DevOps Pipeline Design Document

HealthSmart Medication Management System

Date: 1/19/2025

Version: 1.1.1

Author: Joshua D’Agostino

Revision History			
Version	Author	Version Description	Date Completed
1.1.1	Joshua D'Agostino	Initial version of the document	1/19/2025

Table of Contents

1. Introduction

The DevOps Pipeline Documentation for the HealthSmart Medication Management System provides an overview of the automated pipeline that supports the development, testing, deployment, and CI/CD processes of the HealthSmart application. This document outlines the tools processes, and workflows that form the backbone of the DevOps pipeline, ensuring the efficient and reliable delivery of software. It was designed in mind to facilitate collaboration between developers, operators, and quality assurance testers, providing an automated process for continuous improvements to the system.

1.1 Purpose

The purpose of this document is to describe the architecture and flow of the DevOps pipeline for the HealthSmart Medication Management System. The pipeline aims to automate and accelerate the build, test, and deployment processes for the system, contributing to continuous improvement in software quality, timely feature release, and reduced time-to-market. This documentation will detail the steps, technologies, and

practices employed within the pipeline to maintain consistent and high-quality software delivery that satisfies the needs of the healthcare professionals and patients using the HealthSmart system through functionality.

1.2 Scope

This document covers the following aspects of the DevOps Pipeline, excluding non-automated processes, infrastructure provisioning outside of the pipeline, and manual intervention workflows.

- CI/CD Pipeline Overview: A high-level description of the pipeline stages, including the build, and deployment processes
- Tools and Technologies: The software tools used throughout the pipeline, including version control, automation tools, build and test frameworks, deployment strategies, and monitoring tools
- Environment setup: the configuration and management of development, testing, staging, and production environments
- Automation processes: how tasks such as code completion, unit testing, integration, security checks, and deployment are automated through GitHub workflows

1.3 Audience

This document is intended for the following audiences:

- Developers: To understand how their code is integrated into the pipeline, tested, and deployed to production
- Operators: To oversee the environment configurations, deployment strategies, and maintenance of infrastructure
- Quality Assurance: To review and ensure the quality and security of the pipeline through testing automation and continuous monitoring
- Infrastructure operators: To configure, monitor and manage the pipeline's operation and troubleshoot issues in the deployment pipeline

1.4 Intended Use

This document serves as a guide for setting up, maintaining, and troubleshooting the DevOps pipeline for the HealthSmart Medication Manage System. It is intended to help all relevant teams understand the flow of the DevOps pipeline and their responsibilities in maintaining its efficiency and reliability, providing insights for improvement and resolution of issues.

2. DevOps Pipeline Overview

The HealthSmart Medication Management System follows a traditional three-stage DevOps pipeline: Commit, Acceptance, and Production.

When a developer commits and pushes code to the application repository, the commit stage pulls the code and publishes a release candidate in the container registry. This ensures that the code is ready for testing in subsequent stages, as it has been containerized into a portable format that can be deployed across different environments. Once completed, it publishes an event – new release candidate available – to trigger the acceptance stage to pull the release candidate.

In the acceptance stage, the release candidate is tested to ensure both quality and alignment with the acceptance criteria. Automated test suites will be run against the commit to ensure the code meets both quality standards and acceptance criteria. These tests include unit testing, integration testing, security checking, and other validation required to ensure that the code is stable and compliant with the project's requirements. After the release candidate passes the acceptance stage, it moves to the production stage.

Once the production stage is triggered – acceptance stage passed – it will pull the containerized Docker image and wrap it in a high-level abstraction object for an orchestrator. This ensures that the release candidate takes advantage of cloud-native features, such as scalability, high availability, and self-healing, as it is deployed into the production environment. When finished, the production stage pushes the release candidate to the deployment repository, where it can be tracked and monitored as part of the live environment

3. Tools, Frameworks, and Technologies

Tool	Description
Paketo Buildpacks Builder	Allows containerization of Spring Boot applications without the need for a Dockerfile, extra dependencies, or manual configuration
Grype	A vulnerability scanner for container images that ensures the images are secure and free of known vulnerabilities
Docker Compose	Used to define and run multi-container Docker applications. It can help simulate the application's runtime environment for local development and testing.
GitHub Actions	A CI/CD tool that automates workflows for building, testing, and deploying code in the pipeline.
GitHub Agent	A component that assists in managing the interaction between GitHub and other pipeline tools, triggering subsequent stages in the process

Spring Boot Gradle	A build tool that compiles and builds Spring Boot applications, integrating with the pipeline for smooth packaging and deployment
Junit 5	A testing framework for Java applications that facilitates writing and executing unit tests to validate functionality
Mockito	A mocking framework used alongside Junit for creating mock objects in unit tests
Testcontainers	A tool that provides lightweight, disposable containers for integration testing. It helps simulate environments like databases or external services during tests.
GitHub Repository	A service provided by GitHub for storing and managing Docker container images and other container artifacts

4. Environment Setup Strategies

4.1. Overview

The developers at HealthSmart Medication Management System understand that a release candidate that works in one environment may not necessarily be suitable for another. For instance, a release candidate tested in the development environment might be optimized for speed and quick feedback but may not meet the standards required in the production environment. To mitigate the risk of deploying an unstable or inadequate release to production, the system will profile release candidates for each environment – Development, Testing, Staging, and Production. This ensures that each environment receives a tailored version of the candidate, which is configured and tested to meet the requirements of that environment.

