

# UNIVERSITY OF RWANDA



## African Centre of Excellence in Data Science

### Cloud computing and web development

#### Final Exam Project Report

**Group 10**

**NSHUNGUYIMFURA Abou-Bakar : 223028093**

**UWIMBABAZI Rosine : 213001787**

**NSABIYERA Enock : 21300178**

**December 2024**

## 1. Introduction

The Inventory Management System is a web application developed in Django to manage inventory, streamline transactions, and offer real-time analytics.

By integrating cloud services and big data tools like Hadoop and Kafka, the system aims to enhance scalability, data processing efficiency, and real-time responsiveness.

## 2. Case Study and Objectives

**Case Study:** The Inventory Management System designed for a retail business struggling with manual inventory tracking and data inconsistencies.

Problems included:

- Difficulty monitoring stock levels.
- Inefficient transaction logging.
- Lack of real-time insights into sales trends and profits.

**Objectives:**

- Develop a web-based inventory management system.
- Integrate an authentication system for secure user access.
- Provide CRUD functionality for managing inventory and transactions.
- Build a dashboard for analytics and reporting.
- Leverage AWS MySQL database for reliable storage and accessibility.
- Use Hadoop for batch processing of large inventory datasets.
- Implement Apache Kafka for real-time data streaming and updates.

## 3. Web Application Development

**Framework:**

- Backend: Django (Python) for server-side logic and ORM integration.
- Frontend: HTML, CSS, and JavaScript for responsive and interactive user interfaces.

**Features:**

**Authentication System:**

- Secure login functionality using Django's built-in authentication system.
- Validation to prevent unauthorized access to sensitive data and features.

## CRUD Operations:

Manage inventory data:

- **Create:** Add new items to the inventory.
- **Read:** View all items with details like stock balance, purchase price, and sale price.
- **Update:** Edit item details (e.g., prices, quantities).
- **Delete:** Remove obsolete items from the database.

Inventory Management System						
Dashboard	Items	Transactions	Stock Balance	Suppliers	Categories	Logout
Item List						
Add Item						
Item Name	Unit	Purchase Price	Sale Price	Supplier	Category	Actions
laptop	pce	230.00	300.00	VTC Technology	IT Equipments	<a href="#">Edit</a>   <a href="#">Delete</a>
Printers	pce	150.00	200.00	VTC Technology	IT Equipments	<a href="#">Edit</a>   <a href="#">Delete</a>
Scanner	pce	320.00	400.00	VTC Technology	IT Equipments	<a href="#">Edit</a>   <a href="#">Delete</a>
Phone	pce	480.00	600.00	Kigali phone KT	Phone	<a href="#">Edit</a>   <a href="#">Delete</a>
Apples	kg	2000.00	2500.00	JT Supply	Fresh Produce	<a href="#">Edit</a>   <a href="#">Delete</a>
Laptop bag	pce	15000.00	16000.00	Magasin Sport Class	Bags	<a href="#">Edit</a>   <a href="#">Delete</a>
<a href="#">Back to Home</a>						

## Dashboard:

Display meaningful analytics such as:

- Total Purchase Value
- Total Sales Value
- Total Profit
- Top-selling items.
- Date selector to view weekly and monthly figures.
- Real-time updates using Kafka integration.

## Dynamic Forms:

Update fields like price and profit dynamically based on user input and item selection.

The screenshot displays two side-by-side 'Add New Transaction' forms within the 'Inventory Management System' interface. The top navigation bar includes links for Dashboard, Items, Transactions, Stock Balance, Suppliers, Categories, and Logout. The left form is for 'Phone' with a 'Stock IN' transaction type. It has a 'Quantity' field set to 2, a 'Price' field set to 480.00, and a 'Profit' field set to 600. The right form is also for 'Phone' but with a 'Stock OUT' transaction type. It has a 'Quantity' field set to 2, a 'Price' field set to 600, and a 'Profit' field set to 240.00. Both forms include a 'Date' field with a calendar icon and a green 'Add Transaction' button at the bottom.

