DevOps Project Documentation: Java App CI/CD to EC2 with Terraform and Ansible

Phase 1: Infrastructure Provisioning with Terraform

Objective

Provision an EC2 instance with a public IP using Ubuntu AMI, including: - SSH key pair - Security group for NGINX/HTTP and SSH - Required instance size and tags

What We Did

- Created a modular Terraform setup with modules for EC2, key pair, and security group.
- Deployed the EC2 instance in the `eu-north-1a` availability zone.
- Used a custom SSH key named `java-project-01-key`.
- Used Terraform state commands like `terraform state list` to inspect current resources.
- Destroyed previous EC2-related resources using `terraform destroy -target` commands.
- Reapplied the Terraform plan to create a clean Ubuntu-based EC2 instance.

Issues Faced

- The EC2 instance originally reverted to Amazon Linux after account reinstatement.
- SSH key mismatches due to old keys being associated with the previous instance.

Fixes Applied

- We deleted previous Terraform-managed resources.
- Generated new SSH keys and uploaded the public key into `~/.ssh/authorized_keys` on the EC2 instance.
- Recreated the EC2 infrastructure from scratch using Terraform.

Phase 2: CI Pipeline with GitHub Actions

Objective

Implement a CI pipeline that builds a Java WAR file, runs tests, scans with SonarQube and Trivy, and uploads artifacts to GitHub Releases and Docker Hub.

What We Did

- Created `.github/workflows/ci-project-01.yml`.
- Set up Maven build using `mvn clean verify`.
- Uploaded the `.war` file as a GitHub Action artifact.

- Integrated SonarQube with JaCoCo test coverage using secrets for `SONAR_TOKEN` and `PROJECT_KEY`.
- Used `ncipollo/release-action@v1` to publish releases based on branch:
- For `master`: Created production-style release `v1.0.X`
- For `project-01`: Created `dev-` prereleases.
- Scanned Docker image with Trivy and pushed to Docker Hub.

Issues Faced

- `.war` file was not found in some cases due to incorrect path.
- CI didn't upload `.war` for feature branches initially.
- Trivy scan failed due to incorrect Docker build context.

Fixes Applied

- Ensured Maven build was happening in correct working directory.
- Updated artifact upload paths to match generated files.
- Modified GitHub Actions to allow uploads from both `master` and `project-01`.
- Fixed Trivy step by correctly specifying Docker context.

Phase 3: CD Pipeline with GitHub Actions and Ansible

Objective

Trigger deployment to EC2 instance using Ansible whenever a GitHub Release is published.

What We Did

- Created `.github/workflows/cd-project-01.yml`.
- Used the `release` trigger to start the deployment pipeline.
- Downloaded the latest `.war` file from GitHub Release.
- Copied the `.war` to EC2 using `scp`.
- Cloned the `ps-terraform-live-envs` repo to retrieve the Ansible playbook.
- Installed Ansible on the runner.
- Ran playbook that:
- Installed Java
- Created `tomcat` user
- Downloaded and extracted Tomcat
- Deployed WAR into `/webapps/` folder
- Restarted Tomcat

Issues Faced

- CD did not trigger even after release creation.
- `.war` file copy failed due to incorrect SSH key or user.
- Ansible playbook failed due to missing files or wrong inventory path.

Fixes Applied

- Confirmed that release events come from `master` (not `project-01`) branch.
- Added debug steps to confirm `.war` download and repo cloning.
- Overwrote `inventory.ini` dynamically within the GitHub Actions job.
- Ensured the correct private key was used for SSH by setting it in the GitHub secret and writing it to the runner.

Phase 4: End-to-End Validation and Cleanup

Objective

Validate full CI/CD pipeline and ensure automation is hands-free.

What We Did

- Verified CI pipeline works on code push.
- Triggered CD pipeline automatically on GitHub Release creation.
- Validated application deployment to EC2 via browser and logs.

Issues Faced

- Some files were missing due to incorrect repo paths (e.g., capitalization issues like `Java-project-01` vs `java-project-01`).
- Manual SSH worked but CD failed due to path mismatches.

Fixes Applied

- Corrected folder paths in playbook and workflow.
- Added fallback checks (`|| echo not found`) in clone step.

Key Learnings

- Terraform resource targeting and cleanup are crucial when infrastructure needs a full reset.
- GitHub Actions workflows should be made branch-aware and release-aware.
- SSH debugging and EC2 access issues are most commonly caused by:
- Wrong key path
- Wrong user (`ec2-user` vs `ubuntu`)
- Changed IP after instance recreation
- Ansible can dynamically deploy WAR to Tomcat without installing a full systemd service.

Next Steps (Optional Enhancements)

- Parameterize environment-based releases using tags (e.g., `v1.0.X` for prod, `sit-` for SIT).
- Add rollback strategy via versioned WAR backups.
- Enable health checks and monitoring after deployment.
- Use GitHub Environments to isolate secrets and approval workflows.

Final Deliverables

- Fully automated CI/CD pipeline using GitHub Actions.
- Java app deployed to EC2 using Terraform + Ansible.
- WAR artifact scanned, tested, versioned, and deployed securely.

> ✅ This project is ready for real-world use and can be extended to staging and production environments with minimal changes.