OnlyVans

Release 1.0

Jacek Miecznikowski, Kamil Danielski, Mateusz Kalenik

Jun 01, 2024

CONTENTS:

This is the documentation for the OnlyVans project. This project is a Django project that is meant to be a platform for creators to sell their content to their fans. The project is in development and is not yet ready for production.

CHAPTER

ONE

REQUIREMENTS

1.1 Functional Requirements

1.2 User Management

1. **User Registration**:

- Users should be able to register an account using a unique username, email, and password.
- Users should not be able to register with an existing username or email.

2. **User Roles**:

- The system should support two types of users: Clients and Content Creators.
- Users should not be able to switch roles.

3. **User Login**:

- Users should be able to log in using their registered username and password.

4. **User Profile**:

- Users should be able to view and update their profile information including username, email, profile picture, and bio.
- Creators should be able to connect their Stripe account in the profile settings to receive payments or create paid content.
- Users should be able to change their password.
- Users should not be able to delete their account.
- Users should be able to view other users' profiles.

1.3 Content Management

5. **Create Tier**:

- Content Creators should be able to create up to 12 subscription tiers with different benefits and prices.
- Each tier should have an option to allow or disallow messaging permissions.

6. **Create Post**:

- Creators should be able to create new posts including a title, text, and optional media files (images/videos).
- Posts can be marked as free or assigned to a specific subscription tier.

7. **Delete Post**:

- Content Creators should be able to delete their posts.

8. **View Posts**:

- Clients should be able to view posts they have access to based on their subscriptions and free posts.
- Creators should be able to view all their posts.

9. **Like**:

- Users should be able to leave a like on posts.
- Users should be able to unlike posts.
- Users should be able to view likes on posts.

1.4 Subscription Management

10. **Subscribe to Tier**:

- Clients should be able to subscribe to a Creator's tier using their account balance.
- Clients should be able to view the benefits of each tier before subscribing.
- Clients should not be able to subscribe to the same Creator multiple times.
- Clients should not be able to subscribe to a tier if they do not have enough points in their account balance.

11. **Manage Subscriptions**:

- Clients should be able to view and manage their active subscriptions.
- Clients should be able to extend or cancel their subscriptions.
- Subscriptions should be automatically renewed at the end of the billing period.
- If the balance is insufficient, the subscription should be expired.

1.5 Financial Management

12. **Wallet**:

- Users should have a wallet that keeps track of their balance in points.
- Clients should be able to add points to their wallet using a payment method.
- Creators should be able to withdraw points from their wallet to their Stripe account.

13. **Transactions**:

- Users should be able to view their transaction history including deposits, withdrawals, and subscription payments.

1.6 Messaging

14. **Direct Messaging**:

- Users with appropriate permissions should be able to send direct messages to each other.
- Messaging permissions should be based on the subscription tier benefits.

1.7 Event Logging

15. **Event Log**:

- The system should log significant user actions such as profile updates, password changes, post creations, and other important events.

1.8 Non-Functional Requirements

1.9 Performance

1. **Response Time**:

- The system should ensure that the response time for any user action does not exceed 3 seconds under normal load conditions.

2. **Scalability**:

- The system should be able to handle a growing number of users and increased load without performance degradation.

1.10 Security

3. **Data Protection**:

- User data should be encrypted in transit and at rest.
- Passwords should be hashed using a strong hashing algorithm.

4. **Authentication and Authorization**:

- The system should implement secure authentication and authorization mechanisms.
- Users should not be able to access unauthorized resources.

1.11 Usability

5. **User Interface**:

- The system should have a user-friendly and intuitive interface.
- The system should provide clear feedback messages for user actions.

6. **Accessibility**:

- The application should be accessible to users with disabilities, following WCAG guidelines.

1.12 Reliability

7. **Uptime**:

- The system should guarantee at least 99.9% uptime.
- Regular backups should be scheduled to prevent data loss.

1.13 Maintainability

8. **Code Quality**:

- The codebase should follow best practices for readability, modularity, and documentation.
- Automated tests should cover at least 80% of the codebase to ensure reliability.

9. **Documentation**:

- The system should have comprehensive documentation for developers, deployment instructions, and user guides.

1.14 Compatibility

10. **Browsers**:

- The application should be compatible with major web browsers (Chrome, Firefox, Safari, Edge).
- The application should support the latest versions of major web browsers (Chrome, Firefox, Safari, Edge).
- The application should not require browser plugins or extensions.

11. **Devices**:

- The application should be responsive and usable on desktops, tablets, and mobile devices.

1.15 Monitoring

12. **Monitoring**:

- The system should include monitoring tools to track application performance, errors, and user activity.

CHAPTER

TWO

DATABASE CHOICE AND DIAGRAMS

This application is divided into several apps, each one with its own set of models and views. The following diagrams show the class diagram for each app. Application uses PostgreSQL as a main database, and SQLite for testing purposes.

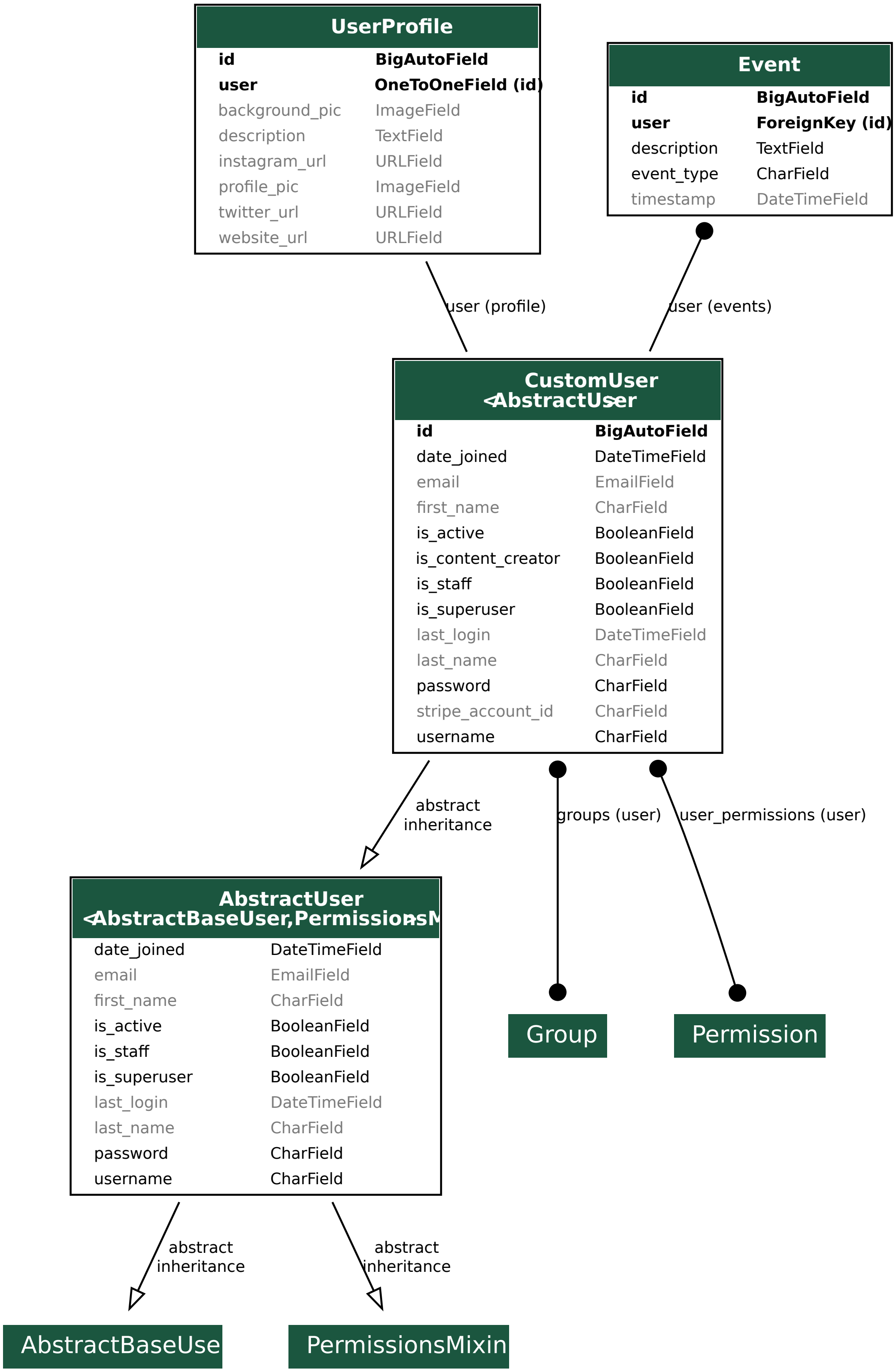
PostgreSQL was selected as the primary database for several reasons:

1. **Reliability**: PostgreSQL is known for its stability and robustness, making it a reliable choice for managing critical data.
2. **Feature-Rich**: It offers a rich set of features, including support for advanced data types, full-text search, and powerful indexing capabilities.
3. **Scalability**: PostgreSQL can efficiently handle large volumes of data and high-concurrency workloads, ensuring the application can scale as needed.
4. **Standards Compliance**: It adheres to SQL standards, which promotes compatibility and eases the integration with other systems.
5. **Community Support**: PostgreSQL has a strong, active community that contributes to its continuous improvement and provides extensive documentation and support resources.
6. **Open Source**: As an open-source database, PostgreSQL offers cost-effectiveness without compromising on quality and features.

