**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**

**Hà Nội, tháng - 2025**

**Bảng phân công công việc**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TT** | **Student Code** | **Full Name** | **Work Content** | **Evaluate (base 10)** | **Note** |
| 1 | BIT230500 | Nguyễn Hải Anh | Docker Compose (Database + Grafana + promotheus 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 |  |

**MỤC LỤC**

[DANH SÁCH TỪ, KÝ HIỆU VIẾT TẮT 4](#_Toc202204148)

[DANH SÁCH HÌNH VẼ 5](#_Toc202204149)

[MỞ ĐẦU 6](#_Toc202204150)

[CHƯƠNG 1. GIỚI THIỆU BÀI TOÁN 7](#_Toc202204151)

[CHAPTER 2. IMPLEMENTATION SOLUTION 8](#_Toc202204152)

[CHAPTER 3. IMPLEMENTATION RESULTS 9](#_Toc202204153)

[TÀI LIỆU THAM KHẢO 12](#_Toc202204154)

[PHỤ LỤC 13](#_Toc202204155)

# DANH SÁCH TỪ, KÝ HIỆU VIẾT TẮT

|  |  |
| --- | --- |
| **Symbol** | **Meaning** |
| BRD | Business Requirement Document |
| UC | Use case |
| BPF | Business Process Flow |
| PJ | Project |
| App | Application |
| OS | Operating System |
| F | Function |
| CRUD | Create, Read, Update, Delete |

# 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)** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**1.4. 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). |
| 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. Installation and Operation of Tools in the Local Environment**

To enable CI/CD automation locally, several tools were installed and configured:

|  |  |  |
| --- | --- | --- |
| **Tool** | **Purpose** | **Configuration** |
| **Jenkins** | Automates CI/CD | Installed on Windows, set to run as a local service, and exposed to GitHub using **Ngrok** to receive webhooks. |
| **Ngrok** | Create public URL | Used to expose the local Jenkins server via a secure tunnel (e.g., http://random-id.ngrok.io), allowing GitHub webhook integration. |
| **Git & GitHub** | Source control & trigger | Code is stored in GitHub. A webhook is configured to trigger Jenkins on each push to the main branch. |
| **Maven** | Build tool | Used to compile Java source code and generate the WAR file. Configured as Maven 3.9.9 in Jenkins Global Tool Configuration. |
| **JDK** | Java Development Kit | Configured JDK 21 as the default runtime environment in Jenkins. |
| **Apache Tomcat** | Web server | Installed locally at D:\apache-tomcat-11.0.7, used to host the compiled .war file. |
| **Docker** | Containerization | Used to build container images and run containers locally. Connected with Docker Hub for image storage and distribution. |
| **Docker Compose** | Multi-container management | Helps orchestrate multiple containers (e.g., app + database) during deployment. |
| **Docker Hub** | Image repository | Jenkins is configured to log in and push Docker images using securely stored credentials (hubdocker). |

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

To automate the process of building, deploying, and distributing the application, we designed a CI/CD pipeline that integrates **Jenkins**, **Docker**, and **GitHub**. This workflow ensures that every change in the source code is automatically built, tested (optional), packaged into a Docker image, deployed, and pushed to Docker Hub.

**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**

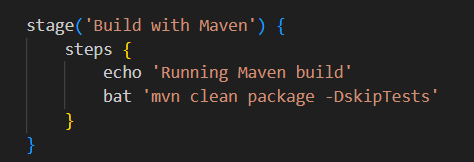
The CI/CD pipeline is defined using Declarative Pipeline Syntax and contains multiple stages, each responsible for a specific task:

**Stage 1: Clone**

Jenkins clones the latest source code from the GitHub repository into the Jenkins workspace. 

