Documentation for Review Prediction – Backend

1. Project Overview

- This backend is built using Django REST Framework (DRF).
- It provides REST APIs for review prediction.
- The backend integrates with an AI model service (built by the AI team) to analyze reviews.
- The frontend team will consume these APIs to display predictions.

2. Project Structure

```
project-root/
│
├── backend/
│   ├── food_review_project/     # Main Django project
│   │   ├── reviews/             # Django app
│   │   │   ├── models.py
│   │   │   ├── serializers.py
│   │   │   ├── views.py
│   │   │   ├── urls.py
│   │   │   └── admin.py
│   │   │
│   │   ├── settings.py
│   │   ├── urls.py
│   │   └── wsgi.py
│   │
│   ├── manage.py               # Django management script
│
└── venv/                       # Virtual environment
```

3. Backend Responsibilities

The backend handles:

1. Receiving review input (text) from users via API.
2. Calling the AI model service to predict rating.
3. Storing the review in the database.
4. Formatting the API response.

4. Review Model (Database)

```
from django.db import models
class Review(models.Model):
    text = models.TextField()
    predicted_rating = models.FloatField()
    created_at = models.DateTimeField(auto_now_add=True)
    def __str__(self):
        return f"Rating: {self.predicted_rating} | {self.text[:150]}"
```

- **text**: The review written by user.
- **predicted_rating**: Rating predicted by AI model.
- **created_at**: Timestamp of submission.

5. serializers.py

```
from rest_framework import serializers

from .models import Review
```

```
class ReviewSerializer(serializers.ModelSerializer):
    """
    Serializer for Review model with enhanced fields and validation
    """
    rating_display = serializers.SerializerMethodField()
    text_preview = serializers.SerializerMethodField()

    class Meta:
        model = Review
        fields = [
            'id',
            'text',
            'predicted_rating',
            'created_at',
            'rating_display',
            'text_preview'
        ]
        read_only_fields = ['id', 'created_at', 'rating_display', 'text_preview']


    def get_rating_display(self, obj):

        """Format rating for display"""

        return f"{round(obj.predicted_rating, 1)}/5.0"


    def get_text_preview(self, obj):

        """Return first 150 characters; add '...' only if longer than 150"""

        if len(obj.text) > 150:

            return obj.text[:150] + "..."

        return obj.text


    def validate_text(self, value):

        """Validate review text"""

        if not value or not value.strip():

            raise serializers.ValidationError("Review text cannot be empty.")

        if len(value.strip()) < 5:
            raise serializers.ValidationError("Review text must be at least 5 characters
long.")
        if len(value) > 5000:
            raise serializers.ValidationError("Review text cannot exceed 5000
characters.")
        return value.strip()

    def validate_predicted_rating(self, value):
        """Validate predicted rating"""
        if value < 1.0 or value > 5.0:
            raise serializers.ValidationError("Rating must be between 1.0 and 5.0.")
        return round(value, 2)

class ReviewListSerializer(serializers.ModelSerializer):
    """
    Lightweight serializer for listing reviews (optimized for performance)
    """
    rating_display = serializers.SerializerMethodField()
```

```
text_preview = serializers.SerializerMethodField()


class Meta:
    model = Review
    fields = [
        'id',
        'text_preview',
        'predicted_rating',
        'created_at',
        'rating_display'
    ]


def get_rating_display(self, obj):
    """Format rating for display"""
    return f"{round(obj.predicted_rating, 1)}/5.0"
def get_text_preview(self, obj):
    """Get a short preview of the review text truncated to 150 characters"""
```

```
        return obj.text[:150] + "..." if len(obj.text) > 150 else obj.text


class ReviewCreateSerializer(serializers.ModelSerializer):
    """
    Serializer for creating new reviews (used in prediction endpoint)
    """
    class Meta:
        model = Review
        fields = ['text', 'predicted_rating']

    def validate_text(self, value):
        """Validate review text for creation"""
        if not value or not value.strip():
            raise serializers.ValidationError("Review text is required.")
        cleaned = value.strip()
        if len(cleaned) < 3:
            raise serializers.ValidationError("Review text is too short.")
        return cleaned


    def create(self, validated_data):
        """Create review with additional processing"""
        return Review.objects.create(**validated_data)
```

