

**BỘ GIÁO DỤC VÀ ĐÀO TẠO  
TRƯỜNG ĐẠI HỌC CMC**

---



## **DEVELOPMENT AND OPERATIONS (DEVOPS)**

**Topic: Software Deployment of a Web-Based Personal Budgeting System**

**Group of students:**

**Trần Quỳnh Anh - BIT230008**

**Nguyễn Hải Anh - BIT230500**

**Đoàn Anh Vũ - BIT230458**

## Work Assignment

TT	Student Code	Full Name	Work Content	Evaluate (base 10)	Note
1	BIT230500	Nguyễn Hải Anh	Docker Compose (Database + Grafana + prometheus Container)	10	Host computer + Help with every other content
2	BIT230008	Trần Quỳnh Anh	Jenkins CI + Docker Containerize	9,75	Leader – in charge with report and slide
3	BIT230458	Đoàn Anh Vũ	MinIO + Public container to Docker hub	9,5	

## TABLE OF CONTENTS

LIST OF FIGURES .....	4
LIST OF ABBREVIATIONS .....	5
INTRODUCTION .....	6
1.1. Background and Rationale for Choosing the Topic.....	6
1.2. Objectives.....	6
1.3. Scope of the Study .....	6
CHAPTER 1. PROBLEM INTRODUCTION .....	8
1.1. General Introduction .....	8
1.2. Proposed Problem .....	8
1.3. Implementation Plan .....	9
1.4. Technologies and Tools Used .....	9
CHAPTER 2. IMPLEMENTATION SOLUTION .....	11
2.1. Requirement Analysis .....	11
2.3. Deployment Process.....	11
2.3.1. CI/CD Deployment using Jenkins, Docker and GitHub .....	11
2.3.2. CI/CD Workflow using Jenkins, Docker and GitHub.....	12
CHAPTER 3. IMPLEMENTATION RESULTS .....	19
3.1. System Architecture Overview .....	19
3.2 Result Evaluation .....	19
3.3. Screenshots.....	19
CONCLUSION.....	22

## LIST OF FIGURES

Figure 1: Clone code from GitHub .....	13
Figure 2: Build with Maven.....	13
Figure 3: Deploy WAR to Local Tomcat .....	14
Figure 4: Build Docker Image .....	14
Figure 5: Dockerfile .....	15
Figure 6: Run Docker Compose .....	15
Figure 7: File docker-compose.yml.....	17
Figure 8: Run MinIO container .....	17
Figure 9: MinIO container running on Docker with ports 9000 and 9001 exposed.....	17
Figure 10: Run docker container .....	18
Figure 11: Spring Boot application container running with port mapping 8091:8080 .....	18
Figure 12: Push Docker Image .....	18
Figure 13: Jenkins pipeline run triggered by GitHub push with successful build execution	20
Figure 14: All container is created and run sequentially automatically .....	20
Figure 15: Project succesfully deployed at port 8091 (Image in profile is got from MinIO)	21
Figure 16: Prometheus graph in Grafana tracking http request and 4xx error .....	21

## LIST OF ABBREVIATIONS

Symbol	Meaning
CI	Continuous Integration
CD	Continuous Deployment / Delivery
SCM	Source Code Management
VCS	Version Control System
UI	User Interface
API	Application Programming Interface
OS	Operating System
IDE	Integrated Development Environment
YAML	Yet Another Markup Language
WAR	Web Application Archive
JDK	Java Development Kit
SDK	Software Development Kit
CLI	Command Line Interface
SQL	Structured Query Language
DBMS	Database Management System
DB	Database
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IP	Internet Protocol
URL	Uniform Resource Locator
Git	A distributed version control system
GitHub	A web-based platform for version control using Git
Jenkins	An open-source automation server for CI/CD
Docker	A containerization platform
Docker Hub	A cloud-based repository for Docker images
Docker Compose	A tool for defining and running multi-container Docker applications
MinIO	A high-performance object storage system
Prometheus	An open-source monitoring and alerting toolkit
Grafana	An open-source platform for monitoring and observability
UI	User Interface

# INTRODUCTION

## 1.1. Background and Rationale for Choosing the Topic

In today's digital era, effective financial management is increasingly supported by personal budgeting applications. These systems help individuals track their income and expenses, set financial goals, and make informed decisions about their money. However, building such applications is only part of the challenge; deploying them reliably, securely, and efficiently is equally important. This is where DevOps practices come into play.