Profiling involves configuring environment specific settings, using mock services, and adjusting other parameters that may vary depending on the environment. Each release candidate will be tested under conditions that closely simulate what it will encounter in its respective environment. To streamline the configuration process and avoid confusion, profiles will be defined and managed directly in the application code using Spring Boot profiles.

By specifying the active profile (e.g., spring.profiles.active=dev or spring.profiles.active=test), Spring Boot will automatically load the appropriate configuration for the environment, ensuring that the release candidate has the right settings for the environment in which it is deployed. This approach eliminates the need for hardcoding configuration values, providing a flexible and dynamic way to handle environment-specific needs.

From the outset, the developers at HealthSmart leverage best practices, such as:

1. Avoiding hardcoded configuration values, which allows configurations to be easily adjusted for different environments
2. Utilizing testcontainers to replicate the production environment more accurately in both the development and testing stages. This helps ensure consistency in how the application behaves across environments
3. Running comprehensive test suites that include unit, integration, and end-to-end tests tailed for each environment

4.2 Environment-Specific Profiling Process

In the Development environment, the focus is on speed and iteration. Developers can leverage simplified configurations and mock services to quickly validate code changes.

In the Testing environment, the application undergoes automated validation through extensive test suites that stimulate the production conditions, ensuring unit and integration tests are properly executed.

Staging candidates closely mirror the Production environment by using real data, comprehensive integration testing, load testing, and security validation.

Finally, Production candidates are deployed with configurations designed for high availability, including advanced infrastructure settings through ConfigSecrets (for sensitive data management) and robust security measures.

4.3 Benefits of Profiling

By applying these tailored configurations, profiles, and validation processes for each environment, HealthSmart ensures that every release candidate is properly tested and validated before moving to the next stage in the pipeline. Spring Boot's ability to manage profiles within the code allows for flexible, streamlined configurations that can be easily adjusted for different environments. This approach minimizes risk and increases confidence that the release candidate will perform as expected in production.

5. Pipeline Architecture and Workflow

The developers at HealthSmart Medication Management System recognize the value of a deployment pipeline that automates the containerization, testing, and deployment of release candidates. This approach ensures that extensive testing and quality control are integrated throughout the development lifecycle. Before implementing the pipeline, it is essential to define the jobs that comprise its workflow. Each stage of the pipeline consists of several jobs, each with a distinct purpose. By designing the pipeline to encompass all components of the system, it can be reasonably ensured that the deployed release candidates will meet user requirements, satisfy non-functional requirements, leverage cloud-native features, and maintain high standards of quality, maintainability, and readability.

5.1 Commit Stage

The HealthSmart Medication Management System is designed to scale and leverage cloud-native features. To achieve this, services and their components will be built and containerized into Docker images. The build process is optimized for efficiency using Spring Boot's layered-JAR feature, which enables caching and reuse of unchanged layers during the image build.

Instead of using traditional Dockerfiles for containerization, Cloud Native Buildpacks will be used to build the Docker image. Given that the system is built with Spring Boot, the Paketo Buildpacks Builder is a natural choice, as it incorporates the Java Runtime Environment (JRE) and utilizes the layered JAR built by Spring Boot.

To manage the vast number of containers required by the Medication Management System, Docker Compose will be used in the local environment. This tool simplifies container management by centralizing control. Eventually, Kubernetes will be employed or orchestration at scale in the deployment environment, but this is outside the scope of this section.

The build process will follow a continuous integration (CI) pipeline, starting with the static code analysis, compilation, unit tests, and integration tests. Once these steps are successfully completed, the application will be packaged into an executable artifact and published. The process ends with the production of a container image that can be reused in subsequent stages of the deployment pipeline, up to and including production.

If at any point in the pipeline (from commit to release candidate) a test fails, the release candidate is rejected. This ensures that issues are caught early, allowing for rapid feedback and quicker refactoring of the code.

