Milestone 4 (Team 3)

Product Summary

Blue Bird is a web platform for Fulda students and faculty to buy, sell or share for free various digital media (images, music, videos, documents), to be used among other things for student projects.

List of final product functions:

- New users should be able to register using their university email addresses.
- Users must log in to the system using their university email.
- Users must be able to upload medias.
- User Users must be able to search for media using simple text and/or tags.
- User must be able to filter media based on media type.
- Users must be able to view the Media Information such as preview, description, uploaded by, upload date, tags, and price.
- User must be able to send a message to the seller of a media.
- Seller must be able to see the messages received and reply back to the buyer.
- Admin must be able to see medias uploaded by a user,
- Every media must be approved by the admin first before going live on the website.
- Admin must be able to get tags for every media through image recognition.
- Admin must be able to disapprove any media at any time if found inappropriate.
- Owner of the media must be able to publish, unpublish or delete uploaded media anytime.
- Owner of the media must be able to update the media prices, name and description
- User must be able to download the free media.
- User must be able to buy the paid media.

Unique Features:

- Extracting tags through the image recognition and displaying it with the media to admin to moderate the inappropriate and explicit content uploaded on website.
- Editor’s choice.

URL of the deployed application: http://192.168.72.72/

Usability Test Plan

Test objectives:

For our website Blue Bird, we have decided to test the functionality of Search along with the filters that we have on the website. Our main concern is to check that if a naive/new user tries to use our application for the first time, how easily/comfortably he/she can use the application without any problems. We want to make sure that the design of the application is user-friendly in every possible way so that they don't need any kind of manual to use our application, even for the very first time. We check for things like: "Will users easily find the search box in its present location?" or "Will the user be able to find the filter functionality and use it easily along with the search option that he has."

Test background and setup

System setup: To perform this test Internet connection and a device with a web browser.

Starting point: In order to conduct usability testing the starting point for a user is to open a browser and enter our website's URL.

Who are the intended users: The intended users of our application are Fulda students and faculty members. To make the usability testing more realistic, we have selected student of HS Fulda who is not part of our project, directly or indirectly.

URL of the system to be tested: http://192.168.72.72/

What is to be measured: We want to measure the how easy to use our website is; we plan to work further if we find some of the things in the Usability testing is not so easy or comfortable for our users to find/navigate through. So, for user Satisfaction evaluation we will be using the Likert Test which includes assessing opinions, attitudes, or behaviours of our users.

Usability Task description: In order to perform the test, the tester first need to navigate to the URL provided above and then they should try to search something by putting text like the name or tag of the media that they are looking for into the search box then they should observe that did they find anything relevant or not and to carry forward this test further they can also filter this result according to media type. Apart from this testing tester should also check this page by loading in various sizes of devices like mobile, tablet, laptop to check how UI is getting rendered on different devices. Finally, they should evaluate the result. To measure effectiveness, first, the tester should check, are results actually according to what they wrote in the search box and if the tester has selected any specific media type, then, is the result filtered according to the given media type?

Questionnaires

Question - 1. How satisfied were you with the UI and Functionality of searching page?

1. Very dissatisfied
2. Somewhat dissatisfied.
3. Neither satisfied nor dissatisfied.
4. Somewhat satisfied.
5. Very satisfied.

Question - 2. How well does this page meet your needs?

1. It did not meet my needs at all.
2. It met very few of my needs.
3. It met some of my needs.
4. It met the majority of my needs.
5. It met all of my needs.

Question - 3. How intuitive did you find using Search Box and Filtering Options?

1. Not intuitive at all.
2. Not very intuitive.
3. Somewhat intuitive.
4. Mostly intuitive.
5. Extremely intuitive.

Tester 1: Pratik Kakadiya (Fellow classmate who is not part of GDSD Project this semester)

Test Results:

Question - 1. How satisfied were you with the UI and Functionality of searching page?

1. Very dissatisfied
2. Somewhat dissatisfied.
3. Neither satisfied nor dissatisfied.

4. Somewhat satisfied.

5. Very satisfied. ✓

Question - 2. How well does this page meet your needs?

1. It did not meet my needs at all.
2. It met very few of my needs.
3. It met some of my needs.
4. It met the majority of my needs. ✓
5. It met all of my needs.

Question - 3. How intuitive did you find using Search Box and Filtering Options?

1. Not intuitive at all.
2. Not very intuitive.
3. Somewhat intuitive.
4. Mostly intuitive.
5. Extremely intuitive. ✓

Tester 2: Mohit Dalal (Fellow classmate who is not part of GDSD Project this semester)

Test Results:

Question - 1. How satisfied were you with the UI and Functionality of searching page?

1. Very dissatisfied
2. Somewhat dissatisfied.
3. Neither satisfied nor dissatisfied.
4. Somewhat satisfied.
5. Very satisfied. ✓

Question - 2. How well does this page meet your needs?

1. It did not meet my needs at all.

2. It met very few of my needs.

3. It met some of my needs.

4. It met the majority of my needs.

5. It met all of my needs. ✓

Question - 3. How intuitive did you find using Search Box and Filtering Options?

1. Not intuitive at all.

2. Not very intuitive.

3. Somewhat intuitive.

4. Mostly intuitive.

5. Extremely intuitive. ✓

Tester 3: Yogeeta Sharama (Fellow classmate who is not part of GDSD Project this semester)

Test Results:

Question - 1. How satisfied were you with the UI and Functionality of searching page?

1. Very dissatisfied

2. Somewhat dissatisfied.

3. Neither satisfied nor dissatisfied.

4. Somewhat satisfied.

5. Very satisfied. ✓

Question - 2. How well does this page meet your needs?

1. It did not meet my needs at all.

2. It met very few of my needs.

3. It met some of my needs.

4. It met the majority of my needs.

5. It met all of my needs. ✓

Question - 3. How intuitive did you find using Search Box and Filtering Options?

1. Not intuitive at all.
2. Not very intuitive.
3. Somewhat intuitive.
4. Mostly intuitive.
5. Extremely intuitive. ✓

QA Test Plan

Test objectives:

The objective is to test the correctness and robustness of the search feature in the Blue Bird. The central idea was to regressively test the search with various filters and text searches from an end-user point of view.

System setup: To perform this test Internet connection and a device with a web browser.