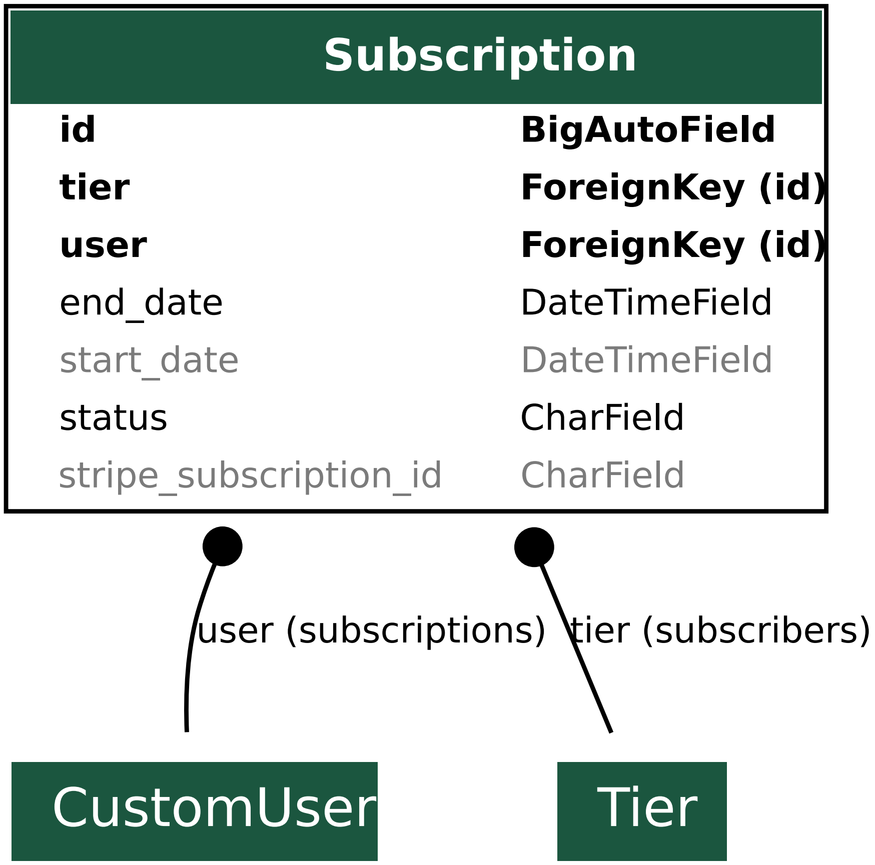
SQLite was chosen for testing purposes due to its simplicity and ease of use. It is a lightweight, serverless database that can be easily set up and used for testing purposes. SQLite is well-suited for testing because it runs in-memory and does not require any configuration, making it convenient for running tests quickly and efficiently.

The following diagrams illustrate the model structure of each app in the project:

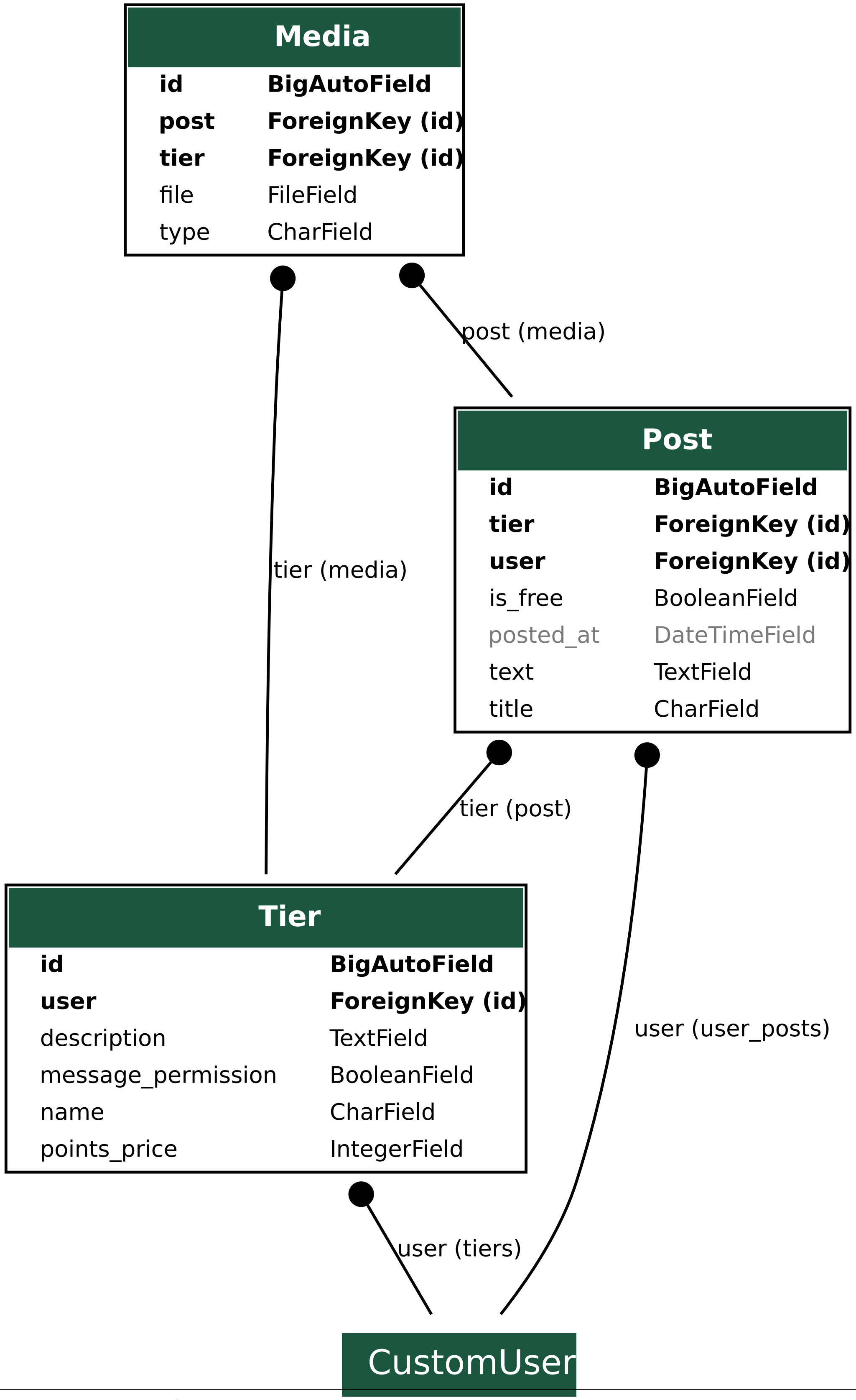
2.1 Account App Class Diagram



2.2 Client App Class Diagram



2.3 Creator App Class Diagram



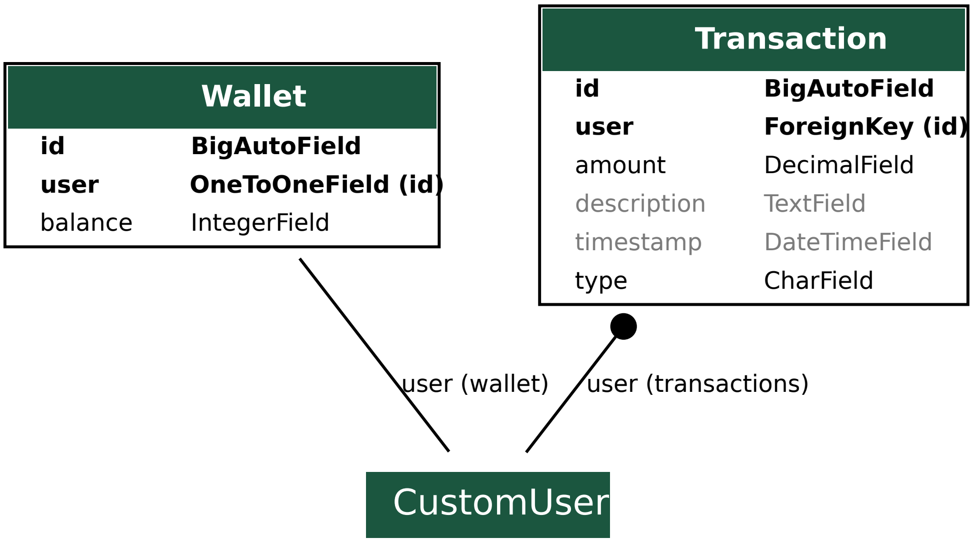
Media	
id	**BigAutoField**
post	**ForeignKey (id)**
tier	**ForeignKey (id)**
file	FileField
type	CharField