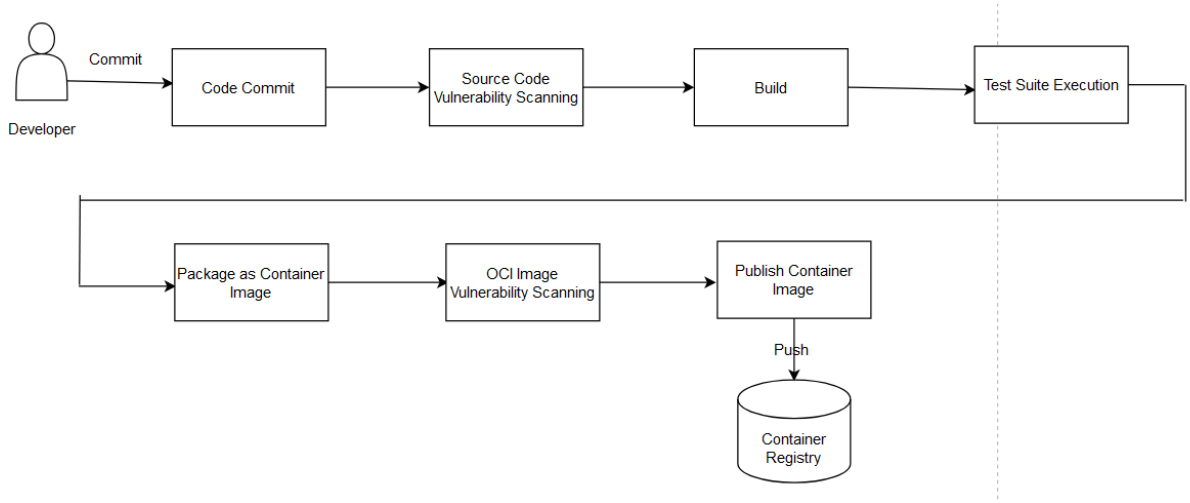

Figure 5-1 Commit Stage for Medication Management System

Below are the steps involved in the commit stage:

1. Code Commit: Code is pushed to the repository
2. Source Code Vulnerability Scanning: Automated security checks are run to identify potential vulnerabilities in the codebase
3. Build: The application is compiled and prepared for containerization
4. Test Suite Execution: The test suite runs both unit and integration tests to ensure code quality and functionality
5. Package as Container Image: The application is containerized as a Docker image

6. OCI Image Vulnerability Scanning: The container image undergoes a security scan to check for vulnerabilities
7. Publish Container Image: The final container image is pushed to the container registry for use in further stages of deployment

Once the container image is successfully built and pushed to the container registry, it triggers the Acceptance Stage. This stage is responsible for ensuring the image meets all necessary requirements before it can proceed to the Production stage. The details of the Acceptance Stage and its associated tests are discussed in the following section.

5.2 Acceptance Stage

The Acceptance Stage of the pipeline places a strong emphasis on API end-to-end testing, given the involvement of multiple services in their interaction. For example, testing a secured API endpoint requires successful integration across various components such as the Edge Service, Redis, Keycloak, the Angular SPA application, and the relevant business service. In addition to API functionality, the stage also addresses cross-cutting concerns such as observability and logging aggregation. These aspects play a critical role in satisfying the non-functional requirements defined by stakeholders, ensuring that the system is not only functional, but also robust and well-suited to meet operational needs.

5.2.1 Test Quality Assurance

To maintain the quality of tests in the pipeline, the test suite itself will undergo rigorous quality checks before it proceeds to the acceptance stage. This process ensures that all commits are evaluated against a set of reliable, well-written tests, thereby preventing faulty or incomplete tests from impacting the acceptance process. By validating the test suite, it ensures that only high-quality test code is used to verify the system's functionality. Here are the steps involved in this:

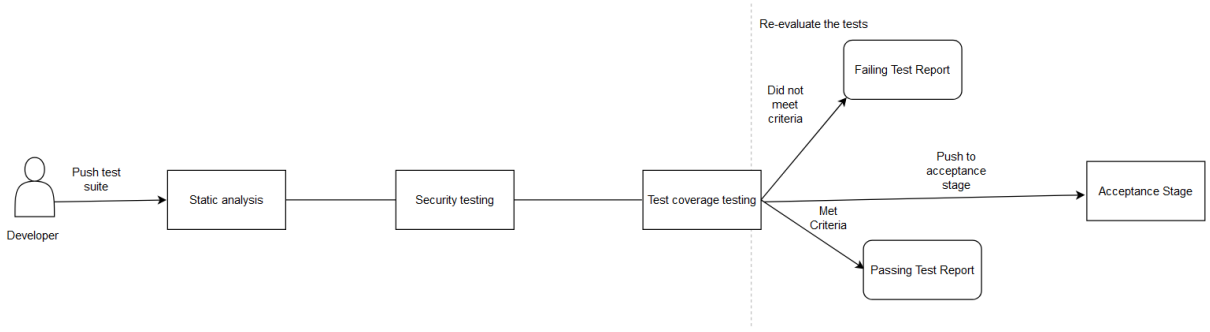

Figure 5.1 Test quality control process

1. Push Test Suite

The developer pushes a new test suite or updates an existing one.