6. views.py

```
import re
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
from django.db.models import Count, Avg
from django.utils import timezone
from .models import Review
from .serializers import ReviewSerializer
from food_review_project.bert_model.predict import predict_rating
def format_review_date(created_at):
    """Format the review date like 'Reviewed in India on 30 June 2025'"""
    if not created_at:
        return "Date not available"


    if timezone.is_aware(created_at):
        local_time = timezone.localtime(created_at)
    else:
        local_time = created_at


    formatted_date = local_time.strftime("%d %B %Y")
    return f"Reviewed in India on {formatted_date}"


def validate_review_text(text: str) -> tuple[bool, str]:
    if not text or not text.strip():
        return False, "Review text is required"


    cleaned_text = text.strip()
```

```
if len(cleaned_text) < 10:
    return False, "Review must be at least 10 characters long"
if len(cleaned_text) > 5000:
    return False, "Review cannot exceed 5000 characters"


words = cleaned_text.split()
if len(words) < 3:
    return False, "Review must contain at least 3 words"
if cleaned_text.isdigit():
    return False, "Review cannot be just numbers"
if len(words) < 3 and len(cleaned_text) < 15:
    return False, "Please provide a more detailed review"


meaningful_chars = sum(1 for c in cleaned_text if c.isalpha())
if meaningful_chars < 5:
    return False, "Review must contain meaningful text about the food"
invalid_patterns = ['test', 'asdf', '123', 'abc', 'hi', 'hello', 'ok', 'good', 'nice']
```

```
    if cleaned_text.lower().strip() in invalid_patterns:
        return False, "Please provide a genuine review about the food"


    food_keywords = [
        'food', 'taste', 'delicious', 'flavor', 'meal', 'dish', 'restaurant', 'eat',
        'ate', 'cook', 'cooked', 'spicy', 'sweet', 'salty', 'hot', 'cold', 'fresh',
        'pizza', 'burger', 'rice', 'chicken', 'beef', 'fish', 'vegetable', 'fruit',
        'drink', 'coffee', 'tea', 'juice', 'bread', 'cake', 'dessert'
    ]
    if len(words) <= 5:
        contains_food_context = any(keyword in cleaned_text.lower() for keyword in
food_keywords)
        if not contains_food_context:
            return False, "Please write a review about food (e.g., 'The pizza was
amazing with great cheese')"
    return True, ""


def clean_text(text: str) -> str:
    text = re.sub(r'[^A-Za-z0-9\s.,!?\'"-]', '', text)

    text = re.sub(r'\s+', ' ', text).strip()

    return text



def generate_customer_insights(reviews):

    """Generate simplified customer insights without category breakdowns"""

    if not reviews:

        return {

            "summary": "No customer feedback available yet.",

            "key_points": [],

            "common_themes": [],

            "generated_from": "No reviews available",

            "overall_sentiment": "neutral",

            "confidence_score": 0,

            "note": "confidence_score is generated by backend for internal use;
frontend and AI team may ignore it if not needed."}

    total_reviews = len(reviews)
    avg_rating = sum(r.predicted_rating for r in reviews) / total_reviews



    if avg_rating >= 4.0:
        summary = f"Customers are generally very satisfied with the food. Based on {total_reviews} reviews, most customers recommend it."
        overall_sentiment = "positive"
    elif avg_rating >= 3.0:
        summary = f"Customers have mixed experiences with the food. Based on {total_reviews} reviews, many customers recommend it."
        overall_sentiment = "mixed"
    else:
        summary = f"Customers have expressed concerns about the food. Based on {total_reviews} reviews, improvements are needed."
        overall_sentiment = "negative"



    key_points = []
    for r in reviews[:5]:
        snippet = r.text.strip()
        if len(snippet) > 100:
            snippet = snippet[:97] + "..."
        key_points.append(snippet)

    from collections import Counter
    words = [word.lower() for r in reviews for word in r.text.split()]
    common_words = [word for word, count in Counter(words).most_common(5) if
len(word) > 3]
    confidence_score = min(total_reviews * 2.5 + abs(avg_rating - 3.0) * 10, 95)
    return {
        "summary": summary,
        "key_points": key_points,
        "common_themes": common_words,
        "generated_from": f"Analysis of {total_reviews} customer reviews",
        "overall_sentiment": overall_sentiment,
        "confidence_score": round(confidence_score, 1),
        "note": "confidence_score is generated by backend for internal use; frontend
and AI team may ignore it if not needed."
```

```
    }



@api_view(['POST'])

def predict_review(request):

    review_text = request.data.get('review_text')

    if not review_text:

        return Response(

            {"error": "review_text is required", "suggestion": "Please provide a review
about the food"},

            status=status.HTTP_400_BAD_REQUEST

        )



    is_valid, error_message = validate_review_text(review_text)

    if not is_valid:

        return Response(

            {

                "error": error_message,
                "suggestion": "Please write a detailed review about the food (e.g., 'The
pizza was delicious with great cheese and crispy crust')",
                "examples": [
                    "The food was amazing, especially the pasta with rich tomato sauce",
                    "Great service and the burger was perfectly cooked and juicy",
                    "Disappointing meal, the chicken was dry and lacked flavor"
                ]
            },
            status=status.HTTP_400_BAD_REQUEST
        )

    original_text = review_text
    cleaned_text = clean_text(review_text)
    if not cleaned_text:
        return Response({"error": "Invalid review text after cleaning"},
status=status.HTTP_400_BAD_REQUEST)

    try:
        predicted_rating = predict_rating(cleaned_text)

