EzCars Project Operational Handbook (Team1)

Bhavin Himatkumar Goswami

Pace University

Capstone Project

Course Number: CS691

1. Abstract

The EzCars Project Operational Handbook is designed to assist team members involved in the development and management of the EzCars application, an innovative automotive management system for vehicle rentals. This document provides essential insights into the project's architecture, deployment strategies, and operational practices, serving as a helpful resource for navigating the project's complexities.

Key areas covered include the configuration of Jenkins for continuous integration and deployment, the implementation of Blue Ocean for an improved user interface, and the integration of Slack notifications for timely updates on project activities. Additionally, the handbook addresses server management issues, particularly the rationale behind upgrading the AWS EC2 instance from t2.micro to t2.medium to meet Jenkins' resource requirements.

While this handbook does not encompass every aspect of the project, it offers valuable guidance and practices that can facilitate smoother workflows and enhance collaboration among team members. By leveraging the information within this document, team members can better understand their roles and contribute effectively to the success of the EzCars project.

2. Introduction

The EzCars Project represents a significant step forward in the automotive rental industry, offering a comprehensive solution for vehicle management that streamlines operations and enhances user experience. This project aims to develop a full-stack application that facilitates vehicle rentals through a user-friendly interface, robust backend architecture, and seamless integration with various services.

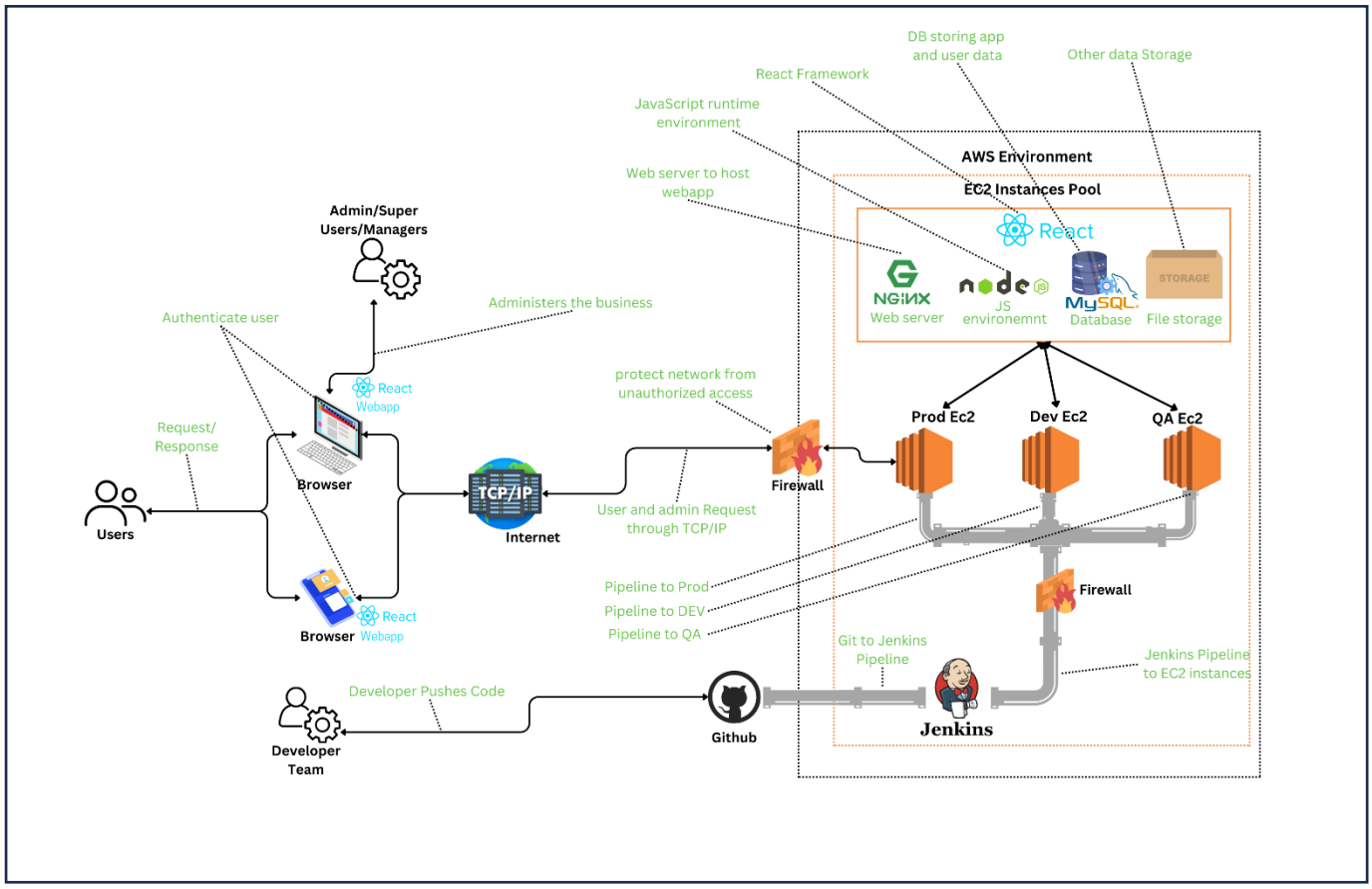
As the development team navigates the complexities of this project, effective collaboration and a shared understanding of the operational processes are crucial. The EzCars Project Operational Handbook serves as a foundational resource to guide team members through the essential components of the project, including deployment practices, system configurations, and integration strategies.