With the growing demand for continuous integration, continuous delivery (CI/CD), and automated deployment pipelines, DevOps has become an essential approach for modern software development and deployment. By choosing the topic "Software Deployment of a Web-Based Personal Budgeting System", this project aims to apply DevOps principles in a real-world scenario—deploying a financial web application that is both functional and maintainable. The topic was selected to gain hands-on experience with tools like Docker, Jenkins, GitHub Actions, and cloud services, ensuring that the application is always in a deployable state with minimal manual intervention.

## 1.2. Objectives

The main objectives of this report are:

- To design and implement a DevOps pipeline for the deployment of a personal budgeting web application.
- To automate the build, test, and deployment processes using modern DevOps tools and technologies.
- To ensure reliable and consistent software delivery using containerization (Docker) and orchestration tools if applicable.
- To explore best practices in DevOps for small-scale personal finance software systems.
- To demonstrate the benefits of continuous integration and continuous deployment (CI/CD) pipelines in maintaining application quality and uptime.

## 1.3. Scope of the Study

This study focuses primarily on the deployment aspects of the personal budgeting system using DevOps practices. The scope includes:

- Setting up the infrastructure needed for deployment (e.g., Docker, CI/CD pipelines).
- Automating the build and deployment process of the budgeting system.
- Using version control (Git) and integrating it with CI/CD tools.

- Testing the application deployment across different environments (development, staging, production).
- Monitoring and logging basic system metrics post-deployment.

## **CHAPTER 1. PROBLEM INTRODUCTION**

### **1.1. General Introduction**

In the digital age, managing personal finances through traditional spreadsheets or manual methods is becoming increasingly inefficient. To address this, web-based personal budgeting systems are emerging as essential tools that allow users to track their income, expenses, and financial goals in an intuitive and accessible way.

However, developing a useful budgeting application is not enough. Ensuring that it is reliably deployed, frequently updated, and easily scalable is a critical part of delivering long-term value to users. This challenge lies in the software deployment process, which is often time-consuming, error-prone, and lacks consistency when performed manually.

With the rapid evolution of cloud infrastructure and DevOps methodologies, modern software deployment can now be streamlined and automated. By integrating tools like Docker, CI/CD pipelines, and cloud hosting, developers can ensure that applications are deployed faster, with fewer errors, and are easier to maintain.

This project aims to explore and implement the deployment of a Web-Based Personal Budgeting System using DevOps principles to demonstrate how such practices can enhance software delivery and lifecycle management.

### **1.2. Proposed Problem**

The main problem addressed in this study is the inefficiency and unreliability of traditional software deployment methods, especially when applied to web-based applications like personal finance management systems. Without automation, deployment processes are prone to:

- Configuration mismatches between development and production environments.
- Long downtime during updates or releases.
- High manual effort in testing, building, and deploying applications.
- Difficulty in scaling or rolling back when failures occur.

To solve these issues, this project proposes designing and implementing a DevOps-based deployment pipeline for a personal budgeting web application. The solution involves:

- Containerizing the application using Docker to ensure consistency across environments.



- Setting up CI/CD pipelines (e.g., using Jenkins or GitHub Actions) to automate build, test, and deployment stages.
- Hosting the application on a server or cloud platform with version control integration for smooth release cycles.
- Monitoring deployment to track performance and failures.

By addressing the deployment process as the central challenge, this project highlights the importance of DevOps in reducing risks, improving software quality, and accelerating delivery in modern web development.