        analysis = {

            "review_text": original_text,

            "cleaned_text": cleaned_text,

            "predicted_rating": round(predicted_rating, 2),

            "rating_out_of_5": f"{round(predicted_rating, 1)}/5.0 ⭐",

            "timestamp": None

        }

    except Exception as e:

        return Response(

            {"error": f"Prediction failed: {str(e)}"},

            status=status.HTTP_500_INTERNAL_SERVER_ERROR

        )

    try:

        review = Review.objects.create(text=original_text,
predicted_rating=predicted_rating)

        analysis["id"] = review.id
        analysis["timestamp"] = review.created_at.isoformat() if review.created_at
else None

        analysis["formatted_date"] = format_review_date(review.created_at)

        serializer = ReviewSerializer(review)

        analysis["database_record"] = serializer.data

    except Exception as e:

        analysis["database_error"] = f"Failed to save to database: {str(e)}"

    return Response(analysis, status=status.HTTP_201_CREATED)



@api_view(['GET'])

def review_list(request):

    try:

        page = int(request.GET.get('page', 1))

        page_size = int(request.GET.get('page_size', 50))

        offset = (page - 1) * page_size



        all_reviews = Review.objects.all().order_by('-id')
```

```
total_reviews = all_reviews.count()

reviews = all_reviews[offset:offset + page_size]



enhanced_reviews = []

for review in reviews:

    serializer = ReviewSerializer(review)

    review_data = serializer.data

    full_stars = int(review.predicted_rating)

    half_star = 1 if (review.predicted_rating - full_stars) >= 0.5 else 0

    star_display = "⭐" * full_stars

    if half_star:

        star_display += "⭐"

    review_data.update({

        "rating_display": f"{round(review.predicted_rating, 1)}/5.0",

        "stars": star_display,

        "star_count": int(round(review.predicted_rating)),

        "formatted_date": format_review_date(review.created_at)
        })

        enhanced_reviews.append(review_data)

    average_rating =
round(all_reviews.aggregate(avg=Avg("predicted_rating"))["avg"] or 0, 2)

    distribution =
all_reviews.values("predicted_rating").annotate(count=Count("id"))

    star_counts = {i: 0 for i in range(1, 6)}

    for d in distribution:

        star_counts[int(round(d["predicted_rating"]))] += d["count"]



    star_percent = {

        star: round((count / total_reviews) * 100) if total_reviews else 0

        for star, count in star_counts.items()

    }

    total_pages = (total_reviews + page_size - 1) // page_size

    has_next = page < total_pages

    has_previous = page > 1

    customer_insights = generate_customer_insights(all_reviews)
```

```
response_data = {
    "reviews": enhanced_reviews,
    "pagination": {
        "current_page": page,
        "page_size": page_size,
        "total_pages": total_pages,
        "total_count": total_reviews,
        "has_next": has_next,
        "has_previous": has_previous
    },
    "summary": {
        "total_reviews": total_reviews,
        "average_rating": average_rating,
        "reviews_on_page": len(enhanced_reviews),
        "star_distribution": star_counts,
        "star_distribution_percent": star_percent
    },
            "customer_insights": customer_insights,
            "status": "success"
        }
        return Response(response_data, status=status.HTTP_200_OK)
    except ValueError as e:
        return Response({
            "error": f"Invalid pagination parameters: {str(e)}",
            "reviews": [], "pagination": {"current_page": 1, "total_count": 0}, "status":
"error"
        }, status=status.HTTP_400_BAD_REQUEST)
    except Exception as e:
        return Response({
            "error": f"Failed to fetch reviews: {str(e)}", "status": "error"
        }, status=status.HTTP_500_INTERNAL_SERVER_ERROR)
@api_view(['GET'])
def customer_insights(request):
    """Dedicated endpoint for customer insights"""
    try:
        reviews = Review.objects.all()
        insights = generate_customer_insights(reviews)
        return Response({
            "customer_insights": insights,
            "status": "success"
        })
    except Exception as e:
        return Response({
            "error": f"Failed to generate customer insights: {str(e)}",
            "status": "error"
        }, status=status.HTTP_500_INTERNAL_SERVER_ERROR)


@api_view(['GET'])
def review_status_summary(request):
    """Returns only the summary data for the status bar"""
    reviews = Review.objects.all()

    total_reviews = reviews.count()

    if total_reviews == 0:

        return Response({

            "summary": {

                "total_reviews": 0,

                "average_rating": 0,

                "star_distribution": {},

                "star_distribution_percent": {}

            },

            "status": "success"

        }

    average_rating = round(reviews.aggregate(avg=Avg("predicted_rating"))["avg"]
or 0, 2)

    star_counts = {i: 0 for i in range(1, 6)}

    for r in reviews:

        star = round(r.predicted_rating)
        star = min(max(star, 1), 5)
        star_counts[star] += 1
    star_percent = {star: round((count / total_reviews) * 100, 1) for star, count in
star_counts.items()}
    return Response({
        "summary": {
            "total_reviews": total_reviews,
            "average_rating": average_rating,
            "star_distribution": star_counts,
            "star_distribution_percent": star_percent
        },
        "status": "success"
    })


@api_view(['DELETE'])
def clear_all_reviews(request):
    try:
        count = Review.objects.count()

        Review.objects.all().delete()

        return Response({"message": f"Deleted {count} reviews successfully"})

    except Exception as e:

        return Response({"error": f"Failed to clear reviews: {str(e)}"},
status=status.HTTP_500_INTERNAL_SERVER_ERROR)
```

