Using ec2 instances as sneaker bid bots pt 2.

Date: 2023-11-27
By: csr13

Download as PDF

Part two consists of the following:

- Creating the endpoints for biding and handling payment directly with Stripe.
- Storing useful datas on the appropriate database tables, for which product this bid is on, size of the shoe, color, and other datas.
- Creating the ec2/t2.micro instances raising them, obtaining their ip, and then putting them in an off state (so they don't generate expenses)

First the frontend for any stripe integrated site will require an endpoint to fetch stripe public key -- to use on the frontend code, in order to verify account origins and esure that your stripe account is valid. Only then you can start processing payments; if the fetched key is incorrect or outdated nothing will work.

API Endpoint for fetching public and for generating a checkout session for stripe payment

I will include imports only on this snippet -- added comments for readablity.

```
import datetime
import json
import logging
import time
import threading

import boto3
import requests
import stripe
from django.conf import settings
from rest_framework.authentication import SessionAuthentication
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response
from rest_framework import status
from rest_framework.views import APIView

from aws.models import UserAwsCheckoutResource
from bids.models import ProductBid, UserProductBid
from notifications.models import UserNotification
from payments.exceptions import InvalidBid
from users.models import User

# ==============================================================
# Required for Checkout Service
# ==============================================================


logger = logging.getLogger(__name__)


class GetStripePublicKey(APIView):
    authentication_classes = [SessionAuthentication]
    permission_classes = [IsAuthenticated]

    def get(self, request, *args, **kwargs):
        data = {
            "stripePublicKey": settings.STRIPE_PUBLIC_KEY,
        }
        return Response(data=data, status=status.HTTP_200_OK)


class CreateStripeCheckoutSession(APIView):
    authentication_classes = [SessionAuthentication]
    permission_classes = [IsAuthenticated]

    def get_checkout_session(self, product_bid: ProductBid, bid_total, user):
        try:
            if settings.DEBUG:
                domain = f"http://{settings.ALLOWED_HOSTS[0]}:8000"
            else:
                domain = f"http://{settings.ALLOWED_HOSTS[0]}"

            # ==================================================
            # Get the real bid total of the product bid.
            # =================================================

            bid_total = float(bid_total)
            product_price = float(product_bid.product.price)
            bid_only = float(bid_total - product_price)

            # ===================================================
            # Handle bids that are not over the minimum threshold
            # ===================================================

            if int(bid_only) <= 25:
                raise InvalidBid("Invalid bid placement.")

            unit_amount = int(bid_only * 100)

            checkout_session = stripe.checkout.Session.create(
                api_key=settings.STRIPE_SECRET_KEY,
                payment_method_types=["card"],
                line_items=[
                    {
                        "price_data": {
                            "currency": "usd",
                            # Product bid amount includes bid fee
                            # via load bids command.
                            "unit_amount": unit_amount,
                            "product_data": {
                                "name": product_bid.product.name,
                            },
                        },
                        "quantity": 1,
                    },
                ],
                metadata={
                    "unit_amount": unit_amount,
                    "product_id": product_bid.pk,
                    "user_id": user.pk
                },
                mode="payment",
                success_url=f"{domain}/success-payment/{product_bid.pk}/",
                cancel_url=f"{domain}/bid/{product_bid.product.pk}/",
            )
        except Exception as error:
            return False, "Unable to generate checkout session."
        return True, checkout_session

    def post(self, request, *args, **kwargs):
        try:
            if request.data.get("bidId") is None:
                data = {"status": "error", "message": "Missing pid"}
                return Response(data=data, status=status.HTTP_400_BAD_REQUEST)

            if request.data.get("bidTotal") is None:
                data = {"status": "error", "message": "Missing bid amount"}
                return Response(data=data, status=status.HTTP_400_BAD_REQUEST)

            bid_total = request.data.get("bidTotal")
            product_bid = request.data["bidId"]
            product_bid = ProductBid.objects.get(pk=product_bid)
            release_date = product_bid.product.get_meta()["releaseDate"][:10]
            year, month, day = [int(x) for x in release_date.split("-")]
            release_date = datetime.datetime(year=year, month=month, day=day)

            # ==========================================
            # Handle a purchase past release date
            # ==========================================

            if datetime.datetime.today() > release_date:
                data = {
                    "status": "error",
                    "message": (
                        "Bid closed"
                    )
                }
                return Response(data=data, status=status.HTTP_400_BAD_REQUEST)

            checkout_session = self.get_checkout_session(
                product_bid, bid_total, request.user
            )

            # ==============================================
            # Handle an error on generating checkout session
            # ==============================================

            if not checkout_session[0]:
                data = {
                    "status": "error",
                    "message": settings.ERROR_CODES["002"]
                }
                return Response(data=data, status=status.HTTP_400_BAD_REQUEST)

            data = {"session": checkout_session[1].id}

        except Exception as error:
            data = {
                "status": "error",
                "message": settings.ERROR_CODES["001"]
            }
            return Response(data=data, status=status.HTTP_400_BAD_REQUEST)

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

For context I will add some database table models for ProductBid model, the name is self explanatory; this is for keeping track of bids made by certian users for certain products (Nike Sneakers)

Here are the models used on the checkout session for keeping track of things.

