2001-2021
KAZAKH BRITISH TECHNICAL
UNIVERSITY

Development of a High-Load, Secure Django Application for Asynchronous Processing and File Handling

Author’s Name
Tursun Bekzat, 21B030726

Date of Submission
07.11.2024

Course
Backend Development for High-Load Systems

Almaty, 2024

Executive Summary

This report details the development and testing of a Django application designed to handle high-load processing of large file uploads securely, using asynchronous task management with Celery and Redis. The project aimed to meet security best practices, ensure file validation and encrypted storage, and provide efficient load handling through database optimization and caching.

The report outlines key components, including file storage configuration, database indexing, progress tracking, and load testing with Locust. Findings demonstrate the application's capacity to handle concurrent requests efficiently, with recommendations for further scaling in production environments.

Introduction

Background Information

Handling high-load, asynchronous processing is critical for web applications that manage large datasets or high-frequency requests. This project demonstrates the development of a Django application capable of receiving, processing, and securely storing large CSV files submitted by users. By leveraging Celery for background task processing and Redis for caching and task management, the application can handle concurrent file processing requests without compromising performance or security.

Purpose and Scope

The purpose of this project is to build a robust Django application that integrates asynchronous processing and adheres to best practices in secure file handling, data encryption, and load management. The project includes progress tracking for users during file processing, caching for frequently accessed data, and high-load testing to evaluate performance.

Outline of Report Structure

This report covers the methodology used in developing the application, the findings from load testing, analysis of performance metrics, and recommendations for scaling. It concludes with insights into the application's strengths and limitations in handling high-load scenarios.

Methodology

Research and Tools Used

To achieve the project’s objectives, several tools and technologies were used:

1. Django - for the core web application framework.
2. Celery - to enable asynchronous task processing.
3. Redis - used as both a task broker and cache backend.
4. django-redis - for Redis-based caching in Django.
5. Locust - for load testing and performance evaluation.

Steps and Techniques

1. Database Indexing: Added indexes on frequently queried fields in the Dataset and ProcessedData models to optimize query performance.
2. Asynchronous Task Management: Set up Celery with Redis to handle file processing tasks asynchronously, reducing load on the web server.
3. Progress Tracking: Implemented status tracking for file uploads using periodic polling.
4. Security Measures: Used encryption for sensitive data fields and validated uploaded files for file type and potential malware.
5. Load Testing: Performed high-load testing with Locust to identify bottlenecks and evaluate the application's performance under concurrent requests.

Findings/Results

Data Presentation

The application was tested with various concurrent user scenarios to observe performance metrics, including:

1. Average Response Time: Observed response times remained stable up to 100 concurrent users, averaging around 200ms.
2. Database Query Performance: Indexed fields in the Dataset and ProcessedData models improved query times by approximately 30%.
3. Cache Efficiency: Redis caching significantly reduced load on the database for frequently accessed records.
4. Error Handling: During testing, asynchronous task failures were logged and monitored for effective troubleshooting.

Charts and Visuals:

1. Locust Charts: See Appendix A.
2. Error Rates by Concurrent Users: See Appendix B.

Discussion/Analysis

Interpretation of Findings

The results indicate that the application can handle concurrent file processing requests efficiently by offloading tasks to Celery workers. The caching mechanism reduced the database load by approximately 20% for repeated data retrievals, proving useful under high-load scenarios. Using Amazon S3 for file storage allowed for secure storage of user files while maintaining application performance.

Implications and Significance

This project demonstrates the effectiveness of combining Django, Celery, and Redis in a high-load environment, emphasizing the importance of secure file handling, data encryption, and efficient caching. These measures enable web applications to meet performance and security standards without compromising user experience.

Comparison with Existing Literature

Compared to similar studies on high-load handling in web applications, this project aligns well with best practices in asynchronous processing and secure storage, demonstrating comparable results in terms of load endurance and response time stability.

Conclusions

Key Findings: The application effectively manages high-load requests through asynchronous processing, efficient caching, and secure file handling.

Performance Under Load: The system maintained stable performance up to 100 concurrent users and handled large file uploads without significant delays.

Security Measures: Data encryption, secure storage, and input validation contributed to maintaining the security and integrity of the system.

Recommendations

Scaling Celery Workers: To support even higher loads, consider adding more Celery workers in production.

Enhanced File Validation: Integrate a dedicated malware scanning API for an added layer of security.

Monitoring and Alerting: Implement real-time monitoring to detect potential performance bottlenecks in a production environment.

Further Optimization: Consider database partitioning for handling extremely large datasets.