7. urls.py

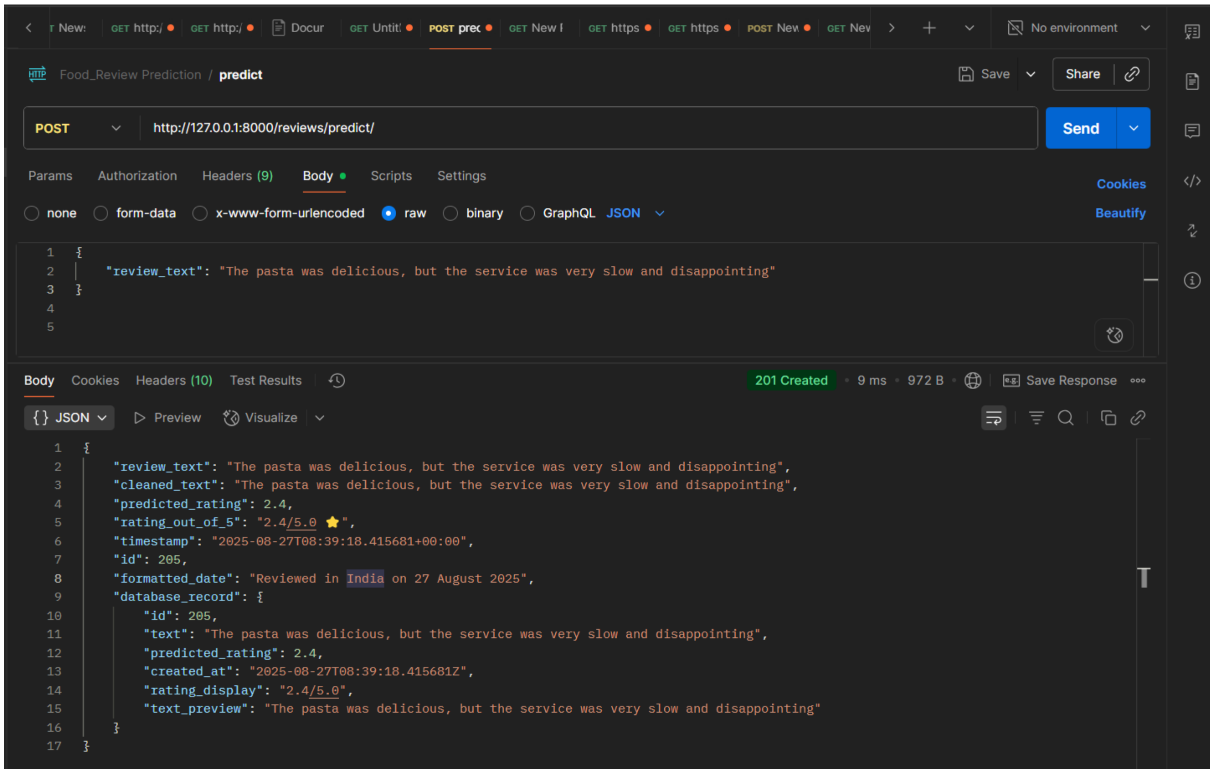
```
from django.urls import path

from . import views

urlpatterns = [

    path('predict/', views.predict_review, name='predict_review'),

    path('', views.review_list, name='review_list'),

    path('status-bar/', views.review_status_summary, name='review_status_bar'),

    path('customer-insights/', views.customer_insights, name='customer_insights'),

]
```