2. Static Analysis

The test suite is analyzed to ensure that it is clean, follows coding standards, and is free from errors.

3. Security Testing

The test suite is checked for security vulnerabilities, such as flaws like hardcoded credentials, insecure testing practices, or potential exploits.

4. Test Coverage

The test suite is evaluated to ensure the code is sufficiently covered. If the test suite doesn't cover all the relevant parts of the system, this would indicate a quality issue.

5.a. Trigger Acceptance Stage

If all checks pass for the test suite, the test suite will be passed to the acceptance stage for testing a system's component or operation

5.b. Trigger Failure Stage

If the test suite fails to meet the required standard, the job fails and alerts the developer to fix the issues.

5.2.2. API Endpoint Testing

The API layer serves as the contract for the system's interfaces, exposing business use cases through REST endpoints. API endpoints are typically invoked by a user interface (UI) interacting with the presentation layer. This allows users to perform actions such as viewing health information, reporting a medication issue, all of which trigger the business logic in the service layer. These endpoints directly map to methods in the service layer, which encapsulates the business logic. The service layer, in turn, interacts with the data persistence layer to manage domain entities and data storage.

Testing the API ensures that the endpoints function as expected, communicate properly with the service layer, and enforce necessary security mechanisms (e.g., authentication, authorization)

5.2.2.1 Data Persistence Layer Testing

One of the major advantages of using Spring Boot is its ability to run integration tests by loading only the specific Spring components that are required for each slice of the application. In the context of this situation, tests will be written for the data slice, which focuses on validating the functionality of the data persistence layer.

Since the system uses Spring Data JDBC for data persistence and PostgreSQL as the data store, integration tests for the data persistence layer will utilize Spring Data JDBC to test interactions between the application and the database. For example, a test class such as MedicationManagementJdbcTests will be responsible for verifying the correct mapping and data operations between the Medication Management service's repository and data store.

To simulate a production-like environment during testing, Testcontainers will be used to manage the PostgreSQL container. This approach allows the tests to run against an isolated, containerized PostgreSQL instance, ensuring that the tests reflect a real-world production environment.

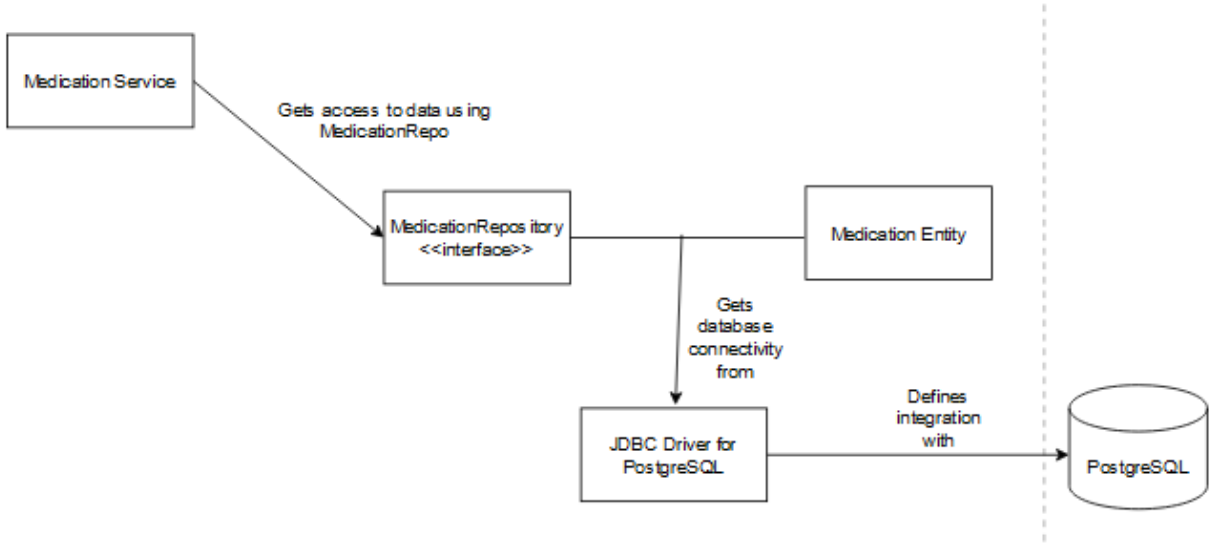

Figure 5-2 Data Persistence Layer of Medication Service

As illustrated above, the Medication Service interacts with the MedicationRepository CRUD interface to retrieve data from the data store. The MedicationRepository establishes database connectivity through the JDBC driver for PostgreSQL, which facilitates integration with PostgreSQL. To ensure these components work together seamlessly, an integration testing approach would involve spinning up containers for both the Medication Service and the PostgreSQL data store. This setup allows for testing the following critical points:

1. Medication Service requests to the Medication Repository
2. Medication Repository database connectivity via the JDBC Driver for PostgreSQL
3. JDBC Driver for PostgreSQL integration with the PostgreSQL

For the persistence layer, integration testing would leverage the @ActiveProfiles(“integration”) annotation in the appropriate application test class, such as MedicationServiceApplicationTests.