Post	
id	**BigAutoField**
tier	**ForeignKey (id)**
user	**ForeignKey (id)**
is_free	BooleanField
posted_at	DateTimeField
text	TextField
title	CharField

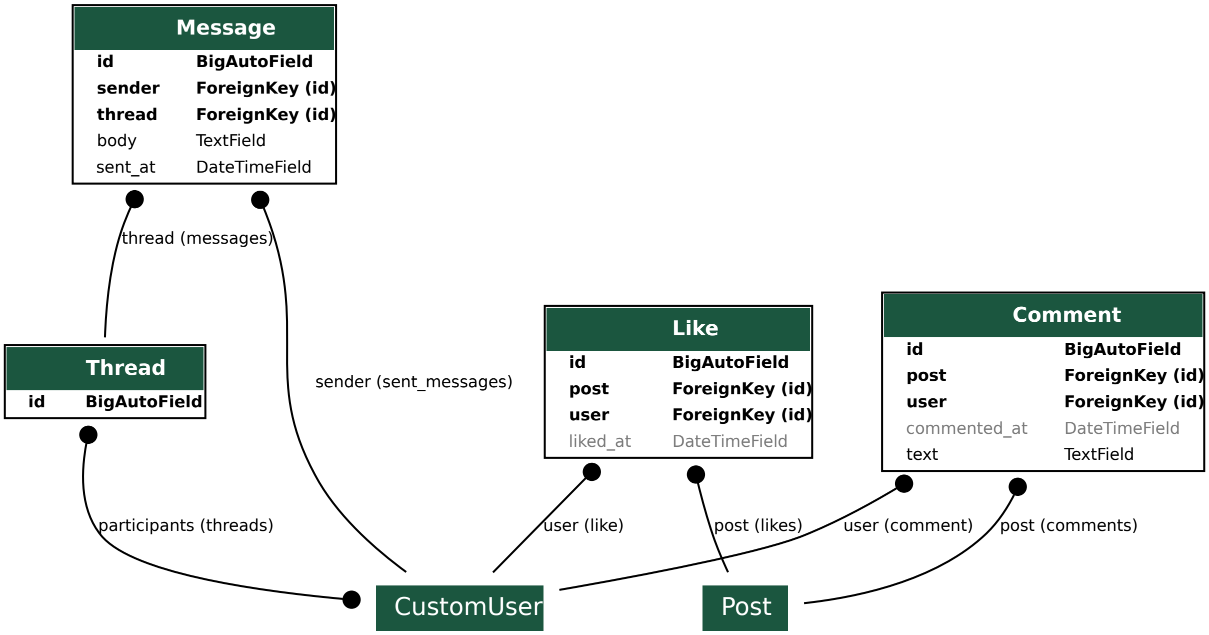
Tier	
id	**BigAutoField**
user	**ForeignKey (id)**
description	TextField
message_permission	BooleanField
name	CharField
points_price	IntegerField

CustomUser

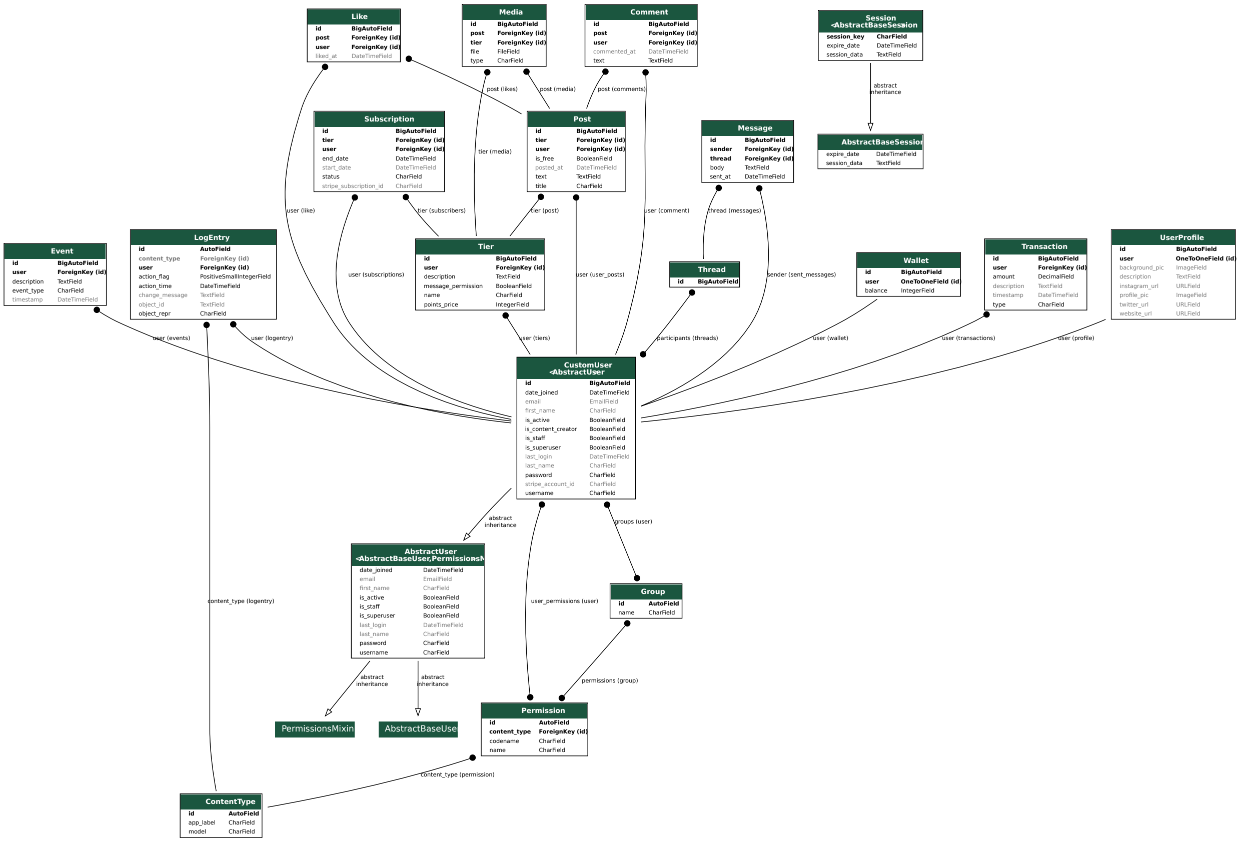
2.4 Finances App Class Diagram



2.5 Interactions App Class Diagram



2.6 Project Class Diagram



Like	
id	BigAutoField
post	ForeignKey (id)
user	ForeignKey (id)
liked_at	DateTimeField

Media	
id	BigAutoField
post	ForeignKey (id)
tier	ForeignKey (id)
file	FileField
type	CharField

Comment	
id	BigAutoField
post	ForeignKey (id)
user	ForeignKey (id)
commented_at	DateTimeField
text	TextField

Session <AbstractBaseSession>	
session_key	CharField
expire_date	DateTimeField
session_data	TextField

Subscription	
id	BigAutoField
tier	ForeignKey (id)
user	ForeignKey (id)
end_date	DateTimeField
start_date	DateTimeField
status	CharField
stripe_subscription_id	CharField

Post	
id	BigAutoField
tier	ForeignKey (id)
user	ForeignKey (id)
is_free	BooleanField
posted_at	DateTimeField
text	TextField
title	CharField

Message	
id	BigAutoField
sender	ForeignKey (id)
thread	ForeignKey (id)
body	TextField
sent_at	DateTimeField

AbstractBaseSession	
expire_date	DateTimeField
session_data	TextField

Event	
id	BigAutoField
user	ForeignKey (id)
description	TextField
event_type	CharField
timestamp	DateTimeField

LogEntry	
id	AutoField
content_type	ForeignKey (id)
user	ForeignKey (id)
action_flag	PositiveSmallIntegerField
action_time	DateTimeField
change_message	TextField
object_id	TextField
object_repr	CharField

Tier	
id	BigAutoField
user	ForeignKey (id)
description	TextField
message_permission	BooleanField
name	CharField
points_price	IntegerField