8. API Workflow

1. User submits review text → **POST /reviews/predict/**
2. Backend sends text to AI model service → gets prediction.
3. Backend saves review in database.
4. Backend returns JSON response.

9. API Testing (Screenshots)

<u>Predict Review</u> (POST /reviews/predict/)



POST http://127.0.0.1:8000/reviews/predict/

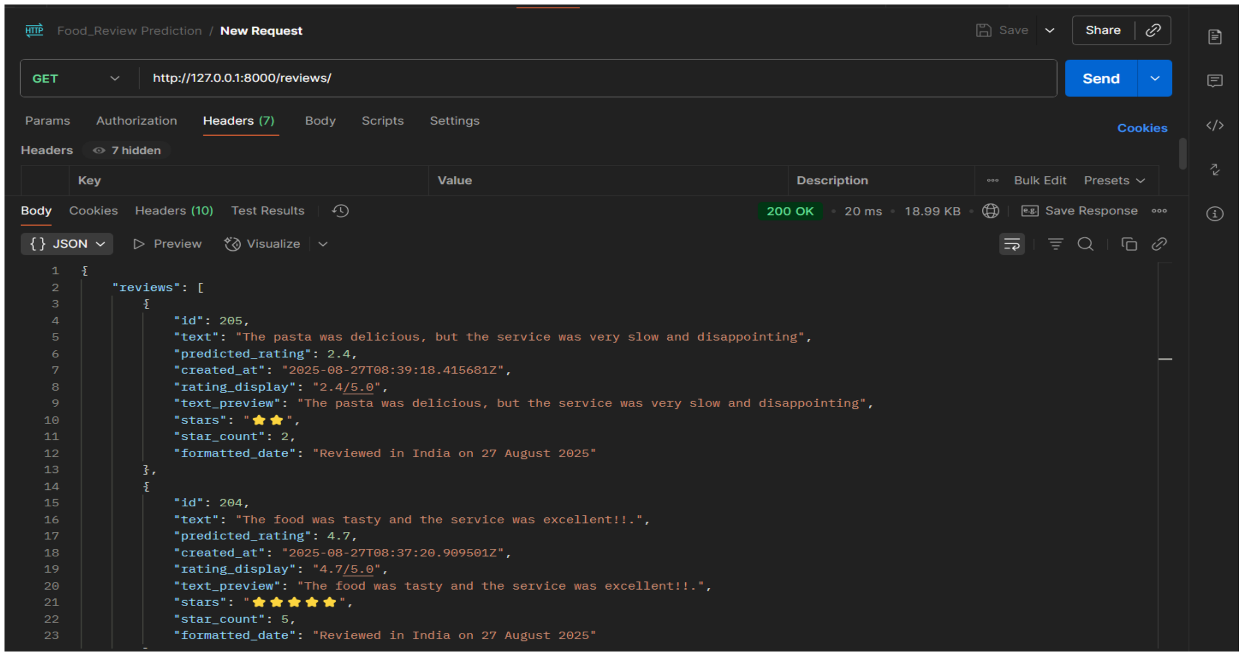
Body: [✓ raw] JSON

```
{
    "review_text": "The pasta was delicious, but the service was very slow and disappointing"
}
```

201 Created • 9 ms • 972 B

```
{
    "review_text": "The pasta was delicious, but the service was very slow and disappointing",
    "cleaned_text": "The pasta was delicious, but the service was very slow and disappointing",
    "predicted_rating": 2.4,
    "rating_out_of_5": "2.4/5.0 ⭐",
    "timestamp": "2025-08-27T08:39:18.415681+00:00",
    "id": 205,
    "formatted_date": "Reviewed in India on 27 August 2025",
    "database_record": {
        "id": 205,
        "text": "The pasta was delicious, but the service was very slow and disappointing",
        "predicted_rating": 2.4,
        "created_at": "2025-08-27T08:39:18.415681Z",
        "rating_display": "2.4/5.0",
        "text_preview": "The pasta was delicious, but the service was very slow and disappointing"
    }
}
```