URLs of the system to be tested: http://192.168.72.72/

http://192.168.72.72/myads

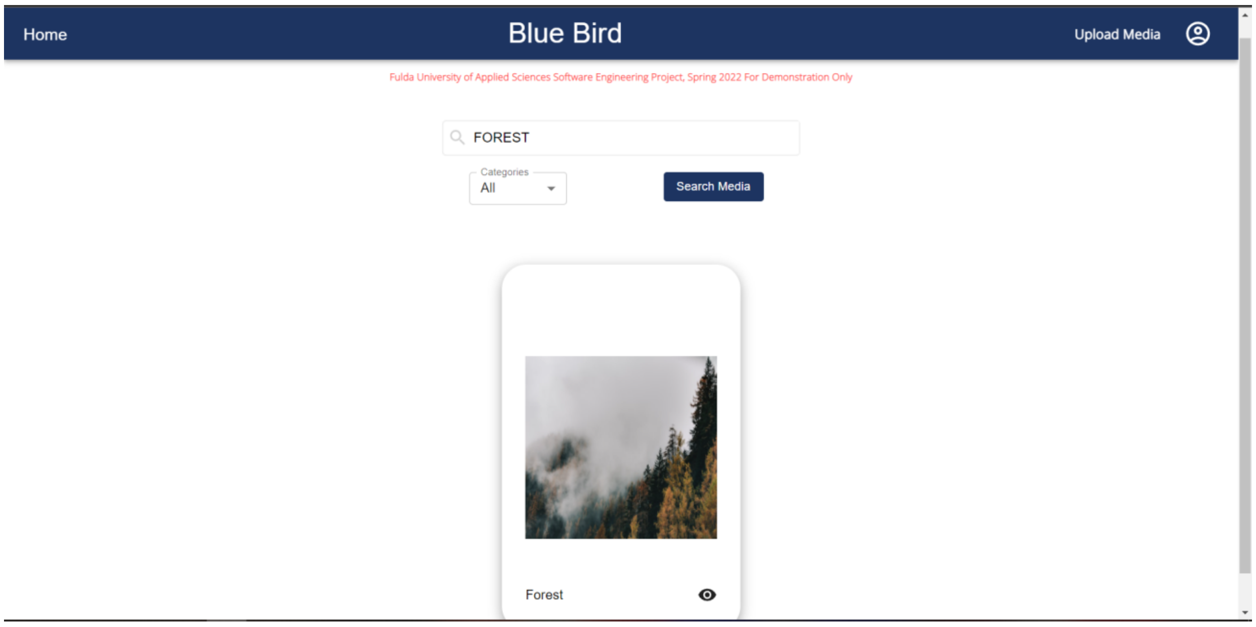
Feature to be tested: The search feature was taken into consideration while applying the various testing methods.

Sr. No.	Test Description	Input	Expected Output	Actual Output	Result	Google Chrome	Mozilla Firefox
1.	Responsiveness ss	Resolution Ipad Air	Screen adapt to size	Screen adapted to size	Passed	✓	✓
2.	Responsiveness ss	Resolution Hp Notebook Full HD(1920*1080)	Screen Adapt to size	Screen adapted to size	Passed	✓	✓
3.	Case sensitive Media Search	FOREST	List the media named Forest	List the media named Forest	Passed	✓	✓
4.	Filter by Category	Category: Image	List all Image medias	List all Image medias	Passed	✓	✓
5.	Media search	Search: Forest	List all	Listed all	Passed	✓	✓

	coupled with Category	Category: Image	Image medias named Forest	Image medias Named Forest			
6.	Media Search by Tag	Search: New	List all medias having “New” Tag	Listed all medias having “New” Tag	Passed	✓	✓
7.	Filter by Category	Category: Approved	List all medias Approved by admin	Listed all medias Approved by admin	Passed	✓	✓
8.	Media search coupled with Category	Search: Audio Category: Disapproved	List all medias not Approved by admin	Listed all medias not Approved by admin	Passed	✓	✓

Result:

All the various test cases drafted were found to be working as expected on the subject of test. Some of the results are formulated below for reference.



Fulda University of Applied Sciences Software Engineering Project, Spring 2022 For Demonstration Only