Thread	
id	BigAutoField

Wallet	
id	BigAutoField
user	OneToOneField (id)
balance	IntegerField

Transaction	
id	BigAutoField
user	ForeignKey (id)
amount	DecimalField
description	TextField
timestamp	DateTimeField
type	CharField

UserProfile	
id	BigAutoField
user	OneToOneField (id)
background_pic	ImageField
description	TextField
instagram_url	URLField
profile_pic	ImageField
twitter_url	URLField
website_url	URLField

CustomUser <AbstractUser>	
id	BigAutoField
date_joined	DateTimeField
email	EmailField
first_name	CharField
is_active	BooleanField
is_content_creator	BooleanField
is_staff	BooleanField
is_superuser	BooleanField
last_login	DateTimeField
last_name	CharField
password	CharField
stripe_account_id	CharField
username	CharField

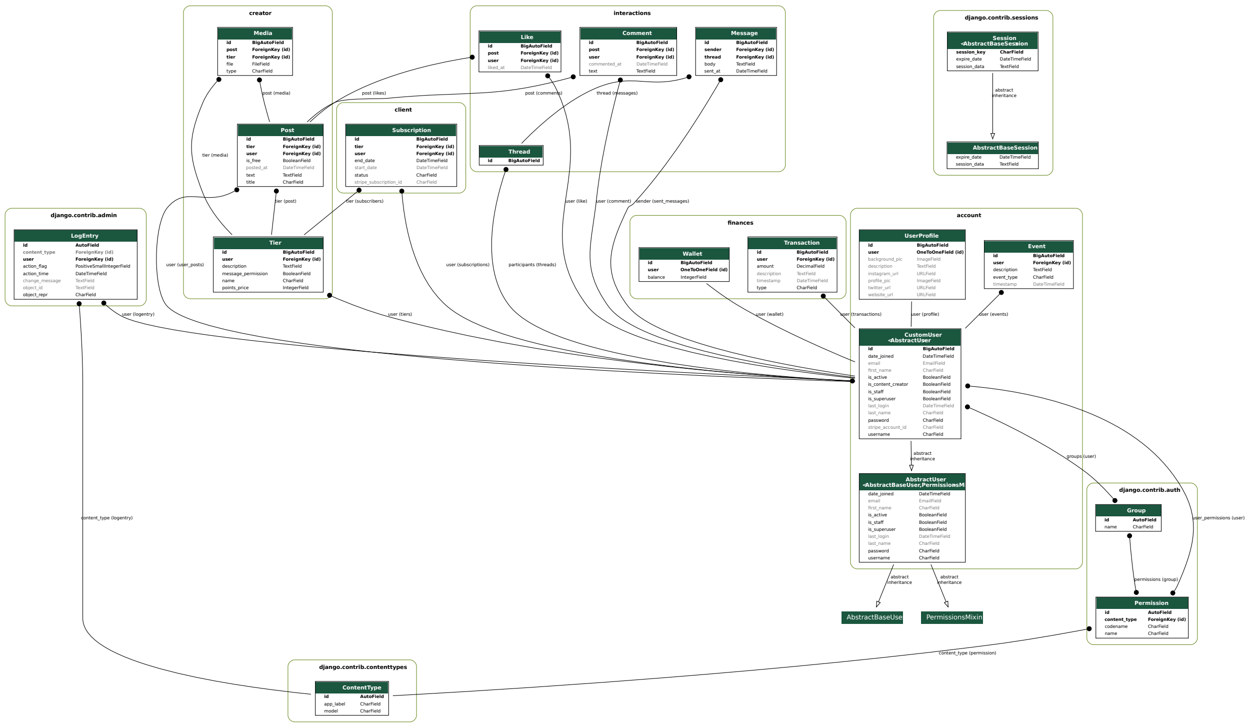
AbstractUser <AbstractBaseUser,PermissionsM	
date_joined	DateTimeField
email	EmailField
first_name	CharField
is_active	BooleanField
is_staff	BooleanField
is_superuser	BooleanField
last_login	DateTimeField
last_name	CharField
password	CharField
username	CharField

Group	
id	AutoField
name	CharField

Permission	
id	AutoField
content_type	ForeignKey (id)
codename	CharField
name	CharField

ContentType	
id	AutoField
app_label	CharField
model	CharField

2.7 Project Class Diagram (App Specific)



CHAPTER

THREE

DOCKER COMPOSE

You can use Docker Compose to easily run the project in a containerized environment.

Just run the following command:

```
docker-compose up
```

To turn off the project, run the following command:

```
docker-compose down
```

The project will be available at http://127.0.0.1:8000/

It would be a good idea to create a superuser to access the Django admin.

```
docker-compose run django python manage.py createsuperuser
```

If you don’t want to use Docker Compose, you can run the project with the following commands:

```
python -m venv venv

.\venv\Scripts\activate

pip install -r minimal-requirements.txt

cd onlyvans

python manage.py makemigrations

python manage.py migrate

python manage.py createsuperuser

python manage.py runserver
```

CHAPTER

FOUR

TESTING METHODS AND APPROACHES

In this project, comprehensive testing has been implemented to ensure the functionality, performance, and reliability of the system. The testing strategies include unit testing, integration testing, and manual testing. Below is an overview of the testing methods and approaches used:

4.1 Unit Testing

Unit tests were used to validate the functionality of individual components of the system. These tests were designed to ensure that each function, method, and class performs as expected in isolation from the rest of the system.

- **Scope**: Unit tests cover various parts of the application, including user management, content management, subscription management, financial management, and messaging.
- **Tools**: Django's built-in test framework and *pytest* are used for writing and executing unit tests.
- **Execution**: Unit tests are executed using the command *python manage.py test* or with *pytest* for more advanced testing scenarios. There were also custom test management commands to run specific test cases.

4.2 Integration Testing

Integration tests are used to verify that different components of the system work together as expected. These tests are essential for detecting issues that may arise when individual modules interact with each other.

- **Scope**: Integration tests focus on critical workflows, such as user registration and login, post creation and deletion, subscription processes, and financial transactions.
- **Tools**: Django's test framework, along with *pytest*.
- **Execution**: Integration tests were executed similarly to unit tests using Django's test management commands.

4.3 Manual Testing

Manual testing was performed to validate the user interface and overall user experience. This involves navigating through the application, performing various actions, and verifying that the application behaves as expected.

- **Scope**: Manual testing included verifying the correctness of web pages, form submissions, user interactions, and the overall look and feel of the application.
- **Tools**: Manual testing was performed using web browsers, with test scenarios documented in a testing checklist.

4.4 Test Documentation

For detailed information about the unit tests, including the specific tests implemented, test cases, and their expected outcomes, please refer to the tests.rst.

4.5 Summary

The testing strategy employed in this project ensures that the application is robust, reliable, and performs as expected. By combining unit testing, integration testing, and manual testing, we have aimed to cover all critical aspects of the system and ensure a high level of quality and user satisfaction.

CHAPTER

FIVE

MODELS

5.1 Account Models

class account.models.**CustomUser**(**args*, ***kwargs*)

Bases: `AbstractUser`

Model representing a custom user, extending Django's default user model (AbstractUser).

Fields:

- is_content_creator (BooleanField): Indicates whether the user is a content creator. Defaults to False
- stripe_account_id (CharField): Stripe account ID associated with the user. Can be blank or null.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

class account.models.**Event**(**args*, ***kwargs*)

Bases: `Model`

Model representing an event associated with a user.

Fields:

- user (ForeignKey): The user who triggered the event. Deletes events if the user is deleted.
- event_type (CharField): Type of the event.
- description (TextField): Description of the event.
- timestamp (DateTimeField): Time when the event occurred. Automatically set to the current time when created.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

class account.models.**UserProfile**(**args*, ***kwargs*)

Bases: `Model`

Model representing a user profile, associated one-to-one with the custom user (CustomUser).

Fields:

- user (OneToOneField): The user to whom this profile belongs. Deletes the profile if the user is deleted.
- profile_pic (ImageField): Profile picture of the user. Can be blank or null.
- background_pic (ImageField): Background picture of the user's profile. Can be blank or null.
- description (TextField): A brief description of the user. Can be blank.
- website_url (URLField): URL of the user's website. Can be blank or null.
- twitter_url (URLField): URL of the user's Twitter profile. Validated by *validate_twitter_url*. Can be blank or null.
- instagram_url (URLField): URL of the user's Instagram profile. Validated by *validate_instagram_url*. Can be blank or null.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

`account.models.`**`create_or_update_user_profile`**(*sender*, *instance*, *created*, ***kwargs*)

Signal receiver that creates or updates a UserProfile whenever a CustomUser is created or updated.

Parameters

- **sender** (*class*) – The model class sending the signal.
- **instance** (`CustomUser`) – The actual instance being saved.
- **created** (*bool*) – Whether this instance is being created.
- ****kwargs** – Additional keyword arguments.

