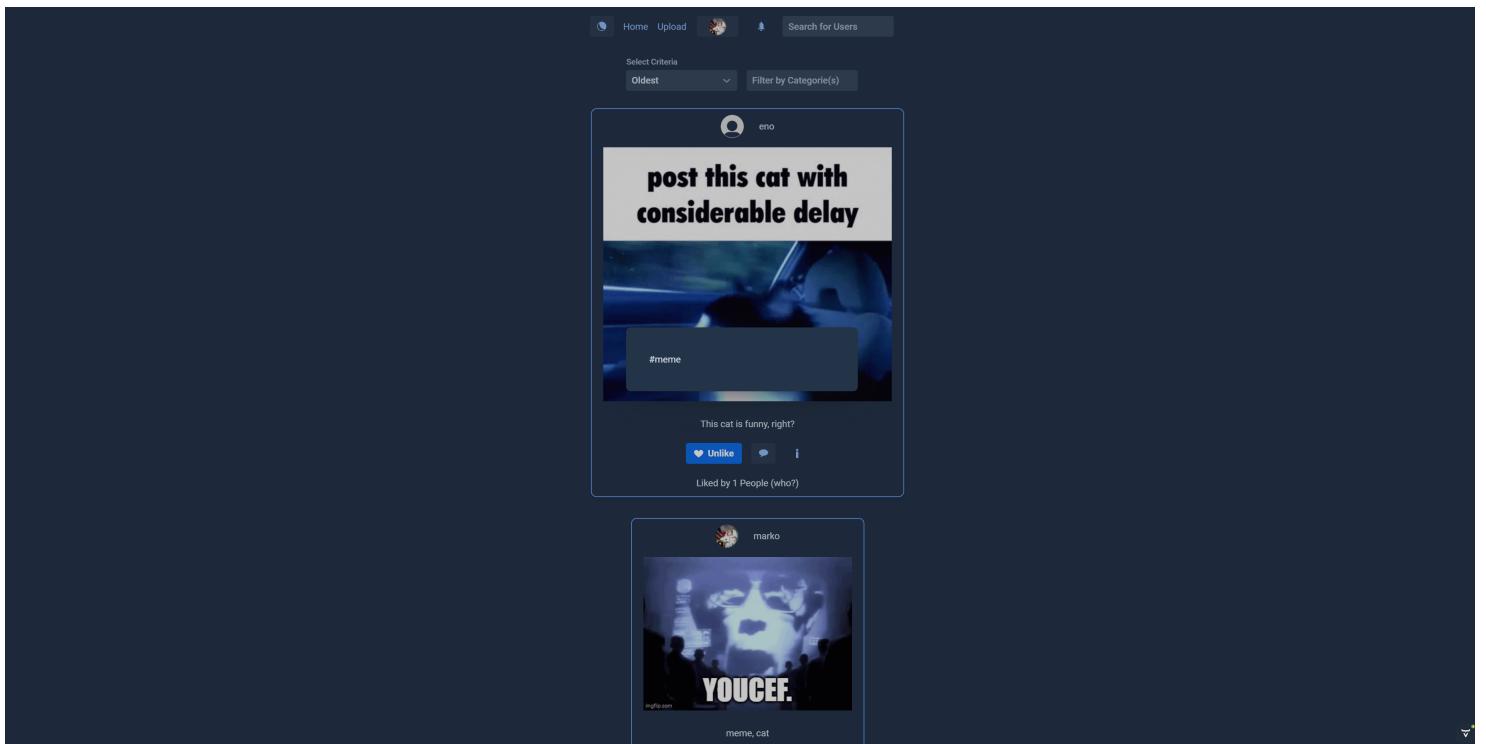


Abbildung 5: Hashtags / Kategorien eines GIFs.