Figure 1. Case sensitive Media Search

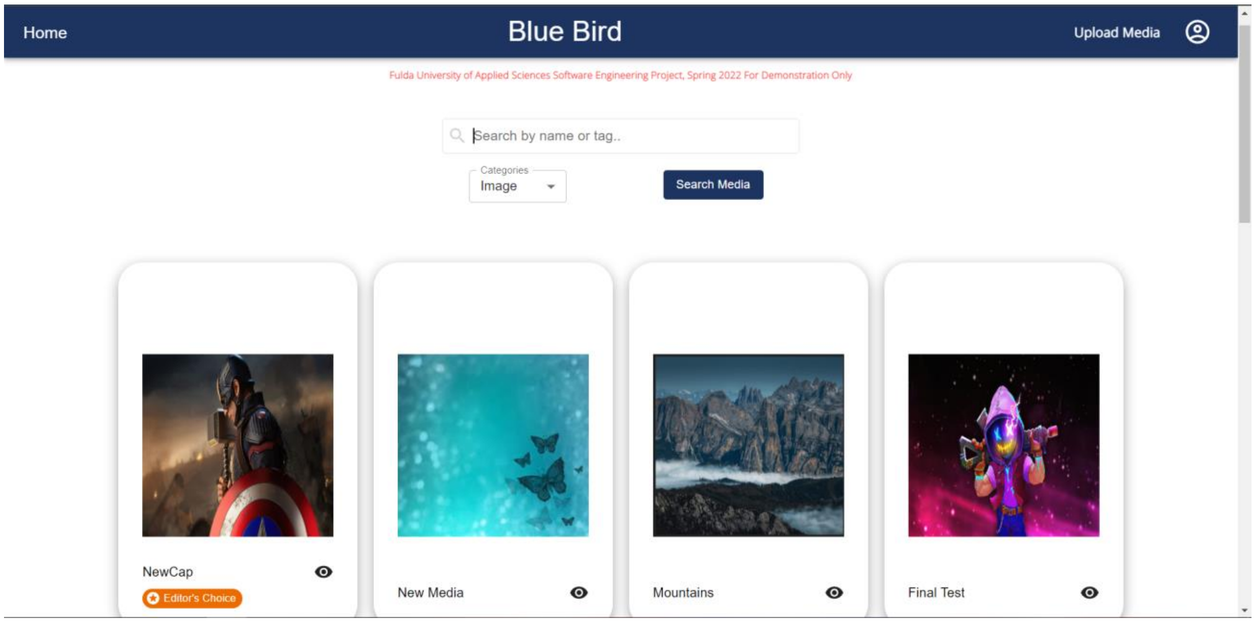


Figure 2. Media Search by Category

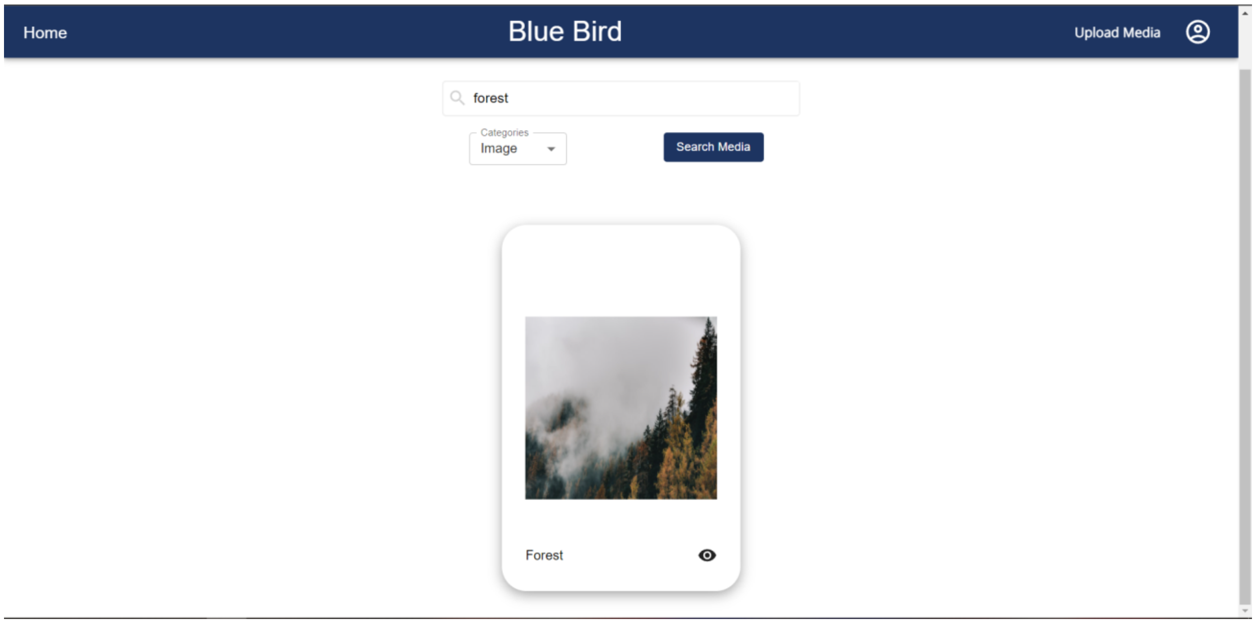


Figure 3. Media Search by text and category both

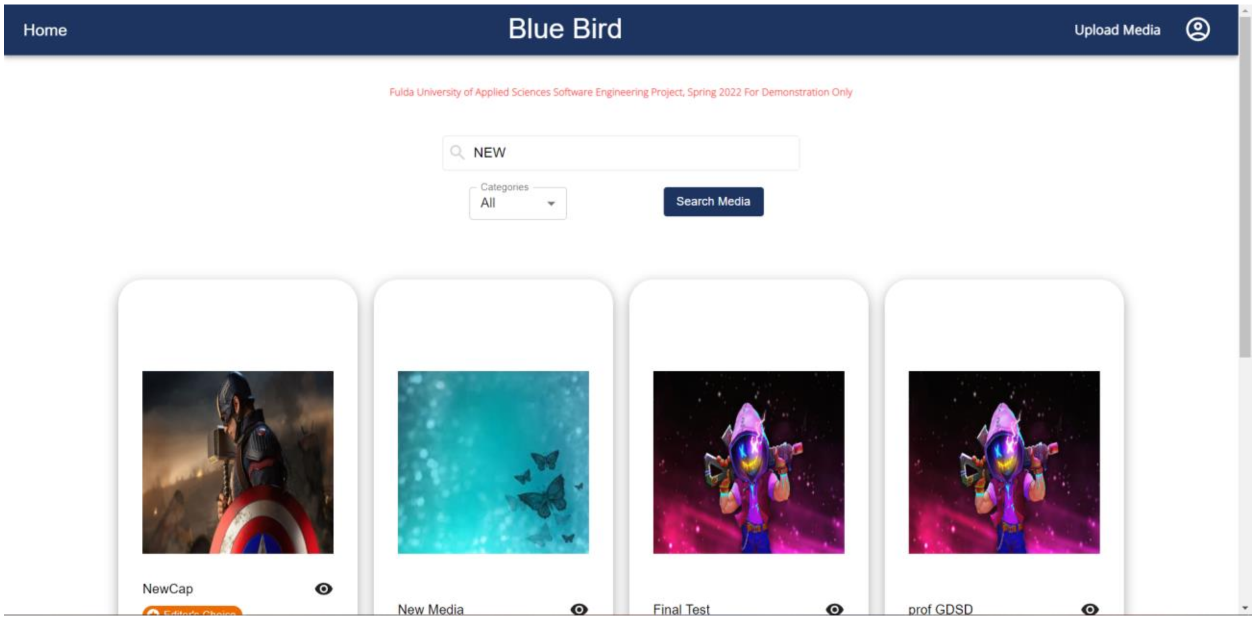

Home
Blue Bird
Upload Media
Fulda University of Applied Sciences Software Engineering Project, Spring 2022 For Demonstration Only
NEW
Categories
All
Search Media
NewCap
New Media
Final Test
prof GDSD