5.2 Client Models

`class client.models.`**`Subscription`**(**args*, ***kwargs*)

Bases: `Model`

Model representing a subscription to a content creator's tier.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

clean()

Custom validation for the Subscription model.

Raises

ValidationError – If end date is not after the start date, if end date is in the past, or if there is already an active subscription for the same creator.

is_expired()

Checks if the subscription is expired based on the end date.

Returns

True if the subscription is expired, False otherwise.

Return type
bool

5.3 Creator Models

class `creator.models.`**`Media`**`(*args, **kwargs)`

Bases: `Model`

Model representing media files associated with posts.

Attributes: - post: The post to which the media file is attached. - file: The media file itself. - type: The type of media (image or video). - tier: The tier associated with the media file.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

save`(*args, **kwargs)`

Override the save method to determine the type of media and assign the tier.

class `creator.models.`**`Post`**`(*args, **kwargs)`

Bases: `Model`

Model representing a post made by a user.

Attributes: - title: The title of the post. - text: The content of the post. - posted_at: The date and time the post was made. - is_free: Boolean indicating if the post is free. - tier: The tier associated with the post. - user: The user who created the post.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

property comments_count

Returns the number of comments on the post.

property likes_count

Returns the number of likes for the post.

class `creator.models.`**`Tier`**`(*args, **kwargs)`

Bases: `Model`

Model representing a subscription tier.

Attributes: - name: The name of the tier. - points_price: The price in points for subscribing to the tier. - description: A description of the tier. - user: The user who created the tier. - message_permission: Boolean indicating if the tier includes messaging permissions.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

5.4 Finances Models

`class finances.models.Transaction(*args, **kwargs)`

Bases: `Model`

Model representing a transaction made by a user.

Fields:

user (ForeignKey): The user who made the transaction. Deletes transactions if the user is deleted. type (CharField): The type of the transaction. Choices are 'PURCHASE', 'SUBSCRIPTION', 'DONATION', 'WITHDRAWAL'. amount (DecimalField): The amount of the transaction, allowing for up to 10 digits with 2 decimal places. timestamp (DateTimeField): The time when the transaction was made. Automatically set to the current date and time when the transaction is created. description (TextField): A description of the transaction. Can be blank or null.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

`class finances.models.Wallet(*args, **kwargs)`

Bases: `Model`

Model representing a user's wallet.

Fields:

user (OneToOneField): The user to whom this wallet belongs. Deletes the wallet if the user is deleted. balance (IntegerField): The current balance of points in the wallet. Defaults to 0.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

5.5 Interactions Models

`class interactions.models.Comment(*args, **kwargs)`

Bases: `Model`

Represents a comment on a post by a user.

Fields: - user: A foreign key to the CustomUser who made the comment. - post: A foreign key to the Post that was commented on. - text: The content of the comment. - commented_at: The timestamp when the comment was made.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

class `interactions.models.`**`Like`**`(`**args*, ***kwargs*`)`

Bases: `Model`

Represents a like on a post by a user.

Fields: - user: A foreign key to the CustomUser who liked the post. - post: A foreign key to the Post that was liked. - liked_at: The timestamp when the post was liked.

Meta: - unique_together: Ensures that a user can like a post only once.

exception **`DoesNotExist`**

Bases: `ObjectDoesNotExist`

exception **`MultipleObjectsReturned`**

Bases: `MultipleObjectsReturned`

class `interactions.models.`**`Message`**`(`**args*, ***kwargs*`)`

Bases: `Model`

Represents a message in a thread.

Fields: - thread: A foreign key to the Thread in which the message is sent. - sender: A foreign key to the CustomUser who sent the message. - body: The content of the message. - sent_at: The timestamp when the message was sent.

Methods: - __str__(): Returns a string representation of the message, indicating the sender and the thread.

exception **`DoesNotExist`**

Bases: `ObjectDoesNotExist`

exception **`MultipleObjectsReturned`**

Bases: `MultipleObjectsReturned`

class `interactions.models.`**`Thread`**`(`**args*, ***kwargs*`)`

Bases: `Model`

Represents a thread of messages between two users.

Fields: - participants: A many-to-many relationship with CustomUser, indicating the participants in the thread.

Methods: - __str__(): Returns a string representation of the thread, listing the usernames of the participants. - get_other_participant(user): Given a user, returns the other participant in the thread. - clean(): Validates that a thread can only have two participants.

exception **`DoesNotExist`**

Bases: `ObjectDoesNotExist`

exception **`MultipleObjectsReturned`**

Bases: `MultipleObjectsReturned`

`clean``()`

Validates that the thread can only have two participants.

Raises

`ValidationError` – If the thread has more than two participants.

`get_other_participant``(`*user*`)`

Returns the other participant in the thread, given one of the participants.

Parameters

`user` (`CustomUser`) – The user for whom to find the other participant.

Returns
The other participant in the thread.

Return type
CustomUser

CHAPTER

SIX

VIEWS

6.1 Account Views

`account.views.`**`change_password`**`(`*request*`)`

Handles the password change for a logged-in user.

If the request method is POST, it will process the form for changing the user's password. If the form is valid, the user's password will be updated and an event will be created to log the change. Success and error messages will be displayed accordingly.

Parameters

`request` (*`HttpRequest`*) – The HTTP request object containing user data.

Returns

Renders the 'account/change_password.html' template with

the password form, or redirects to the 'update-profile' page after successful form submission.

Return type

HttpResponse

Forms:

UserPasswordChangeForm: Form for changing the user's password.

`account.views.`**`create_stripe_account`**`(`*request*`)`

View for creating a Stripe account for the user.

Parameters

`request` (*`HttpRequest`*) – The request object.

Returns

A redirect to Stripe account onboarding or a redirect to the update profile page.

Return type

HttpResponse

`account.views.`**`event_history`**`(`*request*`)`

View for displaying the user's event history.

Parameters

`request` (*`HttpRequest`*) – The request object.

Returns

The rendered event history page.

Return type
HttpResponse

`account.views.home(request)`

View for the home page. Redirects authenticated users based on their role.

Parameters
`request` (*HttpRequest*) – The request object.

Returns
The rendered home page or a redirect.

Return type
HttpResponse

`account.views.profile(request, username)`

View for displaying a user's profile. Handles different visibility of posts based on subscription status.

Parameters

- `request` (*HttpRequest*) – The request object.
- `username` (*str*) – The username of the profile to view.

Returns
The rendered profile page.

Return type
HttpResponse

`account.views.register(request)`

View for user registration. Handles the registration form and user creation.

Parameters
`request` (*HttpRequest*) – The request object.

Returns
The rendered registration page or a redirect.

Return type
HttpResponse

`account.views.update_profile(request)`

Handles the profile update for a logged-in user.

If the request method is POST, it will process the forms for updating the user profile. If the forms are valid, the user's profile will be updated and an event will be created to log the update. Success and error messages will be displayed accordingly.

Parameters
`request` (*HttpRequest*) – The HTTP request object containing user data.

Returns

Renders the 'account/update_profile.html' template with
the user and profile forms, or redirects to the same page after successful form submission.

Return type
HttpResponse

Forms:
CustomUserUpdateForm: Form for updating the CustomUser model. UserProfileForm: Form for updating the UserProfile model.

`account.views.userlogin`(*request*)

View for user login. Handles the authentication form and user login.

Parameters
`request` (*HttpRequest*) – The request object.

Returns
The rendered login page or a redirect.

Return type
HttpResponse

`account.views.userlogout`(*request*)

View for user logout. Logs out the user and redirects to the home page.

Parameters
`request` (*HttpRequest*) – The request object.

Returns
A redirect to the home page.

Return type
HttpResponse

6.2 Client Views

`client.views.cancel_subscription`(*request*, *subscription_id*)

Cancel a user's subscription.

Sets the subscription status to 'CANCELLED' and creates relevant events.

Parameters

- `request` – The HTTP request object.
- `subscription_id` – The ID of the subscription to cancel.

Returns
Redirects to the subscriptions page with a success message.

`client.views.dashboard`(*request*)

Display the client's dashboard with posts from followed creators.

Retrieves active subscriptions of the user, followed creators, and their posts, then paginates the posts and returns the dashboard view.

Parameters
`request` – The HTTP request object.

Returns
Rendered dashboard HTML page with posts and liked posts.

`client.views.discover_creators`(*request*)

Display the discover creators page with various categories of creators.

Fetches and displays top, new, most active, and random creators. Also handles search functionality.

Parameters
`request` – The HTTP request object.

Returns
Rendered discover creators HTML page with categorized creators and search results.

`client.views.`**`extend_subscription`**`(`*request*, *subscription_id*`)`

Extend a user's subscription.

Verifies user wallet balance, deducts points, and extends the subscription.

