Pet Clinic

Automated Deployment Pipeline

Deirdre Lee

devops T00126583

Contents

[Stage 1: Code and Tools 3](#_Toc162521305)

[Stage 1.1: Summary of Application 3](#_Toc162521306)

[Stage 1.2: Tool Chain 4](#_Toc162521307)

[Stage 1.3: Flow Diagram 5](#_Toc162521308)

[Stage 2: Continuous Integration 5](#_Toc162521309)

[Stage 2.1: Branching Strategy 5](#_Toc162521311)

[Stage 2.2: Install & Configure Continuous Integration 6](#_Toc162521312)

[Stage 2.3: Automate Build Process 6](#_Toc162521313)

[Stage 2.4: Configure Git Authentication 6](#_Toc162521314)

[Stage 2.4: Unit Test Execution 7](#_Toc162521315)

[Stage 2.5: Code Quality Tools 7](#_Toc162521316)

[Stage 3: Building the Code & Configuring the Pipeline 7](#_Toc162521317)

[Stage 3.1: Task Pipelines 7](#_Toc162521318)

[Stage 3.2: Deploy Application 7](#_Toc162521319)

[Stage 3.3: Build Pipeline for CI Lifecycle 7](#_Toc162521320)

[Stage 4: End-to-End Automation Of The Application Delivery Lifecycle 7](#_Toc162521321)

[Stage 4.1: Configuration Management Standardisation 7](#_Toc162521322)

[Stage 4.2: Docker Installation and Configuration 7](#_Toc162521323)

[Stage 4.3: Creation of CentOS Containers 7](#_Toc162521324)

[Stage 5: Cloud Provisioning and Configuration Management 7](#_Toc162521325)

[Stage 5.1: Resource Provisioning in a Cloud Environment 7](#_Toc162521326)

[Stage 5.2: Installing the Runtime Environment 8](#_Toc162521327)

[Stage 6: Deploying Application (AWS, Azure, and Docker) 8](#_Toc162521328)

[Stage 6.1: Environment Selection 8](#_Toc162521329)

[Stage 6.2: Configuration Management Preparation 8](#_Toc162521330)

[Stage 6.3a: Cloud Environment Deployment 8](#_Toc162521331)

[Stage 6.3b: Container-Based Deployment 8](#_Toc162521332)

[Stage 6.4: Deployment Configuration 8](#_Toc162521333)

[Stage 6.5: Deployment Execution 8](#_Toc162521334)

[Stage 6.5: Post-Deployment Testing 9](#_Toc162521335)

[Stage 6.6: Monitoring and Maintenance 9](#_Toc162521336)

[Stage 7: Monitoring Infrastructure and Applications 9](#_Toc162521337)

[Stage 8: Orchestrating Application Deployment 9](#_Toc162521338)

[Stage 8.1: Configure Build Jobs for Checkout and Execution: 9](#_Toc162521339)

[Stage 8.2: Implement Compilation and Unit Test Execution: 9](#_Toc162521340)

[Stage 8.3: Provision Runtime Environment: 9](#_Toc162521341)

[Stage 8.4: Configure Permissions in New Instances: 10](#_Toc162521342)

[Stage 8.5: Automate Deployment Process: 10](#_Toc162521343)

# Stage 1: Code and Tools

## Stage 1.1: Summary of Application

Pet clinic is an application developed using the Spring Boot framework, which can be built using either Maven or Gradle, with the help of included wrapper scripts. (spring-projects, n.d.) Its web interface consists of a homepage and a navigation bar with three options: the ‘owners/find’ endpoint, where you can add an owner using a form or search for one using the last name; A ‘/vets’ endpoint, listing all vets along with options to view the results as XML or JSON; and an ‘/oups’ endpoint demonstrating the page shown when an exception occurs.

Many components from the Spring stack were used in the construction of this project, including Spring MCV, which provides a model-view-controller architecture, and Spring Data JPA, facilitating interaction with the associated database. (Singh Raina & Giraldo, 2022). Thymeleaf serves as the template engine, used for server-side rendering. Additionally, the in-memory H2 database is used by default to store data during runtime. The Junit framework is used for the unit testing.

The application was sourced from the Spring Projects repository. (spring-projects, n.d.)