Figure 4. Media Search by Tag

Code Review

Gul Code Review of Hadil

```
# imports can be sorted in something like this
# 1. native imports e-g os, io
# 2. third party modules (django, rest_framework)
# 3. local imports
# Make sure you don't push the unused imports to remote
from cProfile import label
from itertools import permutations
import json
from copy import deepcopy

from rest_framework import status
from rest_framework.decorators import api_view
from rest_framework.permissions import IsAdminUser, IsAuthenticated
from rest_framework.response import Response
from rest_framework import generics

from rest_framework import filters


from django.core.exceptions import ObjectDoesNotExist

from django.db import IntegrityError


from static_content.s3_service import upload_file

# why not to use () for imports longer then 120 characters for better readability

# e-g (

# MediaSerializer, AttachmentSerializer, AttachmentUploadSerializer, OrderSerializer, RatingsSerializer)

# or may be split the dependencies to separate lines


from static_content.serializers.serializers import MediaSerializer, AttachmentSerializer, \

AttachmentUploadSerializer, OrderSerializer, RatingsSerializer

from static_content.rekognition_service import get_labels, get_tags


from static_content.models import Media, Attachment, Order, Ratings

from static_content.filters import MediaFilter

from django_filters.rest_framework import DjangoFilterBackend



# DRF Viewset implementation would help to minimize these two classes (MediaList, MediaDetail) into
one class

# Doc strings for Views and mixins would also contribute in better readability and understanding of code

class MediaList(generics.ListCreateAPIView):

queryset = Media.objects.filter(is_enabled=True, is_approved=True, is_published=True)

serializer_class = MediaSerializer

filter_backends = [DjangoFilterBackend, ]

filterset_class = MediaFilter
```

```
def get_serializer_context(self):
context = super(MediaList, self).get_serializer_context()
context.update({"request": self.request})
return context

def create(self, request):
serializer = MediaSerializer(data=request.data, context=self.get_serializer_context())
serializer.is_valid(raise_exception=True)
media = serializer.save(owner=self.request.user, )
attachments = request.data.get("attachments")
# instead of using multiple return statements why not keep the reference to response and return at end
# unused variable
tags = []
if len(attachments) == 0:
return Response({"error": "attachment field must not be empty"})
for id in attachments:
try:
attachment = Attachment.objects.get(id=id)
if attachment.media is None:
attachment.media = media
attachment.save()
else:
media.delete()
return Response({"error": "attachment with id {id} is already associated with a "
"media object".format(id=id)}, status=status.HTTP_400_BAD_REQUEST)
# except block must only take the targeted statement e.g in this case only Attachment.objects.get(id=id)
except ObjectDoesNotExist:
media.delete()
return Response({"error": "attachment with id {id} does not exist".format(id=id)},
status=status.HTTP_400_BAD_REQUEST)
return Response(serializer.data, status=status.HTTP_201_CREATED)

def list(self, request, *args, **kwargs):
queryset = self.filter_queryset(self.get_queryset())
page = self.paginate_queryset(queryset)
if page is not None:
serializer = self.get_serializer(page, many=True)
return self.get_paginated_response(serializer.data)

search_key = request.query_params.get("search", None)
if search_key:
queryset = search_media(queryset, search_key)
serializer = self.get_serializer(queryset, many=True,
context=self.get_serializer_context())
return Response(serializer.data, status=status.HTTP_200_OK)

serializer = self.get_serializer(queryset, many=True)
return Response(serializer.data, status=status.HTTP_200_OK)

class MediaDetail(generics.RetrieveUpdateDestroyAPIView):
queryset = Media.objects.filter(is_enabled=True)
serializer_class = MediaSerializer

def get_serializer_context(self):
context = super(MediaDetail, self).get_serializer_context()
context.update({"request": self.request})
return context
```

```
def delete(self, request, pk):

media = Media.objects.get(pk=pk)

media.is_enabled = False

media.save()

return Response(status=status.HTTP_204_NO_CONTENT)



class AttachmentCreate(generics.CreateAPIView):

queryset = Attachment.objects.all()

serializer_class = AttachmentSerializer


def create(self, request):

serializer = AttachmentUploadSerializer(data=request.data)

if serializer.is_valid():

file = request.data.get("file")

file_name = request.data.get("name", file.name)

labels = []

# only images are uploaded for fetching the labels

if "image" in file.content_type:

copied_file = deepcopy(file)

try:

labels = get_labels(copied_file.read())

except Exception as e:

print(e)

uri = upload_file(file)

media_id = request.data.get("media")

if media_id:

attachment = Attachment.objects.create(

name=file_name, format=file.content_type.split("/")[1], uri=uri,
media=Media.objects.get(id=media_id), type=file.content_type.split("/")[0],

labels=labels

)

else:

attachment = Attachment.objects.create(

name=file_name,

format=file.content_type.split("/")[1],

uri=uri,

type=file.content_type.split("/")[0],

labels=labels

)

return Response(AttachmentSerializer(attachment).data, status=status.HTTP_201_CREATED)

else:

return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)



class AttachmentDetail(generics.RetrieveUpdateDestroyAPIView):

queryset = Attachment.objects.all()

serializer_class = AttachmentSerializer

def get_serializer_context(self):

context = super(AttachmentDetail, self).get_serializer_context()

context.update({"request": self.request})

return context



class NotApprovedMediaListView(generics.ListCreateAPIView):

permission_classes = IsAdminUser,
serializer_class = MediaSerializer

def get_serializer_context(self):

context = super(NotApprovedMediaListView, self).get_serializer_context()

context.update({"request": self.request})

return context

def get_queryset(self):

is_approved: str = self.request.GET.get("is_approved")

if is_approved:

is_approved = json.loads(is_approved)

return Media.objects.filter(is_approved=is_approved)

else:

return Media.objects.all()

def create(self, request, *args, **kwargs):

media_ids: list = request.data.get("ids")

approve: bool = request.data.get("approve", True)

if not media_ids:

return Response(data={"error": "please provide 'ids' list to approve"})

Media.objects.filter(id__in=media_ids).update(is_approved=approve)

return Response(MediaSerializer(Media.objects.all(), many=True,
context=self.get_serializer_context()).data)

class MyMediasList(generics.ListAPIView):

"""

View for listing medias owned by the currently authenticated user.

"""

queryset = Media.objects.filter(is_enabled=True)
```

```
serializer_class = MediaSerializer

def get_queryset(self):
return Media.objects.filter(owner=self.request.user)


class OrderCreate(generics.CreateAPIView):
"""
View for creating orders.
"""
queryset = Order.objects.all()
serializer_class = OrderSerializer

def create(self, request, pk):
try:
media = Media.objects.get(pk=pk)
buyer = self.request.user
price = media.cost
order = Order.objects.create(media=media, buyer=buyer, price=price)
media.was_bought = media.was_bought + 1
order_serializer = OrderSerializer(order, context={"request": self.request})
media.save()
return Response(order_serializer.data, status=status.HTTP_201_CREATED)
except ObjectDoesNotExist:
return Response({"error": "media with id {pk} does not exist".format(pk=pk)},
status=status.HTTP_400_BAD_REQUEST)
except IntegrityError:
return Response({"error": "order has been already placed"}, status=status.HTTP_201_CREATED)
```

```
class OrderList(generics.ListAPIView):
"""
View for listing existing orders.
"""

def get_serializer_context(self):
context = super(OrderList, self).get_serializer_context()
context.update({"request": self.request})
return context

queryset = Order.objects.all()
serializer_class = OrderSerializer
permission_classes = [IsAdminUser]


class MyOrdersList(generics.ListAPIView):
"""
View for listing orders made by the currently authenticated user.
"""

def get_serializer_context(self):
context = super(MyOrdersList, self).get_serializer_context()
context.update({"request": self.request})
return context

queryset = Order.objects.all()
serializer_class = OrderSerializer
```

```
def get_queryset(self):
return Order.objects.filter(buyer=self.request.user)


# Don't leave such commented code in the code base unless it's going to be reused
# class MediaSearch(generics.ListAPIView):
# """
# View for searching media
# """
# queryset = Media.objects.filter(is_enabled=True, is_approved=True, is_published=True)
# serializer_class = MediaSerializer
#
# def get_queryset(self):
# search_key = self.request.query_params["search"]
# search_words = [word.strip() for word in search_key.split(" ")]
# qs = Media.objects.none()
# qs2 = Media.objects.filter(is_enabled=True)
# for word in search_words:
# qs = qs | qs2.filter(name__icontains=word) | \
# qs2.filter(description__icontains=word) | \
# qs2.filter(tags__name__icontains=word) | \
# qs2.filter(owner__first_name__icontains=word) | \
# qs2.filter(owner__last_name__icontains=word)
#
# return qs


# why not define it inside the View as it's used only once
def search_media(queryset, search_key):
search_words = [word.strip() for word in search_key.split(" ")]
```

```
qs = Media.objects.none()

# qs2 = Media.objects.filter(is_enabled=True)

qs2 = queryset

for word in search_words:

qs = qs | qs2.filter(name__icontains=word) | \

qs2.filter(description__icontains=word) | \

qs2.filter(tags__name__icontains=word) | \

qs2.filter(owner__first_name__icontains=word) | \

qs2.filter(owner__last_name__icontains=word)


return qs
```