Parameters

- **`request`** – The HTTP request object.
- **`subscription_id`** – The ID of the subscription to extend.

Returns
Redirects to the subscriptions page with a success or error message.

`client.views.`**`select_tier`**`(`*request*, *username*`)`

Display the select tier page for a specific creator.

Fetches the creator and their tiers, then returns the select tier view.

Parameters

- **`request`** – The HTTP request object.
- **`username`** – The username of the creator.

Returns
Rendered select tier HTML page with the creator and their tiers.

`client.views.`**`subscribe_to_tier`**`(`*request*, *username*, *tier_id*`)`

Subscribe the user to a specific tier of a creator.

Checks for an active subscription, verifies user wallet balance, deducts points, and creates the subscription. If the user or creator doesn't have a wallet, it creates one.

Parameters

- **`request`** – The HTTP request object.
- **`username`** – The username of the creator.
- **`tier_id`** – The ID of the tier to subscribe to.

Returns
Redirects to the dashboard or select tier page with a success or error message.

`client.views.`**`subscriptions`**`(`*request*`)`

Display the user's active subscriptions.

Fetches and displays the active subscriptions of the user.

Parameters
`request` – The HTTP request object.

Returns
Rendered subscriptions HTML page with the user's active subscriptions.

6.3 Creator Views

`creator.views.`**`create_post`**`(`*request*`)`

Handle the creation of a new post by the creator.

If the request method is POST, it validates and saves the post and its associated media files. Otherwise, it displays an empty form for creating a post.

Parameters
`request` – The HTTP request object.

Returns
The rendered create post page or redirects to the dashboard on successful creation.

Return type
HttpResponse

`creator.views.`**`create_tier`**`(`*request*`)`

Handle the creation of a new tier by the creator.

If the request method is POST, it validates and saves the new tier. Otherwise, it displays an empty form for creating a tier.

Parameters
`request` – The HTTP request object.

Returns
The rendered create tier page or redirects to the tiers page on successful creation.

Return type
HttpResponse

`creator.views.`**`dashboard`**`(`*request*`)`

Display the creator's dashboard with their posts.

Retrieves posts created by the logged-in creator, paginates them, and renders them in the 'creator/dashboard.html' template. Also fetches posts liked by the creator for display.

Parameters
`request` – The HTTP request object.

Returns
The rendered dashboard page.

Return type
HttpResponse

`creator.views.`**`delete_tier`**`(`*request*, *tier_id*`)`

Handle the deletion of a tier by the creator.

Deletes the specified tier if it has no active subscribers. Otherwise, displays an error message.

Parameters

- **`request`** – The HTTP request object.
- **`tier_id`** – The ID of the tier to be deleted.

Returns
Redirects to the tiers page.

Return type
HttpResponse

creator.views.**post_delete**(*request*, *post_id*)

Handle the deletion of a post by the creator.

Deletes the specified post if the logged-in user is the creator of the post. Otherwise, displays an error message.

Parameters

- **request** – The HTTP request object.
- **post_id** – The ID of the post to be deleted.

Returns

Redirects to the dashboard page.

Return type

HttpResponse

creator.views.**tiers**(*request*)

Display the creator's tiers with their active subscribers.

Retrieves tiers created by the logged-in creator and their active subscribers, and renders them in the 'creator/tiers.html' template.

Parameters

request – The HTTP request object.

Returns

The rendered tiers page.

Return type

HttpResponse

6.4 Finances Views

finances.views.**purchase_points**(*request*)

Handles the purchase of points by the user. Displays a form to enter the number of points to purchase, processes the payment using Stripe, and redirects to the Stripe checkout session.

Parameters

request (*HttpRequest*) – The HTTP request object.

Returns

Renders the purchase points form or redirects to the Stripe checkout session.

Return type

HttpResponse

finances.views.**purchase_success**(*request*)

Handles the successful purchase of points. Updates the user's wallet balance, creates a transaction record, and logs the event.

Parameters

request (*HttpRequest*) – The HTTP request object.

Returns

Redirects to the home page with a success message.

Return type

HttpResponse

finances.views.**withdraw_points**(*request*)

Handles the withdrawal of points by the user. Displays a form to enter the number of points to withdraw, processes the withdrawal using Stripe, and updates the user's wallet balance.

Parameters

request (*HttpRequest*) – The HTTP request object.

Returns

Renders the withdrawal form or redirects to the home page with a success message.

Return type

HttpResponse

6.5 Interactions Views

interactions.views.**direct_messages**(*request*)

Display the direct messages for the logged-in user.

This view retrieves all message threads involving the logged-in user, annotates them with the date of the last message, and orders them by this date. Threads are paginated with 20 threads per page.

Only threads where the user has messaging permission with the other participant are included in the context.

Parameters

request (*HttpRequest*) – The HTTP request object.

Returns

The rendered direct messages page with threads.

Return type

HttpResponse

interactions.views.**like_post**(*request*, *post_id*)

Like or unlike a specific post.

If the post is already liked by the user, the like is removed. Otherwise, a new like is added. The view returns a JSON response indicating the success status, the current number of likes, and whether the post is now liked by the user.

Parameters

- **request** (*HttpRequest*) – The HTTP request object.
- **post_id** (*int*) – The ID of the post to be liked or unliked.

Returns

A JSON response with the success status, likes count, and liked status.

Return type

JsonResponse

interactions.views.**view_thread**(*request*, *username=None*, *thread_id=None*)

View and send messages in a specific thread.

If *username* is provided, a thread between the logged-in user and the specified user is retrieved or created. If *thread_id* is provided, the specific thread is retrieved.

Messaging permissions are checked before displaying or sending messages.

Parameters

- **request** (*HttpRequest*) – The HTTP request object.
- **username** (*str, optional*) – The username of the other participant.
- **thread_id** (*int, optional*) – The ID of the thread.

Returns

The rendered view thread page with messages and form.

Return type

HttpResponse

CHAPTER

SEVEN

URLS

7.1 Account URLs

account.urls.**urlpatterns = [<URLPattern '' [name='home']>, <URLPattern 'register' [name='register']>, <URLPattern 'login' [name='login']>, <URLPattern 'logout' [name='logout']>, <URLPattern 'profile/update/' [name='update-profile']>, <URLPattern 'profile/create-stripe-account/' [name='create_stripe_account']>, <URLPattern 'profile/change-password/' [name='change-password']>, <URLPattern 'profile/<str:username>/' [name='profile']>, <URLPattern 'history/' [name='history']>]**

URL configuration for the account application.

Each path function specifies a URL pattern and the view that should handle requests to that pattern. Additionally, a name is provided for each URL pattern to make it easier to refer to them in templates and other parts of the application.

Paths: - ‘’ (home): Handles the home page. View: views.home - ‘register’: Handles user registration. View: views.register - ‘login’: Handles user login. View: views.userlogin - ‘logout’: Handles user logout. View: views.userlogout - ‘profile/update/’: Handles profile updates. View: views.update_profile - ‘profile/create-stripe-account/’: Handles creating Stripe accounts for users. View: views.create_stripe_account - ‘profile/change-password/’: Handles changing user passwords. View: views.change_password - ‘profile/<str:username>/’: Displays user profiles. View: views.profile - ‘history/’: Displays user event history. View: views.event_history

7.2 Client URLs

client.urls.**urlpatterns = [<URLPattern 'dashboard/' [name='dashboard']>, <URLPattern 'discover/' [name='discover_creators']>, <URLPattern 'subscribe/<str:username>/' [name='select-tier']>, <URLPattern 'subscribe/<str:username>/<int:tier_id>/' [name='subscribe-to-tier']>, <URLPattern 'subscriptions/' [name='subscriptions']>, <URLPattern 'subscriptions/extend/<int:subscription_id>/' [name='extend_subscription']>, <URLPattern 'subscriptions/cancel/<int:subscription_id>/' [name='cancel_subscription']>]**

URL configuration for the client application.

Paths: - ‘dashboard/’: Displays the user’s dashboard. View: views.dashboard - ‘discover/’: Allows users to discover new creators. View: views.discover_creators - ‘subscribe/<str:username>/’: Allows users to select a tier to subscribe to for a specific creator. View: views.select_tier - ‘subscribe/<str:username>/<int:tier_id>/’: Subscribes the user to a specific tier of a creator. View: views.subscribe_to_tier - ‘subscriptions/’: Displays the user’s current subscriptions. View: views.subscriptions - ‘subscriptions/extend/<int:subscription_id>/’: Extends the user’s subscription. View: views.extend_subscription - ‘subscriptions/cancel/<int:subscription_id>/’: Cancels the user’s subscription. View: views.cancel_subscription

7.3 Creator URLs

```
creator.urls.urlpatterns = [<URLPattern 'dashboard/' [name='dashboard']>, <URLPattern
'create-post/' [name='create-post']>, <URLPattern 'tiers/' [name='tiers']>, <URLPattern
'tiers/create/' [name='create-tier']>, <URLPattern 'tiers/delete/<int:tier_id>/'
[name='delete-tier']>, <URLPattern 'post/<int:post_id>/delete/' [name='post_delete']>]
```

URL configuration for the creator application.