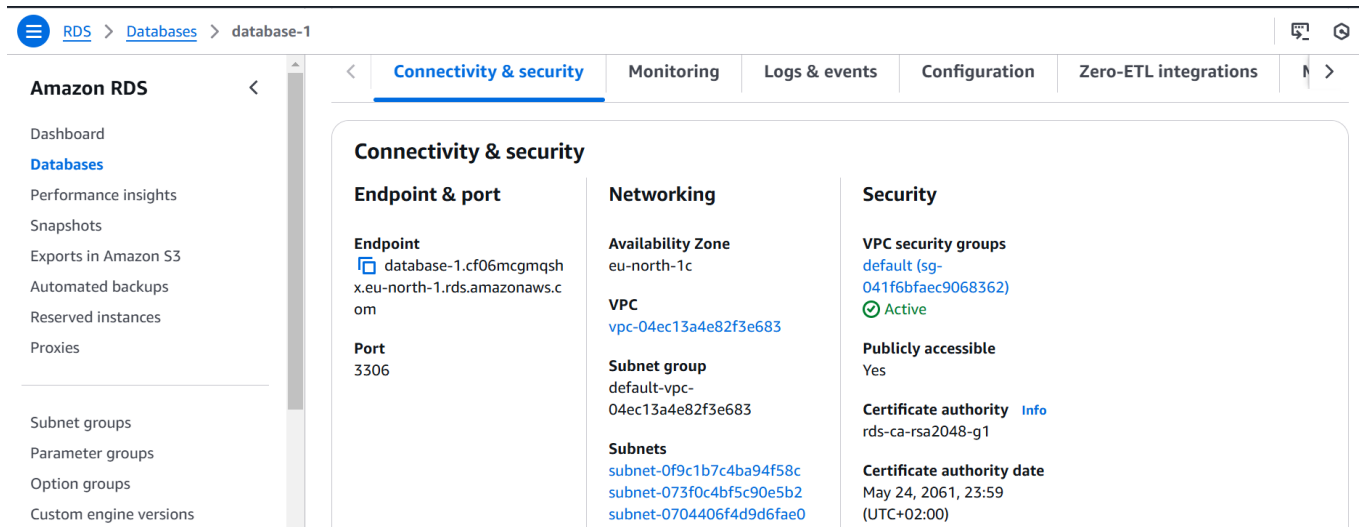
## 4. Cloud Computing Integration

### MySQL Database on AWS

#### Amazon RDS:

- A MySQL database was created and configured using Amazon RDS for robust, scalable data storage.
- Multi-AZ deployment was set up for high availability.
- VPC was configuration

The screenshot shows the AWS Management Console for the 'Stockholm' region, specifically the 'Amazon RDS > Databases' page. The left sidebar contains navigation links for Dashboard, Databases, Performance insights, Snapshots, Exports in Amazon S3, Automated backups, Reserved instances, Proxies, Subnet groups, Parameter groups, and Option groups. The main content area features two informational cards: 'Consider creating a Blue/Green Deployment to minimize downtime during upgrades' and 'Easy path homogeneous data migrations from EC2 database to RDS'. Below these cards is a table titled 'Databases (1)' with columns for DB identifier, Status, Role, Engine, Region, and Size. The table lists one database instance named 'database-1' with a status of 'Available', engine 'MySQL Co...', region 'eu-north-1', and size 'db.t4g.micro'. A top navigation bar includes the AWS logo, a search bar, and a user profile dropdown menu showing 'Account ID 6517-0678-4362' and a 'Sign out' button.