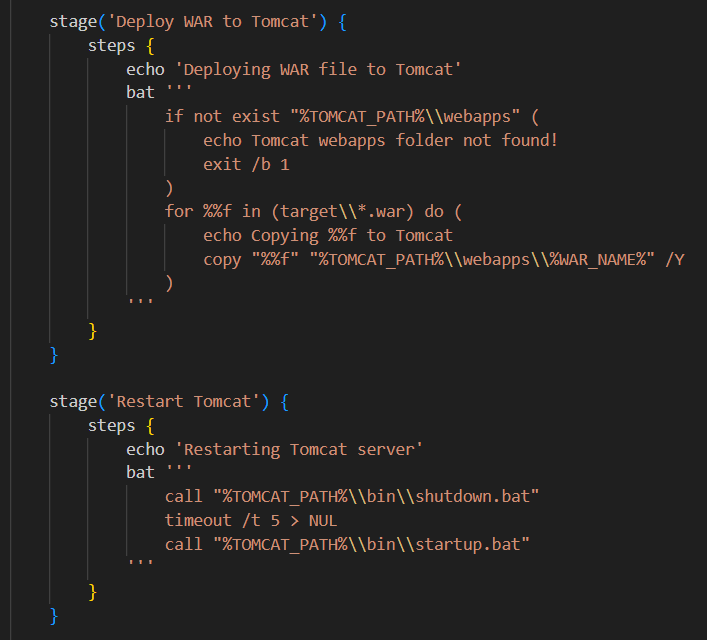
**Stage 2: Build with Maven**

The project is built using Maven inside a container based on maven:3.9.6-eclipse-temurin-21. A .war file is generated using the following command:



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

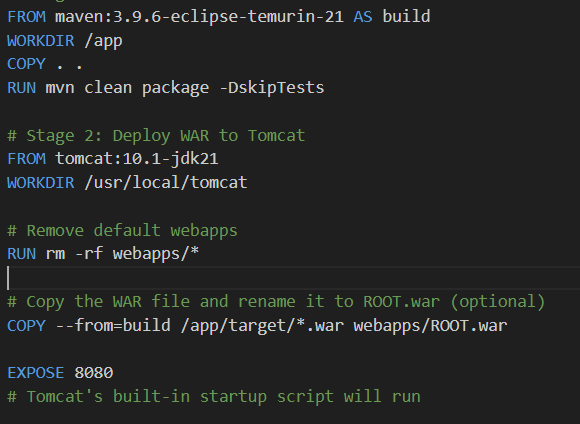
The generated WAR file is copied into the webapps folder of a local Tomcat server. Tomcat is then restarted to redeploy the application. This step currently wont work due to deployment on local machine, not on docker, making database connection invalid.



**Stage 4: Build Docker Image**

The application is containerized using a multi-stage Dockerfile:

* Stage 1: Uses Maven to build the .war file.
* Stage 2: Uses tomcat:10.1-jdk21 to host the application by copying the WAR into the container’s webapps directory.



The resulting Docker image is named springbootapp:latest.

Jenkins will then build the image automatically using the given Dockerfile:

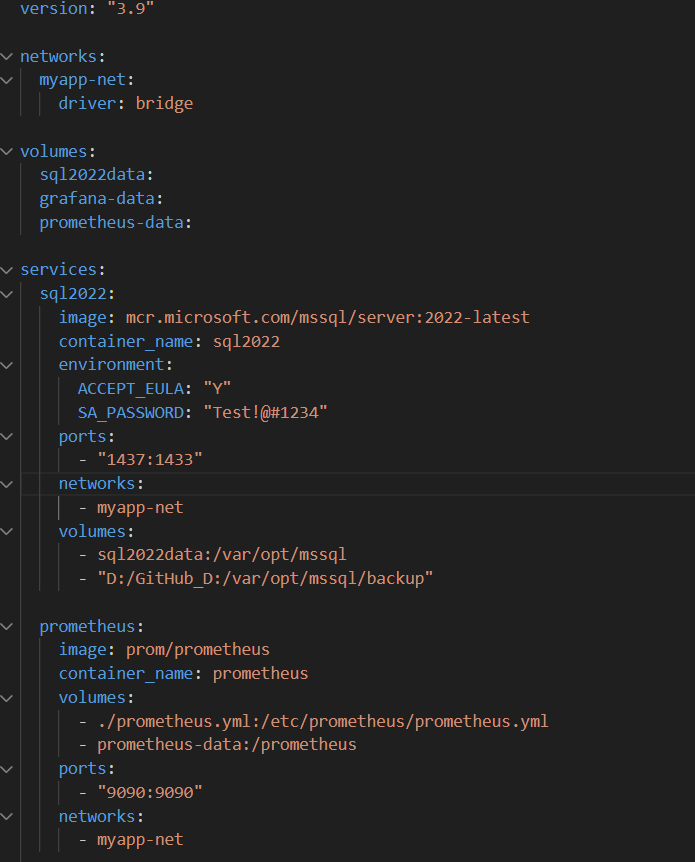


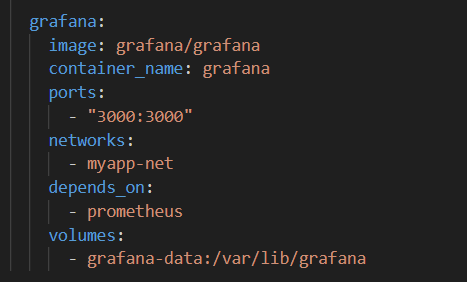
**Stage 5: Docker Compose Deployment and Monitoring Stack Integration**

To manage and deploy multiple dependent services in a unified way, we used **Docker Compose**. This tool simplifies the orchestration of containers including:

* Microsoft SQL Server 2022 (sql2022)
* Prometheus (monitoring & metrics collector)
* Grafana (visualization dashboard)

There are three services defined in a single docker-compose.yml file and share the same Docker network myapp-net.