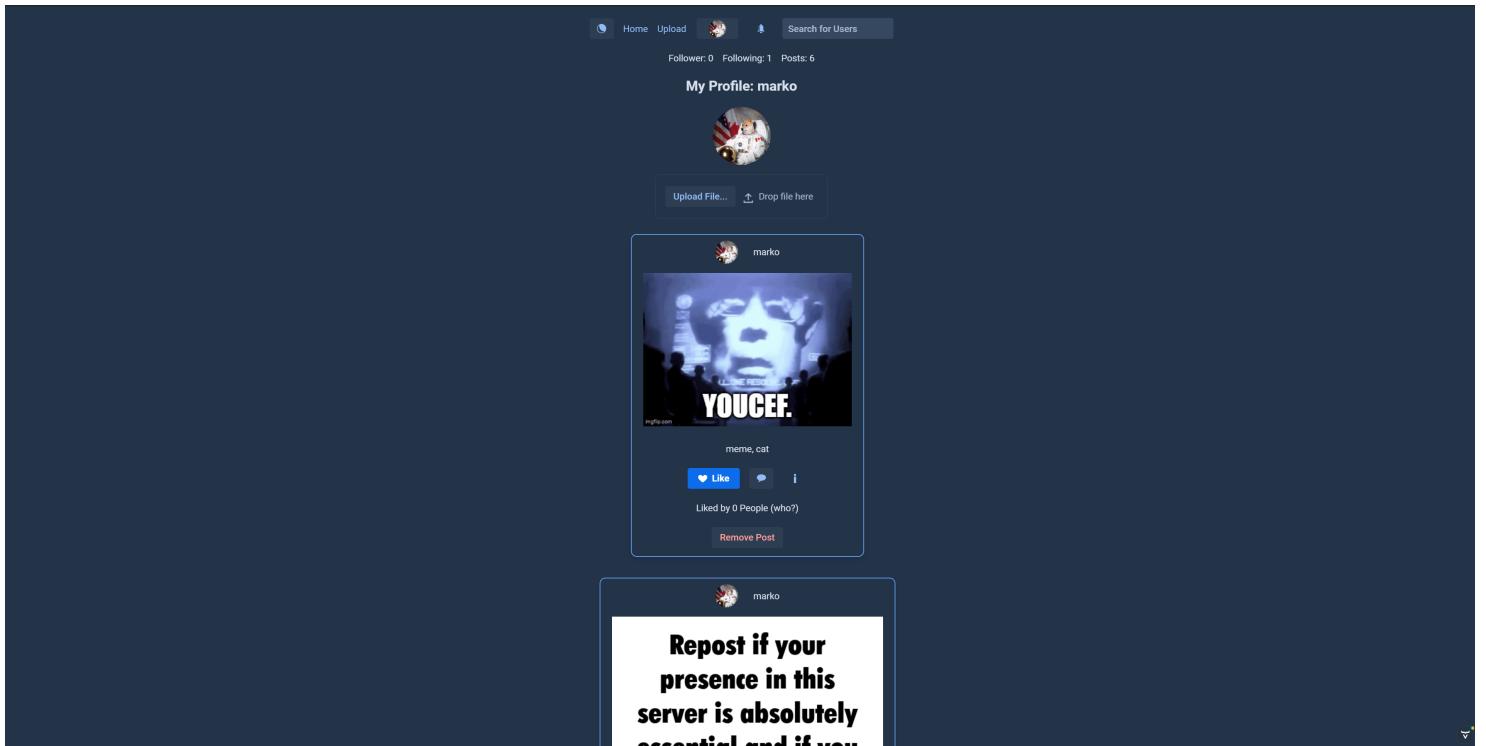


Abbildung 6: Mein Profil.