### Django Integration:

- The Django application was connected to the RDS MySQL database using proper credentials and environment variables.
- The settings.py file was configured for the production database:

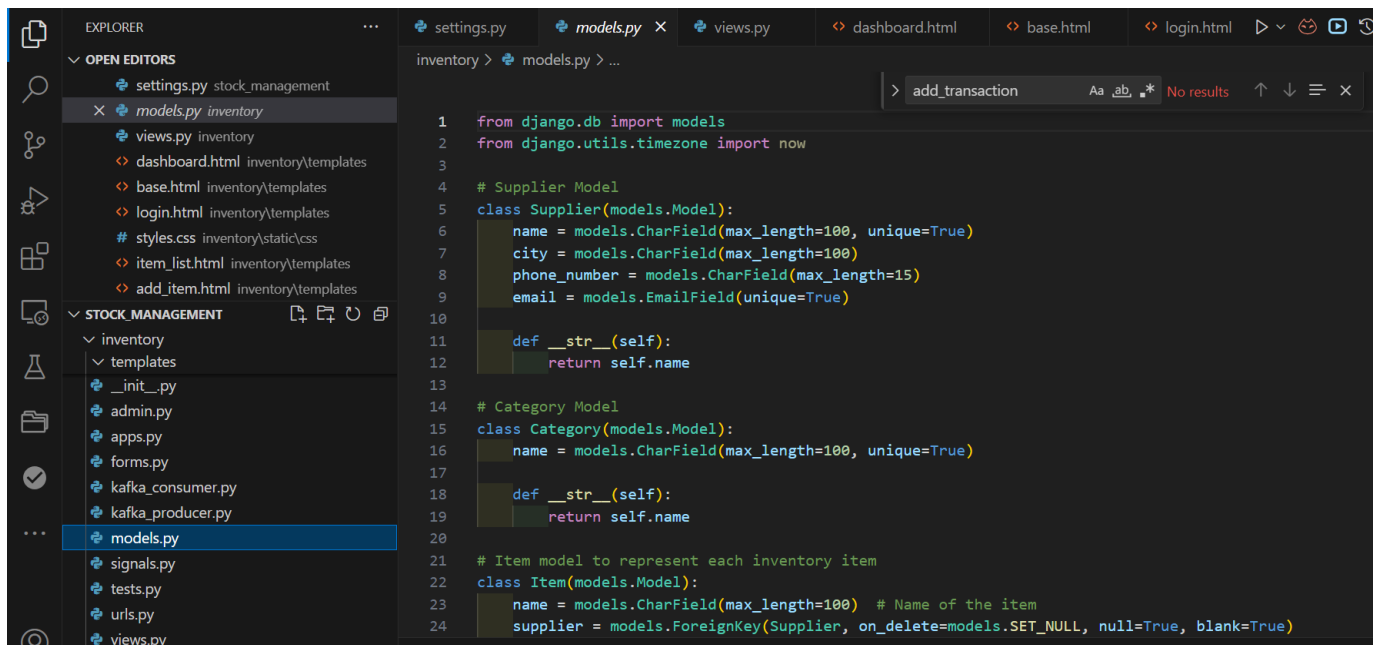
```

settings.py • models.py views.py dashboard.html base.html login.html
stock_management > settings.py > ...
125
126 DATABASES = {
127     'default': {
128         'ENGINE': 'django.db.backends.mysql',
129         'NAME': 'inventory-db',
130         'USER': 'admin',
131         'PASSWORD': 'myPassWord',
132         'HOST': 'database-1.cf06mcgmqshx.eu-north-1.rds.amazonaws.com',
133         'PORT': '3306',
134     }
135 }
136
137 LOGIN_URL = '/login/'
138
139 KAFKA_BROKER_URL = 'localhost:9092'
140
141 STATIC_URL = '/static/'
142 STATICFILES_DIRS = [BASE_DIR / 'static']

```

### Transaction Handling:

- Implemented atomic transactions to ensure data consistency for stock-in and stock-out operations.
- Used Django ORM for efficient query execution.



## 5. Hadoop Integration

### Local Hadoop Setup:

- A single-node Hadoop cluster was set up locally for batch processing.
- HDFS (Hadoop Distributed File System) was used to store transaction datasets.

### MapReduce Workflow:

A MapReduce job was implemented to process sales data and calculate:

- Total sales for each item.
- Total profit for each item.