This handbook outlines the following key areas:

1. **Project Architecture**: A brief overview of the application's structure, highlighting the technologies and frameworks used in both the frontend and backend development.
2. **Deployment**: Guidance on setting up and managing the development and QA environments, including the use of AWS EC2 instances, Jenkins for continuous integration and deployment, and Nginx for serving the application.
3. **Operational Practices**: Best practices for team collaboration, communication, and project management, including the implementation of Blue Ocean for Jenkins and Slack notifications for real-time updates. While this document does not aim to cover every aspect of the project comprehensively, it provides essential insights and guidance that can aid team members in their respective roles, ultimately contributing to the overall success of the EzCars project.

3. Project Architecture

The system architecture consists of a full-stack web application deployed in an AWS environment, utilizing EC2 instances for different environments (Development, QA, and Production) with a continuous integration and deployment pipeline managed via Jenkins and GitHub. The architecture is broken down into the following components:



1. Frontend

- The frontend is built using the **React** framework, served via an **Nginx** web server on each EC2 instance (Dev, QA, and Prod).
- Admins and super users can authenticate and manage the business operations via the React web application hosted on the server.
- Users access the application through their browsers via standard HTTP(S) protocols over TCP/IP, interacting with the web app to receive responses.

2. Backend & Database

- The backend consists of **Node.js** for the runtime environment and a **MySQL** database for storing both application and user data. The backend is deployed across multiple EC2 instances, each serving a different environment (Dev, QA, Prod).
- A dedicated EC2 instance pool hosts these environments, ensuring isolation for development and testing purposes.
- Other storage needs (such as media files or documents) are handled via a separate file storage system also within the AWS environment.

3. Pipeline and Version Control

- **GitHub** is used for version control, where developers push their code. The code is then picked up by **Jenkins**, which manages the continuous integration pipeline.
- Jenkins builds and deploys the application to the appropriate environment (Dev, QA, Prod) based on the pipeline configurations. The pipeline is capable of automatically pushing code to these environments upon successful integration testing.

4. Security and Firewalls

- To protect the network from unauthorized access, a firewall is set up between the user/admin and the EC2 instances.
- Firewalls also segregate access between different environments (Prod, Dev, QA), ensuring that requests and data do not accidentally flow between them.

5. Infrastructure and Scaling

- The AWS environment is scalable, with EC2 instances able to handle various traffic loads in each environment.
- Each environment is separately hosted, and the architecture supports scaling based on demand,making it capable of handling production-level loads without impacting Dev or QA environments.

6. User Interaction

- Users interact with the system through a browser interface where they submit requests and receive responses after authentication.
- Admins have additional access and are responsible for managing and administering the business through a React web interface.

4. How to Install Jenkins on an AWS EC2 Instance

1. **Launch EC2 Instance**

Choose Ubuntu 20.04 LTS and configure security groups to allow SSH (22) and HTTP (80).

2. **Connect via SSH**

ssh -i /path/to/your-key.pem ubuntu@your-ec2-public-ip

3. **Install Java**

sudo apt update

sudo apt install openjdk-11-jdk -y

4. **Install Jenkins**

curl -fsSL https://pkg.jenkins.io/debian/jenkins.io-2023.key | sudo tee/usr/share/keyrings/jenkins-keyring.asc > /dev/null

sudo sh -c 'echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] https://pkg.jenkins.io/debian binary/ > /etc/apt/sources.list.d/jenkins.list'

sudo apt update

sudo apt install jenkins -y

5. **Start Jenkins**

sudo systemctl start jenkins

sudo systemctl enable jenkins

6. **Open Jenkins in Browser**

- Access Jenkins at http://your-ec2-public-ip:8080.
- Get the initial admin password:

sudo cat /var/lib/jenkins/secrets/initialAdminPassword

7. **Configure Security Group**

- Open **port 8080** for Jenkins in your AWS Security Group.