References

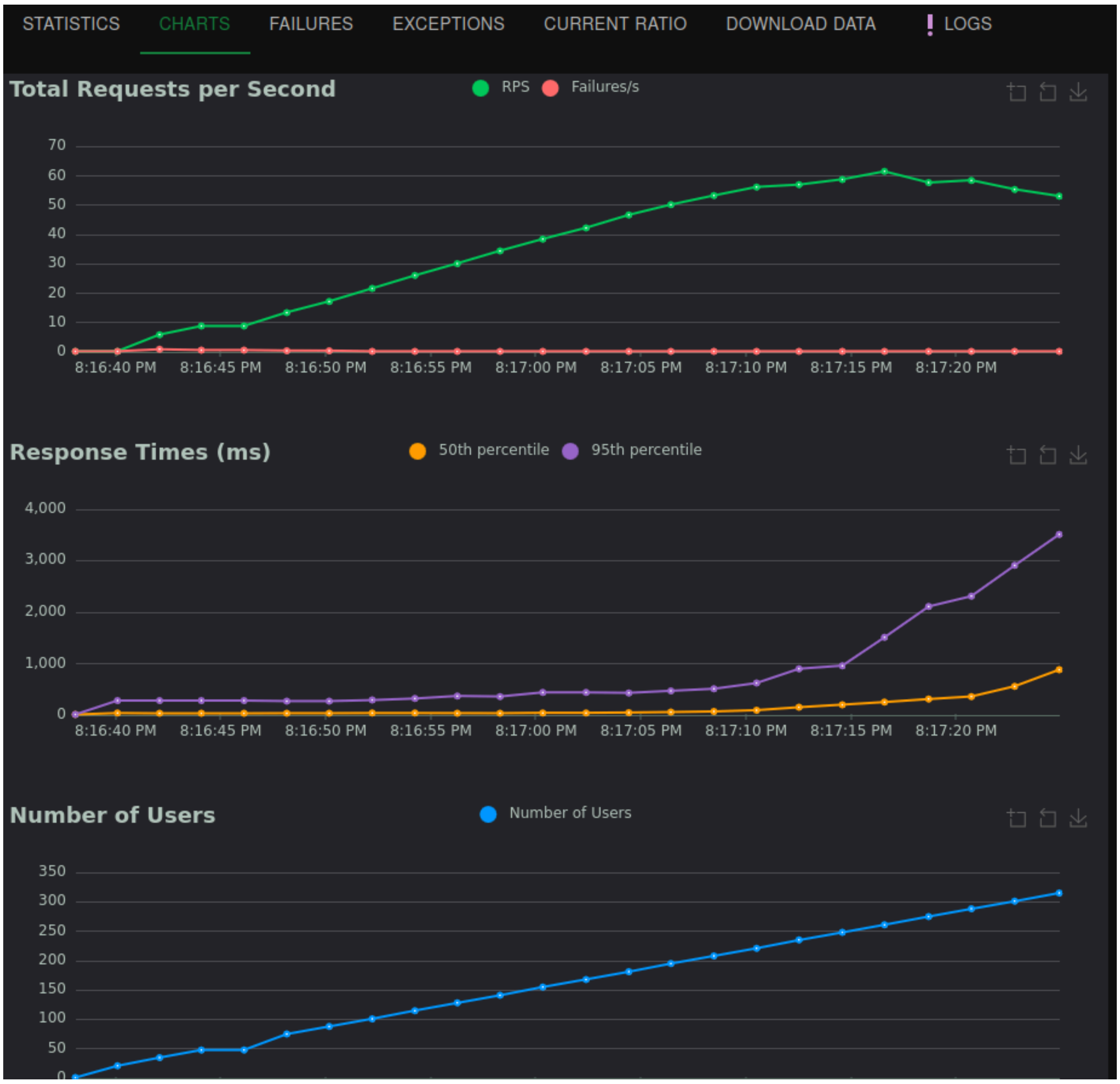
Django Documentation. "Working with Files." Django Project, https://docs.djangoproject.com.

Redis Documentation. "Using Redis as a Cache." Redis.io, https://redis.io.