The following is how workflow was performed:

- Input file uploaded to HDFS: transaction\_data.csv.
- Mapper processed sales records and produced intermediate key-value pairs.
- Reducer aggregated results to calculate totals.

```

C:\Users\PC>hdfs dfs -mkdir /stock_analysis

C:\Users\PC>hdfs dfs -put C:\Users\PC\Desktop\Vm\transaction_data.csv /stock_analysis

C:\Users\PC>hdfs dfs -put C:\Users\PC\Desktop\Vm\mapper.py /stock_analysis

C:\Users\PC>hdfs dfs -put C:\Users\PC\Desktop\Vm\reducer.py /stock_analysis

```

Browse Directory

/stock\_analysis

Go!

Show

25

entries

Search:

<input type="checkbox"/>	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	
<input type="checkbox"/>	-rw-r--r--	PC	supergroup	785 B	Dec 05 00:45	1	128 MB	mapper.py	<div></div>
<input type="checkbox"/>	drwxrwxrwx	hdfs	supergroup	0 B	Dec 06 12:14	0	0 B	output	<div></div>
<input type="checkbox"/>	-rw-r--r--	PC	supergroup	1.57 KB	Dec 05 00:46	1	128 MB	reducer.py	<div></div>
<input type="checkbox"/>	-rw-r--r--	PC	supergroup	362 B	Dec 05 00:45	1	128 MB	transaction_data.csv	<div></div>

Showing 1 to 4 of 4 entries

Previous

1

Next

Browse Directory

/stock\_analysis/output

Go!

Show

25

entries

Search:

<input type="checkbox"/>	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	
<input type="checkbox"/>	-rw-r--r--	hdfs	supergroup	76 B	Dec 06 12:14	1	128 MB	result.txt	<div></div>

Showing 1 to 1 of 1 entries

Previous

1

Next

Hadoop, 2022.

Integration with Django:

- Results of the MapReduce job were displayed on the dashboard for analytics.
- Used Django views to fetch processed data from HDFS and render it in the web application.

The following is the result of the MapReduce job:

Command Prompt

```
C:\Users\PC>hdfs dfs -cat /stock_analysis/output/result.txt
item    total    profit
Laptop  900.0    180.0
Printer 300.0    60.0
Scanner 700.0    100.0
```

View codes for displaying results from HDFS on the web application dashboard.

```
from django.db.models import Sum, F, Q, Value
from django.db.models.functions import Coalesce # Import Coalesce
from datetime import datetime
from hdfs import InsecureClient
from collections import defaultdict
# HDFS Configuration
HDFS_URL = 'http://localhost:9870'
HDFS_USER = 'hdfs'
OUTPUT_FILE = '/stock_analysis/output/result.txt'

client = InsecureClient(HDFS_URL, user=HDFS_USER)

@login_required
def dashboard(request):
    # Fetch MapReduce results from HDFS
    mapreduce_results = []
    try:
        with client.read(OUTPUT_FILE, encoding='utf-8') as reader:
            lines = reader.readlines()[1:] # Skip the header line
            for line in lines:
                item, total, profit = line.strip().split("\t")
                mapreduce_results.append({
                    "item": item,
                    "total": float(total),
                    "profit": float(profit),
                })
```