5. Configure Jenkins Pipeline for CI/CD

Jenkins pipelines can be set up for **QA**, **Dev**, and **Prod** environments. Below is a general pipeline configuration, but keep in mind that the pipeline script may vary depending on the specific requirements of each environment.

Step 1: Create a New Jenkins Pipeline Job

1. In Jenkins, select **New Item**, then choose **Pipeline**.
2. Name your job (e.g., "CI/CD Pipeline") and click **OK**.
3. In the pipeline configuration, under **Pipeline** section, set **Definition** to **Pipeline script**.

Step 2: Example Pipeline Script

The following example shows how you can deploy the application to different environments. Adjust the script for your specific setup (e.g., different build commands or servers):

```
pipeline {
    agent any

    environment {
        DEV_SERVER = 'ec2-user@dev-instance-ip'
        QA_SERVER = 'ec2-user@qa-instance-ip'
        PROD_SERVER = 'ec2-user@prod-instance-ip'
        SSH_KEY = credentials('ssh-private-key')
        APP_DIR = '/var/www/app'
    }

    stages {
        stage('Clone Repository') {
            steps {
                git branch: 'main', url: 'git@github.com:your-repo-url.git'
            }
        }

        stage('Install Dependencies') {
            steps {
                sh 'npm install'
            }
        }

        stage('Run Tests') {
            steps {
                sh 'npm test'
            }
        }

        stage('Build') {
            steps {
                sh 'npm run build'
            }
        }

        stage('Deploy to Dev') {
            when {
                branch 'dev'
            }
```

```
            steps {
                sh """
                scp -i ${SSH_KEY} -r build/* ${DEV_SERVER}:${APP_DIR}
                ssh -i ${SSH_KEY} ${DEV_SERVER} 'sudo systemctl restart nginx'
                """
            }
        }

        stage('Deploy to QA') {
            when {
                branch 'qa'
            }
            steps {
                sh """
                scp -i ${SSH_KEY} -r build/* ${QA_SERVER}:${APP_DIR}
                ssh -i ${SSH_KEY} ${QA_SERVER} 'sudo systemctl restart nginx'
                """
            }
        }

        stage('Deploy to Prod') {
            when {
                branch 'main'
            }
            steps {
                sh """
                scp -i ${SSH_KEY} -r build/* ${PROD_SERVER}:${APP_DIR}
                ssh -i ${SSH_KEY} ${PROD_SERVER} 'sudo systemctl restart nginx'
                """
            }
        }
    }

    post {
        always {
            echo 'Pipeline Completed!'
        }
    }
}
```

Step 3: Configure the Pipeline

- Add your environment variables (EC2 instances, SSH keys, directories) in the **environment** block.
- **Branches**: Each stage (Dev, QA, Prod) is triggered based on the branch being deployed (dev, qa, main).
- **Deployment**: Uses **SCP** and **SSH** to copy the build files to the appropriate environment and restart Nginx to serve the new version of the application.

Step 4: Customize the Script for Different Environments

- Adjust the build, test, or deploy commands according to the requirements of each environment.
- QA, Dev, and Prod environments may have different configurations or scripts to follow.

This configuration can be expanded or modified based on your specific environment setup, ensuring an efficient and automated CI/CD process for your project.

6. Ubuntu Commands for Debugging

This section provides a comprehensive list of useful Ubuntu commands for debugging issues related to the EzCars project. These commands can help identify problems in the system, monitor processes, and check configurations.

Common Debugging Commands

1. **SSH Login to Remote Server**
    - Command: ssh <username>@<hostname_or_IP>
    - Description: Establish a secure shell connection to a remote server. Replace <username> with your username and <hostname_or_IP> with the server's address.
2. **View System Logs**
    - Command: tail -f /var/log/syslog
    - Description: Continuously monitor system log messages for troubleshooting. This is useful for catching errors as they occur.
3. **Check Disk Usage**
    - Command: df -h
    - Description: Display the disk space usage for all mounted filesystems, helping to ensure there is sufficient space for application operations.
4. **Monitor Resource Usage**
    - Command: top
    - Description: View real-time system resource usage, including CPU and memory. Useful for identifying resource hogs.
5. **Check Running Processes**
    - Command: ps aux | grep <process_name>
    - Description: List all running processes and filter for a specific process by name to see if it’s active.
6. **Network Connectivity Test**
    - Command: ping <hostname_or_IP>
    - Description: Check network connectivity to a specific host or IP address to ensure network availability.
7. **View Service Status**
    - Command: systemctl status <service_name>

- Description: Check the status of a specific service (e.g., Jenkins, Nginx) to confirm it's running correctly.

8. **Check Application Logs**
    - Command: tail -f /var/log/<application_name>.log
    - Description: Continuously monitor the application log file for error messages and performance insights.
9. **Check Firewall Rules**
    - Command: sudo ufw status
    - Description: View the current firewall rules and their status to ensure required ports are open.
10. **View Open Ports**
    - Command: sudo netstat -tuln
    - Description: List all open ports and the corresponding services to ensure applications are reachable.
11. **Check Java Version**
    - Command: java -version
    - Description: Verify the installed version of Java to ensure compatibility with your application.
12. **Check Memory Usage**
    - Command: free -m
    - Description: Display memory usage, including total, used, free, and available memory.
13. **Disk I/O Monitoring**
    - Command: iotop
    - Description: Monitor disk I/O usage by processes in real-time, useful for identifying bottlenecks.
14. **Check for Disk Errors**
    - Command: dmesg | grep -i error
    - Description: View kernel-related error messages, including disk-related issues.
15. **Show System Uptime**
    - Command: uptime
    - Description: Display how long the system has been running, along with load averages.

16. **Check TCP Connections**
    - Command: ss -tuln
    - Description: Display all active TCP connections and listening ports, similar to netstat.
17. **Monitor Log Files with Specific Filters**
    - Command: grep 'ERROR' /var/log/<application_name>.log
    - Description: Search for specific keywords (e.g., ERROR) in log files to quickly locate issues.
18. **View System Resource Usage Over Time**
    - Command: sar -u 1
    - Description: Collect and report system activity over time, particularly useful for CPU usage.
19. **Check Installed Packages**
    - Command: dpkg -l
    - Description: List all installed packages, useful for verifying dependencies.
20. **Check Environment Variables**
    - Command: printenv
    - Description: Display all environment variables to ensure they are set correctly.
21. **Killing Processes**
    - Command: kill <pid>
    - Description: Terminate a process using its process ID (PID), which can be found using ps or top.
22. **Remove a Package**
    - Command: sudo apt remove <package_name>
    - Description: Uninstall a specific package from the system. Replace <package_name> with the name of the package to remove.

7. Default Paths

Here are the default paths relevant to the EzCars project setup:

- **Jenkins Home Directory**: /var/lib/jenkins
  - Contains Jenkins configuration files, job data, and workspace.
- **Application Logs**: /var/log/ezcars.log
  - Location for logging output from the EzCars application.
- **Nginx Configuration Directory**: /etc/nginx
  - Contains configuration files for Nginx, including nginx.conf.
- **Deployment Directory**: /var/www/html
  - Default directory where the EzCars application is deployed.
- **Docker Directory**: /var/lib/docker
  - Contains Docker images, containers, and volumes.
- **AWS CLI Configuration Directory**: ~/.aws
  - Contains configuration files for AWS CLI.
- **Java Installation Path**: /usr/lib/jvm/java-11-openjdk-amd64/
  - Default installation path for OpenJDK 11.
- **Node.js Installation Path**: /usr/local/lib/node_modules
  - Default path for globally installed Node.js packages.
- **MySQL Data Directory**: /var/lib/mysql
  - Default location for MySQL database files.
- **SSL Certificates Directory**: /etc/ssl/certs
  - Default directory for storing SSL certificates.

8. Personal Insights and Learnings

Throughout this project, particularly during Sprint 1, I have encountered various challenges and valuable learning experiences that have significantly enhanced my understanding of CI/CD practices and Jenkins configuration.

One of the first issues I faced was related to resource limitations. I initially set up Jenkins on a t2.micro instance, but quickly ran into out of memory issues. Jenkins requires a heap size of 1024 MB, but t2.micro only offers 1 GB of RAM, leading to performance problems. To resolve this, I upgraded the instance to t2.medium, which provided the necessary memory and improved overall stability.

I also explored Blue Ocean, Jenkins' modern interface that simplifies pipeline creation and management. Additionally, I integrated Slack notifications into the pipeline to receive real-time updates on build statuses. Slack notifications streamline communication, allowing me to quickly respond to issues or successful builds without constantly monitoring Jenkins.

One critical learning came during the setup of GitHub webhooks. While configuring the webhook, I spent several hours debugging an issue that stemmed from using the wrong URL. I mistakenly used a remote URL instead of localhost, which caused the webhook to fail. This taught me the importance of correct URL usage (local or remote) during webhook setup. I found that diagramming the architecture and noting the specific URLs for both local and remote access helped me understand the flow better and avoid similar mistakes in the future. While it was a time-consuming challenge, it was also a fun and rewarding experience once I figured it out.
Jenkins also stood out as an extremely flexible tool. I was impressed by its plugin support, which allows for customizable solutions based on specific project needs. For example, I implemented Slack notifications using a plugin and learned how to extend Jenkins' capabilities for better integration and communication.

When configuring GitHub access, I used a Personal Access Token (PAT) by creating a global variable in Jenkins. I set it as secret text to securely manage repository access, which was a straightforward and effective solution for repository authentication.

One final key takeaway was learning the importance of managing disk space on the Jenkins instance. As Jenkins frequently clones repositories and builds applications in the workspace, it's easy for the disk to fill up. I realized that clearing unnecessary builds and removing caches helped maintain optimal performance and prevent space-related issues.

Overall, this project was a great learning experience, full of both challenges and growth opportunities. I found it especially satisfying to troubleshoot and solve complex issues, and I now have a deeper appreciation for the power and flexibility of Jenkins as a CI/CD tool.

9. Current Progress of CI/CD Implementation

As of now, the EzCars project team has made significant strides in establishing a robust Continuous Integration and Continuous Deployment (CI/CD) pipeline. Here's a summary of our current progress:

1. **Jenkins Configuration:**
    - **Setup:** Jenkins has been successfully installed on the development instance, and configurations have been tailored to meet our project requirements.
    - **Plugins:** Key plugins, including the Blue Ocean plugin for enhanced visualization of the CI/CD pipeline, have been integrated to improve user experience and facilitate better monitoring.
2. **Pipeline Development:**
    - **Pipeline Creation:** A declarative Jenkins pipeline has been created to automate the build, test, and deployment processes for the EzCars application.
    - **Stages Defined:** The pipeline currently includes stages for:
        - **Build:** Compiling the application and packaging it into a deployable format.
        - **Test:** Running unit tests to ensure code quality and functionality.
        - **Deploy:** Deploying the application to the designated environment (Dev or QA) using Docker.
3. **Slack Notifications:**
    - **Integration:** Slack notifications have been implemented to keep the team informed about build statuses and deployment events. This integration helps facilitate quick communication and prompt action in case of build failures.
4. **Environment Stability:**
    - **Instance Upgrade:** The t2.micro instance was upgraded to t2.medium to accommodate Jenkins' memory requirements, reducing out-of-memory issues and enhancing overall stability during builds.

5. **Elastic IP Assignment:**
    - **Elastic IPs:** Both the development and QA environments have been assigned Elastic IP addresses, ensuring stable access to the Jenkins interface and other deployed applications.
6. **Collaborative Efforts:**
    - **Team Involvement:** Team members have actively contributed to the pipeline development, ensuring that best practices in CI/CD are followed and lessons learned are documented for future sprints.
7. **Next Steps:**
    - **Monitoring and Optimization:** As we move forward, efforts will be focused on optimizing the CI/CD pipeline for better performance and reliability.
    - **Documentation**: I will be updating this document regularly to reflect new developments and changes in our CI/CD processes, ensuring that all team members have access to the latest information.

9. References

- Jenkins Documentation

  Jenkins Official Documentation: https://www.jenkins.io/doc/

  This was an essential resource for understanding Jenkins configuration, plugin usage, and pipeline setup.
- AWS Documentation

  AWS EC2 Instance Types: https://aws.amazon.com/ec2/instance-types/

  This guide provided detailed information about EC2 instances, helping me decide on upgrading from t2.micro to t2.medium.
- GitHub Documentation

  GitHub Webhooks Guide: https://docs.github.com/en/developers/webhooks-and-events/webhooks

  I referred to this when setting up webhooks and learning about URL configurations.
- Blue Ocean Plugin

  Blue Ocean Jenkins Plugin: https://www.jenkins.io/doc/book/blueocean/

  This source provided insights into using Blue Ocean to improve the user interface for pipeline creation.
- Slack Notifications for Jenkins

  Jenkins Slack Plugin: https://plugins.jenkins.io/slack/

  Used this documentation to implement Slack notifications in the pipeline for real-time build updates.
- Nginx Documentation

  Nginx Web Server Configuration: https://nginx.org/en/docs/

  Referred to for restarting Nginx after deploying builds to ensure the application was served correctly.
- OpenAI GPT (Generative Pre-trained Transformer)

  OpenAI GPT-4 Documentation: https://openai.com/research/gpt-4

  Used ChatGPT, powered by GPT-4, for generating content ideas, structuring the project write-up, and providing technical insights. GPT-4 was instrumental in crafting the "Personal Insights and Learnings" section and refining the overall presentation of the project.