Furthermore, to prevent the system from frequently opening and closing database connections, connection pooling will be implemented. This approach involves the application establishing connections with the database and reusing them, optimizing performance. Part of the testing for an API call will include verifying that connection pooling is functioning correctly. Specifically, the tests will ensure that connections are properly used, and the connection pool is efficiently managed.

Additionally, by utilizing the @DataJdbcTest annotation, each test method will run within a transaction, which will be rolled back at the end of the test ensuring that the database remains clean for subsequent tests.

5.2.2.2. Service/Controller Layer Testing

In the previous section, the Medication Service was introduced. A service defines the use cases for the domain and invokes the repository to access data for each use case. These use cases are exposed through a Rest API. Since this service is imperative, it will use the annotation @RestController provided by Spring MVC to handle incoming HTTP requests for specific resource endpoints. Each use case also requires a method handler to make the use cases available to clients. Therefore, a controller class is needed to handle the respective HTTP requests to an endpoint.

However, this setup alone is not sufficient. It is essential to prevent faults from entering the application, especially in the domain layer. Using Java Bean Validation, constraints and validation rules will be applied to the fields of the domain entity. These rules are enforced when validating the domain object in the object's controller class by using the @Valid annotation whenever a @RequestBody is specified as a method argument. The validation logic will be covered by unit tests to ensure correctness.

Furthermore, errors in a REST API are inevitable, and they will be handled using standard Java exceptions. A @RestControllerAdvice class will be used to define how exceptions are handled when thrown for an endpoint.

To test the API exposed by the respective service, WebTestClient can be used to ensure that the integration between the components is functioning correctly. Moreover, the integration tests can be limited to the service layer slice, excluding the data persistence layer, which saves valuable time during testing. Spring Boot offers the @WebMvcTest annotation for this, which loads a Spring application context in a mock web environment, configures the Spring MVC infrastructure, and includes only beans used by the MVC layer, like @RestController and @RestControllerAdvice. The controller class will be provided as an argument to the @WebMvcTest annotation in a DomainObjectControllerMvcTests class to limit the scope of the context to the beans used by the controller under test.

The domain objects returned by the methods in a DomainObjectController are, by default, parsed into JSON objects. Using the @JsonTest annotation, the JSON serialization and deserialization for domain objects can be tested. @JsonTest annotation loads a Spring

application context and auto-configures the JSON mappers for the specific library in use (by default, Jackson).

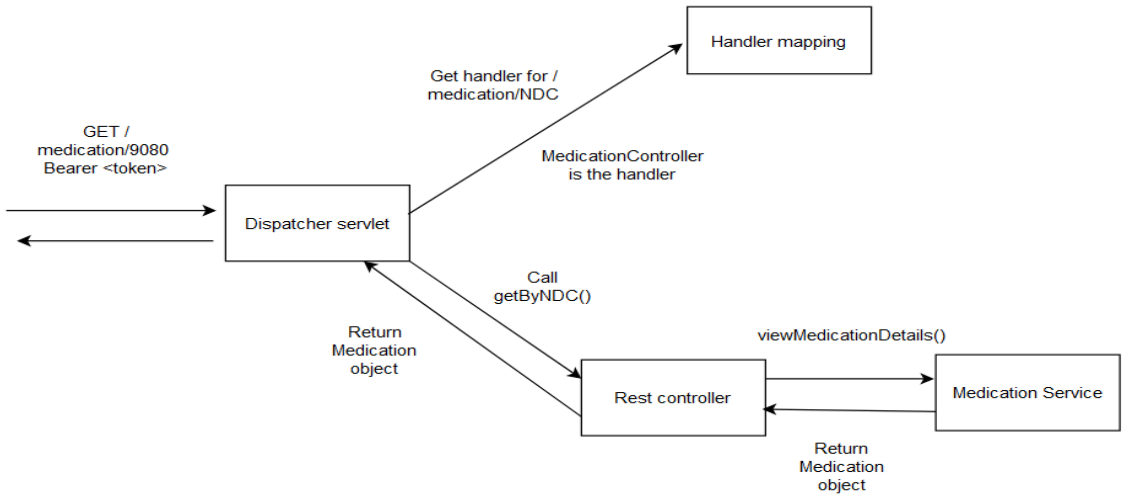

Figure 5-3 MVC Interaction for “GET /medication/NDC” endpoint of Medication Service