Gul Code Review of Trushar

```
# It can be minimized to single class using viewsets by DRF

class RatingsList(generics.ListCreateAPIView):

"""

View for listing and creating ratings.

"""



queryset = Ratings.objects.all()

serializer_class = RatingsSerializer

permission_classes = [IsAuthenticated]



def get_queryset(self):

media = Media.objects.get(pk=self.kwargs['pk'])

return Ratings.objects.filter(media=media)
```

```
# you can also override the save method of RatingsSerializer and do all this business logic there instead of
view

def create(self, request, pk):

try:

media = Media.objects.get(pk=pk)

given_by = self.request.user

stars = request.data.get("stars")

feedback = request.data.get("feedback")

rating = Ratings.objects.create(media=media, given_by=given_by, stars=stars, feedback=feedback)

rating_serializer = RatingsSerializer(rating, context={"request": self.request})

rating.save()

return Response(rating_serializer.data, status=status.HTTP_201_CREATED)

except ObjectDoesNotExist:

return Response({"error": "media with id {pk} does not exist".format(pk=pk)},

status=status.HTTP_400_BAD_REQUEST)



class DeleteRating(generics.DestroyAPIView):

"""

View for deleting ratings.

"""

queryset = Ratings.objects.all()

serializer_class = RatingsSerializer

permission_classes = [IsAuthenticated]



def delete(self, request, pk):

try:
```

```
rating = Ratings.objects.get(pk=pk)

rating.delete()

return Response(status=status.HTTP_204_NO_CONTENT)

except ObjectDoesNotExist:

return Response({"error": "rating with id {pk} does not exist".format(pk=pk)},

status=status.HTTP_400_BAD_REQUEST)
```

Hadil’s Review of Gull’s Code

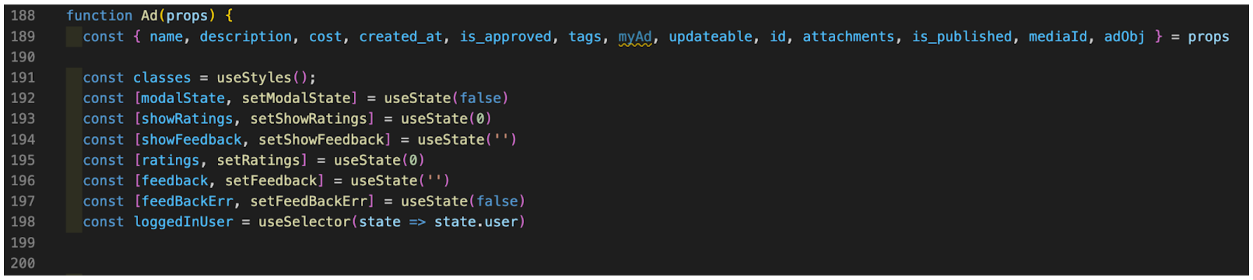
```
# sort imports
import logging
import uuid
import boto3
from django.conf import settings
from botocore.exceptions import ClientError
from PIL import Image
import io


UPLOAD_DIRECTORY = "uploaded_data"


s3_client = boto3.client(
        "s3",
       endpoint_url=settings.AWS_S3_ENDPOINT_URL,
        region_name=settings.AWS_S3_REGION_NAME,
        aws_access_key_id=settings.AWS_ACCESS_KEY,
       aws_secret_access_key=settings.AWS_SECRET_KEY
        )
                              # two new lines are needed here, not one
def upload_file(file_obj):
    """Upload a file to S3 bucket.

    :param file_obj: File Object to upload
    """
    file_ext = file_obj.name.split(".")[-1]
    obj_name = str(uuid.uuid4()) + "." + file_ext

    try:
        s3_client.upload_fileobj(file_obj, settings.AWS_BUCKET_NAME,
f"{UPLOAD_DIRECTORY}/{obj_name}")
```

```
        logging.info("File uploaded with name: {}".format(obj_name), obj_name)

        return obj_name
    except ClientError as e:
        logging.error(e)


def delete_remote_file(uri):
    s3_client = boto3.client("s3", aws_access_key_id=settings.AWS_ACCESS_KEY, # the name could
be changed because it's used above
                            aws_secret_access_key=settings.AWS_SECRET_KEY)
    try:
        s3_client.delete_object(Bucket=settings.AWS_BUCKET_NAME,
Key=f"{UPLOAD_DIRECTORY}/{uri}")
        logging.info("File deleted with name: {}".format(uri))

    except ClientError as e:
        logging.error(e)


def read_image(uri):
    key = f"{UPLOAD_DIRECTORY}/{uri}"
    s3 = boto3.resource('s3')
    image = s3.Object(bucket_name=settings.AWS_BUCKET_NAME, key=key)
    img_data = image.get().get('Body').read()
    return Image.open(io.BytesIO(img_data))


s3_download_client = boto3.client( # this should be defined above on top before all funtions
        "s3",
       endpoint_url=settings.AWS_S3_DOWNLOAD_ENDPOINT_URL,
        region_name=settings.AWS_S3_REGION_NAME,
        aws_access_key_id=settings.AWS_ACCESS_KEY,
       aws_secret_access_key=settings.AWS_SECRET_KEY
        )

def get_public_link(s3_file_name):
    """Getting a publicly accessible link to an asset.

    :param s3_file_name: name of file object which is uploaded before.
    """
    try:
        return s3_download_client.generate_presigned_url("get_object", Params={
            "Bucket": settings.AWS_BUCKET_NAME, "Key":
```

```
f"{UPLOAD_DIRECTORY}/{s3_file_name}"}, ExpiresIn=3600)
    except ClientError as e:
        logging.error(e)
```

Code Review by Deepak:

File Reviewed: MyAds.js
File Owner: Sagar Dhaware

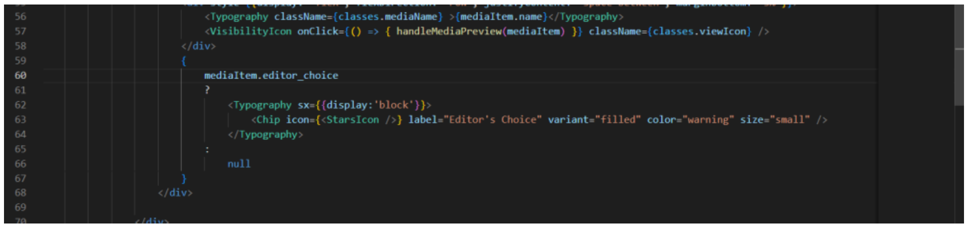


```
188  function Ad(props) {
189    const { name, description, cost, created_at, is_approved, tags, myAd, updateable, id, attachments, is_published, mediaId, adObj } = props
190
191    const classes = useStyles();
192    const [modalState, setModalState] = useState(false)
193    const [showRatings, setShowRatings] = useState(0)
194    const [showFeedback, setShowFeedback] = useState('')
195    const [ratings, setRatings] = useState(0)
196    const [feedback, setFeedback] = useState('')
197    const [feedBackErr, setFeedBackErr] = useState(false)
198    const loggedInUser = useSelector(state => state.user)
199
200
```

On line 189, we should accept all objects in one prop and then destructure them in the component in use.

Code Review by Sagar:

File Reviewed: VerticalPrototype.js
File Owner: Deepak Heman Das

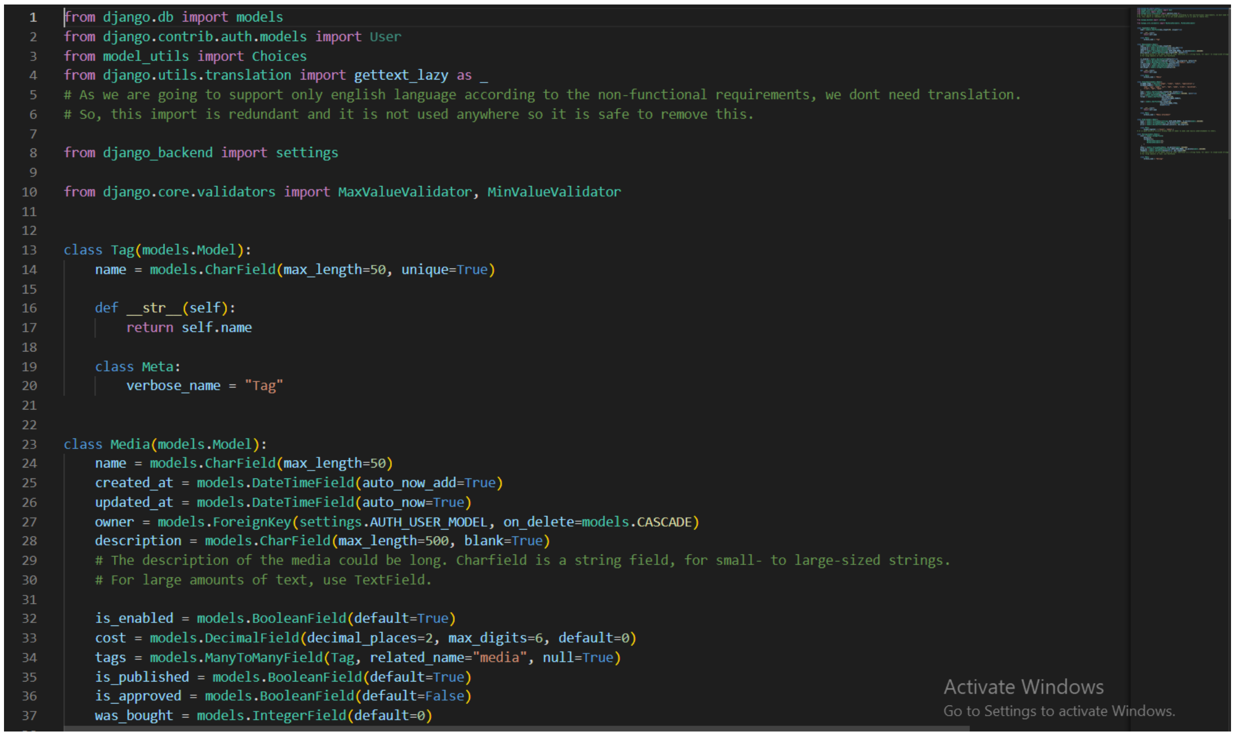


```
56              <Typography className={classes.mediaName} >{mediaItem.name}</Typography>
57              <VisibilityIcon onClick={() => { handleMediaPreview(mediaItem) }} className={classes.viewIcon} />
58            </div>
59            {
60              mediaItem.editor_choice
61              ?
62                <Typography sx={{display:'block'}}>
63                  <Chip icon={<StarsIcon />} label="Editor's Choice" variant="filled" color="warning" size="small" />
64                </Typography>
65              :
66                null
67            }
68          </div>
69
```

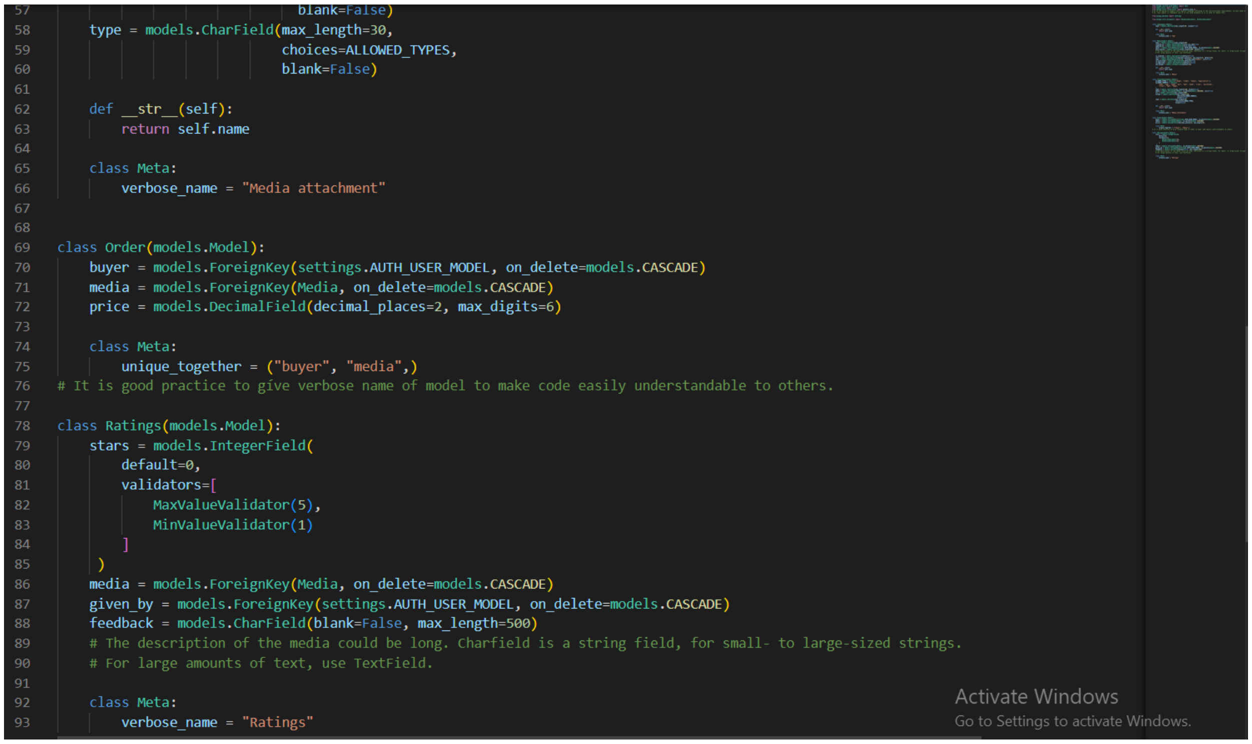
On line 60, instead of having a ternary operator we should just use the && (and) operator, makes the code more readable and clean.