## 6. Apache Kafka Integration

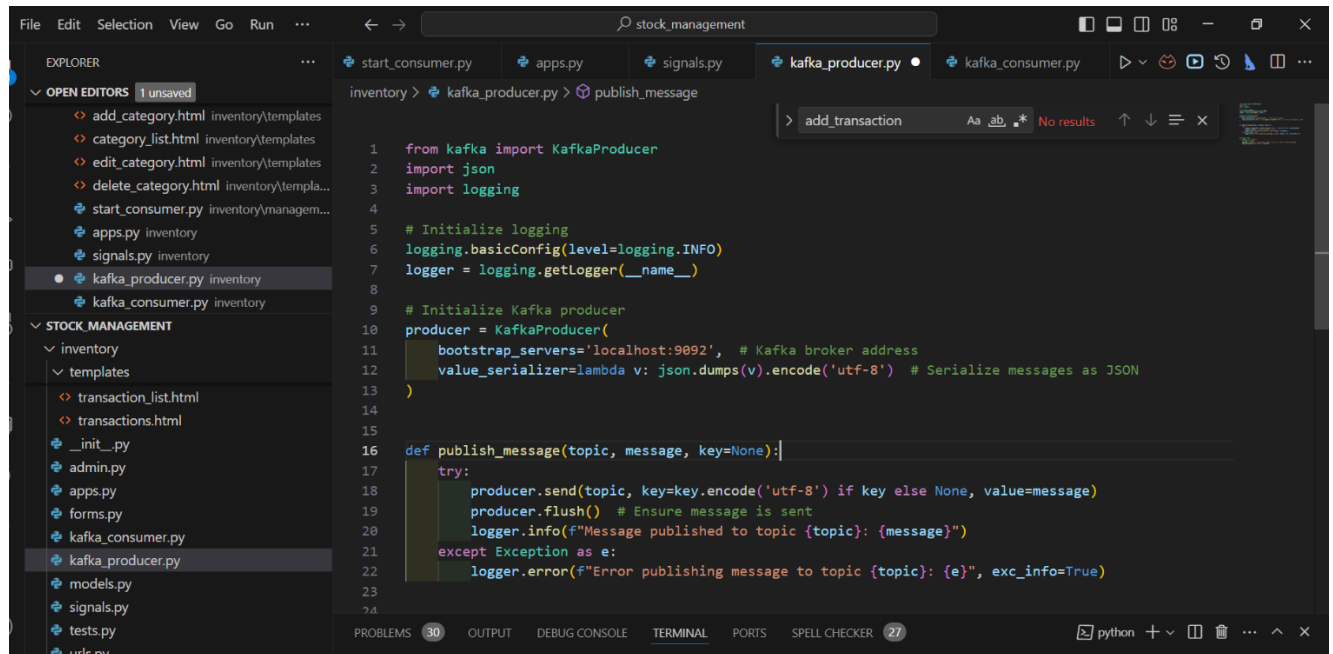
### Setup and Configuration:

- Kafka was set up locally to handle real-time data streaming.
- Created a Kafka topic for publishing stock transactions.



## Data Publishing:

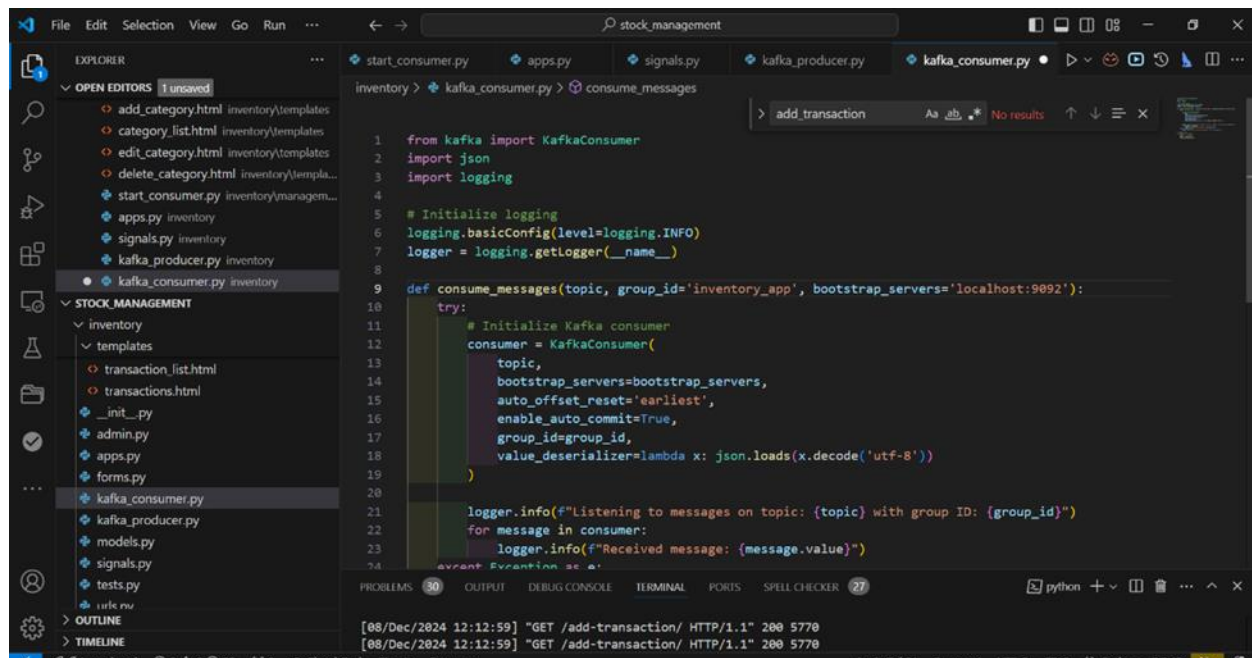
- Stock transactions (e.g., stock-in and stock-out) were published as messages to the Kafka topic:



```
1 from kafka import KafkaProducer
2 import json
3 import logging
4
5 # Initialize logging
6 logging.basicConfig(level=logging.INFO)
7 logger = logging.getLogger(__name__)
8
9 # Initialize Kafka producer
10 producer = KafkaProducer(
11     bootstrap_servers='localhost:9092', # Kafka broker address
12     value_serializer=lambda v: json.dumps(v).encode('utf-8') # Serialize messages as JSON
13 )
14
15
16 def publish_message(topic, message, key=None):
17     try:
18         producer.send(topic, key=key.encode('utf-8') if key else None, value=message)
19         producer.flush() # Ensure message is sent
20         logger.info(f"Message published to topic {topic}: {message}")
21     except Exception as e:
22         logger.error(f"Error publishing message to topic {topic}: {e}", exc_info=True)
```