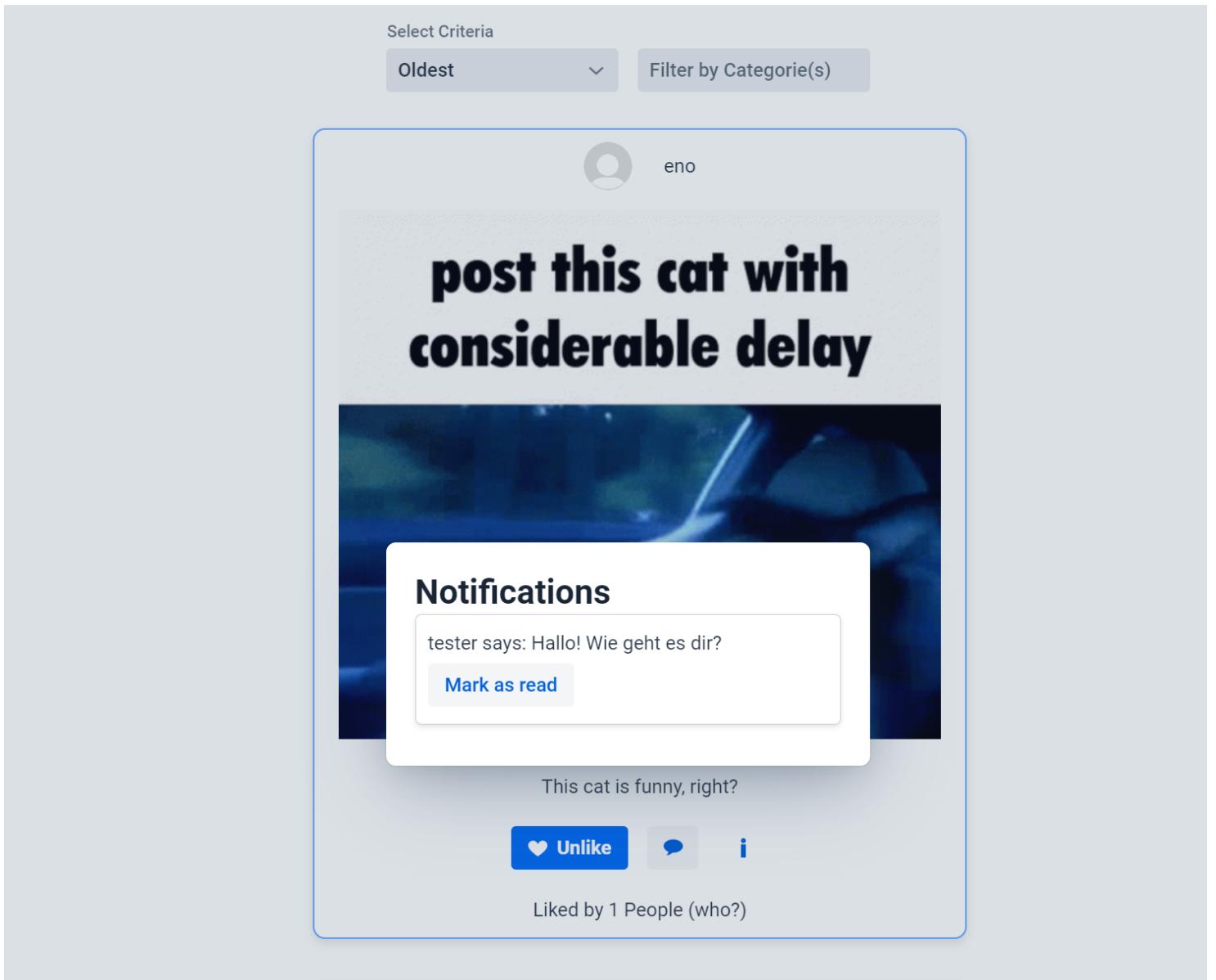


Abbildung 7: Eine Benachrichtigung

