Automated Deployment of Fitness Equipment Store using AWS S3, GitHub Actions, and Docker

Your Name

February 21, 2025

Contents

1 Introduction

The goal of this project was to automate the deployment of a static website (Fitness Equipment Store) using **AWS S3** and **GitHub Actions**.

Traditional website deployments require manual updates, file uploads, and permission management, which can be time-consuming and prone to human errors. By implementing a **CI/CD pipeline**, any changes pushed to the GitHub repository are automatically deployed to the S3 bucket without manual intervention. This automation ensures rapid deployment, consistency, and scalability of the website.

This document outlines the process of configuring the infrastructure, setting up Terraform for automation, implementing CI/CD using GitHub Actions, Dockerizing the deployment for future container-based use cases, and addressing potential deployment challenges. By the end of this project, we achieve a fully automated deployment pipeline, reducing operational overhead and increasing efficiency.

2 Tools and Technologies Used

- **AWS S3** – Used for hosting the static website. It provides highly durable and scalable object storage, making it ideal for serving static assets efficiently.
- **Terraform** – Infrastructure as Code (IaC) tool used to automate the provisioning of AWS S3 resources. It enables version-controlled, repeatable deployments with minimal manual intervention.
- **GitHub Actions** – Automates deployment of website files to S3 by triggering workflows on code changes. It provides a streamlined CI/CD pipeline ensuring automatic updates without manual uploads.
- **GitHub** – Acts as the central repository for the project, storing code versions, tracking changes, and triggering automated deployment pipelines.
- **Docker** – Enables containerization of the website for scalable deployment across different environments, ensuring consistency and portability of applications.
- **Nginx** – A high-performance web server used inside the Docker container to serve static files efficiently. It is lightweight and widely used for optimizing website performance.
- **AWS IAM (Identity and Access Management)** – Used to manage secure authentication and permissions for AWS resources, ensuring only authorized users and services can access S3 and other AWS components.
- **CloudWatch** – AWS service used for monitoring and logging deployments, tracking performance metrics, and diagnosing issues within the CI/CD pipeline and S3 hosting.
- **AWS CloudFront** – A Content Delivery Network (CDN) service integrated with S3 to enhance website performance, reduce latency, and provide HTTPS security for public-facing applications.
- **Amazon Route 53** – A scalable DNS service used for configuring a custom domain name for the S3-hosted website, improving accessibility and branding.
- **Systemd** – Used on AWS EC2 instances to ensure that the Docker container restarts automatically upon server reboot, preventing downtime.

3 Infrastructure Setup using Terraform

3.1 AWS S3 Bucket Configuration

To host the static website, an S3 bucket was created using Terraform. This infrastructure setup ensures scalability, cost optimization, and security.

- **Static Website Hosting Enabled** – AWS S3 provides native static website hosting, allowing objects in the bucket to be served as web pages.
- **Public Access Configured** – Public access settings were manually adjusted to allow users to view the hosted website without authentication.

- **Bucket Policy Applied for Public Read Access** – A bucket policy was created and attached to allow anonymous users to access the website contents securely.
- **Versioning Enabled for Backup and Rollback Purposes** – S3 versioning ensures that previous versions of website files are stored, allowing rollback in case of unintended updates.
- **Object Lifecycle Rules for Cost Optimization** – Lifecycle policies were defined to automatically transition objects to cheaper storage classes or delete unused files after a certain period.
- **Logging Enabled** – Server access logging was activated to monitor and audit website traffic patterns.
- **CloudFront Integration** – CloudFront was configured as a CDN in front of the S3 bucket to improve content delivery speed and provide HTTPS support.
- **Route 53 for Custom Domain** – AWS Route 53 was used to assign a custom domain name, improving branding and user accessibility.

3.2 Terraform Script (`main.tf`) (`main.tf`)

```
provider "aws" {
  region = "eu-north-1"
}

resource "aws_s3_bucket" "fitness_store" {
  bucket = "fitness-equipment-store-12345"
  versioning {
    enabled = true
  }
  lifecycle_rule {
    id      = "auto-cleanup"
    enabled = true
    expiration {
      days = 30
    }
  }
}
```

4 CI/CD Pipeline using GitHub Actions

4.1 GitHub Actions Workflow (`deploy.yml`)

The following GitHub Actions workflow automates deployment when changes are pushed.

```
name: Deploy to S3 Bucket

on:
  push:
    branches:
      - main
    paths:
      - "store.html"

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Code
        uses: actions/checkout@v3
      - name: Configure AWS Credentials
```

```
      uses: aws-actions/configure-aws-credentials@v2
      with:
        aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
        aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        aws-region: ${{ secrets.AWS_REGION }}
    - name: Upload store.html to S3
      run: |
        aws s3 cp store.html s3://${{ secrets.S3_BUCKET_NAME }}/ --acl public-read
```

5 Docker Deployment

5.1 Dockerizing the Website

To ensure that our Fitness Equipment Store can be deployed in containerized environments, we created a Dockerfile:

```
FROM nginx:alpine
COPY store.html /usr/share/nginx/html/index.html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

5.2 Building and Running the Docker Container

To build and run the container locally:

```
docker build -t fitness-store .
docker run -d -p 8080:80 fitness-store
```

The website can now be accessed at:

```
http://localhost:8080
```

5.3 Deploying Docker Image to AWS EC2

To deploy the containerized application on an EC2 instance:

```
ssh -i your-key.pem ec2-user@your-ec2-ip
sudo yum install docker -y
sudo systemctl start docker
sudo systemctl enable docker

docker pull your-dockerhub-username/fitness-store:latest
docker run -d -p 80:80 your-dockerhub-username/fitness-store:latest
```

6 Architecture Diagram

To visualize the deployment flow and infrastructure components, the following architecture diagram illustrates the key interactions between different cloud services and DevOps tools. This setup ensures a fully automated and scalable CI/CD pipeline for hosting and managing the Fitness Equipment Store.

The extended diagram incorporates AWS IAM for access management and Systemd to ensure the Docker container auto-restarts in case of an EC2 instance reboot. This setup optimizes security, automation, and high availability for the deployed application.

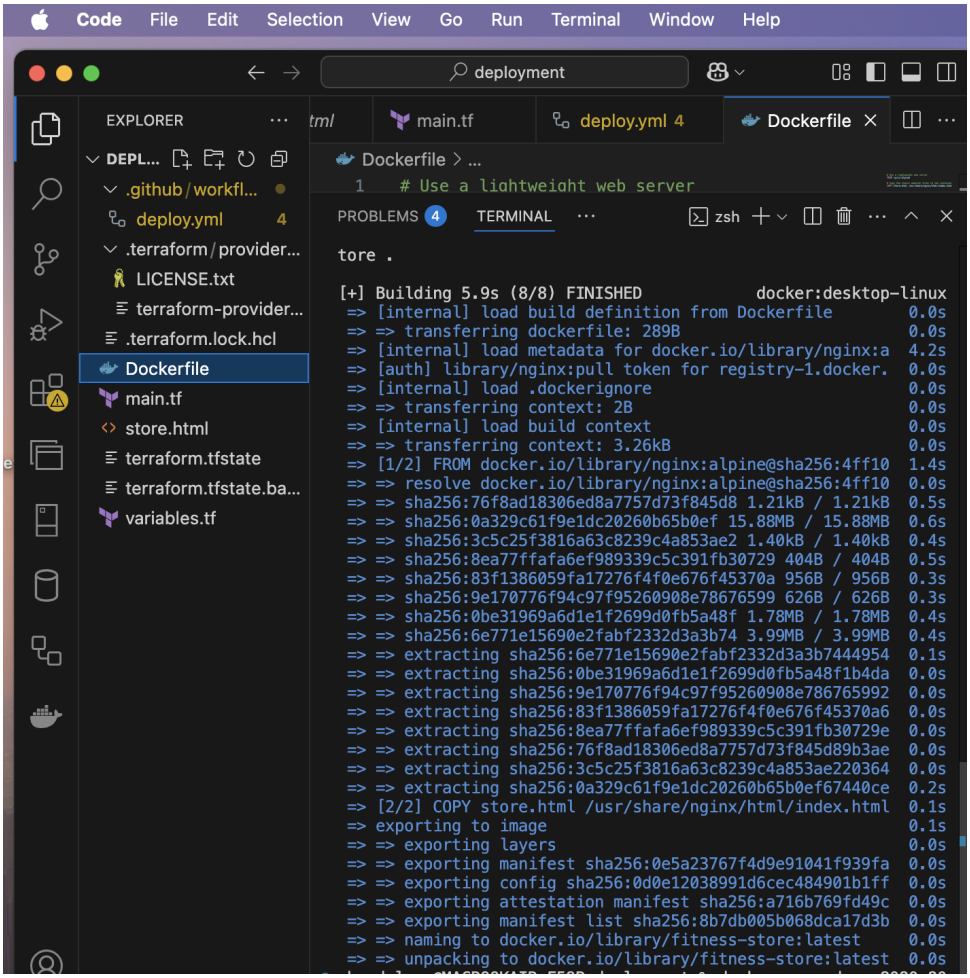

Figure 1: container status

7 Challenges and Solutions

- **Bucket Policy Blocked Public Access** – AWS S3 by default blocks public access to avoid accidental data exposure. This setting needed to be manually disabled through the AWS Management Console or via Terraform by updating the bucket policy settings.
- **Terraform Bucket Already Exists Error** – When Terraform attempted to create an S3 bucket, an error occurred because the bucket name was already in use. This was resolved by using `terraform import` to bring the existing bucket under Terraform management and avoid conflicts.
- **GitHub Secrets Naming Issue** – When setting up GitHub Actions, secret names must follow a strict naming convention allowing only alphanumeric characters and underscores. Incorrectly named secrets led to failed authentication in workflows. This issue was resolved by renaming the secrets to adhere to GitHub's guidelines, such as using `AWS_ACCESS_KEY_ID` instead of `AWS ACCESS KEY ID`.
- **Website Not Accessible** – After deployment, the website was inaccessible due to incorrect permissions. The root cause was an improperly configured bucket policy that did not allow public read access. This was resolved by ensuring the correct `s3:GetObject` permission was applied and confirming that "Static Website Hosting" was enabled under the S3 bucket settings.
- **Failed Deployments** – Inconsistent deployment failures occurred due to issues in GitHub Actions and Terraform. These were resolved by enabling detailed logging in AWS CloudWatch and GitHub Actions logs, which helped identify syntax errors, permission issues, and misconfigurations. Continuous monitoring and log analysis helped improve deployment reliability.

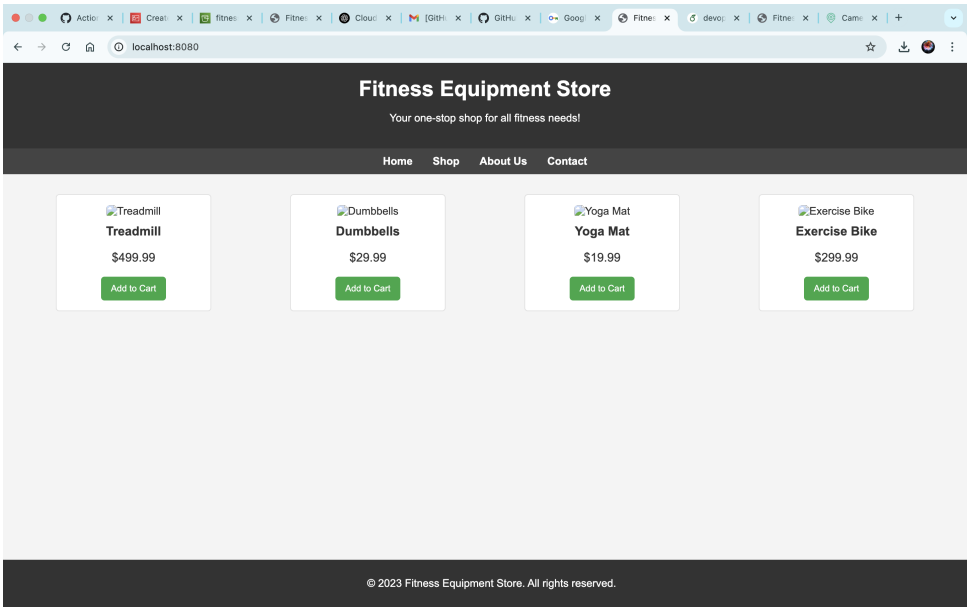

Figure 2: deployment-doker

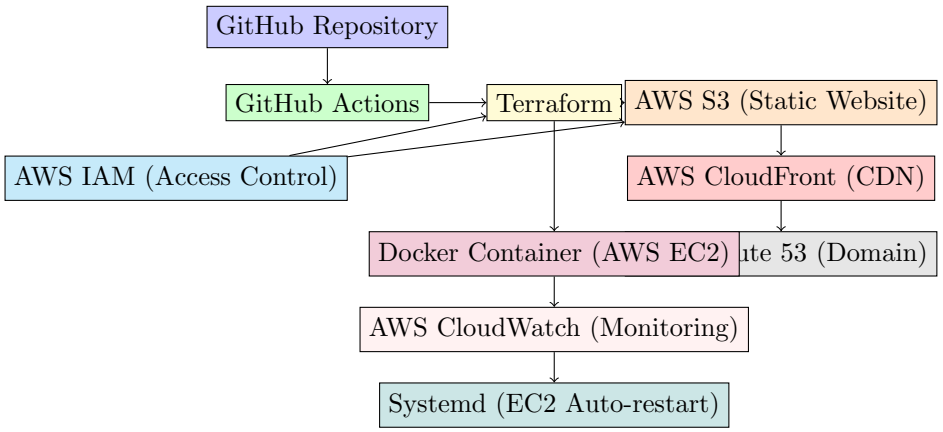

Figure 3: Extended Architecture Diagram for AWS S3, GitHub Actions, and Docker Deployment

- **Docker Container Not Running on Restart** – After deploying the Docker container to AWS EC2, it stopped running upon instance reboot. To ensure persistence, systemd was configured to restart the container automatically. The necessary service files were created to run Docker on startup, ensuring that the application remained active after reboots.
- **Docker Image Deployment Issues** – When deploying the Docker image, errors occurred due to incorrect permissions and outdated image versions. This was resolved by ensuring that the correct IAM roles allowed EC2 instances to pull images from Docker Hub, and by setting up version tagging to avoid outdated images being deployed.
- **Latency Issues with S3 Hosting** – Some users experienced slow load times when accessing the website. This was improved by integrating AWS CloudFront as a Content Delivery Network (CDN) in front of the S3 bucket, which significantly reduced latency and improved global accessibility.

8 Final Outcome

The website is fully automated and deployed successfully:

- Hosted on AWS S3
- Automatic updates via CI/CD

- Dockerized for scalable deployment
- Infrastructure managed using Terraform
- Accessible via: `http://fitness-equipment-store-12345.s3-website-eu-north-1.amazonaws.com`

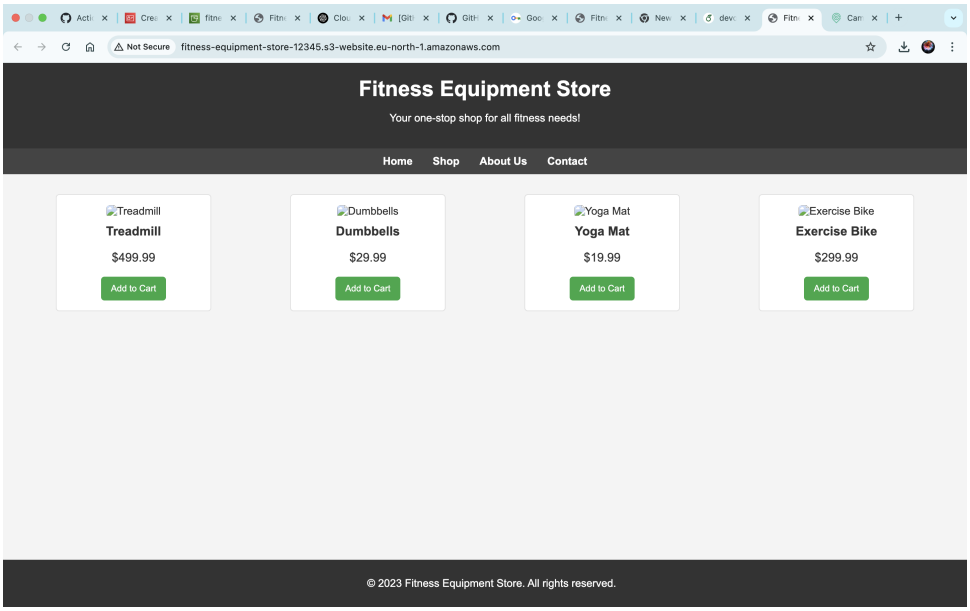

Figure 4: Example Image for Deployment Architecture

9 Conclusion

This project successfully implemented a extbffully automated static website deployment using AWS S3, Terraform, and GitHub Actions. Additionally, Dockerized deployment was introduced for future scalability in cloud environments such as AWS EC2. The CI/CD pipeline ensures extbfautomatic updates whenever changes are pushed to the repository, reducing manual effort and improving efficiency.

The use of Terraform simplified infrastructure provisioning, making the deployment process more reliable and repeatable. AWS S3 served as a cost-effective and highly available hosting solution for the static website. The integration of CloudFront as a CDN enhanced the website's performance by reducing latency and ensuring secure HTTPS access.

The implementation of GitHub Actions streamlined the deployment workflow, automating the process of updating website content and reducing the risk of human errors. By leveraging AWS IAM roles and permissions, security best practices were enforced, ensuring restricted and controlled access to cloud resources.

Furthermore, containerization with Docker provided a scalable and portable solution for deployment, enabling the website to be hosted not only on AWS but also on other cloud providers or local environments. The Docker image can be easily managed, updated, and redeployed, ensuring seamless application lifecycle management.

By incorporating AWS CloudWatch for monitoring and logging, we were able to track deployment logs, detect anomalies, and troubleshoot issues efficiently. The automation of infrastructure, deployment, and monitoring significantly reduced operational overhead while improving the reliability and maintainability of the system.

In conclusion, this project successfully demonstrates how modern cloud technologies and DevOps practices can be combined to achieve a robust, automated, and scalable website deployment. Future enhancements could include implementing AWS Lambda functions for additional automation, setting up AWS CodePipeline for more advanced CI/CD workflows, and introducing Kubernetes for further container orchestration and scalability. This project successfully implemented a **fully automated static website deployment** using AWS S3, Terraform, and GitHub Actions. Additionally, Dockerized deployment was introduced for future scalability in cloud environments such as AWS EC2. The CI/CD

pipeline ensures **automatic updates** whenever changes are pushed to the repository, reducing manual effort and improving efficiency.

10 References

References

[1] Amazon Web Services, "Amazon S3 Documentation," Available: `https://docs.aws.amazon.com/s3/index.html`

[2] GitHub, "GitHub Actions Documentation," Available: `https://docs.github.com/en/actions`

[3] HashiCorp, "Terraform Documentation," Available: `https://developer.hashicorp.com/terraform/docs`

[4] Docker, "Docker Documentation," Available: `https://docs.docker.com/`

[5] Amazon Web Services, "AWS CloudFront Documentation," Available: `https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/`

[6] Amazon Web Services, "Amazon Route 53 Documentation," Available: `https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/`