Get Reviews (GET /reviews/)

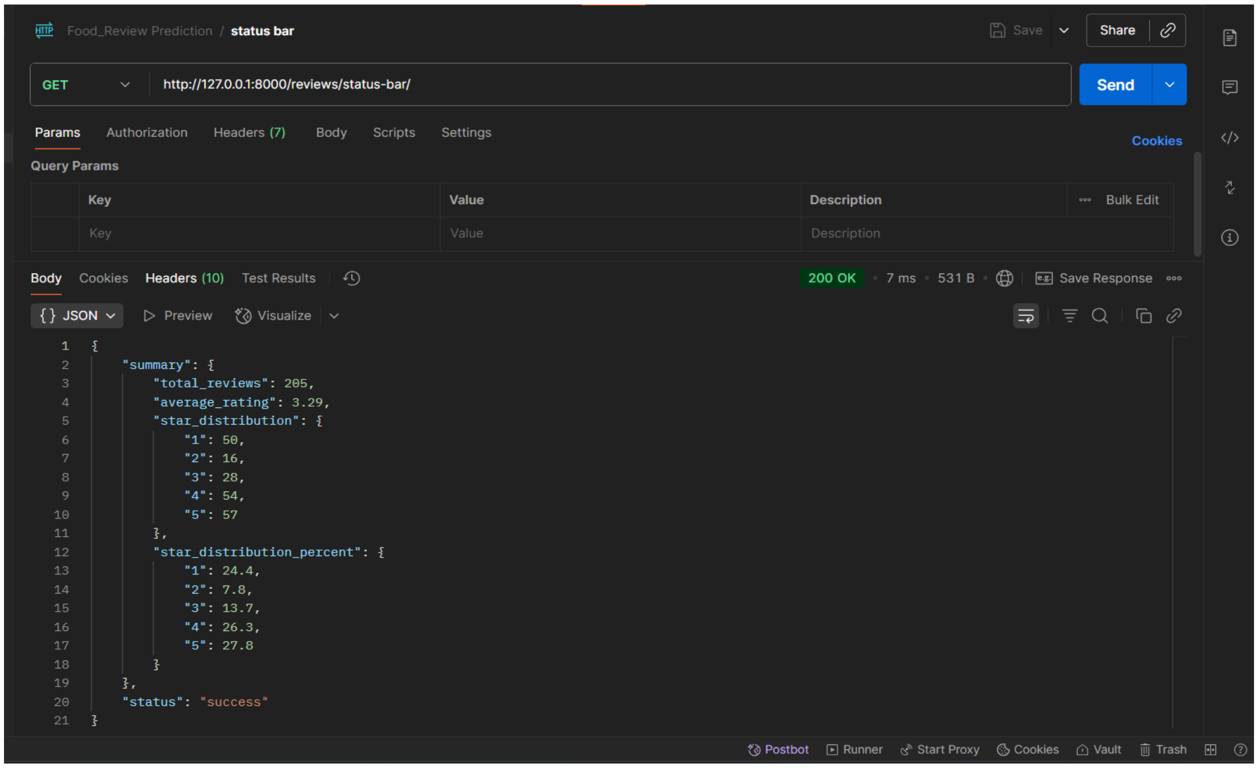


GET http://127.0.0.1:8000/reviews/

200 OK · 20 ms · 18.99 KB

```
{
    "reviews": [
        {
            "id": 205,
            "text": "The pasta was delicious, but the service was very slow and disappointing",
            "predicted_rating": 2.4,
            "created_at": "2025-08-27T08:39:18.415681Z",
            "rating_display": "2.4/5.0",
            "text_preview": "The pasta was delicious, but the service was very slow and disappointing",
            "stars": "⭐⭐",
            "star_count": 2,
            "formatted_date": "Reviewed in India on 27 August 2025"
        },
        {
            "id": 204,
            "text": "The food was tasty and the service was excellent!!.",
            "predicted_rating": 4.7,
            "created_at": "2025-08-27T08:37:20.909501Z",
            "rating_display": "4.7/5.0",
            "text_preview": "The food was tasty and the service was excellent!!.",
            "stars": "⭐⭐⭐⭐⭐",
            "star_count": 5,
            "formatted_date": "Reviewed in India on 27 August 2025"
```