Paths: - 'dashboard/': Handles the creator's dashboard. View: views.dashboard - 'create-post/': Handles the creation of a new post. View: views.create_post - 'tiers/': Displays all tiers created by the creator. View: views.tiers - 'tiers/create/': Handles the creation of a new tier. View: views.create_tier - 'tiers/delete/<int:tier_id>/': Handles the deletion of a tier specified by tier_id. View: views.delete_tier - 'post/<int:post_id>/delete/': Handles the deletion of a post specified by post_id. View: views.post_delete

7.4 Finances URLs

```
finances.urls.urlpatterns = [<URLPattern 'purchase/' [name='purchase']>, <URLPattern
'purchase-success/' [name='purchase-success']>, <URLPattern 'withdraw/'
[name='withdraw']>]
```

URL patterns for the finances application.

This module defines the URL patterns for the views in the finances application, linking URLs to their corresponding view functions.

Patterns: - 'purchase/': Links to the purchase_points view, allowing users to purchase points. - 'purchase-success/': Links to the purchase_success view, handling the post-purchase process. - 'withdraw/': Links to the withdraw_points view, allowing users to withdraw points.

7.5 Interactions URLs

```
interactions.urls.urlpatterns = [<URLPattern 'messages/' [name='direct_messages']>,
<URLPattern 'messages/send/<str:username>/' [name='view_thread_with_user']>, <URLPattern
'messages/thread/<int:thread_id>/' [name='view_thread']>, <URLPattern
'like/<int:post_id>/' [name='like_post']>]
```

URL configuration for the interactions application.

Each path function specifies a URL pattern and the view that should handle requests to that pattern. Additionally, a name is provided for each URL pattern to make it easier to refer to them in templates and other parts of the application.

Paths: - 'messages/': Displays direct messages for the logged-in user. View: views.direct_messages - 'messages/send/<str:username>/': Displays a message thread with a specific user. View: views.view_thread - 'messages/thread/<int:thread_id>/': Displays a specific message thread by thread ID. View: views.view_thread - 'like/<int:post_id>/': Allows the logged-in user to like a specific post. View: views.like_post

CHAPTER

EIGHT

FORMS

8.1 Account Forms

class account.forms.**CustomUserChangeForm**(**args*, ***kwargs*)

Bases: `UserChangeForm`

Form for updating a user. Inherits from Django's UserChangeForm.

Meta:
model: Specifies the model to use for this form (CustomUser). fields: The fields to include in the form.

property media

Return all media required to render the widgets on this form.

class account.forms.**CustomUserCreationForm**(**args*, ***kwargs*)

Bases: `UserCreationForm`

Form for creating a new user. Inherits from Django's UserCreationForm.

Meta:
model: Specifies the model to use for this form (CustomUser). fields: The fields to include in the form.

clean_username()

Validates that the username is not in the list of reserved usernames.

Returns
The cleaned username.

Return type
str

Raises
ValidationError – If the username is reserved.

property media

Return all media required to render the widgets on this form.

class account.forms.**CustomUserUpdateForm**(**args*, ***kwargs*)

Bases: `ModelForm`

Form for updating CustomUser details. This form allows updating the email, first name, and last name of the user. If the user has a Stripe account ID, the field is included but set to read-only and disabled.

Meta:
model: Specifies the model to use for this form (CustomUser). fields: Specifies the fields to include in the form.

clean()

Cleans the form data. Ensures that stripe_account_id is not required if it doesn't exist for content creators.

Returns
The cleaned data.

Return type
dict

property media

Return all media required to render the widgets on this form.

class account.forms.**UserPasswordChangeForm**(**args*, ***kwargs*)

Bases: `PasswordChangeForm`

Form for changing a user's password. Inherits from Django's PasswordChangeForm.

Meta:
model: Specifies the model to use for this form (CustomUser). fields: The fields to include in the form.

property media

Return all media required to render the widgets on this form.

class account.forms.**UserProfileForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

Bases: `ModelForm`

Form for updating user profile details. Inherits from Django's ModelForm.

Meta:
model: Specifies the model to use for this form (UserProfile). fields: The fields to include in the form.

property media

Return all media required to render the widgets on this form.

8.2 Creator Forms

class creator.forms.**MediaForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)

Bases: `Form`

Form for uploading media files.

Fields:

- files: Multiple file field for uploading media files.

property media

Return all media required to render the widgets on this form.

class creator.forms.**PostForm**(**args*, ***kwargs*)

Bases: `ModelForm`

Form for creating and editing posts.

Fields:

- title: The title of the post.
- text: The content of the post.
- is_free: Boolean indicating if the post is free.
- tier: The tier associated with the post.

Labels:

- title: 'Title'
- text: 'Content'
- is_free: 'Is this a free post?'
- tier: 'Choose a tier'

`clean()`

Clean and validate the form data. Ensure 'tier' is not required for free posts and required for paid posts.

`property media`

Return all media required to render the widgets on this form.

`class creator.forms.TierForm(*args, **kwargs)`

Bases: `ModelForm`

Form for creating and editing tiers.

Fields:

- name: The name of the tier.
- points_price: The points price of the tier.
- description: The description of the tier.
- message_permission: Boolean indicating if messaging is allowed in this tier.

`clean()`

Ensure the user does not exceed the limit of 12 tiers.

`clean_name()`

Validate the uniqueness of the tier name for the user.

`clean_points_price()`

Validate that the points price is greater than zero.

`property media`

Return all media required to render the widgets on this form.

8.3 Finances Forms

class finances.forms.**PurchasePointsForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)

Bases: `Form`

Form for purchasing points.

Fields:

points (ChoiceField): A dropdown field allowing users to select the number of points to purchase. The choices available are 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, and 100000 points.

property media

Return all media required to render the widgets on this form.

class finances.forms.**WithdrawPointsForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)

Bases: `Form`

Form for withdrawing points.

Fields:

points (IntegerField): An integer field allowing users to specify the number of points to withdraw. The minimum value is 1 point.

property media

Return all media required to render the widgets on this form.

8.4 Interactions Forms

class interactions.forms.**MessageForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

Bases: `ModelForm`

Form for creating and sending messages.

This form is linked to the Message model and includes a single field for the message body. The body field is rendered as a textarea with a placeholder and custom row size.

property media

Return all media required to render the widgets on this form.

CHAPTER

NINE

TESTS

9.1 Account Tests

class account.tests.**AccountTests**(*methodName='runTest'*)

Bases: `TestCase`

Test case for the account application.

setUp()

Set up a user and their profile for testing.

test_change_password_view()

Test the change password view. Ensures the form is displayed and processes correctly.

test_create_stripe_account_view()

Test the create Stripe account view. Ensures the Stripe account is created and user is redirected.

test_event_history_view()

Test the event history view. Ensures the event history is displayed correctly.

test_home_view_redirects_authenticated_user()

Test that an authenticated user is redirected to the appropriate dashboard.

test_profile_view()

Test the profile view. Ensures the profile page is displayed correctly.

test_register_view()

Test the registration view. Ensures the form is displayed and processes correctly.

test_update_profile_view()

Test the update profile view. Ensures the form is displayed and processes correctly.

test_userlogin_view()

Test the login view. Ensures the form is displayed and processes correctly.

test_userlogout_view()

Test the logout view. Ensures the user is logged out and redirected.

9.2 Client Tests

class client.tests.**ClientTests**(*methodName='runTest'*)

Bases: `TestCase`

setUp()

Set up test data for each test case. This includes creating a client user, a creator user, a tier, and wallets for both users.

test_cancel_subscription()

Test canceling a subscription to ensure its status is updated to 'CANCELLED'.

test_dashboard()

Test the dashboard view to ensure it loads correctly for a logged-in client user.

test_discover_creators()

Test the discover creators view to ensure it loads correctly for a logged-in client user.

test_extend_subscription()

Test extending a subscription to ensure the end date and wallet balances are updated correctly.

test_select_tier()

Test the select tier view to ensure it loads correctly for a given creator.

test_subscribe_to_tier()

Test subscribing to a tier, ensuring the subscription is created and balances are updated.

test_subscription_model_validation()

Test the validation logic of the Subscription model to ensure invalid subscriptions are caught.

test_subscription_unique_constraint()

Test the unique constraint of the Subscription model to prevent multiple active subscriptions to the same tier.

test_subscriptions()

Test the subscriptions view to ensure it displays active subscriptions correctly.

test_wallet_balance_deduction()

Test that subscribing to a tier correctly deducts points from the client's wallet.

test_wallet_balance_insufficient()

Test subscribing to a tier when the client's wallet balance is insufficient.