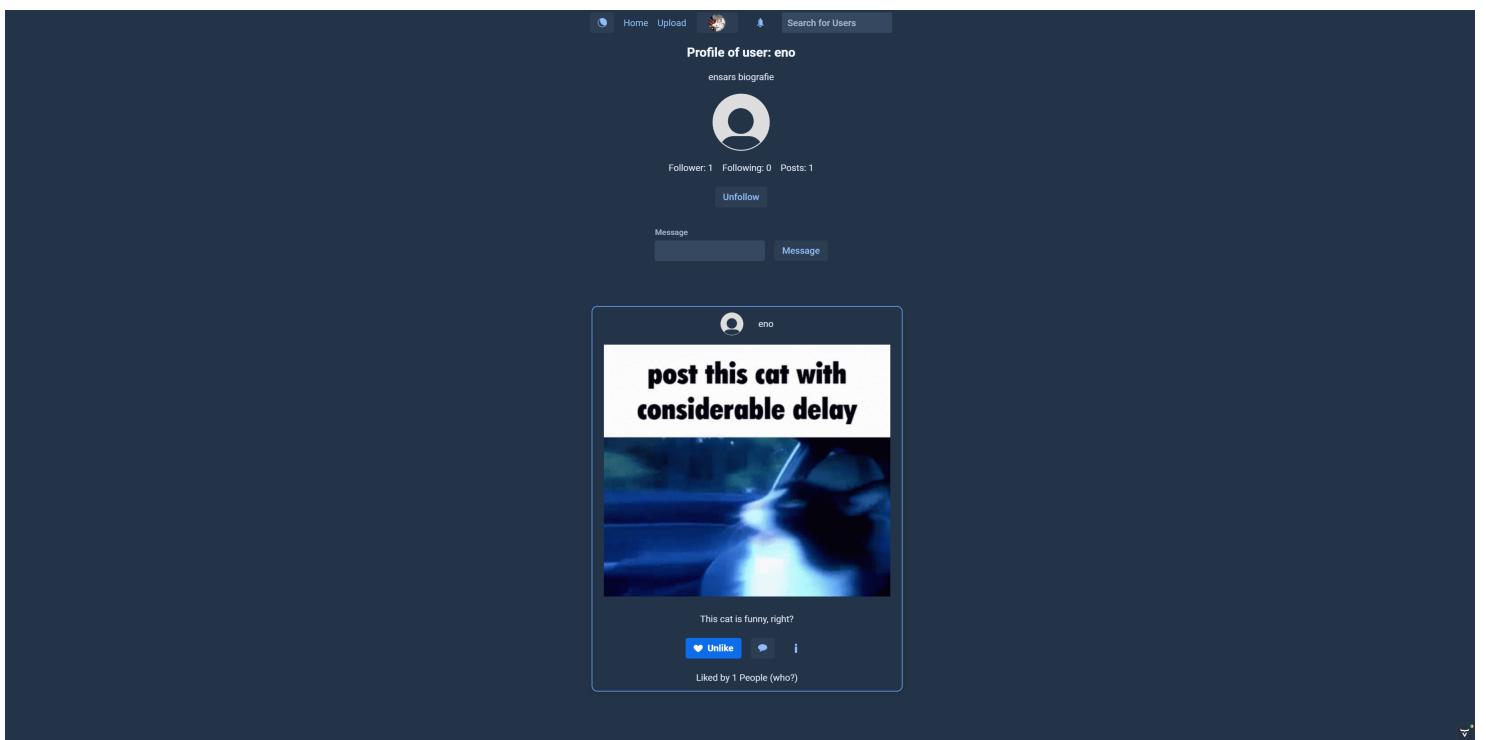


Abbildung 8: Ein anderer Nutzer.