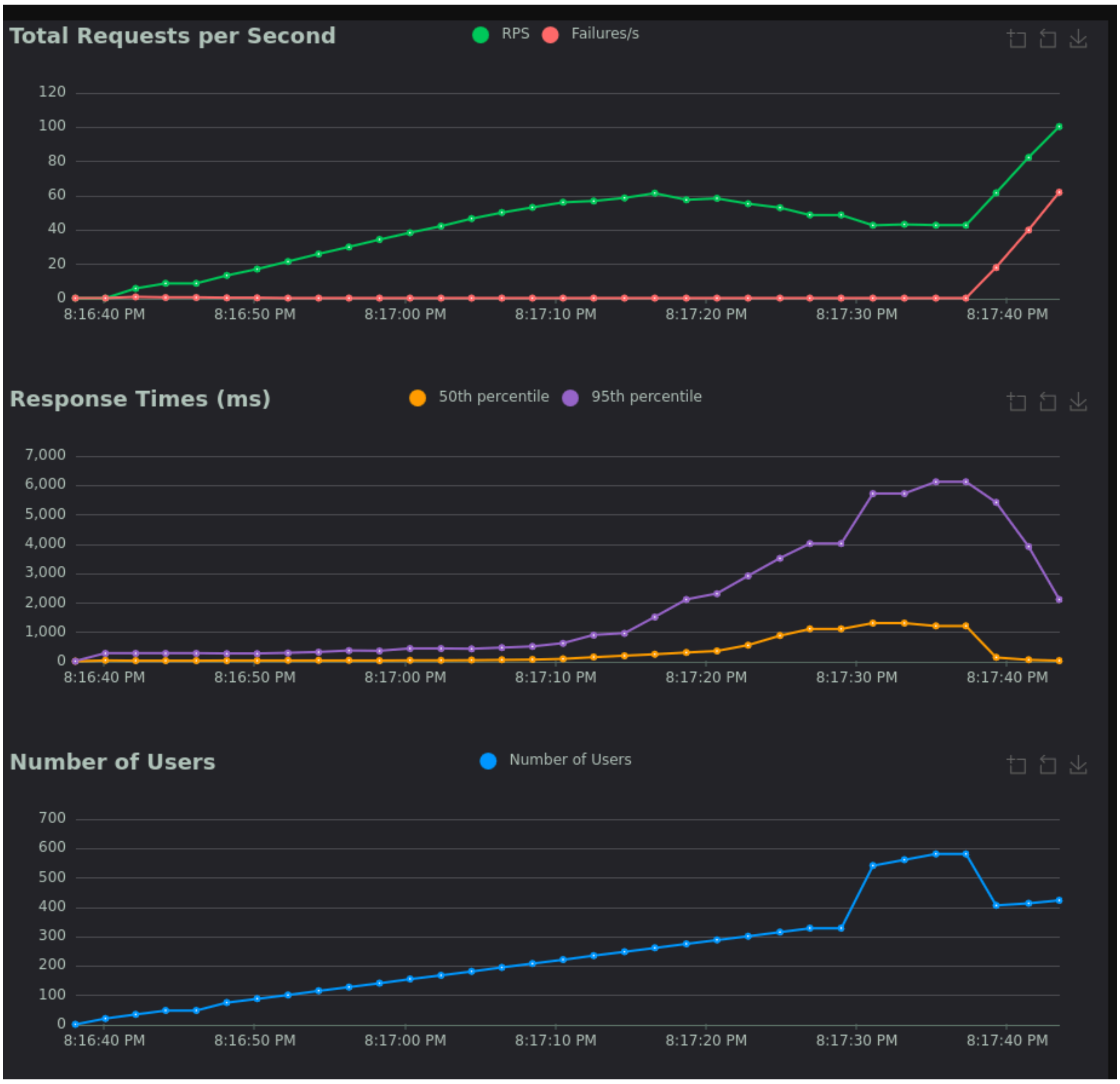
Celery Documentation. "Distributed Task Queue." Celery Project, https://docs.celeryproject.org.