### 1.3. Implementation Plan

No	Task	Deliverables	Person in charge	Duration (week)
1	Docker Compose Setup	Docker Compose file integrating Database, Grafana, and Prometheus	Nguyễn Hải Anh	2
2	CI/CD with Jenkins and Docker	Jenkinsfile and Dockerfile for automating build and deployment	Trần Quỳnh Anh	2
3	MinIO Integration & Docker Hub	MinIO configuration and pushing container image to Docker Hub	Đoàn Anh Vũ	2
4	Documentation & Presentation	Final report and presentation slides	Trần Quỳnh Anh	1

### 1.4. Technologies and Tools Used

Tool/Technology	Purpose	Role in the Project
Git & GitHub	Version control	Used to manage source code, track changes, and integrate with CI/CD pipeline for automated deployments.
Docker	Containerization	Used to package the application and its dependencies into a consistent and portable container image, ensuring it runs the same across environments.

Docker Compose	Multi-container setup	Used to define and manage multi-container applications (e.g., application + database) for local development and testing.
Jenkins / GitHub Actions	CI/CD automation	Used to automate the build, test, and deployment pipeline. (Choose one depending on what you used).
Ngrok	Create public URL	Used to expose the local Jenkins server via a secure tunnel for receiving GitHub webhooks.
Apache	Web server / reverse proxy	Serves the frontend and handles routing, reverse proxy, and load balancing (if applicable).
SQL Server	Database	Used as the backend database for storing transaction data, budgets, categories, etc.
Spring Boot	Backend Framework	The core backend logic of the application, handling APIs, user logic, and interaction with the database.
HTML/CSS/JavaScript	Frontend technologies	Technologies used to build the user interface of the personal budgeting system.
Visual Studio Code	Development environment	Code editor used for writing and debugging the application code.
Postman	API testing	Used to test backend APIs before and after deployment.
Grafana	System monitoring dashboard	Used to visualize system and application metrics collected by Prometheus in real time through customizable dashboards.
Prometheus	Metrics collection & monitoring	Collects time-series data and metrics from application and infrastructure components for system health monitoring.
MinIO	Object storage	Provides high-performance, S3-compatible object storage for storing application files, reports, or backups.
Kubernetes (K8s)	Container orchestration	Automates deployment, scaling, and management of Docker containers across a distributed cluster.

## **CHAPTER 2. IMPLEMENTATION SOLUTION**

### **2.1. Requirement Analysis**

The personal finance management system was designed and implemented based on the following functional and non-functional requirements:

- Functional Requirements:
  - Allow users to record, update, and delete personal income and expenses.
  - Display categorized financial summaries (e.g., by time period or expense type).
  - Provide reporting and statistical visualization (charts, tables).
  - Support user authentication and role-based access control
- Non-Functional Requirements:
  - The system must operate smoothly on modern web browsers.
  - Data must be securely stored and efficiently retrieved from a relational database (SQL Server).
  - The development workflow must support automated build, test, and deployment using CI/CD (Jenkins integrated with GitHub).
  - The application must be containerized using Docker to ensure consistency across environments, simplify deployment, and enable future scalability (e.g., deployment via Docker Compose or orchestration platforms).
  - System components (e.g., backend, frontend, database) must be defined as Docker services and run in isolated containers.

These requirements ensure that the system not only fulfills user needs but is also maintainable, portable, and ready for real-world deployment scenarios using modern DevOps practices.

### **2.3. Deployment Process**

#### **2.3.1. CI/CD Deployment using Jenkins, Docker and GitHub**

The Continuous Integration and Continuous Deployment (CI/CD) process for this project is built using Jenkins, Docker, and GitHub. The goal of the pipeline is to automate the entire lifecycle from code integration to deployment using containerized infrastructure and Tomcat.