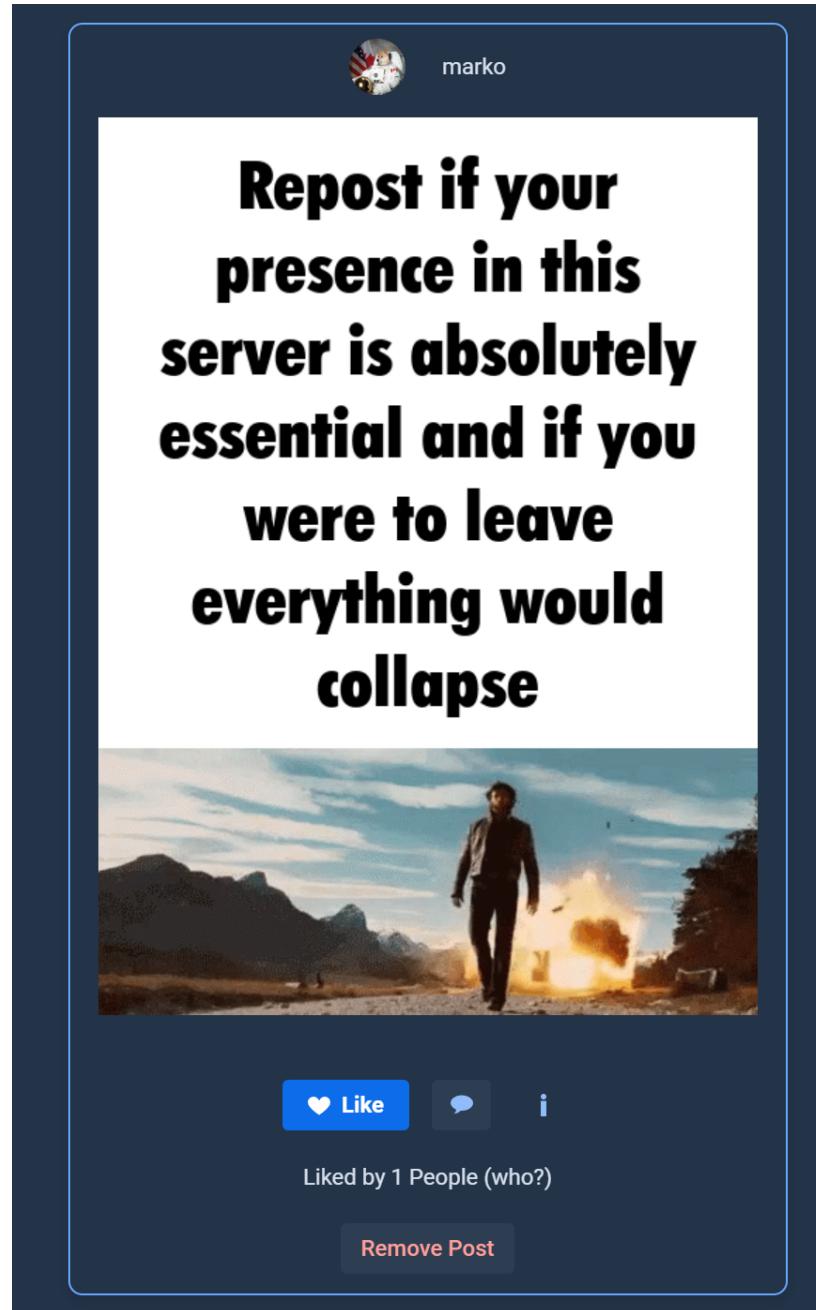


Abbildung 9: Nutzer können ihre eigene Beiträge löschen, während Moderatoren alle Beiträge löschen können.

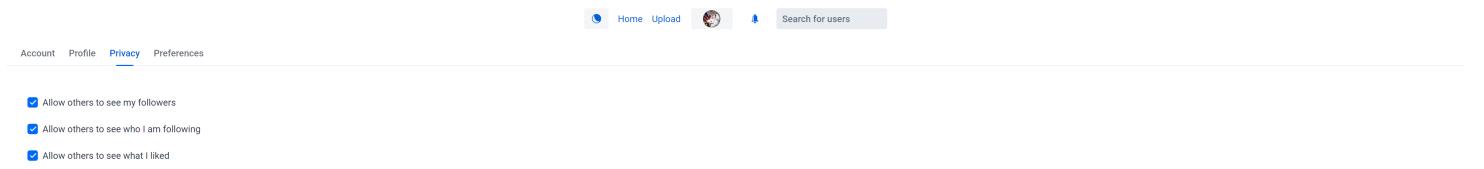


Abbildung 10: Ein Beispiel der Einstellungen.