```
from django.db import models
from django.conf import settings

from sneakers.models import Product


class Bid(models.Model):
    inital_fee = models.FloatField(default=1.00, null=True)
    amount = models.FloatField(default=0.00, null=True)
    start_date = models.DateTimeField(auto_now_add=True)
    end_date = models.DateField(null=True)

    def __str__(self):
        return str(self.pk)


class ProductBid(models.Model):
    bid = models.ForeignKey(Bid, on_delete=models.CASCADE)
    product = models.OneToOneField(Product, on_delete=models.CASCADE)
    active_bid = models.BooleanField(default=False, null=True)
    def __str__(self):
        return f"Product - {self.product.name}"


class UserProductBid(models.Model):
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL, on_delete=models.CASCADE
    )
    product_bid = models.ForeignKey(ProductBid, on_delete=models.CASCADE)
    bid_amount = models.FloatField(default=0.00, null=True)
    created_at = models.DateTimeField(auto_now_add=True, null=True)

    def __str__(self):
        return f"{self.user.email}'s bid of {self.product_bid.product.name}"

    def get_product_bid_product_price(self):
        return self.product_bid.product.price

    def get_product_bid_amount(self):
        return self.bid_amount
```

Preety simple relational models, one model Bid, that handles the information for the bid made, which stores usefful things.

`Bid`

- Initial fee -- charged a default amount of 1.00 usd.
- Bid Amount -- well the amount that was bid.
- Start of the bid.
- End of the bid.

`ProductBid`

- A foreign key relation to a Bid object.
- Product which the bid is taking pair with.
- A boolean flag to know if the bid is active or not, for usage all around.

`UserProductBid`

- User, the user that placed this product bid.
- ProductBid, the product bid object is related here with the user.
- Bid Amount that this used placed.
- Timestamp of the bid.

Here is the Product model, which is imported on the above module and is a key component of relationship making.

```
class ProductImage(models.Model):
    class Meta:
        ordering = ["-created_at"]
        verbose_name = "Product Image"
        verbose_name_plural = "Product Images"

    name = models.CharField(max_length=255, null=True)
    image = models.ImageField(upload_to="uploads/products/%Y/%m/%d")
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.name


class Product(models.Model):

    class Meta:
        ordering = ["-created_at"]
        verbose_name = "Product"
        verbose_name_plural = "Products"

    brand = models.CharField(max_length=255, null=True)
    name = models.CharField(max_length=255, null=True)
    price = models.FloatField(default=0, null=True)
    product_id = models.CharField(max_length=255, null=True)
    meta = models.TextField(default="{}")
    slug = models.CharField(max_length=255, unique=True, null=True)
    created_at = models.DateTimeField(auto_now_add=True)
    release_date = models.DateTimeField()
    images = models.ManyToManyField(ProductImage, related_name="product_images")
```

These models are self explanatory, same goes as the field names for the columns on this database table, posting them here to better explain what is happening.

The last part is the post payment action for creating the ec2/t2.micro instances that the user just paid for, in addition to the cost of the sneaker that the user placed a bid on, and all the fees for this service.

I took some time to comment the code, this is a refactored version of the tests done on part 1 of this series, this code is executed when Stripe process the payment, and the payment is a success, it triggers this listener webhook.

This code then handles the creation of the t2/ec2 instances for the bots to execute this job of trying to get the new Sneaker before it sells out, replicating parellel processes executing the same action, with more probabilities of getting one pair of sneakers.