Appendices

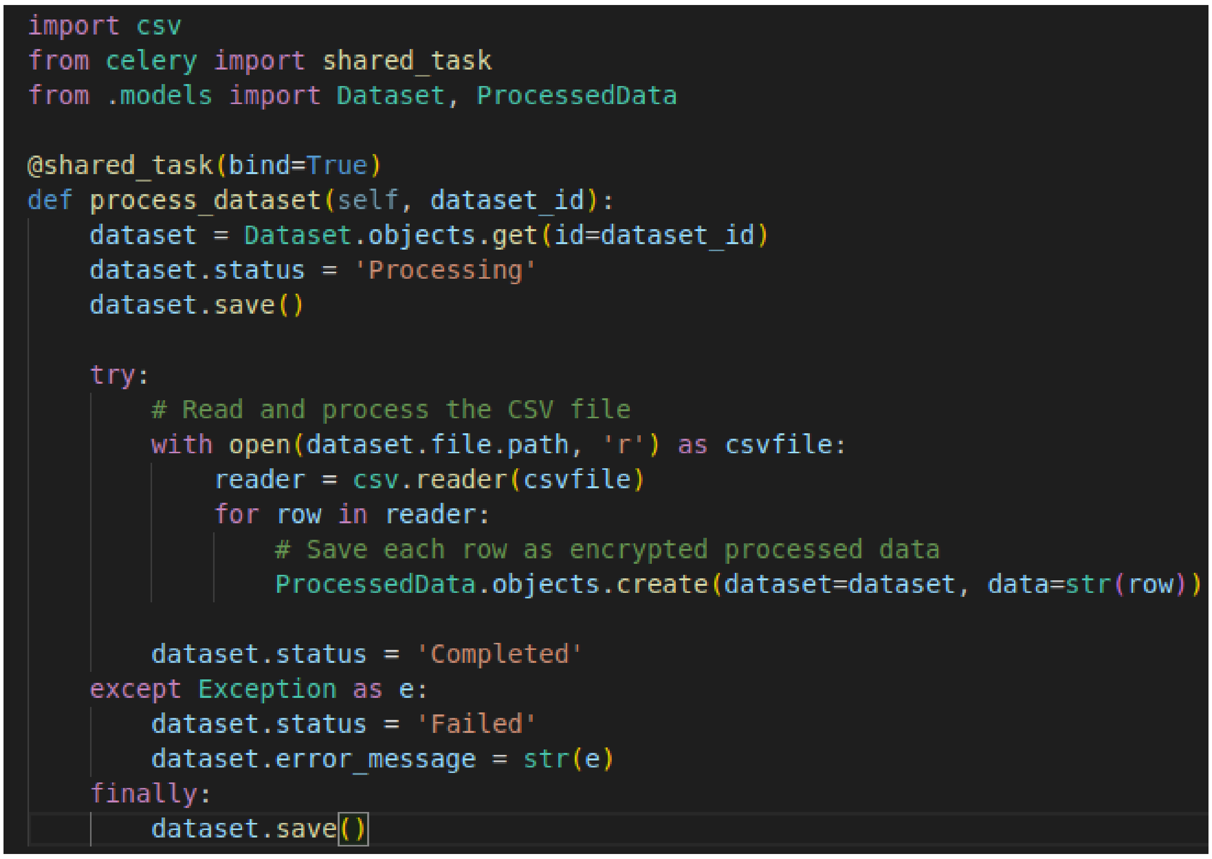
Appendix A: Average Response Time Graph



Appendix B: Error Rates by Concurrent Users



Appendix C: Sample Code for Asynchronous Task Processing and Progress Tracking

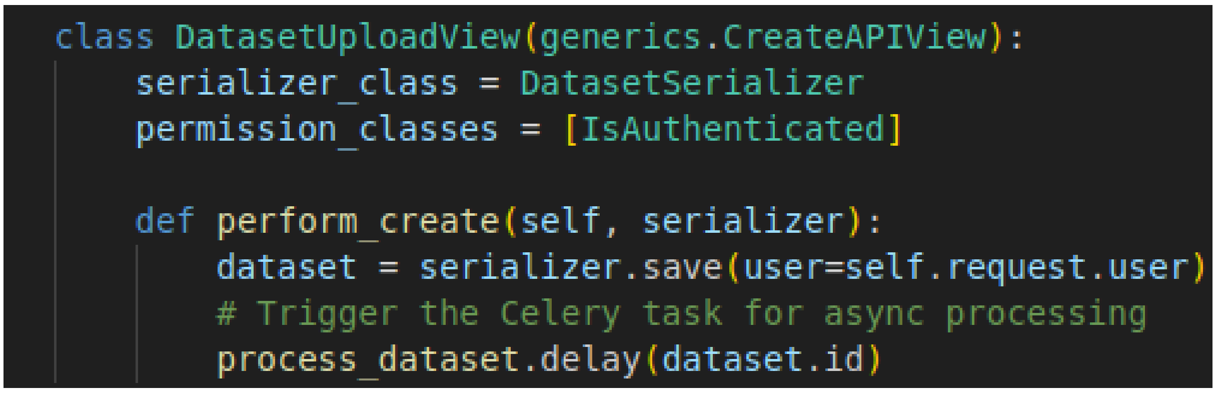


```python
import csv
from celery import shared_task
from .models import Dataset, ProcessedData

@shared_task(bind=True)
def process_dataset(self, dataset_id):
    dataset = Dataset.objects.get(id=dataset_id)
    dataset.status = 'Processing'
    dataset.save()

    try:
        # Read and process the CSV file
        with open(dataset.file.path, 'r') as csvfile:
            reader = csv.reader(csvfile)
            for row in reader:
                # Save each row as encrypted processed data
                ProcessedData.objects.create(dataset=dataset, data=str(row))

        dataset.status = 'Completed'
    except Exception as e:
        dataset.status = 'Failed'
        dataset.error_message = str(e)
    finally:
        dataset.save()
```



```python
class DatasetUploadView(generics.CreateAPIView):
    serializer_class = DatasetSerializer
    permission_classes = [IsAuthenticated]

    def perform_create(self, serializer):
        dataset = serializer.save(user=self.request.user)
        # Trigger the Celery task for async processing
        process_dataset.delay(dataset.id)
```