## Data Consumption:

- A Kafka consumer was implemented to consume messages from a Kafka topic and update the dashboard in real-time:



```
1 from kafka import KafkaConsumer
2 import json
3 import logging
4
5 # Initialize logging
6 logging.basicConfig(level=logging.INFO)
7 logger = logging.getLogger(__name__)
8
9
10 def consume_messages(topic, group_id='inventory_app', bootstrap_servers='localhost:9092'):
11     try:
12         # Initialize Kafka consumer
13         consumer = KafkaConsumer(
14             topic,
15             bootstrap_servers=bootstrap_servers,
16             auto_offset_reset='earliest',
17             enable_auto_commit=True,
18             group_id=group_id,
19             value_deserializer=lambda x: json.loads(x.decode('utf-8'))
20         )
21
22         logger.info(f"Listening to messages on topic: {topic} with group ID: {group_id}")
23         for message in consumer:
24             logger.info(f"Received message: {message.value}")
25     except Exception as e:
26         logger.error(f"Error consuming message from topic {topic}: {e}", exc_info=True)
```

[08/Dec/2024 12:12:59] "GET /add-transaction/ HTTP/1.1" 200 5770  
[08/Dec/2024 12:12:59] "GET /add-transaction/ HTTP/1.1" 200 5770

## 7. Challenges and Solutions

### Database Connectivity Issues:

- **Challenge:** Connection errors during AWS MySQL integration.
- **Solution:** Verified RDS endpoint, security groups, and IAM roles to ensure proper access.

### Real-Time Streaming Implementation:

- **Challenge:** Setting up Kafka and integrating with the application.
- **Solution:** Used Python's Kafka library and debugged network issues with Zookeeper.

### Hadoop Integration:

- **Challenge:** Processing datasets efficiently.
- **Solution:** Optimized MapReduce logic and ensured proper HDFS configurations.

## 8. Conclusion

The Inventory Management System successfully achieves its objectives by providing a secure, scalable, and efficient solution for inventory management. The integration with AWS ensures high availability, and the implemented features address the critical requirements of stock validation, profit calculation, and user-friendly interactions.

The challenges faced during development were resolved with thoughtful design decisions and robust technical implementations, resulting in a highly functional and user-focused application.

**End!!!!**