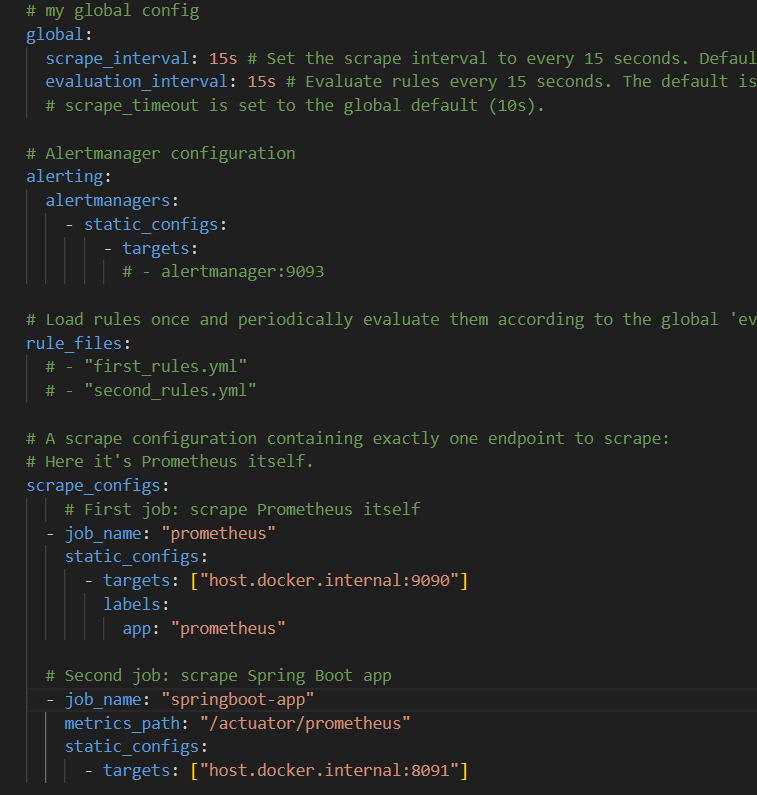




This file creates isolated containers and persistent volumes for data storage. The services are connected via a shared bridge network myapp-net.

**Monitoring Setup with Prometheus and Grafana**

* To monitor the health, performance, and metrics of the application, we integrated Prometheus and Grafana.
* Prometheus Configuration: Prometheus is configured to scrape metrics from:
  + Itself (for internal health)
  + Spring Boot application via the /actuator/prometheus endpoint

****

The Spring Boot app must include the Micrometer and Prometheus actuator dependencies to expose metrics. Prometheus runs on localhost:9090, allowing developers to query system metrics directly.

**Grafana Dashboard**

Grafana connects to Prometheus as a data source and visualizes key metrics. It runs on port 3000 and provides:

* Real-time dashboards
* Alerting (optional)
* Custom panels for memory, CPU, DB connections, etc.
* Default login: admin / admin

**Stage 6: Run MinIO container:**

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

A screen shot of a computer

AI-generated content may be incorrect.

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.

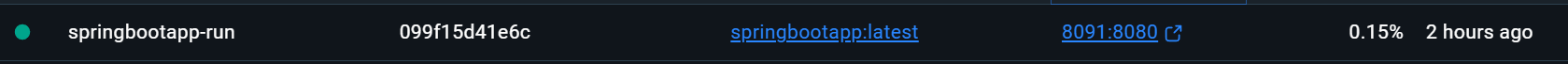


**Stage 7: Run Docker container:**

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.

A screen shot of a computer

AI-generated content may be incorrect.



**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.

A computer code with text

AI-generated content may be incorrect.

# 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**

**A screenshot of a computer

AI-generated content may be incorrect.**

*Any push to git repository will run automatically run in Jenkins (If webhooks available)*

# A screenshot of a computer AI-generated content may be incorrect.

# *All container is created and run sequentialy automatically*

# A screenshot of a computer AI-generated content may be incorrect.

# *Project succesfully deployed at port 8091 (Image in profile is got from MinIO)*

# A screenshot of a computer AI-generated content may be incorrect.

# *Promotheus graph in Grafana tracking http request and 4xx error*PHỤ LỤC