When code is pushed to the main branch of the GitHub repository, a webhook triggers Jenkins, which then executes a defined pipeline (in Jenkinsfile) to perform the following:

- Clone the updated source code
- Build the project using Maven
- Deploy the application to Apache Tomcat
- Build a Docker image of the application

- Push the Docker image to Docker Hub
- Run the application using Docker and Docker Compose

This automation improves reliability, consistency, and delivery speed.

### **2.3.2. CI/CD Workflow using Jenkins, Docker and GitHub**

This section provides an in-depth explanation of the CI/CD workflow, illustrating how Jenkins, Docker, and GitHub are integrated to automate the entire process—from code integration to deployment and monitoring.

The Jenkins pipeline is defined in a Jenkinsfile, which orchestrates multiple stages including building, deploying, containerizing, and pushing the application image to Docker Hub. Below is the detailed step-by-step workflow:

#### **2.3.3.1. Source Code Management with GitHub**

- The entire application source code is hosted on GitHub at:  
<https://github.com/HAnh3112/JavaWebFinal.git>
- Jenkins uses the Git plugin to automatically clone the latest code from the main branch whenever the pipeline is triggered.

#### **2.3.3.2 Webhook Integration using Ngrok**

Since Jenkins was hosted locally without a public IP address, we used Ngrok to expose Jenkins to the internet in order to receive webhook events from GitHub. This enabled true continuous integration, where each git push automatically triggered the Jenkins pipeline.

Steps:

- **Ngrok** was used to expose Jenkins on a temporary public URL.
- This URL was added to the GitHub repository as a webhook endpoint under Settings → Webhooks.
- GitHub then sent a POST request to Jenkins every time code was pushed to the main branch.
- Jenkins processed the webhook and started the pipeline automatically.

This integration simulates a real-world CI/CD environment, even when running Jenkins on a local machine.

### 2.3.3.3 Jenkins Pipeline Overview

The CI/CD pipeline is defined using Declarative Pipeline Syntax and contains multiple stages, each responsible for a specific task:

#### Stage 1: Clone

When code is pushed to the main branch of the GitHub repository, a webhook automatically triggers Jenkins to start the pipeline. The pipeline begins by cloning the repository:

```
stages {
    stage('Clone') {
        steps {
            echo 'Cloning source code from GitHub'
            git branch: 'main', url: 'https://github.com/HAnh3112/JavaWebFinal.git'
        }
    }
}
```

*Figure 1: Clone code from GitHub*

#### Stage 2: Build with Maven

Jenkins then uses Maven (configured as Maven 3.9.9 in Jenkins global settings) to compile the Java source code and package it into a .war file, skipping tests to speed up the process:

```
stage('Build with Maven') {
    steps {
        echo 'Running Maven build'
        bat 'mvn clean package -DskipTests'
    }
}
```

*Figure 2: Build with Maven*

#### Stage 3: Deploy WAR to Local Tomcat

After building, the .war file is copied to the local Apache Tomcat server (installed at D:\apache-tomcat-11.0.7). Tomcat is restarted to reflect the new deployment:

```

stage('Deploy WAR to Tomcat') {
    steps {
        echo 'Deploying WAR file to Tomcat'
        bat '''
            if not exist "%TOMCAT_PATH%\webapps" (
                echo Tomcat webapps folder not found!
                exit /b 1
            )
            for %%f in (target\*.war) do (
                echo Copying %%f to Tomcat
                copy "%%f" "%TOMCAT_PATH%\webapps\%WAR_NAME%" /Y
            )
        '''
    }
}

stage('Restart Tomcat') {
    steps {
        echo 'Restarting Tomcat server'
        bat '''
            call "%TOMCAT_PATH%\bin\shutdown.bat"
            timeout /t 5 > NUL
            call "%TOMCAT_PATH%\bin\startup.bat"
        '''
    }
}