```
class SBListener(APIView):

    resource = boto3.resource('ec2', region_name="us-east-1")
    client = boto3.client('ec2', region_name="us-east-1")

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

        payload = request.body

        # ====================================================
        # Get the signature header to authenticate that the
        # origin of this request is Stripe.
        # ===================================================

        try:
```

```python
            sig_header = request.META["HTTP_STRIPE_SIGNATURE"]
            event = None
        except KeyError:
            return Response(
                data={"message": "Not Allowed"},
                status=status.HTTP_400_BAD_REQUEST
            )

        # =====================================================
        # Construct the stripe event to handle it accordingly
        # =====================================================

        try:
            event = stripe.Webhook.construct_event(
                payload,
                sig_header,
                settings.STRIPE_WEBHOOK_SECRET
            )

        except ValueError as error:
            data = {"status": "error", "code": str(error)}
            return Response(data=data, status=status.HTTP_400_BAD_REQUEST)
        except stripe.error.SignatureVerificationError as error:
            data = {"status": "error", "code": str(error)}
            return Response(data=data, status=status.HTTP_400_BAD_REQUEST)

        if event["type"] == "checkout.session.completed":
            session = event["data"]["object"]

            # =====================================================
            # Get all the things required to create bid record for
            # the customer on the db.
            # =====================================================

            try:
                customer_email = session["customer_details"]["email"]
                product_id = session["metadata"]["product_id"]
                user_id = session["metadata"]["user_id"]
                unit_amount = session["metadata"]["unit_amount"]
                product_bid = ProductBid.objects.get(pk=product_id)
                user = User.objects.get(pk=user_id)
                user_product_bid = UserProductBid.objects.create(
                    user=user,
                    product_bid=product_bid,
                    bid_amount=(float(unit_amount) / 100)
                )
            except Exception as error:
                if settings.DEBUG:
                    data = {"status": "error", "message": str(error)}
                else:
                    data = {"status": "error", "code": "006"}
                return Response(data=data, status=status.HTTP_400_BAD_REQUEST)

            self.user = user
            self.product = product_bid.product

            bid_amount = int(
                float(unit_amount) / 100
            )

            ##############################################################################
            # We have a lower end limit of 25 usd per bid on bots of this action.
            # This means everybody must buy 25 or more usd on bots (ec2/t2) instances.
            # We don't take crypto only USD, fucking noobs, suck my dick.
            ##############################################################################

            if bid_amount < 25:

                ##############################################################################
                # Here on production it handles refund, this is actually handled on checkout.
                # So its rare it actually hits this condition.
                ##############################################################################

                raise NotImplementedError()

            number_of_instances = (
                5 * len(
                    [x for x in range(1, bid_amount + 1) if x % 25 == 0]
                )
            )

            ############################################
            # Unlikeley but we shall verify eitherway.
            ############################################

            if number_of_instances <= 0:
                raise NotImplementedError()

            ############################################
            # Create instances.
            ############################################

            instances = self.resource.create_instances(
                ImageId="ami-027ece036eecc0eea",
                MaxCount=number_of_instances, # Using the number of instancers paid for.
                MinCount=1,
                InstanceType="t2.micro",
                TagSpecifications=[
                    {
                        "ResourceType": "instance",
                        "Tags": [
                            {
                                "Key": "USER",
                                "Value": user.email
                            }
                        ],
                    },
                ],
                KeyName="puppeteer",
                NetworkInterfaces=[
                    {
                        'SubnetId': settings.CHECKOUT_SUBNET_ID,
                        'DeviceIndex': 0,
                        'AssociatePublicIpAddress': True,
                        'Groups': [
                            settings.CHECKOUT_SECURITY_GROUP_ID,
                        ]
                    }
                ],
                UserData=settings.INIT
            )

            for instance in instances: # Create the resource to keep track and fetch puiblic IP for command and control of this bot.
                UserAwsCheckoutResource.cook_user_aws_checkout_resource(
                    **dict(
                        user=user,
                        instance_id=instance.id,
                        public_ip="-",
                        product=self.product,
                        meta=instance.meta.data,
                    )
                )

            #######################################################################################
            # This job is daemonized, meaning it runs in the background so it won't stall execution
            # Of this webhook
            ########################################################################################

            def step_one(instances, user):
                iterator = 0
                # Make range dynamic in case a user buys 200+ proxies
                while iterator <= 500:
                    current_instances = []

                    for instance in instances:
                        current_instances.append(
                            self.resource.Instance(id=instance.id)
                        )

                    states_and_ids = []
                    for each in current_instances:
                        states_and_ids.append(
                            [
                                each.state['Name'],
                                each.id
                            ]
                        )

                    logger.info(states_and_ids)
                    if not all(
                        list(
                            map(
                                lambda x: x[0] == "running", states_and_ids
                            )
                        )
                    ):
                        time.sleep(5)
                        iterator += 1
                        continue

                    # =================================
                    # Stop the instances
                    # =================================

                    for each in current_instances:
                        each.stop()

                    # ================================
                    # Administrative tasks
                    # ================================

                    for inst in current_instances:

                        try:
                            ip = inst.public_ip_address
                            if ip is None or ip == "":
                                ip = inst.private_ip_address
                                if ip is None or ip == "":
                                    ip = "-"
                        except Exception as error:
                            logger.error(str(error))
                            ip = "-"

                        try:
                            user_instance = UserAwsCheckoutResource.objects.get(
                                user=user,
                                instance_id=inst.id,
                            )
                        except Exception:
                            logger.exception("No userawscheckoutresource {inst.id}")
                            continue

                        # =====================================
                        # Save the new state and the ip to the
                        # user instance resource
                        # =====================================

                        user_instance.is_active = False
                        user_instance.public_ip = ip
                        user_instance.save()

                    break

            # ===========================================
            # Start the job.
            # ============================================

            job = threading.Thread(
                target=step_one,
                args=[
                    instances,
                    user
                ]
            )
            job.start()

            UserNotification.objects.create(
                user=self.user,
                data=json.dumps(
                    {
                        "status": "success",
                        "type": settings.EVENT_TYPES["RESOURCE_CREATED"],
                        "message": (
                            f"Proxy created for bid "
                            f"{user_product_bid.product_bid.product.name}"
                        )
                    }
                )
            )

            data = {"status": "success", "message": "Bid placed"}
            return Response(data=data, status=status.HTTP_200_OK)

        data = {"status": "error", "message": "Invalid type"}
        return Response(data=data, status=status.HTTP_400_BAD_REQUEST)
```

Lastly, here is an example replica of how a task would be sent to any ec2/t2 instance running purchase on behalf of the user using static data. The public IP would be obtained from a `UserAwsCheckoutResource` and replaced to be dynamically called. For example, call this endpoint for each Checkout resource for a UserProductBid with the ProductBid and Product data as functional arguments, or in this case as request body data.

```
class DemoCheckout(APIView):
    permission_classes = [IsAdmin, IsCronjobRunner]

    # Place Holder datas, these are updated inside the endpoint.
    address = {
        "id": 2,
        "type": "",
        "first_name": "Carlota",
        "last_name": "Gorda",
        "address_line_1": "Km 20",
        "address_line_2": " El Camino Real",
        "city": "Encinitas",
        "state": "CA",
        "postal_code": "90005",
        "country": "United States",
        "phone_number": "1234567891"
    }

    task_data = {
        "id": 1,
        "site_id": "1",
        "size": "7",
        "url": (
            "https://www.nike.com/t/revolution-5-womens-running-shoe-wide-"
            "hC41Vf/BQ3207-111"
        ),
        "billing_address_id": 2,
        "shipping_address_id": 2
    }

    def post(self, request, *args, **kwargs):
        """
        This is called via cron for each user.
        """
        # Here extract all task data and update the task data
        user = request.data.get("email")
        user = User.objects.get(email=email)
        self.address.update({"type": user.preferred_address})
        if self.address["type"] == "billing":

            self.address.update({
                "type": "billing",
                "address_line_1": user.billing_address_line_1,
                "address_line_2": user.billing_address_line_2,
                "city": user.billing_city,
                "state": user.billing_state,
                "postal_code": user.billing_postal_code,
                "country": user.billing_country.name,
                "email_address": user.email,
            })

        elif self.address["type"] == "shipping":
            self.address.update({
                "type": "shipping",
                "address_line_1": user.shipping_address_line_1,
                "address_line_2": user.shipping_address_line_2,
                "city": user.shipping_city,
                "state": user.shipping_state,
                "postal_code": user.shipping_postal_code,
                "country": user.shipping_country.name,
                "email_address": user.email,
            })


        with requests.Session() as session:

            url = "http://34.228.52.8:8888/checkoutService/Addresses"
            resp = session.post(url, data=self.address)

            # Your caller code should handle errors responses, for refund and complaints, to keep track of failures and errors.
```

```python
        if resp.status_code != 200:
            return Response(
                data={"message": "unable to create address"},
                status=status.HTTP_400_BAD_REQUEST
            )

        url = "http://18.207.255.212/checkoutService/Tasks"
        resp = session.post(url, data=self.task_data)
        if resp.status_code != 200:
            return Response(
                data={"message": "unable to create task"},
                status=status.HTTP_400_BAD_REQUEST
            )

        ##################################################
        # Card data is encrypted pre hitting a db commit.
        ##################################################

        final_data = {
            "card_friendly_name": user.card_friendly_name,
            "cc_number": user.cc_number_enc,
            "cc_expiry": user.cc_expiry_enc,
            "cc_code": user.cc_code_enc,
        }