Code Review by Trushar:

File Reviewed: models.py (static_content)
File Owner: Hadil Bader

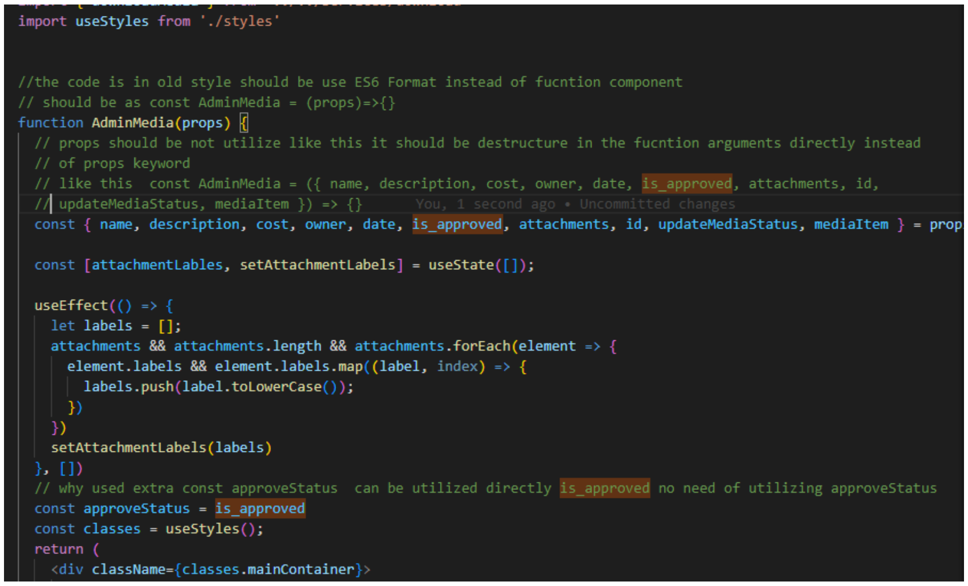


```python
from django.db import models
from django.contrib.auth.models import User
from model_utils import Choices
from django.utils.translation import gettext_lazy as _
# As we are going to support only english language according to the non-functional requirements, we dont need translation.
# So, this import is redundant and it is not used anywhere so it is safe to remove this.

from django_backend import settings

from django.core.validators import MaxValueValidator, MinValueValidator


class Tag(models.Model):
    name = models.CharField(max_length=50, unique=True)

    def __str__(self):
        return self.name

    class Meta:
        verbose_name = "Tag"


class Media(models.Model):
    name = models.CharField(max_length=50)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    owner = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    description = models.CharField(max_length=500, blank=True)
    # The description of the media could be long. Charfield is a string field, for small- to large-sized strings.
    # For large amounts of text, use TextField.

    is_enabled = models.BooleanField(default=True)
    cost = models.DecimalField(decimal_places=2, max_digits=6, default=0)
    tags = models.ManyToManyField(Tag, related_name="media", null=True)
    is_published = models.BooleanField(default=True)
    is_approved = models.BooleanField(default=False)
    was_bought = models.IntegerField(default=0)
```

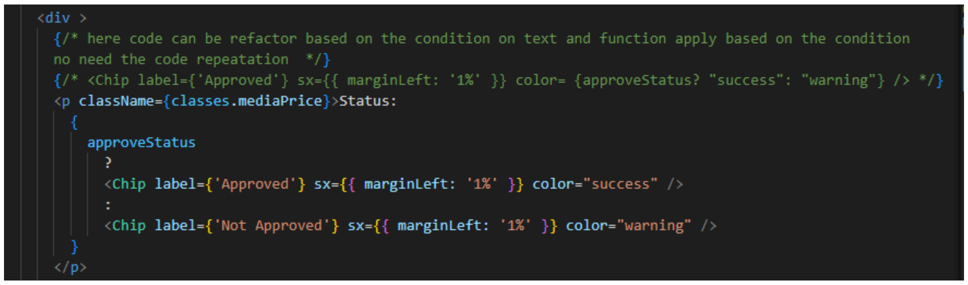


```python
                               blank=False)
    type = models.CharField(max_length=30,
                            choices=ALLOWED_TYPES,
                            blank=False)

    def __str__(self):
        return self.name

    class Meta:
        verbose_name = "Media attachment"


class Order(models.Model):
    buyer = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    media = models.ForeignKey(Media, on_delete=models.CASCADE)
    price = models.DecimalField(decimal_places=2, max_digits=6)

    class Meta:
        unique_together = ("buyer", "media",)
# It is good practice to give verbose name of model to make code easily understandable to others.

class Ratings(models.Model):
    stars = models.IntegerField(
        default=0,
        validators=[
            MaxValueValidator(5),
            MinValueValidator(1)
        ]
     )
    media = models.ForeignKey(Media, on_delete=models.CASCADE)
    given_by = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    feedback = models.CharField(blank=False, max_length=500)
    # The description of the media could be long. Charfield is a string field, for small- to large-sized strings.
    # For large amounts of text, use TextField.

    class Meta:
        verbose_name = "Ratings"
```