```

*Figure 3: Deploy WAR to Local Tomcat*

#### Stage 4: Build Docker Image

Using a multi-stage Dockerfile, Jenkins builds a Docker image of the application. The first stage compiles the app with Maven, and the second stage deploys it on a Tomcat base image:

```

stage('Build Docker Image') {
    steps {
        bat '''
            docker build -t springbootapp:latest -f "%WORKSPACE%\Dockerfile" .
        '''
    }
}

```

*Figure 4: Build Docker Image*

Relevant part of the Dockerfile:

```

FROM maven:3.9.6-eclipse-temurin-21 AS build
WORKDIR /app
COPY . .
RUN mvn clean package -DskipTests

# Stage 2: Deploy WAR to Tomcat
FROM tomcat:10.1-jdk21
WORKDIR /usr/local/tomcat

# Remove default webapps
RUN rm -rf webapps/*

# Copy the WAR file and rename it to ROOT.war (optional)
COPY --from=build /app/target/*.war webapps/ROOT.war

EXPOSE 8080
# Tomcat's built-in startup script will run

```

*Figure 5: Dockerfile*

### Stage 5: Launch Environment with Docker Compose

To orchestrate services such as the app, SQL Server, Prometheus, and Grafana, Docker Compose is used. Jenkins executes the following commands to stop any existing setup and relaunch everything:

```

stage('Run Docker Compose') {
    steps {
        bat '''
            docker compose down
            docker compose up -d --build
        '''
    }
}

```

*Figure 6: Run Docker Compose*

Key services in docker-compose.yml:

- sql2022: Microsoft SQL Server 2022
- prometheus: Monitoring system
- grafana: Visualization dashboard

```
version: "3.9"

networks:
  myapp-net:
    driver: bridge

volumes:
  sql2022data:
  grafana-data:
  prometheus-data:

services:
  sql2022:
    image: mcr.microsoft.com/mssql/server:2022-latest
    container_name: sql2022
    environment:
      ACCEPT_EULA: "Y"
      SA_PASSWORD: "Test!@#1234"
    ports:
      - "1437:1433"
    networks:
      - myapp-net
    volumes:
      - sql2022data:/var/opt/mssql
      - "D:/GitHub_D:/var/opt/mssql/backup"

  prometheus:
    image: prom/prometheus
    container_name: prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus-data:/prometheus
    ports:
      - "9090:9090"
    networks:
      - myapp-net
```



```

grafana:
  image: grafana/grafana
  container_name: grafana
  ports:
    - "3000:3000"
  networks:
    - myapp-net
  depends_on:
    - prometheus
  volumes:
    - grafana-data:/var/lib/grafana

```

Figure 7: File docker-compose.yml

### Stage 6: Run MinIO container:

Jenkins pipeline will startup MinIO container. If it already started, this stage will simply be skipped

```

stage('Run MinIO Container') {
  steps {
    bat '''
      docker start minio || echo MinIO already running, skipping startup...
    '''
  }
}

```

Figure 8: Run MinIO container

MinIO will be run in 2 default ports: 9000 and 9001. Port 9000 is port for accessing data locally from other container, while port 9001 is admin panel, allow us to manage and monitor it.

● minio	e64d2263aff4	<a href="#">minio/minio</a>	<a href="#">9000:9000</a> <a href="#">9001:9001</a> <a href="#">Show less</a>
---------	--------------	-----------------------------	-------------------------------------------------------------------------------------

Figure 9: MinIO container running on Docker with ports 9000 and 9001 exposed

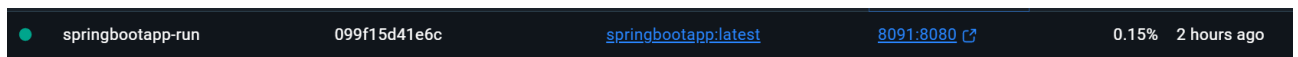
### Stage 7: Run Docker container:

Project's container will now be created and run on port 8091 using the newest version of image we created. If the container already exists, it will remove the older container and

create new one. The container is also placed in the same network “myapp-net” to be able to access to other services.

```
stage('Run Docker Container') {
    steps {
        bat '''
            docker rm -f springbootapp-run || echo "Container not found, skipping removal"
            docker run -d --name springbootapp-run --network myapp-net -p 8091:8080 springbootapp:latest
            ...
        '''
    }
}
```

*Figure 10: Run docker container*



*Figure 11: Spring Boot application container running with port mapping 8091:8080*

### Stage 8: Push Docker image to Dockerhub:

After passing all previous step, it implies that the image created work perfectly and ready to be pushed to public. Jenkins pipeline will then push it to public Dockerhub, allow another computer to pull back and run. Note that we are not using cloud for database, hence the pulling computer needs to have the database to be able to run.

```
stage('Push Docker Image') {
    steps {
        script {
            // push Docker image to Docker Hub
            docker.withRegistry('https://index.docker.io/v1/', DOCKERHUB_CREDENTIALS) {
                docker.image("${DOCKER_IMAGE_NAME}:${DOCKER_TAG}").push()
            }
        }
    }
}
```

*Figure 12: Push Docker Image*

## **CHAPTER 3. IMPLEMENTATION RESULTS**

### **3.1. System Architecture Overview**

The system consists of the following main components, all running as isolated containers within a Docker Compose network:


- Spring Boot Application (Custom WAR deployed on Tomcat)
- Microsoft SQL Server 2022 (SQL Database)
- Prometheus (Monitoring and metrics collection)
- Grafana (Visualization and dashboard)
- Docker Compose (Service orchestration)
- MinIO (Only as a Local Files Storage)


All services are connected via a custom Docker network (myapp-net), allowing inter-service communication and network isolation.


### **3.2 Result Evaluation**

- After successful deployment and integration of monitoring tools, the following results were observed:
- Fast Deployment: The application and supporting services were automatically deployed within minutes using Jenkins pipeline.
- System Observability: Prometheus successfully collected application metrics. Grafana visualized them using real-time dashboards.
- Maintainability: The environment can be replicated or scaled easily using container orchestration.
- Data Persistence: SQL Server data was preserved across container restarts using Docker volumes.


### **3.3. Screenshots**


#111 (17:37:14, 6 thg 8, 2025)



Started by GitHub push by tinytqa


This run spent:

- 9.8 sec waiting;
- 1 min 39 sec build duration;
- 1 min 49 sec total from scheduled to completion.


Revision: fe9236a9a72d284d9049a13ec8d555961ec4cb91  
Repository: <https://github.com/HAnh3112/JavaWebFinal>

- refs/remotes/origin/main


Changes

- 1. report devops update ([details](#) / [githubweb](#))

Figure 13: Jenkins pipeline run triggered by GitHub push with successful build execution












minio	e64d2263aff4	<a href="#">minio/minio</a>	<a href="#">9000:9000</a>  <a href="#">Show all ports (2)</a>	0.07%	2 hours ago	
springbootapp-run	099f15d41e6c	<a href="#">springbootapp:latest</a>	<a href="#">8091:8080</a> 	0.16%	1 hour ago	
javawebfinal	-	-	-	1.83%	1 hour ago	
grafana	fa1650c6831b	<a href="#">grafana/grafana</a>	<a href="#">3000:3000</a> 	0.43%	1 hour ago	
sql2022	cf7f49d8c931	<a href="#">mssql/server:2022-latest</a>	<a href="#">1437:1433</a> 	1.4%	1 hour ago	
prometheus	aa48bf59de3c	<a href="#">prom/prometheus</a>	<a href="#">9090:9090</a> 	0%	1 hour ago	