        '''
        Final data looks like this, cc data is encrypted before even saving it, because PCI compliance requires it, otherwise, an audit can go wrong.
        final_data = {
            "card_friendly_name": "Mastercard",
            "cc_number": "gAAAAABgkKDt-7o6p8CvPYGKYxV3fOa2zE5jqx0h0lAEZ8f0oyOKe38qs7E8LuROn0MVicOMLNiimY6JEmu_-YqoA-xonwCIeaJ9OAIdsgcRcrrTBkORMEE=",
            "cc_expiry": "gAAAAABgkKE8q84txUVTYW9TVG1P0Yrkjfhrn5ggnb_YRarmdHJhJEB_K_wFMkMUGvyr9uesWpSxCT3HaA7ii224DjOHq5OdgQ==",
            "cc_code": "gAAAAABgkKG_PT4L5mw7Kbm1ULUZ3M3iqmCjqly_d5bky486vwo3uNdJdaPLQN435x1OIUvT0NPI7OVkUeYjrrCJCLPVfnffIA=="
        }
        '''
        url = "http://18.207.255.212/checkoutService/Tasks/1/start?id=1"
        resp = session.post(url, data=final_data)

        ############################################################################
        # Same here, caller code should handle these errors for refunds and track of
        # Failures.
        #############################################################################

        if resp.status_code != 200:
            return Response(
                data={"message": "unable to start task"},
                status=status.HTTP_400_BAD_REQUEST
            )

        return Response(
            data={
                "status": "success",
                "message": "address_created"
            },
            status=status.HTTP_200_OK
        )
```

For clarity here is `UserAwsCheckoutResource` code -- it has some methods that are rareley used but I still left them there.

```python
import json
import ast

from django.db import models
from django.conf import settings
from django.core.exceptions import ObjectDoesNotExist

from sneakers.models import Product


class UserAwsCheckoutResource(models.Model):

    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    instance_id = models.CharField(max_length=255, null=True)
    public_ip = models.CharField(max_length=255, null=True)
    is_active = models.BooleanField(default=False, null=True)
    created_at = models.DateTimeField(auto_now_add=True)
    meta = models.TextField(default='{}', null=True)
    product = models.ForeignKey(Product, on_delete=models.CASCADE, null=True)

    @classmethod
    def cook_user_aws_checkout_resource(cls, **kwargs):
        try:
            resource = cls.objects.get(instance_id=kwargs["instance_id"])
        except (cls.DoesNotExist, ObjectDoesNotExist):
            resource = cls.objects.create(**kwargs)
        return resource

    def get_meta(self):
        try:
            return json.loads(self.meta)
        except json.JSONDecodeError:
            return ast.literal_eval(self.meta)
        return {}

    def get_instance_state(self):
        try:
            return settings.EC2_RESOURCE.Instance(self.instance_id).state
        except Exception as error:
            return bool(0)
        return bool(1)

    def _terminate(self):
        try:
            settings.EC2_RESOURCE.Instance(self.instance_id).terminate()
        except Exception as error:
            return bool(0)
        return bool(1)
    def _boot(self):
        try:
            settings.EC2_RESOURCE.Instance(self.instance_id).start()
        except Exception as error:
            return bool(0)
        return bool(1)
```

Part two includes key components of the program, it does not make sense for me to post entire solutions because I am just being nice here and providing a "more than what you see on the web how tos", because we made money out of this already around -- 2 years ago.

If you want full solution or custom one, contact me directly, but bring a good budget.

For part three I will write about how to secure users credit card information in order to comply with PCI standards, how to store your key to encrypt users credit cards as secure as possible. In addition I am going to write about how to scrape nike products, since nike.com uses React and Nike uses a local store "storage" which often wont allow scrappers to scrape, because it dinamically loads, and selenium usage because of bot detection frontend libraries, will fuck you up.However, React is shit, and there's lots of flaws, and this makes it okay way to scrape React sites if you know how to get local storage store object and just traverse the 'store;.

This post is part of a series, check out the other parts of this series of notes

Using ec2 instances as sneaker bid bots pt 3.

Using ec2 instances as sneaker bid bots pt 1.

Related Notes

1) Real Time Language Translation Agent System for Call Centers

Date published: 2023-11-16

voip telephony python systems

2) Using ec2 instances as sneaker bid bots pt 2.

Date published: 2023-11-27

bots python aws series

3) Whatsapp chatbot with Python and Twilio

Date published: 2023-11-15

bots python whatsapp business

4) Backend Celery task manager dashboard via Flower

Date published: 2023-10-29

backend tasks python

5) Using ec2 instances as sneaker bid bots pt 3.

Date published: 2023-11-28

bots python aws series

6) Using ec2 instances as sneaker bid bots pt 1.

Date published: 2023-11-21

bots python aws series