Status Bar (GET /reviews/status-bar)

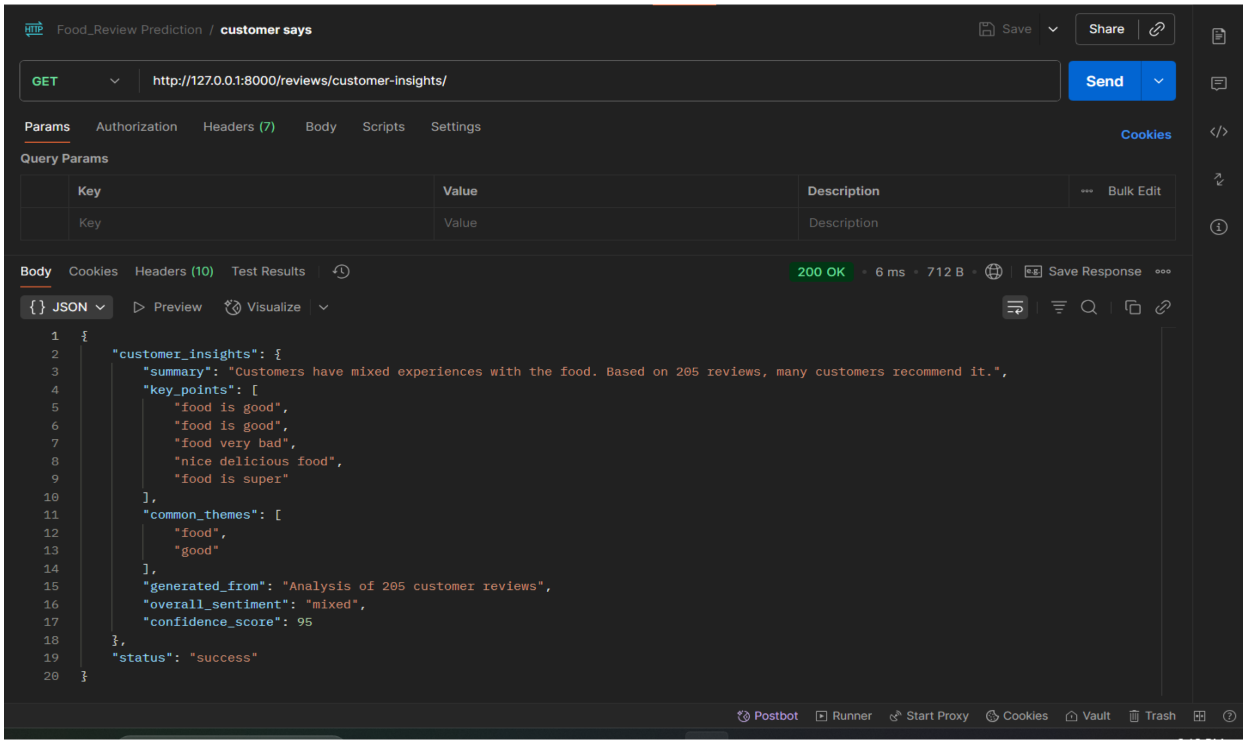


GET http://127.0.0.1:8000/reviews/status-bar/

200 OK · 7 ms · 531 B

```
{
    "summary": {
        "total_reviews": 205,
        "average_rating": 3.29,
        "star_distribution": {
            "1": 50,
            "2": 16,
            "3": 28,
            "4": 54,
            "5": 57
        },
        "star_distribution_percent": {
            "1": 24.4,
            "2": 7.8,
            "3": 13.7,
            "4": 26.3,
            "5": 27.8
        }
    },
    "status": "success"
}
```

Customer Insights (GET /reviews/customers-insights)