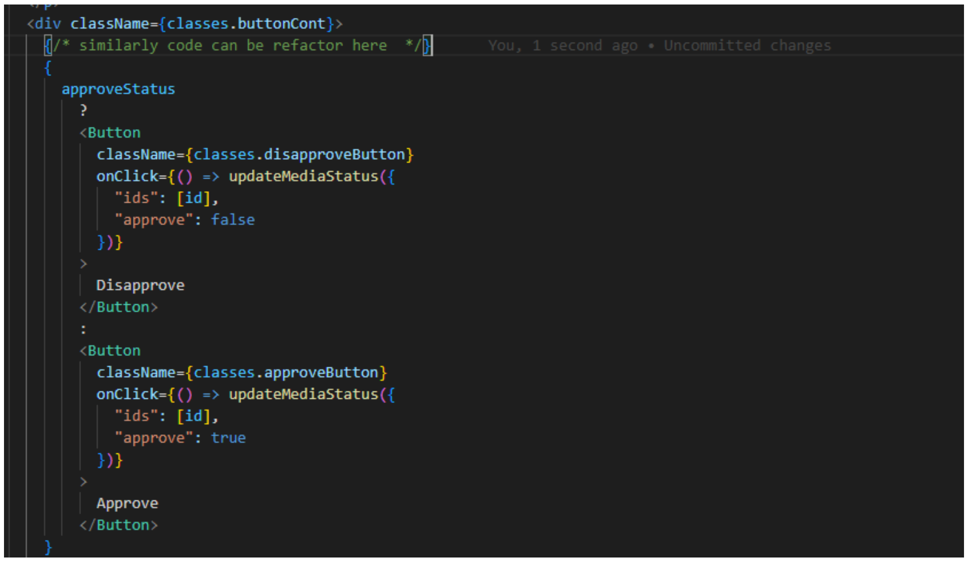
Matee Review Code of Deepak Haman Das



```
import useStyles from './styles'


//the code is in old style should be use ES6 Format instead of fucntion component
// should be as const AdminMedia = (props)=>{}
function AdminMedia(props) {
  // props should be not utilize like this it should be destructure in the fucntion arguments directly instead
  // of props keyword
  // like this  const AdminMedia = ({ name, description, cost, owner, date, is_approved, attachments, id,
  // updateMediaStatus, mediaItem }) => {}       You, 1 second ago • Uncommitted changes
  const { name, description, cost, owner, date, is_approved, attachments, id, updateMediaStatus, mediaItem } = prop

  const [attachmentLables, setAttachmentLabels] = useState([]);

  useEffect(() => {
    let labels = [];
    attachments && attachments.length && attachments.forEach(element => {
      element.labels && element.labels.map((label, index) => {
        labels.push(label.toLowerCase());
      })
    })
    setAttachmentLabels(labels)
  }, [])
  // why used extra const approveStatus  can be utilized directly is_approved no need of utilizing approveStatus
  const approveStatus = is_approved
  const classes = useStyles();
  return (
    <div className={classes.mainContainer}>
```



```
    <div >
      {/* here code can be refactor based on the condition on text and function apply based on the condition
      no need the code repeatation  */}
      {/* <Chip label={'Approved'} sx={{ marginLeft: '1%' }} color= {approveStatus? "success": "warning"} /> */}
      <p className={classes.mediaPrice}>Status:
        {
          approveStatus
            ?
            <Chip label={'Approved'} sx={{ marginLeft: '1%' }} color="success" />
            :
            <Chip label={'Not Approved'} sx={{ marginLeft: '1%' }} color="warning" />
        }
      </p>
```



```
  <div className={classes.buttonCont}>
    {/* similarly code can be refactor here  */}        You, 1 second ago • Uncommitted changes
    {
      approveStatus
        ?
        <Button
          className={classes.disapproveButton}
          onClick={() => updateMediaStatus({
            "ids": [id],
            "approve": false
          })}
        >
          Disapprove
        </Button>
        :
        <Button
          className={classes.approveButton}
          onClick={() => updateMediaStatus({
            "ids": [id],
            "approve": true
          })}
        >
          Approve
        </Button>
    }
```

Self-check on best practices for security

Major Assets to Protect

1. User content
    - Password
2. Media content
    - Media file
3. Secured API Access
4. Secured Infrastructure

Major Threats on Assets and Protection Mechanism

1. User Content

We aren't saving the password of the users as simple plain text. The passwords are encrypted using PBKDF2 algorithm with SHA256 hash.

2. Media content

The media files are saved in encrypted format in our ubuntu instance. Even if someone gets access to the filesystem of instance the files aren't saved in readable format.

3. Secured API Access

We are using JWT for REST APIs authentication. All APIs which aren't needed to be public are secured with role-based security. Admin specific tasks can only be done by admin user only.

4. Secured Infrastructure

We are using the best industry standards to secure our infrastructure. We are using Docker to make it cloud-native application. Moreover, below nginx reverse proxy most of the services aren't accessible outside of the instance and they are getting used inside it e.g websocket, api, database, localstack etc.

we have also validated search bar input for up to 40 alphanumeric characters.

Self-check: Adherence to original Non-functional specs – performed by team leads

1. Application shall be developed, tested and deployed using tools and servers approved by Class CTO and as agreed in Milestone 0. Application delivery shall be from chosen cloud server. **DONE**

2. Application shall be optimized for standard desktop/laptop browsers e.g. must render correctly on the two latest versions of two major browser. **DONE**

3. All or selected application functions must render well on mobile devices. **DONE**

4. Data shall be stored in the database on the team's deployment cloud server. **DONE**

5. Full resolution free media shall be downloadable directly, and full resolution media for selling shall be obtained after contacting the seller/owner. **DONE**

6. No more than 50 concurrent users shall be accessing the application at any time. **DONE**

7. Privacy of users shall be protected, and all privacy policies will be appropriately communicated to the users. **DONE**

8. The language used shall be English (no localization needed). **DONE**

9. Application shall be very easy to use and intuitive. **DONE**

10. Application should follow established architecture patterns. **DONE**

11. Application code and its repository shall be easy to inspect and maintain. **DONE**

12. Google analytics shall be used (optional for Fulda teams). **DONE**

13. No e-mail clients shall be allowed. **DONE**

14. Pay functionality, if any (e.g. paying for goods and services) shall not be implemented nor simulated in UI. **DONE**

15. Site security: basic best practices shall be applied (as covered in the class) for main data items. **DONE**

16. Application shall be media rich (images, video etc.). Media formats shall be standard as used in the market today. **DONE**

17. Modern SE processes and practices shall be used as specified in the class, including collaborative and continuous SW development. **DONE**

18. For code development and management, as well as documentation like formal milestones required in the class, each team shall use their own GitHub to be set-up by class instructors and started by each team during Milestone 0. **DONE**

19. The application UI (WWW and mobile) shall prominently display the following exact text on all pages "Fulda University of Applied Sciences Software Engineering Project, Spring 2022 For Demonstration Only" at the top of the WWW page. (Important to not confuse this with a real application). **DONE**