Figure 14: All container is created and run sequentially automatically

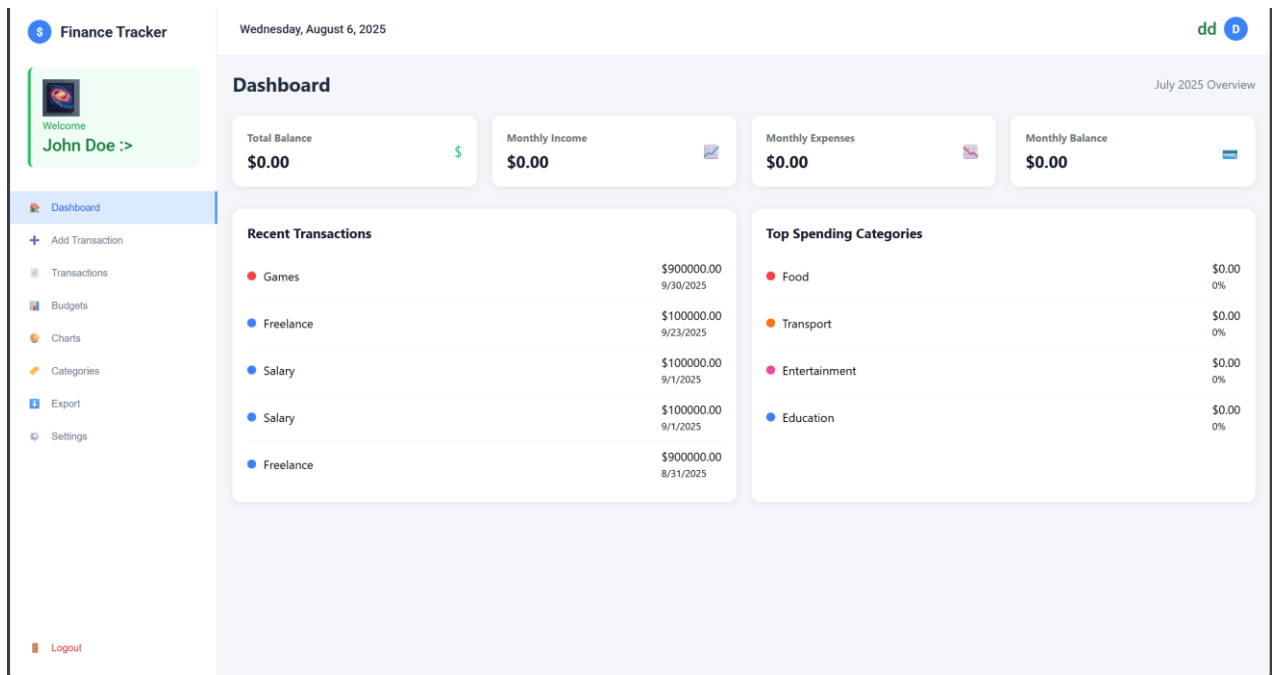


Figure 15: Project succesfully deployed at port 8091 (Image in profile is got from MinIO)



Figure 16: Prometheus graph in Grafana tracking http request and 4xx error

## CONCLUSION

In this project, we have successfully implemented a CI/CD pipeline that automates the process of building, testing, and deploying applications using Jenkins, Docker, and GitHub. By applying containerization through Docker and orchestration via Docker Compose, we were able to deploy and manage multiple services including a SQL Server database, Jenkins, MinIO object storage, Prometheus for monitoring, and Grafana for visualization.

The integration of Jenkins with GitHub enabled continuous integration, ensuring that every code change pushed to the repository was automatically built and tested. This reduces manual intervention, minimizes errors, and accelerates the development workflow. The use of Docker and Docker Hub allowed for consistent and scalable deployment across environments.

Moreover, with the help of Prometheus and Grafana, the system's performance and health were effectively monitored, enabling proactive debugging and system optimization. The implementation of MinIO demonstrated our ability to work with object storage in a cloud-native manner.

Through this project, we not only gained practical experience with modern DevOps tools but also improved our collaboration and problem-solving skills. Each team member contributed significantly, and the project was completed successfully within the planned timeframe.

This CI/CD system provides a strong foundation for future projects, offering automation, reliability, and scalability in software development and deployment workflows.