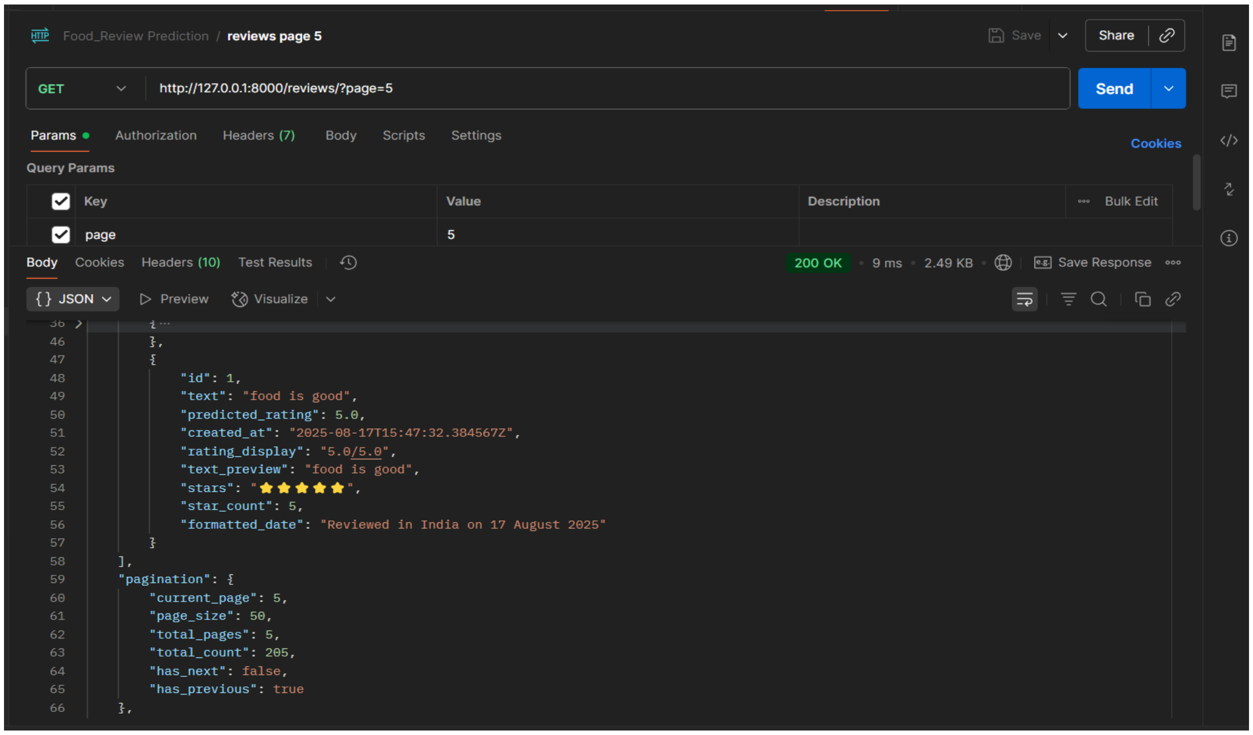
GET http://127.0.0.1:8000/reviews/customer-insights/

Key	Value	Description
Key	Value	Description

Body — 200 OK · 6 ms · 712 B — JSON

```
{
    "customer_insights": {
        "summary": "Customers have mixed experiences with the food. Based on 205 reviews, many customers recommend it.",
        "key_points": [
            "food is good",
            "food is good",
            "food very bad",
            "nice delicious food",
            "food is super"
        ],
        "common_themes": [
            "food",
            "good"
        ],
        "generated_from": "Analysis of 205 customer reviews",
        "overall_sentiment": "mixed",
        "confidence_score": 95
    },
    "status": "success"
}
```

Pagination



GET http://127.0.0.1:8000/reviews/?page=5

	Key	Value	Description
✓	page	5	

Body — 200 OK · 9 ms · 2.49 KB — JSON

```
        {…
        },
        {
            "id": 1,
            "text": "food is good",
            "predicted_rating": 5.0,
            "created_at": "2025-08-17T15:47:32.384567Z",
            "rating_display": "5.0/5.0",
            "text_preview": "food is good",
            "stars": "⭐⭐⭐⭐⭐",
            "star_count": 5,
            "formatted_date": "Reviewed in India on 17 August 2025"
        }
    ],
    "pagination": {
        "current_page": 5,
        "page_size": 50,
        "total_pages": 5,
        "total_count": 205,
        "has_next": false,
        "has_previous": true
    },
```

10. Installation & Setup Instructions

1. Clone the repository.
2. Create and activate a virtual environment:
3. python -m venv venv
4. source venv/bin/activate   # Mac/Linux
5. venv\Scripts\activate   # Windows
6. Install dependencies:
7. pip install -r requirements.txt
8. Run database migrations:
9. python manage.py migrate
10. Start development server:
11. python manage.py runserver

11. API Endpoints Summary

Method	**Endpoint**	**Description**
POST	/reviews/predict/	Submit a review for prediction
GET	/reviews/	List all reviews (paginated).
GET	/reviews/status-bar	Get summary data for status bar
GET	/reviews/customer-insights	Get customer insights summary

12. Dependencies / Requirements

- **Python**: 3.9+
- **Django**: 5.x
- **Django REST Framework**: 3.x
- **Requests** (for external AI API calls)