As shown in the interaction, an operation is initiated to obtain medication with the ID 9080, including the Bearer token for authorization purposes. The Dispatcher servlet retrieves the method handler for /medication/{NDC} from the handler mapping. This handler is mapped to the MedicationController, so the handler mapping informs the dispatcher servlet. The Dispatcher servlet then sends the getByNdc() call to the REST controller, which invokes the viewMedicationDetails() use case in the Medication Service. After querying the data store, the Medication Service returns a Medication object to the REST controller. The controller

subsequently returns this object to the Dispatcher servlet, which sends it as an HTTP response with a JSON body.

In the DevOps pipeline, a comprehensive testing strategy ensures the application's functionality, including the service and controller layers, works as expected. This testing strategy is integrated into the pipeline to verify code quality and reliability through automated tests. Below is an outline of the testing approach used, with the previously mentioned scenario in mind.

1. Unit testing the Service Layer (Medication Service)

The Medication Service defines the business logic and use cases for the domain. To ensure this layer works as expected:

- Unit tests are written using Spring Boot Test's Mockito to mock dependencies and to isolate service logic
- Validation tests ensure that constraints and validation rules are correctly enforced on the Medication domain entity using Java Bean Validation annotations. The @Valid annotation is used in the controller layer to trigger these validations during method invocation
- Unit tests verify that the service layer properly interacts with the repository and returns the expected domain objects

2. Integration testing the Controller Layer

The controller layer handles incoming HTTP requests and delegates the logic to the service layer. To test the interaction between the controller and service:

- WebTestClient is used to simulate HTTP requests to the controller endpoints and validate the responses
- WebMvcTest is employed to isolate the controller layer for fast integration tests. It loads only the necessary Spring MVC beans and skips loading the entire application context. This assists in limiting the scope and speeding up the tests.
- Integration tests verify that the controller processes requests correctly, interacts with the service layer, and returns the expected HTTP responses
- The tests also validate that input data is processed according to the validation rules, rejecting invalid input and returning error responses appropriately

3. Error Handling with @RestControllerAdvice

Errors in the REST API are handled using @RestControllerAdvice. To ensure robust error handling:

- Unit tests simulate error scenarios (e.g., invalid input or exceptions) and verify that @RestControllerAdvice catches these exceptions and returns appropriate HTTP status codes and error messages
- Tests ensure that specific exceptions trigger the correct responses, such as a 400 Bad Request or 500 Internal Server Error

4. Testing JSON Serialization and Deserialization

The @JsonTest annotation is used to test the serialization and deserialization of domain objects into JSON format. This ensures that data returned from the API is correctly structured for frontend consumption.

- @JsonTest is employed to verify that domain objects are serialized into JSON as expected and that the JSON structure matches the frontend's requirements
- Tests ensure that customize serialization logic (e.g., date formatting) is applied correctly

5. Controller Unit Tests with @WebMvcTest

In addition to the integration tests, unit tests are written for the controller logic:

- These tests check whether the controller methods invoke the correct service methods and return the expected HTTP responses
- They also verify that when invalid data is provided, the controller triggers validation errors, returning appropriate error responses.

5.2.2.3 Security Context Testing

Building on top of the service/controller layer for services inside the application, the authentication flow is supported by the OIDC protocol in the system. When a user calls an endpoint secured by the Edge Service, the Edge Service (OAuth2 client) redirects the browser to Keycloak for authentication. Keycloak authenticates the user and then redirects the browser back to the Edge Service, along with an authorization code. The Edge Service tells Keycloak to exchange the authorization code for an ID token, which contains information about the authenticated user. The Edge Service initializes an authenticated user session with the browser, based on a session cookie. After the user is authenticated, Keycloak sends a JWT to the Edge Service containing information about the newly authenticated user. The Edge Service validates the JWT's signature and inspects it to retrieve claims about the user. Finally, it establishes an authenticated session with the user's browser, using a session cookie whose identifier is mapped to the JWT. The Edge Service is registered as an OAuth2 client in Keycloak.

Now, the user, through the browser client, can invoke a use case by sending a request to the Edge Service, which is the OAuth2 client. In turn, the Edge Service routes the request to the appropriate service to retrieve the specific data stored for that user, as determined by the access token. After validating the token signature, the service searches for the user's data in the data store for that operation. Once the data is retrieved, it is returned to the user, and the Edge Service relays the response back to the user.