## Stage 1.2: Tool Chain

|  |  |
| --- | --- |
|  |  |
| Source Code Repositories | GitHub |
| Build Tools | Maven |
| Continuous Integration | GitHub Actions |
| Configuration Management to Set Runtime Environment | Ansible  Puppet  Chef |
| Resource Provisioning Tools  Cloud  Containers | AWS  Azure  Google Cloud Platform  Docker  Google Kubernetes Engine (GKE) |
| Continuous Delivery | GitHub Actions |
| Continuous Deployment | GitHub Actions |
| Continuous Monitoring | Prometheus  Grafana SonarCloud |
| Continuous Feedback | Slack  Email |
| Continuous Improvement | Kanban  Jira |

## Stage 1.3: Flow Diagram

## 

# Stage 2: Continuous Integration



## Stage 2.1: Branching Strategy

Before proceeding I decided to set up a branching strategy based on Git Flow. (Thummala, 2023) The main branch will serve as the production-ready codebase, where only stable and thoroughly tested code will be merged. Direct commits will not be permitted to maintain code integrity and stability.

The development branch, on the other hand, will act as the integration point for all project changes, including bug fixes and the addition of new features. A workflow will be implemented on this branch to automate the build and testing process, ensuring that changes are thoroughly validated before integration. A list of branches are shown in Fig 2.1.1.



Fig 2.1.1 Branches list

The introduction of these branches required the setup of a branch protection rule on GitHub to guarantee smooth integration and adherence to the established workflow. The rule entails that a pull request be approved before merging, as well as passing status checks.

## Stage 2.2: Install & Configure Continuous Integration

As I am using GitHub for version control, I decided to use GitHub Actions to implement continuous integration in the project.

GitHub Actions is a CI/CD platform for automating tasks and workflows in your GitHub repositories. It uses YAML files to define workflows triggered by events like pushes or pull requests. Actions are reusable applications, covering repetitive or complex tasks. (GitHub, 2024)

It involved creating a workflow file, which automates the testing and deployment of the project whenever changes are pushed to the repository. I first had to create a directory in the project called **.github/workflows** and within this directory created a yaml file called **maven-build.yml**, as shown in Fig 2.2.1. This file will contain all the steps to be performed as part of a build.



Fig 2.2.1 Setting up CI in GitHub Actions

## Stage 2.3: Automate Build Process

The **maven-build.yaml** file sets up a workflow for automating the build process of the **PetClinic** Java project using Maven. This starter workflow, along with many others, is provided on GitHub. It is triggered by pushes or pull requests to the main branch of the project. The job is called ‘build’ and performs basic build steps, including running the latest version of Ubuntu, checking out the code, and setting up the JDK with Java 17. It caches the Maven packages to speed up future builds using **actions/cache** to cache the **.m2/repository** directory, which is populated based on the **pom.xml** file. The final step runs the Maven wrapper with the package option. This compiles the source code, runs tests, and packages the application into a JAR file. Slight updates to the default file were made to suit the project's needs.

## Stage 2.4: Configure Git Authentication

As git authentication has been previously set up on my laptop with GitHub it was not necessary to configure this step.

## Stage 2.5: Configure GitHub Actions to send notifications

I installed the application GitDailies on GitHub. Through this service, I was able to set up daily alerts about the project status, as well as add two additional alerts which would send an email to my email regarding the success or failure of the GitHub Actions workflow, as can be seen in Fig 2.5.1.

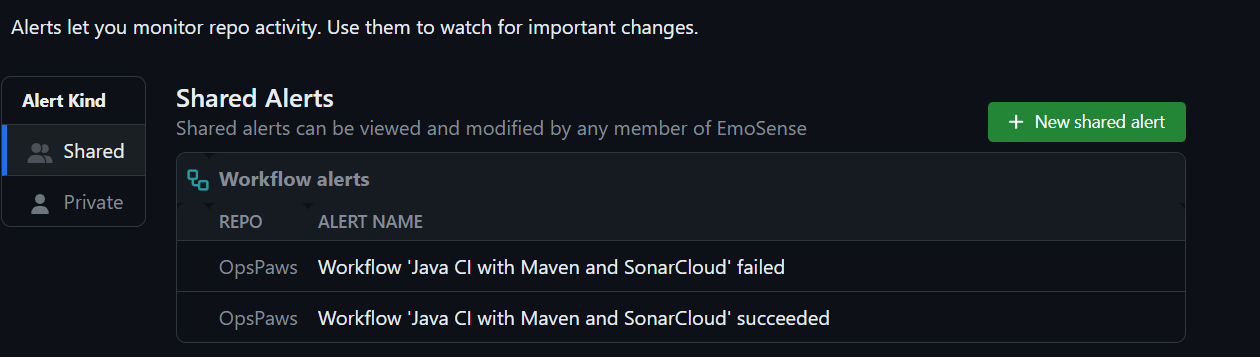


Fig 2.5.1 GitDailies Email Alerts

## Stage 2.6: Unit Test Execution

Using the Maven wrapper command **./mvn test** I ran the tests, of which 45 were run. Two tests were skipped and the rest passed as can be seen in Fig 2.6.1.



Fig 2.6.1 ‘mvn test’ result output

The next step was to add steps to the workflow file to add the running of the unit tests as a job, the result of which can be seen in Fig 2.6.2. A surefire report is also generated and will be accessible in the ‘Artifacts’ section of GitHub actions, as shown in Fig 2.6.3. This can be downloaded as a zip file and contains the test reports. (Casperson, 2023)

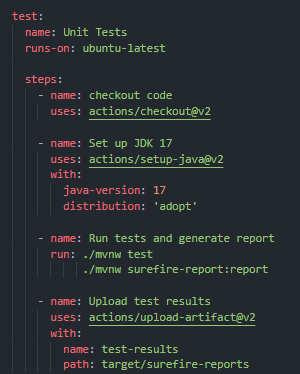


Fig 2.6.2 maven-build.yaml

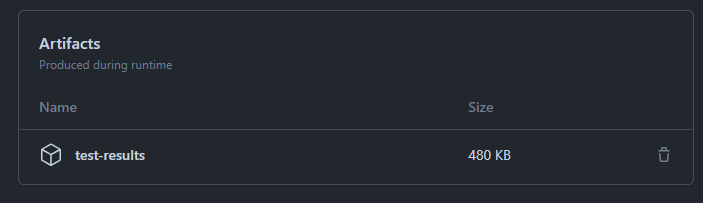
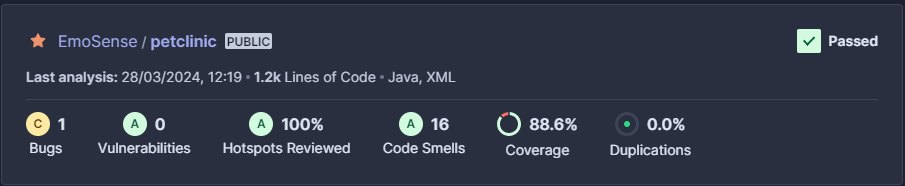


Fig 2.6.3 Artifacts, including test results

## Stage 2.7: Code Quality Tools



I decided to use SonarCloud for code quality analysis and reporting because I have some experience working with it and wanted to further my skills using it. I added the project to my SonarCloud account and updated various settings, including creating a new quality gate for the project. Then, I added a SonarCloud Analysis job to the workflow file, as shown in Figure 2.7.1. This task builds the project, analyses the source code and sends results to SonarCloud.

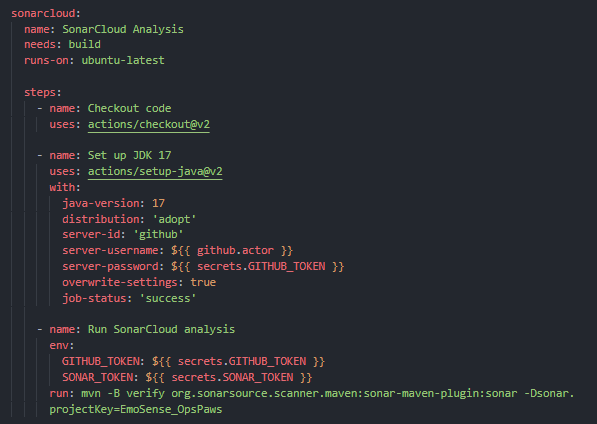


Fig 2.7.1 SonarCloud job

The link to the project in SonarCloud is as follows:

<https://sonarcloud.io/summary/new_code?id=EmoSense_OpsPaws>

# Stage 3: Building the Code & Configuring the Pipeline

## Stage 3.1: Task Pipelines

Up until now, stages of the Java pipeline have been defined, including the build stage and quality check, using GitHub actions. All that remains for this part of the project is to deploy the application to a test environment.

## Stage 3.2: Deploy Application

I decided to create a Docker image for the web application and then deploy this to Azure as it seemed more efficient and flexible. The first step was to create a Dockerfile with the application, configuring the Dockerfile to copy and run the application once the container starts. I then built the Docker image using the Docker CLI and the following command docker build -t pet-clinic .

Continue

Create azure container registry

Build and push docker image

Deploy to ACI

Monitor and manage

https://www.hanselman.com/blog/penny-pinching-in-the-cloud-deploying-containers-cheaply-to-azure

* Deploy the application to a suitable web or application server.
* Ensure appropriate configuration for deployment.

## Stage 3.3: Build Pipeline for CI Lifecycle

* Set up a build pipeline that encompasses all stages of continuous integration.
* Include processes such as compilation, testing, and packaging within the pipeline.

# Stage 4: End-to-End Automation Of The Application Delivery Lifecycle

## Stage 4.1: Configuration Management Standardisation

## Stage 4.2: Docker Installation and Configuration

## Stage 4.3: Creation of CentOS Containers

# Stage 5: Cloud Provisioning and Configuration Management

## Stage 5.1: Resource Provisioning in a Cloud Environment

* Identify the cloud environment for resource provisioning.
* Determine the resources needed to support the sample application.

## Stage 5.2: Installing the Runtime Environment

* Install the necessary components required to run the sample application.
* Ensure compatibility and functionality with the chosen cloud environment.
* Verify successful installation and runtime environment configuration.

# Stage 6: Deploying Application (AWS, Azure, and Docker)

## Stage 6.1: Environment Selection

* Determine the target environment for deployment, considering options like cloud services (AWS, Azure, Google Cloud), container platforms (Docker, Kubernetes), or on-premises servers.

## Stage 6.2: Configuration Management Preparation

* Ensure that the configuration management tool has completed all necessary preparations for deployment.
* Verify that the application's configurations, dependencies, and environment variables are correctly set up and packaged.

## Stage 6.3a: Cloud Environment Deployment

* If deploying to a cloud environment:
  + Access the chosen cloud platform's management console.
  + Create or select the appropriate instance types, virtual machines, or containers for deployment.
  + Configure networking, security groups, access control policies, and firewall rules as needed.

## Stage 6.3b: Container-Based Deployment

* If deploying using containers:
  + Set up a container orchestration platform like Docker Swarm or Kubernetes.
  + Build container images for the sample application.
  + Push container images to a container registry like Docker Hub or a private registry.

## Stage 6.4: Deployment Configuration

* Define deployment configurations such as environment variables, secret management, and application scaling policies.
* Configure load balancers, auto-scaling groups, or service discovery mechanisms based on deployment requirements.

## Stage 6.5: Deployment Execution

* Initiate the deployment process, either manually or through automated deployment pipelines.
* Monitor the deployment progress and handle any errors or issues that arise during deployment.

## Stage 6.5: Post-Deployment Testing

* Conduct thorough testing to ensure that the application functions correctly in the new environment.
* Perform integration tests, end-to-end tests, and performance tests to validate the deployment's integrity.

## Stage 6.6: Monitoring and Maintenance

* Set up monitoring and logging tools to track the application's performance and health in the new environment.
* Implement alerts and notifications for critical events and performance degradation.
* Establish a maintenance schedule for routine updates, patches, and infrastructure optimizations.

# Stage 7: Monitoring Infrastructure and Applications

* Detail the necessity of continuous monitoring within the end-to-end automation process.
* Emphasise its critical role in ensuring service availability and application functionality.
* Highlight the significance of monitoring various aspects including cloud resources, application servers, and application performance.
* Stress the importance of early issue detection and timely resolution to prevent service disruptions.
* Explain how monitoring aids in optimizing resource utilization, identifying performance bottlenecks, and enhancing user experience.
* Illustrate the overarching goal of continuous monitoring: to increase services and application availability by proactively managing and addressing potential issues.

# Stage 8: Orchestrating Application Deployment

## Stage 8.1: Configure Build Jobs for Checkout and Execution:

* Set up build jobs to initiate the checkout of source code repositories and execute the build pipeline.

## Stage 8.2: Implement Compilation and Unit Test Execution:

* Include compilation tasks within the build jobs to compile the source code effectively.
* Integrate unit test execution to validate the functionality and integrity of the application.

## Stage 8.3: Provision Runtime Environment:

* Incorporate tasks to install the runtime environment, such as deploying Linux on Amazon EC2 instances, to support application execution.

## Stage 8.4: Configure Permissions in New Instances:

* Define configuration steps to configure permissions within the newly created instances.
* Ensure secure access and operation of the runtime environment and application.

## Stage 8.5: Automate Deployment Process:

* Automate the deployment process within the build jobs to streamline application rollout.
* Enable seamless deployment of the application to the provisioned runtime environment.