Admin Dashboard																																				
Enter SQL statement (preferably update or delete; tables will update after page refresh)																																				
aurora_users																																				
<table border="1"> <thead> <tr> <th>user_id</th><th>username</th><th>email</th><th>password</th></tr> </thead> <tbody> <tr> <td>3402</td><td>t</td><td>t@gmail.com</td><td>\$2a\$10\$hXSst4BESVkJxmVNBO9vYeyKXbcIybSKucovHuWJeS7XYrejDH7WO</td></tr> <tr> <td>3352</td><td>marko</td><td>marko@gmail.com</td><td>\$2a\$10\$zEaUxyOhrPQeMaJcMoZYSuWfe4lNQf4mrRFPAz4sYjmXMKp260VW</td></tr> <tr> <td>3353</td><td>eno</td><td>eno@gmail.com</td><td>\$2a\$10\$GP6D1EbRJgMcWj0zm9IxOsFBVi7GQ9RZUfFmeB2iC.hshQMDLXg6</td></tr> </tbody> </table>					user_id	username	email	password	3402	t	t@gmail.com	\$2a\$10\$hXSst4BESVkJxmVNBO9vYeyKXbcIybSKucovHuWJeS7XYrejDH7WO	3352	marko	marko@gmail.com	\$2a\$10\$zEaUxyOhrPQeMaJcMoZYSuWfe4lNQf4mrRFPAz4sYjmXMKp260VW	3353	eno	eno@gmail.com	\$2a\$10\$GP6D1EbRJgMcWj0zm9IxOsFBVi7GQ9RZUfFmeB2iC.hshQMDLXg6																
user_id	username	email	password																																	
3402	t	t@gmail.com	\$2a\$10\$hXSst4BESVkJxmVNBO9vYeyKXbcIybSKucovHuWJeS7XYrejDH7WO																																	
3352	marko	marko@gmail.com	\$2a\$10\$zEaUxyOhrPQeMaJcMoZYSuWfe4lNQf4mrRFPAz4sYjmXMKp260VW																																	
3353	eno	eno@gmail.com	\$2a\$10\$GP6D1EbRJgMcWj0zm9IxOsFBVi7GQ9RZUfFmeB2iC.hshQMDLXg6																																	
aurora_gifs																																				
<table border="1"> <thead> <tr> <th>gif_id</th><th>description</th><th>publish_date</th><th>user_id</th></tr> </thead> <tbody> <tr> <td>7552</td><td>merme, cat</td><td>2024-06-25 18:32:49.172</td><td>3352</td></tr> <tr> <td>7652</td><td>This cat is funny, right?</td><td>2024-06-25 18:37:59.433</td><td>3353</td></tr> <tr> <td>7702</td><td></td><td>2024-06-25 18:41:51.704</td><td>3352</td></tr> <tr> <td>7703</td><td></td><td>2024-06-25 18:42:03.718</td><td>3352</td></tr> <tr> <td>7752</td><td></td><td>2024-06-25 18:45:04.084</td><td>3352</td></tr> <tr> <td>7802</td><td>marko</td><td>2024-06-25 18:50:40.87</td><td>3352</td></tr> <tr> <td>7852</td><td></td><td>2024-06-25 18:52:29.765</td><td>3352</td></tr> </tbody> </table>					gif_id	description	publish_date	user_id	7552	merme, cat	2024-06-25 18:32:49.172	3352	7652	This cat is funny, right?	2024-06-25 18:37:59.433	3353	7702		2024-06-25 18:41:51.704	3352	7703		2024-06-25 18:42:03.718	3352	7752		2024-06-25 18:45:04.084	3352	7802	marko	2024-06-25 18:50:40.87	3352	7852		2024-06-25 18:52:29.765	3352
gif_id	description	publish_date	user_id																																	
7552	merme, cat	2024-06-25 18:32:49.172	3352																																	
7652	This cat is funny, right?	2024-06-25 18:37:59.433	3353																																	
7702		2024-06-25 18:41:51.704	3352																																	
7703		2024-06-25 18:42:03.718	3352																																	
7752		2024-06-25 18:45:04.084	3352																																	
7802	marko	2024-06-25 18:50:40.87	3352																																	
7852		2024-06-25 18:52:29.765	3352																																	
roles																																				
<table border="1"> <thead> <tr> <th>role</th><th>user_id</th></tr> </thead> <tbody> <tr> <td>user</td><td>3352</td></tr> <tr> <td>user</td><td>3353</td></tr> <tr> <td>user</td><td>3402</td></tr> <tr> <td>admin</td><td>3352</td></tr> </tbody> </table>					role	user_id	user	3352	user	3353	user	3402	admin	3352																						
role	user_id																																			
user	3352																																			
user	3353																																			
user	3402																																			
admin	3352																																			
comments																																				
<table border="1"> <thead> <tr> <th>comment_id</th><th>comment_text</th><th>created_at</th><th>gif_id</th><th>user_id</th></tr> </thead> <tbody> <tr> <td>20</td><td>haha!</td><td>2024-06-25T20:07:39.745653</td><td>7652</td><td>3352</td></tr> <tr> <td>21</td><td>😊</td><td>2024-06-25T21:54:02.866724</td><td>7652</td><td>3352</td></tr> </tbody> </table>					comment_id	comment_text	created_at	gif_id	user_id	20	haha!	2024-06-25T20:07:39.745653	7652	3352	21	😊	2024-06-25T21:54:02.866724	7652	3352																	
comment_id	comment_text	created_at	gif_id	user_id																																
20	haha!	2024-06-25T20:07:39.745653	7652	3352																																
21	😊	2024-06-25T21:54:02.866724	7652	3352																																

Abbildung 11: Admin-Ansicht.