As for the endpoints, Role-Based Access Control (RBAC) will be applied using Spring Security. Access to each operation will require a specific role to initiate it, and the

operation will only produce data for a user with a specific ID. Below is the testing arrangement for this layer:

1. Authentication Tests

- Test the OAuth2 authentication flow works properly
- Verify that the Edge Service successfully redirects to Keycloak and correctly exchanges the authorization code for an ID token
- Validate that JWT tokens are properly signed and that the token includes the correct claims

2. RBAC Access Control Tests

- Verify that users with different roles (e.g., PATIENT, PHYSICIAN) have access only to their data and role-based actions
- Test unauthorized access attempts, ensuring that users without the correct role receive a 403 Forbidden or 401 Unauthorized responses

3. Data Auditing Tests

- Simulate data modifications and verify that audit logs are generated accordingly, ensuring proper capture of user actions.
- Verify that audit logs are secured and include necessary details such as the action performed, the user involved, timestamp, and changes made

5.2.2.4 Presentation Layer Testing

The HealthSmart Medication Management System utilizes a Single Page Application (SPA) through Angular for browser clients to interact with business logic. This architecture is important as it abstracts the business logic from the UI, ensuring a better user experience. Authenticated requests from Angular will be routed to the Edge Service, which then forwards these requests to the appropriate service.

Testing is essential to ensure that the API operations invoked from Angular are successfully reaching the Edge Service and are being correctly routed to the appropriate backend service. This will involve verifying that:

1. Requests from Angular clients are correctly routed to the Edge Service
2. Edge Service routing functionality is working as expected to forward requests to the correct downstream services
3. The authentication flow is properly handled, ensuring that only authenticated users can make API requests
4. Data specific to the authenticated user is loaded and returned in the response, ensuring that users only have access to their own data.

5.2.2.5 Overview of API Testing

While this document may be revised in the future to account for changes, this is the general testing flow for APIs of both imperative and reactive services in the system. Reactive

services may have minor variations, but the main concepts are applicable. This section provides an overview of testing done from the presentation layer through the data persistence layer. It represents an end-to-end test for an API. It was designed to be reusable, with minor variations needed to Mockito and Testcontainer objects, so that the API testing process can be streamlined. By testing the following key areas, we can ensure that:

1. API request from Angular reach the Edge Service and are routed to the correct backend services
2. The authentication flow and RBAC ensure that only authorized users can access their own data
3. The service layer processes business logic correctly and interacts with the database to fetch the necessary data
4. Error handling mechanisms respond to invalid inputs and unexpected failures with appropriate HTTP responses
5. Audit logs capture necessary details for tracking user actions and ensuring data security

Fitting in the Pipeline

End-to-end testing of an API endpoint involves multiple layers of the application, and it can be complex. To streamline and automate this process, the testing is integrated into the

Acceptance Stage of the pipeline. This stage verifies that all components of the application interact as expected before the artifact is promoted to the deployment stage.

Testing Flow in the Acceptance Stage

The testing process follows a sequential order, validating each layer of the application to ensure that the complete flow from the presentation layer to the data persistence layer is functional:

1. Presentation Layer: Both the front-end and API gateway is tested to ensure that user requests are correctly processed. This includes validating that the API exposes the right endpoints, and the data is correctly serialized/deserialized using JSON.
2. Security Context: The security and authorization mechanisms are tested, ensuring that the correct permissions and authentication tokens are handled. This verifies that requests are properly authorized and that RBAC is enforced.
3. Controller Layer: The controller layer is tested using integration tests to ensure that incoming HTTP requests are processed correctly. It checks that the controller methods invoke the corresponding service methods and return the correct HTTP responses.
4. Service Layer: The business logic within the service layer is tested, validating that it interacts correctly with the repository and processes data as expected. This layer also handles domain-specific rules, and the service methods must return the correct objects based on input data.

5. Data Persistence Layer: Finally, the data persistence layer is tested to ensure that data is being correctly saved, retrieved, and modified from a database. Integration tests are written to simulate interactions with the data store, ensuring that CRUD operations are correctly executed.

Each of these layers has a corresponding test suite slice, which targets specific components and interactions within the layer. The test suites ensure that each individual part of the system is functioning correctly and that integration points between components are validated.

Workflow in GitHub Actions

In GitHub Actions, each of the above test suites is represented as a separate job. These jobs collectively define the Acceptance Stage of the API building process. The jobs are executed sequentially to verify that each layer works properly before the artifact moves to the deployment stage.

- Triggering Tests: A new artifact produced during the Commit stage triggers the Acceptance Stage. This artifact is the target object for the test suites. The artifact must pass all tests in the Acceptance Stage before it is allowed to proceed to deployment.
- Test Suites Execution: Each job in the pipeline runs its respective test suite, verifying the functionality of the respective application layer. The jobs are configured to execute in the correct order to ensure a short feedback loop.

- Passing Tests: If all test suites pass successfully, the artifact is considered ready for deployment. If any test fails, the pipeline halts, and the issue must be addressed before proceeding.