9.3 Creator Tests

class creator.tests.**CreatorTests**(*methodName='runTest'*)

Bases: `TestCase`

Test cases for the creator application, covering views, models, and constraints.

setUp()

Set up test data for the tests. This includes creating a content creator user, a client user, their wallets, and a subscription tier.

test_create_post()

Test creating a new post.

test_create_tier()

Test creating a new tier.

test_create_tier_invalid_price()

Test creating a tier with an invalid price.

test_dashboard()

Test the creator dashboard view.

test_delete_tier_with_active_subscribers()

Test deleting a tier with active subscribers.

test_invalid_post_creation()

Test creating a post with invalid data.

test_post_delete()

Test deleting a post.

test_subscription_creation()

Test creating a subscription.

test_tier_unique_constraint()

Test the unique constraint for tiers.

test_tiers()

Test the tiers view.

test_wallet_balance_insufficient()

Test subscription with insufficient wallet balance.

9.4 Finances Tests

class finances.tests.**FinancesTests**(*methodName='runTest'*)

Bases: `TestCase`

setUp()

Set up users, clients, and wallets for testing.

test_purchase_points_view_get()

Test the GET method of the purchase_points view. Ensures the form is displayed correctly for client.

test_purchase_points_view_post(*mock_stripe_create*)

Test the POST method of the purchase_points view. Ensures the form processes correctly for client.

test_purchase_success_view(*mock_stripe_retrieve*)

Test the purchase_success view. Ensures points are added to the client's wallet and transaction is recorded.

test_transaction_model()

Test the Transaction model.

test_wallet_model()

Test the Wallet model.

test_withdraw_points_view_get(*mock_stripe_account_retrieve*)

Test the GET method of the withdraw_points view. Ensures the form is displayed correctly for creator.

test_withdraw_points_view_post(*mock_stripe_transfer_create*, *mock_stripe_account_retrieve*)

Test the POST method of the withdraw_points view. Ensures the form processes correctly for creator.

9.5 Interactions Tests

class interactions.tests.**LikeTests**(*methodName='runTest'*)

Bases: `TestCase`

Test cases for liking and unliking posts.

setUp()

Set up test data for like tests. Creates a user, a content creator, a tier, and a post.

test_double_like_post_view()

Test that liking a post twice correctly unlikes the post.

test_like_creation()

Test that a like is correctly created for a post.

test_like_post_view()

Test that the like post view correctly likes a post.

test_unlike_post_view()

Test that the like post view correctly unlikes a post.

class interactions.tests.**MessagingPermissionTests**(*methodName='runTest'*)

Bases: `TestCase`

Test cases for checking messaging permissions between users based on their subscriptions.

setUp()

Set up test data for messaging permission tests. Creates a content creator user, a regular user, wallets, tiers, and subscriptions.

test_has_messaging_permission_with_permission()

Test that messaging permission is granted when the user has a valid subscription with messaging permission.

test_has_messaging_permission_without_permission()

Test that messaging permission is denied when the user does not have a valid subscription with messaging permission.

class interactions.tests.**ThreadMessageTests**(*methodName='runTest'*)

Bases: `TestCase`

Test cases for creating and managing message threads between users.

setUp()

Set up test data for thread and message tests. Creates two users, a thread between them, and messages in the thread.

test_get_other_participant()

Test that the correct other participant is retrieved from the thread.

`test_message_creation()`

Test that messages are correctly created and associated with the thread.

`test_thread_creation()`

Test that a thread is correctly created with two participants.

CHAPTER

TEN

UTILITIES

10.1 Account Utilities

`account.templatetags.custom_filters.`**`ends_with`**`(`*value*, *arg*`)`

Custom template filter that checks if a given value ends with the specified suffix.

Parameters

- **`value`** (*`str`*) – The string to be checked.
- **`arg`** (*`str`*) – The suffix to check for.

Returns

True if the value ends with the specified suffix, False otherwise.

Return type

bool

`account.templatetags.custom_filters.`**`random_greeting`**`()`

Custom template tag that returns a random greeting message. The greeting is selected from a list of common greetings in different languages.

Returns

A random greeting message.

Return type

str

`account.templatetags.custom_filters.`**`starts_with`**`(`*value*, *prefix*`)`

Custom template filter that checks if a given value starts with the specified prefix.

Parameters

- **`value`** (*`str`*) – The string to be checked.
- **`prefix`** (*`str`*) – The prefix to check for.

Returns

True if the value starts with the specified prefix, False otherwise.

Return type

bool

`account.helpers.`**`get_active_subscribers_count`**`(`*user*`)`

Get the total number of active subscribers for a given user.

Parameters

`user` (`CustomUser`) – The user whose active subscribers are being counted.

Returns
The total number of active subscribers across all tiers of the user.

Return type
int

`account.helpers.`**`get_total_likes`**`(`*user*`)`

Get the total number of likes on all posts created by a given user.

Parameters
user (CustomUser) – The user whose posts' likes are being counted.

Returns
The total number of likes across all posts by the user.

Return type
int

`account.helpers.`**`get_total_likes_given`**`(`*user*`)`

Get the total number of likes given by a given user.

Parameters
user (CustomUser) – The user whose given likes are being counted.

Returns
The total number of likes given by the user.

Return type
int

`account.helpers.`**`get_total_subscriptions`**`(`*user*`)`

Get the total number of active subscriptions for a given user.

Parameters
user (CustomUser) – The user whose subscriptions are being counted.

Returns
The total number of active subscriptions for the user.

Return type
int

`account.validators.`**`validate_instagram_url`**`(`*value*`)`

Validates that the given value is a valid Instagram URL.

Parameters
value (*str*) – The URL to validate.

Raises
ValidationError – If the URL is not a valid Instagram URL.

`account.validators.`**`validate_twitter_url`**`(`*value*`)`

Validates that the given value is a valid Twitter URL.

Parameters
value (*str*) – The URL to validate.

Raises
ValidationError – If the URL is not a valid Twitter URL.

10.2 Client Utilities

`client.decorators.`**`client_required`**`(`*view_func*`)`

Decorator to ensure the user is a client (not a content creator).

This decorator checks if the logged-in user has the *is_content_creator* attribute set to *False*. If the user is a client, the view function is executed. Otherwise, the user is redirected to the home page with an error message.

Parameters
`view_func` (*`function`*) – The view function to be wrapped by this decorator.

Returns
The wrapped view function that includes the client check.

Return type
function

`client.tasks.`**`renew_subscriptions`**`()`

Function to renew subscriptions that are due for renewal. This function checks all active subscriptions that have reached their end date and attempts to renew them if the user has enough points in their wallet. If the user does not have enough points, the subscription is marked as expired.

`class` `client.management.commands.renew_subscriptions.`**`Command`**`(`*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*`)`

Bases: `BaseCommand`

Custom management command to renew subscriptions that are due for renewal.

This command can be run using Django's management command system. It calls the *renew_subscriptions* function to renew any subscriptions that are due for renewal and outputs a success message upon completion.

`handle``(`**args*, ***kwargs*`)`

The entry point for the command. Calls the *renew_subscriptions* function and writes a success message to stdout.

10.3 Creator Utilities

`creator.decorators.`**`creator_required`**`(`*view_func*`)`

Decorator to ensure the user is a content creator.

This decorator checks if the logged-in user has the *is_content_creator* attribute set to *True*. If the user is a content creator, the view function is executed. Otherwise, the user is redirected to the home page with an error message.

Parameters
`view_func` (*`function`*) – The view function to be wrapped by this decorator.

Returns
The wrapped view function that includes the content creator check.

Return type
function

`creator.helpers.`**`get_upload_to`**`(`*instance*, *filename*`)`

Determine the upload path for media files based on their file type.

Parameters: - instance: The model instance this file is being attached to. - filename: The name of the file being uploaded.

Returns: - A string representing the upload path for the file, categorized by its type (images or videos).

10.4 Interactions Utilities

`interactions.helpers.`**`has_messaging_permission`**(*sender*, *recipient*)

Checks if there is messaging permission between the sender and recipient.

There are two conditions under which messaging permission is granted: 1. The recipient has an active subscription to one of the sender's tiers that allows messaging. 2. The sender has an active subscription to one of the recipient's tiers that allows messaging.

Parameters

- **`sender`** (`CustomUser`) – The user sending the message.
- **`recipient`** (`CustomUser`) – The user receiving the message.

Returns

True if there is messaging permission between the sender and recipient, False otherwise.

Return type

bool

CHAPTER ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX