You:

I need a view that staff can call on that will say names of users with lots. Does that already exist: import os, io, torch, json from PIL import Image, ImageDraw, ImageFont import torchvision.transforms as transforms from torch import nn, optim from django.http import FileResponse, JsonResponse from django.views.generic import ListView from rest_framework.views import APIView from rest_framework import status, permissions from rest_framework.response import Response from rest_framework.permissions import AllowAny from rest_framework.authentication import SessionAuthentication, BasicAuthentication from django.core.files.storage import default_storage from django.conf import settings from .serializers import CamImageSerializer, LicensePlateReadingSerializer, LPRMetadataNoPasscodeSerializer from .models import CamImage, LotMetadata, CamMetadata, LPRMetadata, LicensePlateReading from django.utils import timezone from datetime import datetime MAX_FOLDER_MB = 1950 def get_mb_folder(camera_name): if os.path.exists(camera_name): return int(os.popen(f"du -sm {camera_name} | awk '{{print $1}}'").read()) # This can limit folder size by image counts instead of folder MB if you choose, this is otherwise not used def get_file_count_folder(camera_name): if os.path.exists(camera_name): files = os.listdir(camera_name) return len(files) def get_oldest_image_filename(camera_name): oldest_file = None oldest_datestamp = datetime.now() if os.path.exists(camera_name): for filename in os.listdir(camera_name): if filename.endswith('.jpg'): # Adjust the file extension as per your filename format date_code = filename.split("_")[-1].split(".")[0] file_datestamp = datetime.strptime(date_code, '%Y%m%d%H%M') if file_datestamp < oldest_datestamp: oldest_datestamp = file_datestamp oldest_file = filename return oldest_file def delete_file_and_lot_image(filename): if os.path.exists(filename): os.remove(filename) try: lot_image = CamImage.objects.get(image__icontains=os.path.basename(filename)) lot_image.delete() print(f'~Successfully deleted {filename}') except CamImage.DoesNotExist: pass # CNN model good at determining if car in spot, from notebook, will separate to another file eventually for organization class CNN(nn.Module): def __init__(self): super(CNN, self).__init__() # Convolutional layer 1 self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1) self.bn1 = nn.BatchNorm2d(64) self.relu1 = nn.ReLU() # Convolutional layer 2 self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1) self.bn2 = nn.BatchNorm2d(128) self.relu2 = nn.ReLU() # Convolutional layer 3 self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1) self.bn3 = nn.BatchNorm2d(256) self.relu3 = nn.ReLU() # Convolutional layer 4 self.conv4 = nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1) self.bn4 = nn.BatchNorm2d(512) self.relu4 = nn.ReLU() # Convolutional layer 5 self.conv5 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1) self.bn5 = nn.BatchNorm2d(512) self.relu5 = nn.ReLU() # Max pool layer self.pool = nn.MaxPool2d(kernel_size=2) # Dropout layer self.dropout = nn.Dropout(p=0.5) # Fully connected layers self.fc1 = nn.Linear(512 * 8 * 8, 1024) self.fc2 = nn.Linear(1024, 512) self.fc3 = nn.Linear(512, 2) def forward(self, x): # Convolutional layer 1 out = self.conv1(x) out = self.bn1(out) out = self.relu1(out) out = self.pool(out) # Convolutional layer 2 out = self.conv2(out) out = self.bn2(out) out = self.relu2(out) out = self.pool(out) # Convolutional layer 3 out = self.conv3(out) out = self.bn3(out) out = self.relu3(out) out = self.pool(out) # Convolutional layer 4 out = self.conv4(out) out = self.bn4(out) out = self.relu4(out) out = self.pool(out) # Convolutional layer 5 out = self.conv5(out) out = self.bn5(out) out = self.relu5(out) out = self.pool(out) # Flatten for fully connected layer out = out.view(out.size(0), -1) # Fully connected layer 1 out = self.fc1(out) out = self.dropout(out) # Fully connected layer 2 out = self.fc2(out) out = self.dropout(out) # Fully connected layer 3 out = self.fc3(out) return out # Originally in Model_Maker notebook, this preps cropped parking spaces for ML processing transform = transforms.Compose([transforms.Resize((256, 256)), # Resize to 256x256 transforms.ToTensor(), # Convert to PyTorch tensor transforms.Normalize((0.5,0.5,0.5,), (0.5,0.5,0.5,)) # Normalize pixel values in the range [-1, 1]]) class ImageUploadView(APIView): authentication_classes = [SessionAuthentication, BasicAuthentication] permission_classes = [AllowAny] def post(self, request, format=None): # Very basic authentication passcode = request.data.get('passcode') if passcode != 'lightsecurity': return Response({'detail': 'Invalid passcode'}, status=status.HTTP_401_UNAUTHORIZED) uploaded_file = request.FILES['image'] # Convert django.core.files.uploadedfile.InMemoryUploadedFile to a cv2 image for ML processing pil_image = Image.open(uploaded_file) filename = uploaded_file.name camera_name, date_code = os.path.splitext(filename)[0].split("_") # Check if an image with the same filename already exists try: lot_image = CamImage.objects.get(image__icontains=filename) # Delete the old file before saving the new one lot_image.image.delete() except CamImage.DoesNotExist: lot_image = CamImage() # Save the new image lot_image.image = uploaded_file lot_image.camera_name = camera_name save_folder = os.path.abspath('./camfeeds/' + camera_name) # Load data from spots.json spots_file_path = os.path.join('models', camera_name, 'spots.json') with open(spots_file_path, 'r') as spots_file: spots_data = json.load(spots_file) labels = {key: False for key in spots_data.keys()} # Get the keys from spots.json and set them in human_labels and model_labels for spot in spots_data.keys(): x, x_w, y, y_h = spots_data[spot] cropped_image = pil_image.crop((x, y, x_w, y_h)) #convert cropped image of spot to form usable by ML model using transform defined above input_tensor = transform(cropped_image) input_tensor = input_tensor.unsqueeze(0) # Add a batch dimension model = CNN() # Replace YourModelClass with the actual class name of your model model_path = os.path.join('models', camera_name, spot + '.pth') #Code for development env # model_state_dict = torch.load(model_path, map_location=torch.device('cpu')) # model.load_state_dict(model_state_dict) #Code for production env model.load_state_dict(torch.load(model_path)) model.eval() # Set the model to evaluation mode with torch.no_grad(): output = model(input_tensor) _, predicted = torch.max(output, 1) # Access the prediction result prediction = predicted.item() if prediction == 0: labels[spot] = True lot_image.human_labels = json.dumps(labels) lot_image.model_labels = json.dumps(labels) lot_image.save() print(f'Image Count: {get_file_count_folder(save_folder)} | Folder MB: {get_mb_folder(save_folder)} | Oldest image: {get_oldest_image_filename(save_folder)}') while (get_mb_folder(save_folder) > MAX_FOLDER_MB): delete_file_and_lot_image(get_oldest_image_filename(save_folder)) return Response({'detail': 'Image successfully stored.'}, status=status.HTTP_201_CREATED) class LatestImageView(APIView): permission_classes = [AllowAny] def get(self, request, format=None): lot_name = request.GET.get('lot') if not lot_name: return Response({'detail': 'Lot or image not specified.'}, status=status.HTTP_400_BAD_REQUEST) try: lot = LotMetadata.objects.get(id=lot_name) cameras = CamMetadata.objects.filter(lot=lot) except LotMetadata.DoesNotExist: return Response({'detail': 'No such lot found.'}, status=status.HTTP_404_NOT_FOUND) camera_names = [camera.name for camera in cameras] try: lot_image = CamImage.objects.filter(camera_name=camera_names[0]).latest('timestamp') except CamImage.DoesNotExist: return Response({'detail': 'No images found for this camera.'}, status=status.HTTP_404_NOT_FOUND) # Get the URL of the image file image_url = default_storage.url(lot_image.image.name) try: previous_image = CamImage.objects.filter(camera_name=camera_names[0], timestamp__lt=lot_image.timestamp).latest('timestamp') previous_image_name_part = previous_image.image.name.split('_')[-1].replace('.jpg', '') except CamImage.DoesNotExist: # If there is no previous image, use the current image name part previous_image_name_part = lot_image.image.name.split('_')[-1].replace('.jpg', '') spots_path = os.path.join('models', camera_names[0], 'spots_view.json') bestspots_path = os.path.join('models', camera_names[0], 'bestspots.json') # Load the contents of the JSON files with open(spots_path, 'r') as spots_file: spots_data = json.load(spots_file) with open(bestspots_path, 'r') as bestspots_file: bestspots_data = json.load(bestspots_file) human_labels = json.loads(lot_image.human_labels) model_labels = json.loads(lot_image.model_labels) # Construct the response data response_data = { 'image_url': image_url, 'timestamp': lot_image.timestamp, 'human_labels': human_labels, 'model_labels': model_labels, 'previous_image_name_part': previous_image_name_part, 'spots': spots_data, 'bestspots': bestspots_data, } return Response(response_data) class SpecificImageView(APIView): permission_classes = [AllowAny] def get(self, request, format=None): lot_name = request.GET.get('lot') image_name_part = request.GET.get('image') if not lot_name or not image_name_part: return Response({'detail': 'Lot or image not specified.'}, status=status.HTTP_400_BAD_REQUEST) try: lot = LotMetadata.objects.get(id=lot_name) cameras = CamMetadata.objects.filter(lot=lot) except LotMetadata.DoesNotExist: return Response({'detail': 'No such lot found.'}, status=status.HTTP_404_NOT_FOUND) camera_names = [camera.name for camera in cameras] image_name = f"camfeeds/{camera_names[0]}/{camera_names[0]}_{image_name_part}.jpg" try: lot_image = CamImage.objects.get(image__icontains=image_name) except CamImage.DoesNotExist: return Response({'detail': 'No images found for this camera.'}, status=status.HTTP_404_NOT_FOUND) # Get the URL of the image file image_url = default_storage.url(lot_image.image.name) # Find the previous and next images by timestamp previous_image = CamImage.objects.filter(camera_name=camera_names[0], timestamp__lt=lot_image.timestamp).order_by('-timestamp').first() next_image = CamImage.objects.filter(camera_name=camera_names[0], timestamp__gt=lot_image.timestamp).order_by('timestamp').first() # Extract the image name part from the previous and next image names previous_image_name_part = previous_image.image.name.split('_')[-1].split('.')[0] if previous_image else image_name_part next_image_name_part = next_image.image.name.split('_')[-1].split('.')[0] if next_image else image_name_part spots_path = os.path.join('models', camera_names[0], 'spots_view.json') bestspots_path = os.path.join('models', camera_names[0], 'bestspots.json') # Load the contents of the JSON files with open(spots_path, 'r') as spots_file: spots_data = json.load(spots_file) with open(bestspots_path, 'r') as bestspots_file: bestspots_data = json.load(bestspots_file)

human_labels = json.loads(lot_image.human_labels) model_labels = json.loads(lot_image.model_labels) # Construct the response data response_data = { 'image_url': image_url, 'timestamp': lot_image.timestamp, 'human_labels': human_labels, 'model_labels': model_labels, 'previous_image_name_part': previous_image_name_part, 'next_image_name_part': next_image_name_part, 'spots': spots_data, 'bestspots': bestspots_data, } return Response(response_data) class LotMenuView(ListView): model = LotMetadata def get_queryset(self): queryset = super().get_queryset() return queryset.values('id', 'name', 'gps_coordinates', 'state', 'zip', 'city') def render_to_response(self, context, **response_kwargs): # We override this method to change the output format to JSON. return JsonResponse(list(context['object_list']), safe=False) class LatestJPGImageFileView(APIView): permission_classes = [AllowAny] def get(self, request, format=None): camera_name = request.GET.get('camera') if not camera_name: return Response({'detail': 'Camera not specified.'}, status=status.HTTP_400_BAD_REQUEST) try: # Filter by '.jpg' extension lot_image = CamImage.objects.filter(camera_name=camera_name, image__endswith='.jpg').latest('timestamp') except CamImage.DoesNotExist: return Response({'detail': 'No JPG images found for this camera.'}, status=status.HTTP_404_NOT_FOUND) # Get the path of the image file image_path = os.path.join(settings.MEDIA_ROOT, lot_image.image.name) # Open the image file and create an Image object image = Image.open(image_path) human_labels = json.loads(lot_image.human_labels) spots_path = os.path.join('models', camera_name, 'spots_view.json') with open(spots_path, 'r') as spots_file: spots_data_view = json.load(spots_file) # Resize the image base_width = 900 w_percent = (base_width / float(image.size[0])) h_size = int((float(image.size[1]) * float(w_percent))) image = image.resize((base_width, h_size), Image.LANCZOS) # Create a draw object draw = ImageDraw.Draw(image) # Define the text and position text = lot_image.timestamp.strftime("%I:%M%p %-m/%-d/%Y").lower().strip() print('Chomp: ' + text) text_position = (image.width - 450, image.height - 50) # Change the position as needed # Define the font (change the font file and size as needed) font = ImageFont.truetype("/usr/share/fonts/truetype/dejavu/DejaVuSans.ttf", 30) # Draw the text on the image draw.text(text_position, text, font=font) # Draw a rectangle for each spot in the spots_data_view for spot, coordinates in reversed(list(spots_data_view.items())): x1, y1, x2, y2 = coordinates correct_coordinates = [x1, x2, y1, y2] correct_coordinates = [x1 * w_percent, x2 * w_percent, y1 * w_percent, y2 * w_percent] # Swap y1 and y2 and scale coordinates # Choose the color of the rectangle based on the value in human_labels color = 'red' if human_labels.get(spot, False) else 'green' draw.rectangle(correct_coordinates, outline=color, width=5) # Save the image to a BytesIO object byte_arr = io.BytesIO() image.save(byte_arr, format='JPEG') byte_arr.seek(0) # seek back to the start after saving # Create a response response = FileResponse(byte_arr, content_type='image/jpeg') # Add anti-caching headers response['Cache-Control'] = 'no-store, no-cache, must-revalidate, max-age=0' response['Pragma'] = 'no-cache' response['Expires'] = '0' # Return the image data as a response return response class LotOwnerDashboardView(APIView): permission_classes = [permissions.IsAuthenticated] def get(self, request, format=None): user = self.request.user allowed_roles = ['Lot Operator', 'Customer Support', 'Lot Specialist', 'Accountant'] if user.role.role_name not in allowed_roles: return Response({"message": "Unauthorized."}, status=status.HTTP_403_FORBIDDEN) # Retrieve the lots associated with the user lots = [] for x in LotMetadata.objects.all(): if str(x.owner) == request.user.email: lots.append(x) lot_cams = {} for lot in lots: cameras = CamMetadata.objects.filter(lot=lot) lot_cams[str(lot)] = cameras camera_names = [camera.name for camera in lot_cams[str(lots[0])]] try: lot_image = CamImage.objects.filter(camera_name=camera_names[0]).latest('timestamp') except CamImage.DoesNotExist: return Response({'detail': 'No images found for this camera.'}, status=status.HTTP_404_NOT_FOUND) # Get the URL of the image file image_url = default_storage.url(lot_image.image.name) try: previous_image = CamImage.objects.filter(camera_name=camera_names[0], timestamp__lt=lot_image.timestamp).latest('timestamp') previous_image_name_part = previous_image.image.name.split('_')[-1].replace('.jpg', '') except CamImage.DoesNotExist: # If there is no previous image, use the current image name part previous_image_name_part = lot_image.image.name.split('_')[-1].replace('.jpg', '') spots_path = os.path.join('models', camera_names[0], 'spots_view.json') bestspots_path = os.path.join('models', camera_names[0], 'bestspots.json') # Load the contents of the JSON files with open(spots_path, 'r') as spots_file: spots_data = json.load(spots_file) with open(bestspots_path, 'r') as bestspots_file: bestspots_data = json.load(bestspots_file) human_labels = json.loads(lot_image.human_labels) model_labels = json.loads(lot_image.model_labels) # Get the LPRMetadata for the lots lpr_metadata_list = LPRMetadata.objects.filter(lot__in=lots) lpr_serializer = LPRMetadataNoPasscodeSerializer(lpr_metadata_list, many=True) # Construct the response data response_data = { 'image_url': image_url, 'timestamp': lot_image.timestamp, 'human_labels': human_labels, 'model_labels': model_labels, 'previous_image_name_part': previous_image_name_part, 'spots': spots_data, 'bestspots': bestspots_data, 'lpr_metadata': lpr_serializer.data # Add the serialized LPRMetadata to the response } return Response(response_data) class GetLotHistory(APIView): permission_classes = [permissions.IsAuthenticated] def get(self, request, format=None): user = self.request.user role_name = user.role.role_name if role_name != 'Lot Operator': return Response({"message": "Unauthorized."}, status=status.HTTP_403_FORBIDDEN) # Retrieve the lots associated with the user lots = LotMetadata.objects.filter(owner=user) cam_names = CamMetadata.objects.filter(lot__in=lots).values_list('name', flat=True) cam_images = CamImage.objects.filter(camera_name__in=cam_names) # Serialize the cam_images using CamImageSerializer serializer = CamImageSerializer(cam_images, many=True) response_data = { 'image_data': serializer.data } return Response(response_data) class OverparkingConfirm(APIView): def get(self, request, lot, cam, spot, startdatetime, enddatetime, format=None): # Convert startdatetime and enddatetime from string to datetime naive_startdatetime = datetime.strptime(startdatetime, '%Y%m%d%H%M') naive_enddatetime = datetime.strptime(enddatetime, '%Y%m%d%H%M') # Make the datetime objects timezone aware startdatetime = timezone.make_aware(naive_startdatetime) enddatetime = timezone.make_aware(naive_enddatetime) # Query the CamImage model to get all instances that meet the conditions cam_images = CamImage.objects.filter(timestamp__range=(startdatetime, enddatetime), camera_name=cam) spots_file_path = os.path.join('models', cam, 'spots_view.json') with open(spots_file_path, 'r') as spots_file: spots_data = json.load(spots_file) # Serialize the cam_images queryset to JSON serializer = CamImageSerializer(cam_images, many=True) # Include the serialized cam_images in the response response_data = { 'crop': spots_data[spot], 'cam_images': serializer.data } return Response(response_data) class GetArchiveView(APIView): permission_classes = [permissions.IsAuthenticated] def get(self, request, format=None): user = self.request.user user_email = str(user) lots = LotMetadata.objects.filter(owner__email=user_email) cams = CamMetadata.objects.filter(lot=lots[0]) try: image = CamImage.objects.filter(camera_name=cams[0]).latest('timestamp') except CamImage.DoesNotExist: return Response({'detail': 'No images found for this camera.'}, status=status.HTTP_404_NOT_FOUND) # Separate date string from full path of latest image full_image_path = str(image.image) parts = full_image_path.split("_") last_part = parts[-1] image_date = last_part.split(".")[0] response_data = { 'email': user_email, 'lot': lots[0].id, 'cam': cams[0].name, 'image': image_date } return Response(response_data) class LicensePlateReadingView(APIView): permission_classes = [AllowAny] def post(self, request, format=None): lpr_name = request.data.get('lpr') try: lpr_metadata = LPRMetadata.objects.get(name=lpr_name) except LPRMetadata.DoesNotExist: return Response({"detail": "Invalid LPR device."}, status=status.HTTP_400_BAD_REQUEST) if lpr_metadata.passcode != request.data.get('passcode'): return Response({"detail": "Incorrect passcode."}, status=status.HTTP_403_FORBIDDEN) serializer = LicensePlateReadingSerializer(data=request.data) if serializer.is_valid(): serializer.save() return Response(serializer.data, status=status.HTTP_201_CREATED) return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST) class RecentLicensePlateReadingsView(APIView): permission_classes = [permissions.IsAuthenticated] def get(self, request, lpr_name, format=None): # Check user's role user = self.request.user allowed_roles = ['Lot Operator', 'Customer Support', 'Lot Specialist', 'Accountant'] if user.role.role_name not in allowed_roles: return Response({"message": "Unauthorized."}, status=status.HTTP_403_FORBIDDEN) # Query the LicensePlateReading model for the given LPRMetadata readings = LicensePlateReading.objects.filter(lpr__name=lpr_name).order_by('-timestamp')[:10] print(readings) # Serialize the readings serializer = LicensePlateReadingSerializer(readings, many=True) return Response(serializer.data, status=status.HTTP_200_OK) class MonthlyLicensePlateReadingsView(APIView): permission_classes = [permissions.IsAuthenticated] def get(self, request, lot_name, year, month, format=None): # Check user's role user = self.request.user allowed_roles = ['Lot Operator', 'Customer Support', 'Lot Specialist', 'Accountant'] if user.role.role_name not in allowed_roles: return Response({"message": "Unauthorized."}, status=status.HTTP_403_FORBIDDEN) # Define the start and end date for the given month start_date = datetime(year, month, 1) print('start date ' + str(start_date)) if month == 12: end_date = datetime(year+1, 1, 1) else: end_date = datetime(year, month+1, 1) print('end date ' + str(end_date)) # Query the LicensePlateReading model for the given month, year, and LPRMetadata readings = LicensePlateReading.objects.filter(lpr=lot_name, timestamp__gte=start_date, timestamp__lt=end_date).order_by('-timestamp') print(readings) # Serialize the readings serializer = LicensePlateReadingSerializer(readings, many=True) print(serializer.data) return Response(serializer.data, status=status.HTTP_200_OK)

ChatGPT:

Based on the provided code, there is no view that directly returns the names of users with lots.

However, the closest thing to a view that provides details about lots is `LotMenuView`. This view returns lot details as a list of JSON objects, where each object contains the `id`, `name`, `gps_coordinates`, `state`, `zip`, and `city` of a lot.

If you want a view that returns the names of users with lots, you can create a new view based on `LotMenuView`. You can modify the `get_queryset` method to return lot owners instead of lot details. This would be achieved by querying the `LotMetadata` model for distinct owners and then returning their names.

Here's an example of what such a view might look like:

```
python
class LotOwnersView(ListView):
    model = LotMetadata

    def get_queryset(self):
        # Get distinct owners of lots
        owners = LotMetadata.objects.values_list('owner__username', flat=True).distinct()
        return owners

    def render_to_response(self, context, **response_kwargs):
        # We override this method to change the output format to JSON.
        return JsonResponse(list(context['object_list']), safe=False)
```

With this view, when you call it, it should return a list of usernames of users who own lots.

You:

Can you make a view that returns all users in the system with lots associated with them? The initial permissions and roles should look like this: permission_classes = [permissions.IsAuthenticated] def get(self, request, format=None): user = self.request.user allowed_roles = ['Lot Operator', 'Customer Support', 'Lot Specialist', 'Accountant'] if user.role.role_name not in allowed_roles: return Response({"message": "Unauthorized."}, status=status.HTTP_403_FORBIDDEN) For reference here is the lots model.py: from django.db import models from django.utils import timezone from django.utils.dateformat import format as dateformat from django.core.files.storage import default_storage from accounts.models import CustomUser def image_upload_path(instance, filename): return f'camfeeds/{instance.camera_name}/{filename}' class CamImage(models.Model): image = models.ImageField(upload_to=image_upload_path) timestamp = models.DateTimeField() camera_name = models.CharField(max_length=255) human_labels = models.TextField(blank=True, null=True) model_labels = models.TextField(blank=True, null=True) def save(self, *args, **kwargs): if not self.timestamp: filename = self.image.name date_code = filename.split("_")[-1].split(".")[0] naive_datetime = timezone.datetime.strptime(date_code, '%Y%m%d%H%M') self.timestamp = timezone.make_aware(naive_datetime) super().save(*args, **kwargs) def __str__(self): return dateformat(self.timestamp, 'm-d-y H:i') def delete(self, using=None, keep_parents=False): # Delete the old file before saving the new one default_storage.delete(self.image.name) super().delete(using=using, keep_parents=keep_parents) class LotMetadata(models.Model): id = models.CharField(max_length=100, primary_key=True) name = models.CharField(max_length=255) owner = models.ForeignKey(CustomUser, on_delete=models.SET_NULL, null=True, blank=True) gps_coordinates = models.CharField(max_length=255, null=True, blank=True) state = models.CharField(max_length=2, null=True, blank=True) zip = models.CharField(max_length=5, null=True, blank=True) city = models.CharField(max_length=255, null=True, blank=True) def __str__(self): return self.name class CamMetadata(models.Model): name = models.CharField(max_length=255, primary_key=True) lot = models.ForeignKey(LotMetadata, on_delete=models.CASCADE) def __str__(self): return self.name class LPRMetadata(models.Model): name = models.CharField(max_length=255, primary_key=True) lot = models.ForeignKey(LotMetadata, on_delete=models.CASCADE) passcode = models.CharField(max_length=255) # You can adjust max_length and other parameters as needed def __str__(self): return self.name class LicensePlateReading(models.Model): lpr = models.ForeignKey(LPRMetadata, on_delete=models.CASCADE) timestamp = models.DateTimeField(auto_now_add=True) plate_number = models.CharField(max_length=10) # Adjust max_length as per your region's plate format def __str__(self): return f"{self.plate_number} at {self.timestamp}" Accounts model.py: from django.db import models from django.contrib.auth.models import AbstractUser from django.core.validators import MinLengthValidator class Role(models.Model): role_name = models.CharField(max_length=50) is_employee = models.BooleanField() def __str__(self): return self.role_name class CustomUser(AbstractUser): role = models.ForeignKey(Role, on_delete=models.SET_NULL, null=True) email = models.EmailField(unique=True) first_name = models.CharField(max_length=255, null=True, blank=True) last_name = models.CharField(max_length=255, null=True, blank=True) company_name = models.CharField(max_length=255, null=True, blank=True) company_address = models.CharField(max_length=255, null=True, blank=True) state = models.CharField(max_length=2, validators=[MinLengthValidator(2)], null=True, blank=True) city = models.CharField(max_length=255, null=True, blank=True) zip = models.CharField(max_length=5, validators=[MinLengthValidator(5)], null=True, blank=True) is_uninitialized = models.BooleanField(default=False) USERNAME_FIELD = 'email' REQUIRED_FIELDS = ['username', 'first_name', 'last_name'] def __str__(self): return self.email Lots serializer: from rest_framework import serializers from .models import CamImage, LicensePlateReading, LPRMetadata class CamImageSerializer(serializers.ModelSerializer): class Meta: model = CamImage fields = ('image', 'timestamp', 'camera_name', 'human_labels', 'model_labels') class LicensePlateReadingSerializer(serializers.ModelSerializer): class Meta: model = LicensePlateReading fields = ('lpr', 'plate_number','timestamp') class LPRMetadataNoPasscodeSerializer(serializers.ModelSerializer): class Meta: model = LPRMetadata exclude = ('passcode',) # class LicensePlateReading(models.Model): # lpr = models.ForeignKey(LPRMetadata, on_delete=models.CASCADE) # timestamp = models.DateTimeField(auto_now_add=True) # plate_number = models.CharField(max_length=10) # Adjust max_length as per your region's plate format # def __str__(self): # return f"{self.plate_number} at {self.timestamp}"

ChatGPT:

To create a view that returns all users in the system with lots associated with them, you can follow these steps:

1. In your views, query all the `LotMetadata` objects.
2. From those objects, extract the associated users.
3. Serialize the user data along with the lot data.
4. Return the serialized data.

Here's a view that implements this:

```
python
from rest_framework import status
from accounts.models import CustomUser
from .models import LotMetadata
from rest_framework.views import APIView
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response

class UsersWithLotsView(APIView):
    permission_classes = [IsAuthenticated]

    def get(self, request, format=None):
        user = self.request.user
        allowed_roles = ['Lot Operator', 'Customer Support', 'Lot Specialist', 'Accountant']
        if user.role.role_name not in allowed_roles:
            return Response({"message": "Unauthorized."}, status=status.HTTP_403_FORBIDDEN)

        # Get all lots and their associated owners
        lots = LotMetadata.objects.all()
        users_with_lots = [{
            'email': lot.owner.email,
            'first_name': lot.owner.first_name,
            'last_name': lot.owner.last_name,
            'company_name': lot.owner.company_name,
            'lot_name': lot.name,
            'lot_id': lot.id
        } for lot in lots if lot.owner]  # if lot.owner ensures that only those lots with associated users are co

        return Response(users_with_lots, status=status.HTTP_200_OK)
```

You can then add this view to your URLs. The view will return the email, first name, last name, company name, lot name, and lot ID for each user with an associated lot. Adjust the fields as necessary based on your requirements.

You:

Using a fetch that works like this: useEffect(() => { const fetchTickets = async () => { const response = await fetch(API_URL + "tickets/get_tickets", { headers: { Authorization: `Token ${localStorage.getItem("token")}`, }, }); if (response.ok) { const data = await response.json(); setTickets(data); } }; fetchTickets(); }, []); Make this this .js file (CSS excluded): import React, { useEffect, useState } from 'react'; import { useNavigate } from 'react-router-dom'; import styled from 'styled-components'; import heroImage from '../images/park-hero.jpg'; import theme from '../theme'; import Footer from "./Footer"; const API_URL = process.env.REACT_APP_API_URL; const ChooseLotDashboard = () => { const navigate = useNavigate(); const handleMenuClick = (selected) => { if (selected === 'lot'){ navigate("/add-lot-invoice"); } if (selected === 'ad'){ navigate("/add-ad-invoice"); } }; return (<HomeContainer> <HeroImage> <ListOrganize> <SubHeading>Choose Create Invoice Type</SubHeading> <LocationList> <LocationItem onClick={() => handleMenuClick('lot')}> Create Lot Operator Invoice </LocationItem> <LocationItem onClick={() => handleMenuClick('ad')}> Create Advertiser Invoice </LocationItem> </LocationList> </ListOrganize> </HeroImage> <Footer /> </HomeContainer>); }; export default ChooseLotDashboard; Show Users with lots as options on the LocationList. Recall the View's url is API_URL + "lots/users_with_lots" and looks like this: class UsersWithLotsView(APIView): permission_classes = [permissions.IsAuthenticated] def get(self, request, format=None): user = self.request.user allowed_roles = ['Lot Operator', 'Customer Support', 'Lot Specialist', 'Accountant'] if user.role.role_name not in allowed_roles: return Response({"message": "Unauthorized."}, status=status.HTTP_403_FORBIDDEN) # Get all lots and their associated owners lots = LotMetadata.objects.all() users_with_lots = [{ 'email': lot.owner.email } for lot in lots if lot.owner] # if lot.owner ensures that only those lots with associated users are considered. return Response(users_with_lots, status=status.HTTP_200_OK)

ChatGPT:

To show the users with lots as options on the `LocationList`, you'll need to:

1. Fetch the data from the API endpoint using a similar approach to the `fetchTickets` method you provided.
2. Store the fetched data in a state variable.
3. Map through the state variable to display each user as an option on the `LocationList`.

Here's how you can modify the `ChooseLotDashboard` component to achieve this:

```jsx
import React, { useEffect, useState } from 'react';
import { useNavigate } from 'react-router-dom';
import styled from 'styled-components';
import heroImage from '../images/park-hero.jpg';
import theme from '../theme';
import Footer from "./Footer";

const API_URL = process.env.REACT_APP_API_URL;

const ChooseLotDashboard = () => {
  const navigate = useNavigate();
  const [usersWithLots, setUsersWithLots] = useState([]);

  useEffect(() => {
    const fetchUsersWithLots = async () => {
      const response = await fetch(API_URL + "lots/users_with_lots", {
        headers: {
          Authorization: `Token ${localStorage.getItem("token")}`,
        },
      });

      if (response.ok) {
        const data = await response.json();
        setUsersWithLots(data);
      }
    };

    fetchUsersWithLots();
  }, []);

  const handleMenuClick = (selected) => {
    if (selected === 'lot'){
      navigate("/add-lot-invoice");
    }
    if (selected === 'ad'){
      navigate("/add-ad-invoice");
    }
  };

  return (
    <HomeContainer>
      <HeroImage>
        <ListOrganize>
          <SubHeading>Choose Create Invoice Type</SubHeading>
          <LocationList>
            <LocationItem onClick={() => handleMenuClick('lot')}>
              Create Lot Operator Invoice
            </LocationItem>
            <LocationItem onClick={() => handleMenuClick('ad')}>
              Create Advertiser Invoice
            </LocationItem>
            {usersWithLots.map(user => (
              <LocationItem key={user.email}>
                {user.email}
              </LocationItem>
            ))}
          </LocationList>
        </ListOrganize>
      </HeroImage>
      <Footer />
    </HomeContainer>
  );
};

export default ChooseLotDashboard;
```

This modification will fetch the list of users with lots when the component mounts and then display them on the `LocationList`. Each user is displayed by their email address as an option. If you want to display other information, adjust the mapped content accordingly.

You:

Basically, I want it to get the parking lot data for the currently logged in user if the role is 'Lot Operator', however I want to get the parking lot data for a user with a supplied (in the request) email address if the role is 'Customer Support', 'Lot Specialist' or 'Accountant'. What's the smartest way to accomplish that? class LotOwnerDashboardView(APIView): permission_classes = [permissions.IsAuthenticated] def get(self, request, format=None): user = self.request.user allowed_roles = ['Lot Operator', 'Customer Support', 'Lot Specialist', 'Accountant'] if user.role.role_name not in allowed_roles: return Response({"message": "Unauthorized."}, status=status.HTTP_403_FORBIDDEN) # Retrieve the lots associated with the user lots = [] for x in LotMetadata.objects.all(): if str(x.owner) == request.user.email: lots.append(x) lot_cams = {} for lot in lots: cameras = CamMetadata.objects.filter(lot=lot) lot_cams[str(lot)] = cameras camera_names = [camera.name for camera in lot_cams[str(lots[0])]] try: lot_image = CamImage.objects.filter(camera_name=camera_names[0]).latest('timestamp') except CamImage.DoesNotExist: return Response({'detail': 'No images found for this camera.'}, status=status.HTTP_404_NOT_FOUND) # Get the URL of the image file image_url = default_storage.url(lot_image.image.name) try: previous_image = CamImage.objects.filter(camera_name=camera_names[0], timestamp__lt=lot_image.timestamp).latest('timestamp') previous_image_name_part = previous_image.image.name.split('_')[-1].replace('.jpg', '') except CamImage.DoesNotExist: # If there is no previous image, use the current image name part previous_image_name_part = lot_image.image.name.split('_')[-1].replace('.jpg', '') spots_path = os.path.join('models', camera_names[0], 'spots_view.json') bestspots_path = os.path.join('models', camera_names[0], 'bestspots.json') # Load the contents of the JSON files with open(spots_path, 'r') as spots_file: spots_data = json.load(spots_file) with open(bestspots_path, 'r') as bestspots_file: bestspots_data = json.load(bestspots_file) human_labels = json.loads(lot_image.human_labels) model_labels = json.loads(lot_image.model_labels) # Get the LPRMetadata for the lots lpr_metadata_list = LPRMetadata.objects.filter(lot__in=lots) lpr_serializer = LPRMetadataNoPasscodeSerializer(lpr_metadata_list, many=True) # Construct the response data response_data = { 'image_url': image_url, 'timestamp': lot_image.timestamp, 'human_labels': human_labels, 'model_labels': model_labels, 'previous_image_name_part': previous_image_name_part, 'spots': spots_data, 'bestspots': bestspots_data, 'lpr_metadata': lpr_serializer.data # Add the serialized LPRMetadata to the response } return Response(response_data)

ChatGPT:

To accomplish this, you can make use of the `request` object to determine which data to fetch.

1. If the role of the logged-in user is `Lot Operator`, you will fetch the lot data for that user.
2. If the role is `Customer Support`, `Lot Specialist`, or `Accountant` and an email address is supplied in the request, you will fetch the lot data for the user with that email address.

Here's how you can modify the `LotOwnerDashboardView` class to achieve this:

```
python
from rest_framework.exceptions import NotFound

class LotOwnerDashboardView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def get(self, request, format=None):
        user = self.request.user
        allowed_roles = ['Lot Operator', 'Customer Support', 'Lot Specialist', 'Accountant']
        if user.role.role_name not in allowed_roles:
            return Response({"message": "Unauthorized."}, status=status.HTTP_403_FORBIDDEN)

        # Decide which user's lots to fetch based on role and request data
        target_user_email = request.user.email  # default to the logged-in user
        if user.role.role_name in ['Customer Support', 'Lot Specialist', 'Accountant']:
            target_user_email = request.query_params.get('email', target_user_email)  # Use provided email or defa

        try:
            target_user = CustomUser.objects.get(email=target_user_email)
        except CustomUser.DoesNotExist:
            raise NotFound(detail="User not found.")

        # Retrieve the lots associated with the target user
        lots = LotMetadata.objects.filter(owner=target_user)
        lot_cams = {str(lot): CamMetadata.objects.filter(lot=lot) for lot in lots}

        # ... rest of your code ...

        return Response(response_data)
```

To fetch data for a specific user when the role is `Customer Support`, `Lot Specialist`, or `Accountant`, you would make a GET request to this view with the email address as a query parameter:

```
ruby
GET /your-endpoint/?email=targetuser@email.com
```

This way, the code checks the role of the authenticated user and decides which data to fetch based on that role and any provided email address.

You:

Do I have to change anything in the urls.py file to accommodate GET /your-endpoint/?email=targetuser@email.com

---

ChatGPT:

No, you don't have to make any changes in the `urls.py` file to accommodate this. The `email` parameter you're adding is a query parameter, and Django does not require you to explicitly define query parameters in the URL configuration.

For example, if your current URL pattern for this view in `urls.py` looks something like this:

```
python
from django.urls import path
from . import views

urlpatterns = [
    path('your-endpoint/', views.LotOwnerDashboardView.as_view(), name='lot_owner_dashboard'),
    # ... other url patterns ...
]
```

You can still make a GET request to:

```
ruby
GET /your-endpoint/?email=targetuser@email.com
```

And the `email` query parameter will be accessible in your view using:

```
python
email = request.query_params.get('email')
```

So, your existing URL configuration should work just fine for this purpose!

You:

How can I make it so <Route path="/operator-dashboard" element={<OperatorDashboard />} /> Can accept an optional argument that specifies an email for dashboards staff are looking at

---

ChatGPT:

To accept an optional argument in your route, you can use route parameters in `react-router`. However, if you're looking to pass an email as a route parameter, I'd recommend using a query parameter instead (as shown previously) for privacy reasons.

If you still want to use route parameters, here's how you can do it:

1. Update your route to accept an optional `email` parameter:

```
jsx
<Route path="/operator-dashboard/:email?" element={<OperatorDashboard />} />
```

The `?` at the end makes the parameter optional.

2. In your `OperatorDashboard` component, you can access the `email` parameter using the `useParams` hook:

```
jsx
import { useParams } from 'react-router-dom';

const OperatorDashboard = () => {
    const { email } = useParams();

    useEffect(() => {
        if (email) {
            // Fetch data for the specified email
        } else {
            // Fetch data for the currently logged-in user
```

```
        }
    }, [email]);

    // Rest of the component
}
```

However, I'd like to reiterate that using an email in the URL might expose sensitive user information. It's generally a better practice to use a user ID or some other non-sensitive identifier, or better yet, a query parameter which doesn't change the base URL of the page and can be more easily omitted when sharing links.

You:

Show me the difference between implementing a route parameter and a query parameter

---

ChatGPT:

Certainly! Let's dive into the difference between route parameters and query parameters using an example of a user profile page.

Route Parameter:

1. Setting up the route: For a route parameter, you'll specify the parameter within the path using `:parameterName`.

```
jsx
<Route path="/user/:userId" element={<UserProfile />} />
```

2. Accessing the parameter in the component: You can access the `userId` using the `useParams` hook.

```
jsx
import { useParams } from 'react-router-dom';

const UserProfile = () => {
    const { userId } = useParams();

    useEffect(() => {
        // Fetch data for userId
    }, [userId]);

    // Rest of the component
}
```

3. Navigating to the route: The URL will look something like `/user/12345`.

Query Parameter:

1. Setting up the route: The route does not need any special setup for query parameters.

```
jsx
<Route path="/user" element={<UserProfile />} />
```

2. Accessing the parameter in the component: You can access the query parameter using the `useLocation` hook.

```
jsx
import { useLocation } from 'react-router-dom';

const UserProfile = () => {
    const location = useLocation();
    const queryParams = new URLSearchParams(location.search);
    const userId = queryParams.get('userId');

    useEffect(() => {
        // Fetch data for userId
    }, [userId]);

    // Rest of the component
}
```

3. Navigating to the route: The URL will look something like `/user?userId=12345`.

Key Differences:

1. **URL Structure**:
    - Route parameters become part of the URL path itself (e.g., `/user/12345`).
    - Query parameters are added after the path, preceded by a ? (e.g., `/user?userId=12345`).
2. **Flexibility**:
    - Route parameters are more rigid, as they have to match a specific pattern in the URL.
    - Query parameters are more flexible. You can have multiple query parameters in any order, and they're optional by default.
3. **Usage**:
    - Route parameters are usually used for essential parts of the application routing, like specifying a resource (e.g., `/user/12345` or `/post/56789`).
    - Query parameters are commonly used for optional filtering, sorting, or other non-essential configurations (e.g., `/search?query=react&page=2`).
4. **Privacy**:
    - Using sensitive information, like emails, in a route parameter can expose user information when URLs are shared. Query parameters can be a bit safer in this aspect, but it's generally best to avoid placing sensitive information in URLs altogether.

Choosing between the two often depends on the context and the use-case. If the parameter is essential for defining the resource, a route parameter might be more appropriate. If it's optional or non-essential, a query parameter might be the better choice.

You:

Make it so a user clicks the email that email gets saved a parameter for the next page and then gets navigated to /operator-dashboard (where the parameter will be used) Set the email that was clicked as a query parameter and then go to /operator-dashboard import React, { useEffect, useState } from 'react'; import { useNavigate } from 'react-router-dom'; import styled from 'styled-components'; import heroImage from '../images/park-hero.jpg'; import theme from '../theme'; import Footer from "./Footer"; const API_URL = process.env.REACT_APP_API_URL; const HomeContainer = styled.div` background-color: white; align-items: center; justify-content: center; height: 100%; `; const SubHeading = styled.h2` font-size: 2rem; width: fit-content; color: white; background-color: rgba(0, 0, 0, 1); padding: 0.5rem 1rem; `; const ListOrganize = styled.div` display: flex; flex-direction: column; align-items: center; justify-content: center; padding-left: 0; `; const HeroImage = styled.div` width: 100%; min-height: 100vh; background-image: url(${heroImage}); background-position-y: top; background-position-x: center; background-size: cover; display: flex; // Set display to flex justify-content: center; // Center horizontally align-items: center; // Center vertically font-size: 1.5rem; margin-bottom: 2rem; `; const UserList = styled.ul` list-style-type: none; list-style-position: inside; margin-right: 0rem; margin-left:0rem; padding-right: 0rem; padding-left:0rem; `; const UserItem = styled.li` font-size: 120%; margin-right: 0rem; margin-left:0rem; padding-right: 0rem; padding-left:0rem; color: white; text-align: center; background-color: rgba(0, 0, 0, 1); padding: 0.5rem 1rem; cursor: pointer; transition: background-color 0.3s ease; &:hover { background-color: ${theme.secondary}; } `; const ChooseLotDashboard = () => { const navigate = useNavigate(); const [usersWithLots, setUsersWithLots] = useState([]); useEffect(() => { const fetchUsersWithLots = async () => { const response = await fetch(API_URL + "lots/users_with_lots/", { headers: { Authorization: `Token ${localStorage.getItem("token")}`, }, }); if (response.ok) { const data = await response.json(); setUsersWithLots(data); } }; fetchUsersWithLots(); }, []); return (<HomeContainer> <HeroImage> <ListOrganize> <SubHeading>Select Lot Operator to View Dashboard</SubHeading> <UserList> {usersWithLots.map(user => (<UserItem key={user.email}> {user.email} </UserItem>))} </UserList> </ListOrganize> </HeroImage> <Footer /> </HomeContainer>); }; export default ChooseLotDashboard;

ChatGPT:

To set the email that was clicked as a query parameter and then navigate to `/operator-dashboard`, you can add an `onClick` handler to the `UserItem`. This handler will call the `navigate` function with the desired path, including the query parameter. Here's how you can accomplish that:

1. Add the `handleUserClick` function to handle the navigation:

```
jsx
const handleUserClick = (email) => {
    navigate(`/operator-dashboard?email=${email}`);
};
```

2. Attach the `handleUserClick` function to the `UserItem`'s `onClick` event:

```
jsx
<UserItem key={user.email} onClick={() => handleUserClick(user.email)}>
    {user.email}
</UserItem>
```

Here's the updated code:

```
jsx
```

```
import React, { useEffect, useState } from 'react';
import { useNavigate } from 'react-router-dom';
import styled from 'styled-components';
import heroImage from '../images/park-hero.jpg';
import theme from '../theme';
import Footer from "./Footer";

// ... (other styled-components)

const ChooseLotDashboard = () => {
    const navigate = useNavigate();
    const [usersWithLots, setUsersWithLots] = useState([]);

    useEffect(() => {
        // ... (fetching data logic)
    }, []);

    const handleUserClick = (email) => {
        navigate(`/operator-dashboard?email=${email}`);
    };

    return (
        <HomeContainer>
            <HeroImage>
                <ListOrganize>
                    <SubHeading>Select Lot Operator to View Dashboard</SubHeading>
                    <UserList>
                        {usersWithLots.map(user => (
                            <UserItem key={user.email} onClick={() => handleUserClick(user.email)}>
                                {user.email}
                            </UserItem>
                        ))}
                    </UserList>
                </ListOrganize>
            </HeroImage>
            <Footer />
        </HomeContainer>
    );
};

export default ChooseLotDashboard;
```

Now, when a user clicks on an email, the app will navigate to `/operator-dashboard` with the clicked email set as a query parameter.

You:

make it so if it gets a email from navigate (`/operator-dashboard?email=${email}`); it adds the email to the request to 'lots/lot_dashboard/'. On the backend it retrieves it with target_user_email = request.query_params.get('email', target_user_email) if (token) { fetch(API_URL + 'lots/lot_dashboard/', { headers: { 'Content-Type': 'application/json', 'Authorization': `Token ${token}`, }, }) .then(response => response.json()) .then(data => { console.log(data.lpr_metadata); setDateOfMostRecentImage(data.timestamp); let bestSpotString = 'None available'; let BestSpotSoFarKey = 99999; for (let spot in Object.keys(data.bestspots)){ if(!data.human_labels[data.bestspots[spot]] & Number(spot) < BestSpotSoFarKey){ bestSpotString = data.bestspots[spot]; BestSpotSoFarKey = Number(spot); } } let totalSpotsFull = 0; for (let key in data.human_labels) { if (data.human_labels[key]){ totalSpotsFull = totalSpotsFull + 1; } } setCurrentCarsParked(totalSpotsFull); setMaxCarsParked(Object.keys(data.human_labels).length) const image = new Image(); image.src = API_URL + "lots" + data.image_url; image.onload = () => { canvas.width = image.width; canvas.height = image.height; context.drawImage(image, 0, 0, canvas.width, canvas.height); context.lineWidth = 9; context.font = "bold 50px Arial"; const entries = Object.entries(data.spots); entries.reverse().forEach(([key, value]) => { const [x1, x2, y1, y2] = value; const width = x2 - x1; const height = y2 - y1; if (key === bestSpotString){ context.strokeStyle = 'green'; context.fillStyle = 'green'; } else if (data.human_labels[key]){ context.strokeStyle = 'red'; context.fillStyle = 'red'; } else { context.strokeStyle = 'blue'; context.fillStyle = 'blue'; } context.strokeRect(x1, y1, width, height); context.fillText(key, x1, y1 - 5); const fetchReadings = async () => { let readings = {}; for (let lpr of data.lpr_metadata) { const response = await fetch(API_URL + `lots/recentreadings/${lpr.name}/`, { headers: { 'Content-Type': 'application/json', 'Authorization': `Token ${token}`, } }); readings[lpr.name] = await response.json(); console.log(readings); } setRecentReadings(readings); } fetchReadings(); }); }

ChatGPT:

To include the email in the request when you navigate to `/operator-dashboard?email=${email}`, you need to extract the email from the URL's query parameters and include it in the fetch request to 'lots/lot_dashboard/'.

Here's how you can do it:

1. First, you need to use the `useLocation` hook from `react-router-dom` to get the current location (URL) of your component.
2. Extract the email from the query parameters using the URLSearchParams API.

3. Include the extracted email in the fetch request.

Let's modify your code to achieve this:

```
jsx
import { useLocation } from 'react-router-dom'; // import useLocation

// ... (rest of your imports)

const OperatorDashboard = () => {
  // ... (your other state definitions and functions)

  const location = useLocation();
  const queryParams = new URLSearchParams(location.search);
  const email = queryParams.get('email');

  useEffect(() => {
    if (token) {
      let url = API_URL + 'lots/lot_dashboard/';
      if (email) {
        url += `?email=${email}`;
      }

      fetch(url, {
        headers: {
          'Content-Type': 'application/json',
          'Authorization': `Token ${token}`,
        },
      })
      .then(response => response.json())
      .then(data => {
        // ... (rest of your logic)
      });
    }
  }, [token, email]);

  // ... (rest of your component)

};
```

This way, if there is an email in the URL's query parameters, it will be included in the fetch request to 'lots/lot_dashboard/', and on the backend, you can retrieve it using `target_user_email = request.query_params.get('email', target_user_email)`.

You:

Now, there's a high chance it will have none, it which case it should not include that information in the API request

---

ChatGPT:

Alright! If there's no email present in the query parameters, we should avoid adding it to the fetch URL.

Here's how you can modify the code to achieve that:

```
jsx
useEffect(() => {
    if (token) {
      // Start with the base URL.
      let url = API_URL + 'lots/lot_dashboard/';

      // If there's an email, append it to the URL.
      if (email) {
        url += `?email=${email}`;
      }

      fetch(url, {
        headers: {
          'Content-Type': 'application/json',
          'Authorization': `Token ${token}`,
        },
      })
```

```
      .then(response => response.json())
      .then(data => {
        // ... (rest of your logic)
      });
    }
  }, [token, email]);
```

This code ensures that if there's no email present, the URL remains unchanged. Only when there's an email in the query parameters will it be appended to the fetch URL.

You:

Okay that worked great. Email is conditionally added to the API call here: if (token) { fetch(API_URL + 'lots/get_lot_history/', { headers: { 'Content-Type': 'application/json', 'Authorization': `Token ${token}`, }, }) Doe the exact same thing for this call here: if (token) { fetch(API_URL + 'lots/get_lot_history/', { headers: { 'Content-Type': 'application/json', 'Authorization': `Token ${token}`, }, })

ChatGPT:

Sure! Just like before, we'll conditionally append the email to the fetch URL if it's present. Here's how you can modify the provided code:

```
jsx
useEffect(() => {
    if (token) {
      // Start with the base URL.
      let url = API_URL + 'lots/get_lot_history/';

      // If there's an email, append it to the URL.
      if (email) {
        url += `?email=${email}`;
      }

      fetch(url, {
        headers: {
          'Content-Type': 'application/json',
          'Authorization': `Token ${token}`,
        },
      })
      .then(response => response.json())
      .then(data => {
        // ... (rest of your logic for processing the data)
      });
    }
  }, [token, email]);
```

The above code will ensure that the `email` query parameter is only appended to the URL when there's an email value present. Otherwise, the fetch request will proceed with the base URL.

You:

Make it so GetLotHistory can optionally receive an email, and if it does user becomes the user with that supplied email instead of the user that sent the request. Implement this the same way it is implemented in LotOwnerDashboardView. Also make it so allowed_roles matches LotOwnerDashboardView. class GetLotHistory(APIView): permission_classes = [permissions.IsAuthenticated] def get(self, request, format=None): user = self.request.user role_name = user.role.role_name if role_name != 'Lot Operator': return Response({"message": "Unauthorized."}, status=status.HTTP_403_FORBIDDEN) # Retrieve the lots associated with the user lots = LotMetadata.objects.filter(owner=user) cam_names = CamMetadata.objects.filter(lot__in=lots).values_list('name', flat=True) cam_images = CamImage.objects.filter(camera_name__in=cam_names) # Serialize the cam_images using CamImageSerializer serializer = CamImageSerializer(cam_images, many=True) response_data = { 'image_data': serializer.data } return Response(response_data) class LotOwnerDashboardView(APIView): permission_classes = [permissions.IsAuthenticated] def get(self, request, format=None): user = self.request.user allowed_roles = ['Lot Operator', 'Customer Support', 'Lot Specialist', 'Accountant'] if user.role.role_name not in allowed_roles: return Response({"message": "Unauthorized."}, status=status.HTTP_403_FORBIDDEN) # Decide which user's lots to fetch based on role and request data target_user_email = user.email # default to the logged-in user if user.role.role_name in ['Customer Support', 'Lot Specialist', 'Accountant']: target_user_email = request.query_params.get('email', target_user_email) # Use provided email or default to the logged-in user try: target_user = CustomUser.objects.get(email=target_user_email) except CustomUser.DoesNotExist: raise NotFound(detail="User not found.") # Retrieve the lots associated with the target user lots = LotMetadata.objects.filter(owner=target_user) lot_cams = {str(lot): CamMetadata.objects.filter(lot=lot) for lot in lots} camera_names = [camera.name for camera in lot_cams[str(lots[0])]] try: lot_image = CamImage.objects.filter(camera_name=camera_names[0]).latest('timestamp') except CamImage.DoesNotExist: return Response({'detail': 'No images found for this camera.'}, status=status.HTTP_404_NOT_FOUND) # Get the URL of the image file image_url = default_storage.url(lot_image.image.name)

try: previous_image = CamImage.objects.filter(camera_name=camera_names[0], timestamp__lt=lot_image.timestamp).latest('timestamp') previous_image_name_part = previous_image.image.name.split('_')[-1].replace('.jpg', '') except CamImage.DoesNotExist: # If there is no previous image, use the current image name part previous_image_name_part = lot_image.image.name.split('_')[-1].replace('.jpg', '') spots_path = os.path.join('models', camera_names[0], 'spots_view.json') bestspots_path = os.path.join('models', camera_names[0], 'bestspots.json') # Load the contents of the JSON files with open(spots_path, 'r') as spots_file: spots_data = json.load(spots_file) with open(bestspots_path, 'r') as bestspots_file: bestspots_data = json.load(bestspots_file) human_labels = json.loads(lot_image.human_labels) model_labels = json.loads(lot_image.model_labels) # Get the LPRMetadata for the lots lpr_metadata_list = LPRMetadata.objects.filter(lot__in=lots) lpr_serializer = LPRMetadataNoPasscodeSerializer(lpr_metadata_list, many=True) # Construct the response data response_data = { 'image_url': image_url, 'timestamp': lot_image.timestamp, 'human_labels': human_labels, 'model_labels': model_labels, 'previous_image_name_part': previous_image_name_part, 'spots': spots_data, 'bestspots': bestspots_data, 'lpr_metadata': lpr_serializer.data # Add the serialized LPRMetadata to the response } return Response(response_data)

ChatGPT:

Alright, let's modify `GetLotHistory` to behave similarly to `LotOwnerDashboardView`. Here's how we can modify `GetLotHistory`:

1. Adjust the permissions to check if the role of the user is in the allowed roles.
2. Decide which user's lots to fetch based on the role and any email passed in the request.

Here's the updated `GetLotHistory` class:

```
python
class GetLotHistory(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def get(self, request, format=None):
        user = self.request.user
        allowed_roles = ['Lot Operator', 'Customer Support', 'Lot Specialist', 'Accountant']
        if user.role.role_name not in allowed_roles:
            return Response({"message": "Unauthorized."}, status=status.HTTP_403_FORBIDDEN)

        # Decide which user's lots to fetch based on role and request data
        target_user_email = user.email  # default to the logged-in user
        if user.role.role_name in ['Customer Support', 'Lot Specialist', 'Accountant']:
            target_user_email = request.query_params.get('email', target_user_email)  # Use provided email or defa

        try:
            target_user = CustomUser.objects.get(email=target_user_email)
        except CustomUser.DoesNotExist:
            raise NotFound(detail="User not found.")

        # Retrieve the lots associated with the target user
        lots = LotMetadata.objects.filter(owner=target_user)
        cam_names = CamMetadata.objects.filter(lot__in=lots).values_list('name', flat=True)
        cam_images = CamImage.objects.filter(camera_name__in=cam_names)

        # Serialize the cam_images using CamImageSerializer
        serializer = CamImageSerializer(cam_images, many=True)
        response_data = {
            'image_data': serializer.data
        }
        return Response(response_data)
```

This code first checks if the user's role is in the allowed roles. If it is, it will then determine which user's lot history to fetch based on the role and any email parameter passed in the request. If the logged-in user's role is one of the staff roles (`Customer Support`, `Lot Specialist`, `Accountant`), it will use the email provided in the query parameter (if present) or default to the logged-in user's email. If the logged-in user's role is `Lot Operator`, it will only fetch the lot history for the logged-in user.

You:

Make it so this fetch: const fetchReadings = async (year, month) => { const token = localStorage.getItem("token"); let readings = {}; const response = await fetch(API_URL + 'lots/lot_dashboard/', { headers: { 'Content-Type': 'application/json', 'Authorization': `Token ${token}`, }, }) Works like this fetch: if (token) { let url = API_URL + 'lots/lot_dashboard/'; if (email) { url += `?email=${email}`; } fetch(url, { headers: { 'Content-Type': 'application/json', 'Authorization': `Token ${token}`, }, })

ChatGPT:

Certainly! To modify the `fetchReadings` function to conditionally append the email as a query parameter (if it exists) to the API endpoint, you can use the following code:

```
javascript
const fetchReadings = async (year, month) => {
    const token = localStorage.getItem("token");
    let readings = {};

    let url = API_URL + 'lots/lot_dashboard/';
    const email = /* retrieve the email value from somewhere, e.g., component state */;
    if (email) {
        url += `?email=${email}`;
    }

    const response = await fetch(url, {
        headers: {
            'Content-Type': 'application/json',
            'Authorization': `Token ${token}`,
        },
    });

    // ... rest of the function logic
}
```

You'll need to ensure that the email value is retrieved or accessible within the `fetchReadings` function. If you have stored the email in the component's state or received it via props, you can access it directly. If not, you might need to pass it as an additional argument to the `fetchReadings` function or retrieve it from another source.

You:

import React, { useEffect, useState } from 'react'; import { useNavigate } from 'react-router-dom'; import styled from 'styled-components'; import heroImage from '../images/park-hero.jpg'; import theme from '../theme'; import Footer from "./Footer"; const API_URL = process.env.REACT_APP_API_URL; const HomeContainer = styled.div` background-color: white; align-items: center; justify-content: center; height: 100%; `; const SubHeading = styled.h2` font-size: 2rem; width: fit-content; color: white; background-color: rgba(0, 0, 0, 1); padding: 0.5rem 1rem; `; const ListOrganize = styled.div` display: flex; flex-direction: column; align-items: center; justify-content: center; padding-left: 0; `; const HeroImage = styled.div` width: 100%; min-height: 100vh; background-image: url(${heroImage}); background-position-y: top; background-position-x: center; background-size: cover; display: flex; // Set display to flex justify-content: center; // Center horizontally align-items: center; // Center vertically font-size: 1.5rem; margin-bottom: 2rem; `; const UserList = styled.ul` list-style-type: none; list-style-position: inside; margin-right: 0rem; margin-left:0rem; padding-right: 0rem; padding-left:0rem; `; const UserItem = styled.li` font-size: 120%; margin-right: 0rem; margin-left:0rem; padding-right: 0rem; padding-left:0rem; color: white; text-align: center; background-color: rgba(0, 0, 0, 1); padding: 0.5rem 1rem; cursor: pointer; transition: background-color 0.3s ease; &:hover { background-color: ${theme.secondary}; } `; const ChooseLotDashboard = () => { const navigate = useNavigate(); const [usersWithLots, setUsersWithLots] = useState([]); useEffect(() => { const fetchUsersWithLots = async () => { const response = await fetch(API_URL + "lots/users_with_lots/", { headers: { Authorization: `Token ${localStorage.getItem("token")}`, }, }); if (response.ok) { const data = await response.json(); setUsersWithLots(data); } }; fetchUsersWithLots(); }, []); const handleUserClick = (email) => { navigate(`/operator-dashboard?email=${email}`); }; return (<HomeContainer> <HeroImage> <ListOrganize> <SubHeading>Select Lot Operator to View Dashboard</SubHeading> <UserList> {usersWithLots.map(user => (<UserItem key={user.email} onClick={() => handleUserClick(user.email)}> {user.email} </UserItem>))} </UserList> </ListOrganize> </HeroImage> <Footer /> </HomeContainer>); }; export default ChooseLotDashboard; Make it so there's a radio button choice above the UserList:: Default option is labeled 'Dashboard' and the secondary option is labeled 'Plates' If Dashboard is select the target user URL remains unchanged (/operator-dashboard?email=${email}`) but if radio button Plates is current selected then the url chages too import React, { useEffect, useState } from 'react'; import { useNavigate } from 'react-router-dom'; import styled from 'styled-components'; import heroImage from '../images/park-hero.jpg'; import theme from '../theme'; import Footer from "./Footer"; const API_URL = process.env.REACT_APP_API_URL; const HomeContainer = styled.div` background-color: white; align-items: center; justify-content: center; height: 100%; `; const SubHeading = styled.h2` font-size: 2rem; width: fit-content; color: white; background-color: rgba(0, 0, 0, 1); padding: 0.5rem 1rem; `; const ListOrganize = styled.div` display: flex; flex-direction: column; align-items: center; justify-content: center; padding-left: 0; `; const HeroImage = styled.div` width: 100%; min-height: 100vh; background-image: url(${heroImage}); background-position-y: top; background-position-x: center; background-size: cover; display: flex; // Set display to flex justify-content: center; // Center horizontally align-items: center; // Center vertically font-size: 1.5rem; margin-bottom: 2rem; `; const UserList = styled.ul` list-style-type: none; list-style-position: inside; margin-right: 0rem; margin-left:0rem; padding-right: 0rem; padding-left:0rem; `; const UserItem = styled.li` font-size: 120%; margin-right: 0rem; margin-left:0rem; padding-right: 0rem; padding-left:0rem; color: white; text-align: center; background-color: rgba(0, 0, 0, 1); padding: 0.5rem 1rem; cursor: pointer; transition: background-color 0.3s ease; &:hover { background-color: ${theme.secondary}; } `; const ChooseLotDashboard = () => { const navigate = useNavigate(); const [usersWithLots, setUsersWithLots] = useState([]); useEffect(() => { const fetchUsersWithLots = async () => { const response = await fetch(API_URL + "lots/users_with_lots/", { headers: { Authorization: `Token ${localStorage.getItem("token")}`, }, }); if (response.ok) { const data = await response.json(); setUsersWithLots(data); } }; fetchUsersWithLots(); }, []); const handleUserClick = (email) => { navigate(`/operator-dashboard?email=${email}`); }; return (<HomeContainer> <HeroImage> <ListOrganize> <SubHeading>Select Lot Operator to View Dashboard</SubHeading> <UserList> {usersWithLots.map(user => (<UserItem key={user.email} onClick={() => handleUserClick(user.email)}> {user.email} </UserItem>))} </UserList> </ListOrganize> </HeroImage> <Footer /> </HomeContainer>); }; export default ChooseLotDashboard; Make it so there's a radio button choice above the UserList:: Default option is labeled 'Dashboard' and the secondary option is labeled 'Plates' If Dashboard is select the target user URL remains unchanged (/operator-dashboard?email=${email}`) but if radio button Plates is current selected then the url chages too import React, { useEffect, useState } from 'react'; import { useNavigate } from 'react-router-dom'; import styled from 'styled-components'; import heroImage from '../images/park-hero.jpg'; import theme from '../theme'; import Footer from "./Footer"; const API_URL = process.env.REACT_APP_API_URL; const HomeContainer = styled.div` background-color: white; align-items: center; justify-content: center; height: 100%; `; const SubHeading = styled.h2` font-size: 2rem; width: fit-content; color: white; background-color: rgba(0, 0, 0, 1); padding: 0.5rem 1rem; `; const ListOrganize = styled.div` display: flex; flex-direction: column; align-items: center; justify-content: center; padding-left: 0; `; const HeroImage = styled.div` width: 100%; min-height: 100vh; background-image: url(${heroImage}); background-position-y: top; background-position-x: center; background-size: cover; display: flex; // Set display to flex justify-content: center; // Center horizontally align-items: center; // Center vertically font-size: 1.5rem; margin-bottom: 2rem; `; const UserList = styled.ul` list-style-type: none; list-style-position: inside; margin-right: 0rem; margin-left:0rem; padding-right: 0rem; padding-left:0rem; `; const UserItem = styled.li` font-size: 120%; margin-right: 0rem; margin-

left:0rem; padding-right: 0rem; padding-left:0rem; color: white; text-align: center; background-color: rgba(0, 0, 0, 1); padding: 0.5rem 1rem; cursor: pointer; transition: background-color 0.3s ease; &:hover { background-color: ${theme.secondary}; } `; const ChooseLotDashboard = () => { const navigate = useNavigate(); const [usersWithLots, setUsersWithLots] = useState([]); useEffect(() => { const fetchUsersWithLots = async () => { const response = await fetch(API_URL + "lots/users_with_lots/", { headers: { Authorization: `Token ${localStorage.getItem("token")}`, }, }); if (response.ok) { const data = await response.json(); setUsersWithLots(data); } }; fetchUsersWithLots(); }, []); const handleUserClick = (email) => { navigate(`/operator-dashboard?email=${email}`); }; return (<HomeContainer> <HeroImage> <ListOrganize> <SubHeading>Select Lot Operator to View Dashboard</SubHeading> <UserList> {usersWithLots.map(user => (<UserItem key={user.email} onClick={() => handleUserClick(user.email)}> {user.email} </UserItem>))} </UserList> </ListOrganize> </HeroImage> <Footer /> </HomeContainer>); }; export default ChooseLotDashboard; Make it so there's a radio button choice above the UserList:: Default option is labeled 'Dashboard' and the secondary option is labeled 'Plates' If Dashboard is select the target user URL remains unchanged (/operator-dashboard?email=${email}`) but if radio button Plates is current selected then the url chages too import React, { useEffect, useState } from 'react'; import { useNavigate } from 'react-router-dom'; import styled from 'styled-components'; import heroImage from '../images/park-hero.jpg'; import theme from '../theme'; import Footer from "./Footer"; const API_URL = process.env.REACT_APP_API_URL; const HomeContainer = styled.div` background-color: white; align-items: center; justify-content: center; height: 100%; `; const SubHeading = styled.h2` font-size: 2rem; width: fit-content; color: white; background-color: rgba(0, 0, 0, 1); padding: 0.5rem 1rem; `; const ListOrganize = styled.div` display: flex; flex-direction: column; align-items: center; justify-content: center; padding-left: 0; `; const HeroImage = styled.div` width: 100%; min-height: 100vh; background-image: url(${heroImage}); background-position-y: top; background-position-x: center; background-size: cover; display: flex; // Set display to flex justify-content: center; // Center horizontally align-items: center; // Center vertically font-size: 1.5rem; margin-bottom: 2rem; `; const UserList = styled.ul` list-style-type: none; list-style-position: inside; margin-right: 0rem; margin-left:0rem; padding-right: 0rem; padding-left:0rem; `; const UserItem = styled.li` font-size: 120%; margin-right: 0rem; margin-left:0rem; padding-right: 0rem; padding-left:0rem; color: white; text-align: center; background-color: rgba(0, 0, 0, 1); padding: 0.5rem 1rem; cursor: pointer; transition: background-color 0.3s ease; &:hover { background-color: ${theme.secondary}; } `; const ChooseLotDashboard = () => { const navigate = useNavigate(); const [usersWithLots, setUsersWithLots] = useState([]); useEffect(() => { const fetchUsersWithLots = async () => { const response = await fetch(API_URL + "lots/users_with_lots/", { headers: { Authorization: `Token ${localStorage.getItem("token")}`, }, }); if (response.ok) { const data = await response.json(); setUsersWithLots(data); } }; fetchUsersWithLots(); }, []); const handleUserClick = (email) => { navigate(`/operator-dashboard?email=${email}`); }; return (<HomeContainer> <HeroImage> <ListOrganize> <SubHeading>Select Lot Operator to View Dashboard</SubHeading> <UserList> {usersWithLots.map(user => (<UserItem key={user.email} onClick={() => handleUserClick(user.email)}> {user.email} </UserItem>))} </UserList> </ListOrganize> </HeroImage> <Footer /> </HomeContainer>); }; export default ChooseLotDashboard; Make it so there's a radio button choice above the UserList:: Default option is labeled 'Dashboard' and the secondary option is labeled 'Plates' If Dashboard is select the target user URL remains unchanged (/operator-dashboard?email=${email}`) but if radio button Plates is current selected then the url chages to (/plate-data?email=${email})

ChatGPT:

Sure! Here's how you can achieve this:

1. Add a `useState` hook to manage the selected radio button's value.
2. Add the radio buttons for 'Dashboard' and 'Plates'.
3. Modify the `handleUserClick` function to navigate to the appropriate URL based on the selected radio button's value.

Here's the modified `ChooseLotDashboard` component:

```
javascript
const ChooseLotDashboard = () => {
  const navigate = useNavigate();
  const [usersWithLots, setUsersWithLots] = useState([]);
  const [selectedOption, setSelectedOption] = useState('Dashboard');  // Step 1

  useEffect(() => {
    const fetchUsersWithLots = async () => {
      const response = await fetch(API_URL + "lots/users_with_lots/", {
        headers: {
          Authorization: `Token ${localStorage.getItem("token")}`,
        },
      });

      if (response.ok) {
        const data = await response.json();
        setUsersWithLots(data);
      }
    };

    fetchUsersWithLots();
  }, []);

  const handleUserClick = (email) => {
    if (selectedOption === 'Dashboard') {
      navigate(`/operator-dashboard?email=${email}`);
    } else if (selectedOption === 'Plates') {
      navigate(`/plate-data?email=${email}`);
    }
  };

  return (
    <HomeContainer>
      <HeroImage>
        <ListOrganize>
```

```
          <SubHeading>Select Lot Operator to View Dashboard</SubHeading>
          {/* Step 2 */}
          <div>
            <input
              type="radio"
              id="dashboard"
              name="option"
              value="Dashboard"
              checked={selectedOption === 'Dashboard'}
              onChange={() => setSelectedOption('Dashboard')}
            />
            <label htmlFor="dashboard">Dashboard</label>

            <input
              type="radio"
              id="plates"
              name="option"
              value="Plates"
              checked={selectedOption === 'Plates'}
              onChange={() => setSelectedOption('Plates')}
            />
            <label htmlFor="plates">Plates</label>
          </div>
          <UserList>
            {usersWithLots.map(user => (
              <UserItem key={user.email} onClick={() => handleUserClick(user.email)}>
                {user.email}
              </UserItem>
            ))}
          </UserList>
        </ListOrganize>
      </HeroImage>
      <Footer />
    </HomeContainer>
  );
};
```

This solution will:

- Render two radio buttons labeled 'Dashboard' and 'Plates'.
- Allow you to select between the two options.
- Navigate to the appropriate URL when a user is clicked based on the selected radio button option.

You:

const styledInput = styled.input` font-size: 2rem; width: fit-content; color: white; `; Why isn't this working they way I'd expect it to?

ChatGPT:

The code you provided is using the `styled-components` library to create a styled `input`. However, there are a few potential reasons why the styles might not be applied as you expect:

1. **CSS specificity**: If there are other styles in your application with higher specificity, they might override the styles you've defined with `styled-components`.
2. **CSS inheritance**: The `input` tag doesn't inherit all properties. For instance, if you set the `color` property to `white`, it will not affect the text color inside the input. Instead, you might want to set the `background-color` to white and adjust the `color` property to something like black or another color for the text.
3. **Parent Styles**: Sometimes, parent styles or global styles can interfere with styled components.
4. **Usage**: Ensure you're actually using the `styledInput` component in your JSX. If you've defined the styled component but are still using a regular `<input>` tag, the styles won't be applied.
5. **Library Installation**: Ensure that you have `styled-components` properly installed and imported.
6. **Syntax Issues**: Ensure there aren't any syntax issues elsewhere in your code that could be causing the styled component to malfunction.

To further diagnose the issue, I'd recommend the following steps:

1. Check if other styles from `styled-components` are working. If not, there might be an issue with how the library is set up.

2. Inspect the input element in the browser's developer tools. See if the styles from `styledInput` are being applied, or if they're being overridden by other styles.
3. Simplify the styles in `styledInput` to just one property to see if it takes effect. This can help in understanding whether it's a particular property causing the issue or the entire styled component.

If you're still facing problems after these steps, please provide more context or a more detailed example, and I'd be happy to assist further!

You:

Here's more context: <styledInput type="radio" id="dashboard" name="option" value="Dashboard" checked={selectedOption === 'Dashboard'} onChange={() => setSelectedOption('Dashboard')} /> <label htmlFor="dashboard">Dashboard</label> <input type="radio" id="plates" name="option" value="Plates" checked={selectedOption === 'Plates'} onChange={() => setSelectedOption('Plates')} />

ChatGPT:

Thank you for providing more context. The issue seems to be with the way you're using the styled component.

In React, components should be named with a capital first letter. This capitalization convention helps differentiate between HTML elements and custom React components. So, in your case, the component `styledInput` should be renamed to `StyledInput`.

Here's how you should modify your code:

1. Rename `styledInput` to `StyledInput`:

```
javascript
const StyledInput = styled.input`
  font-size: 2rem;
  width: fit-content;
  color: white;
`;
```

2. Update the usage in your JSX:

```
javascript
<StyledInput
  type="radio"
  id="dashboard"
  name="option"
  value="Dashboard"
  checked={selectedOption === 'Dashboard'}
  onChange={() => setSelectedOption('Dashboard')}
/>
<label htmlFor="dashboard">Dashboard</label>
<input
  type="radio"
  id="plates"
  name="option"
  value="Plates"
  checked={selectedOption === 'Plates'}
  onChange={() => setSelectedOption('Plates')}
/>
```

With this change, your styled component should now be applied correctly to the input element.

You:

const StyledLabel = styled.label` font-size: 2rem; width: fit-content; color: white; `; Make the circle 50% bigger and make it so there's so there's a bit of space between it and the surround items

---

ChatGPT:

To make the radio button (represented as a circle) 50% bigger, you can adjust its `scale` property using CSS. Additionally, to add space between it and surrounding items, you can use the `margin` property.

Here's how you can achieve both of these:

1. Adjust the scale of the radio input by targeting it within the `StyledLabel`:

```
javascript
const StyledLabel = styled.label`
  font-size: 2rem;
  width: fit-content;
  color: white;

  input[type="radio"] {
    transform: scale(1.5);  // this will make it 50% bigger
    margin-right: 8px;  // space between the radio and its label
    margin-top: 4px;  // adjust this as needed for vertical spacing
    margin-bottom: 4px;  // adjust this as needed for vertical spacing
  }
`;
```

2. Update the usage in your JSX:

```
javascript
<StyledLabel htmlFor="dashboard">
  <input
    type="radio"
    id="dashboard"
    name="option"
    value="Dashboard"
    checked={selectedOption === 'Dashboard'}
    onChange={() => setSelectedOption('Dashboard')}
  />
  Dashboard
</StyledLabel>

<StyledLabel htmlFor="plates">
  <input
    type="radio"
    id="plates"
    name="option"
    value="Plates"
    checked={selectedOption === 'Plates'}
    onChange={() => setSelectedOption('Plates')}
  />
  Plates
</StyledLabel>
```

With these changes, the radio button should be 50% bigger and have some space around it. Adjust the `margin` values as needed to get the desired spacing.

You:

Make it so this code can accept an alternative email of a user to use as a user instead of the user who submitted the request in this View. It should work as it did before if no email is provided: def user_ads_list_view(request): user_ads = Ad.objects.filter(user=request.user) serializer = AdSerializer(user_ads, many=True) serialized_data = serializer.data # Convert image paths to Base64 encoded data for ad in serialized_data: for key in ['top_banner_image1', 'top_banner_image2', 'top_banner_image3', 'side_banner_image1', 'side_banner_image2', 'side_banner_image3']: image_path = ad[key] with open(image_path, "rb") as image_file: base64_encoded = base64.b64encode(image_file.read()).decode('utf-8') ad[key] = f"{base64_encoded}" return Response(serialized_data, status=status.HTTP_200_OK) It should resemble how the below view works, except: allowed_roles = ['Lot Operator', 'Customer Support', 'Lot Specialist', 'Accountant'] should be: allowed_roles = ['Lot Operator', 'Customer Support', 'Advertising Specialist', 'Accountant'] and: if user.role.role_name in ['Customer Support', 'Lot Specialist', 'Accountant']: should be: if user.role.role_name in ['Customer Support', 'Advertising Specialist', 'Accountant']: This is what it should resemble with the user role adjustements: class LotOwnerDashboardView(APIView): permission_classes = [permissions.IsAuthenticated] def get(self, request, format=None): user = self.request.user allowed_roles = ['Lot Operator', 'Customer Support', 'Lot Specialist', 'Accountant'] if user.role.role_name not in allowed_roles: return Response({"message": "Unauthorized."}, status=status.HTTP_403_FORBIDDEN) # Decide which user's lots to fetch based on role and request data target_user_email = user.email # default to the logged-in user if user.role.role_name in ['Customer Support', 'Lot Specialist', 'Accountant']: target_user_email = request.query_params.get('email', target_user_email) # Use provided email or default to the logged-in user try: target_user = CustomUser.objects.get(email=target_user_email) except CustomUser.DoesNotExist: raise NotFound(detail="User not found.") # Retrieve the lots associated with the target user lots = LotMetadata.objects.filter(owner=target_user) lot_cams = {str(lot): CamMetadata.objects.filter(lot=lot) for lot in lots} camera_names = [camera.name for camera in lot_cams[str(lots[0])]]

ChatGPT:

To modify the `user_ads_list_view` function to accept an alternative email and fetch ads for that user, we need to make a few changes:

1. Check the roles of the user making the request to ensure they are allowed to view ads for a user other than themselves.
2. Check for an 'email' query parameter to determine whose ads to retrieve.
3. Fetch the ads for the user specified by the 'email' query parameter or default to the requesting user if no email is provided.

Here's how you can modify the function:

```
python
from rest_framework.decorators import api_view, permission_classes
from rest_framework import permissions, status
from rest_framework.response import Response
from django.core.exceptions import ObjectDoesNotExist

@api_view(['GET'])
@permission_classes([permissions.IsAuthenticated])
def user_ads_list_view(request):
    user = request.user
    allowed_roles = ['Lot Operator', 'Customer Support', 'Advertising Specialist', 'Accountant']

    # Check if the user has an allowed role
    if user.role.role_name not in allowed_roles:
        return Response({"message": "Unauthorized."}, status=status.HTTP_403_FORBIDDEN)

    # Decide which user's ads to fetch based on role and request data
    target_user_email = user.email  # default to the logged-in user
    if user.role.role_name in ['Customer Support', 'Advertising Specialist', 'Accountant']:
        target_user_email = request.query_params.get('email', target_user_email)  # Use provided email or default

    try:
        target_user = CustomUser.objects.get(email=target_user_email)
    except ObjectDoesNotExist:
        return Response({"message": "User not found."}, status=status.HTTP_404_NOT_FOUND)

    # Retrieve the ads associated with the target user
    user_ads = Ad.objects.filter(user=target_user)
    serializer = AdSerializer(user_ads, many=True)
    serialized_data = serializer.data

    # Convert image paths to Base64 encoded data
    for ad in serialized_data:
        for key in ['top_banner_image1', 'top_banner_image2', 'top_banner_image3',
                    'side_banner_image1', 'side_banner_image2', 'side_banner_image3']:

            image_path = ad[key]
            with open(image_path, "rb") as image_file:
                base64_encoded = base64.b64encode(image_file.read()).decode('utf-8')
            ad[key] = f"{base64_encoded}"

    return Response(serialized_data, status=status.HTTP_200_OK)
```

With these changes, the `user_ads_list_view` function should be able to retrieve ads for a user specified by an 'email' query parameter or default to the requesting user if no email is provided, and it will check for the allowed roles as specified.

You:

Can you make it so it fetches ads here: fetch(API_URL + 'ads/user-ads/', { headers: { 'Content-Type': 'application/json', 'Authorization': `Token ${token}`, }, }) With the same param checking logic as when it fetches this api endpoint: if (token) { let url = API_URL + 'lots/lot_dashboard/'; if (email) { url += `?email=${email}`; } fetch(url, { headers: { 'Content-Type': 'application/json', 'Authorization': `Token ${token}`, }, }) With this above: const queryParams = new URLSearchParams(location.search); const email = queryParams.get('email');

ChatGPT:

Certainly! To fetch ads with the same parameter-checking logic, you can modify the fetch code as follows:

```
javascript
const queryParams = new URLSearchParams(location.search);
const email = queryParams.get('email');

const token = localStorage.getItem("token");

if (token) {
  let url = API_URL + 'ads/user-ads/';
  if (email) {
    url += `?email=${email}`;
  }

  fetch(url, {
    headers: {
      'Content-Type': 'application/json',
      'Authorization': `Token ${token}`,
    },
  })
  .then(response => response.json())
  .then(data => {
    // Handle the fetched data here
  })
  .catch(error => {
    console.error("There was an error fetching the ads:", error);
  });
}
```

With the above code, if an `email` query parameter is present in the URL, it will be used when fetching the `user-ads` endpoint. Otherwise, the fetch will proceed without the email parameter, which will then return the ads of the authenticated user.