Here is an example workflow for the Acceptance Stage when an artifact of the Medication Service passes the Commit Stage:

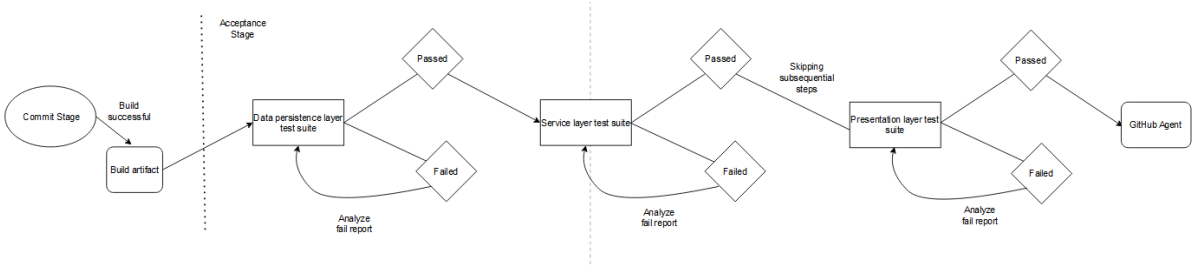

Figure 5-4 Workflow for API Acceptance Testing in Medication Management System

5.2.3. Services dealing with cross-cutting concerns

Although imperative programming has been the focus so far, the Spring reactive stack is the foundation for implementing an API gateway for the HealthSmart Medication Management System. It is used as not only an entry point to the services, but also to handle cross-cutting concerns, such as security, monitoring, and resilience. It's primary purpose is to decouple clients from changes to the internal services' API.

5.3. Production Stage

Once a deliverable has met the quality criteria in both the Commit and Acceptance stages, it will move to the Production stage. The release candidate is then pushed to the Deployment Repository. Upon arrival, this will trigger the GitOps agent to reconcile the actual state with the desired state in the Kubernetes cluster for the service. This process will be fully automated using GitHub Actions.

5.3.1. Configuration

Instead of using environment variables to pass hardcoded configuration to containers running in Kubernetes, the developers have decided to mount a ConfigMap in each service container. This approach was chosen because ConfigMaps provide a structured, maintainable way to store configuration data, making it easier to version and manage configurations over time.

For sensitive data, Kubernetes provides a Secret object to securely store and manage information such as passwords, certificates, tokens, and keys. Like the approach of mounting ConfigMaps in service containers, Pods can consume secrets as configuration files mounted in volumes. To ensure the secrets remain encrypted, the Sealed Secrets project, introduced by Bitnami, will be used. This solution allows secrets to be version-controlled without the risk of exposing sensitive data.

In the next section, configuration management will be discussed. Having a dynamic configuration management system in place is crucial, as the values in a ConfigMap are likely to change depending on the environment and the version of the release candidate.

5.3.2. Configuration Management

The developers at HealthSmart have decided to use Kustomize for configuration management. Kustomize is a declarative tool that helps configure deployments across different environments using a layered approach. This is particularly beneficial for the Medication Management system, given the number of services and environments involved. Rather than deploying applications by applying multiple Kubernetes manifests, Kustomize allows related manifests to be composed together using a Kustomize resource

To manage configurations across multiple environments, Kustomize uses the concepts of bases and overlays. In the context of the Medication Management system, the k8s folder of each service serves as the base. This directory contains a kustomization.yml file that combines Kubernetes manifests with customizations. An overlay is another directory, also containing a kustomization.yml file, that starts from the same base. Each environment can have its own overlay – for instance, a testing overlay and a staging overlay – that customizes configurations based on the shared base Kustomization file

Small, focused patches will be used to customize the configurations of containers as needed.

5.3.3. Automating the production stage

The production stage consists of two key steps:

1. Updating the deployment scripts with the new release version
2. Deploy the application to the production environment

To update the production Kubernetes manifests with the new release version, the source code will be checked out. Next, the production Kustomization file will be updated to reflect the new version for the given application. Finally, the changes will be committed to the service's deployment repository. Whenever there's a change in the application's manifests, Argo CD, a GitOps software agent, will automatically apply the changes in the production Kubernetes cluster.

5.4. Overview

The developers at HealthSmart have carefully considered the processes involved, given the complexity of the system and the quality requirements in place. A continuous deployment pipeline has the potential to streamline processes for each service, shortening the development lifecycle and improving code quality. To summarize the stages, consider the example of the Medication Management Service:

1. Commit Stage: A container image is created and published to the GitHub Container Registry
2. Acceptance Stage: Test suite slices are executed, depending on the component being tested. Afterward, a notification (a custom app_delivery event) is sent to the medication-management-deployment repository.

3. Production Stage: The event triggers an update to the production Kubernetes manifests for the Medication Management Service, and the changes are committed to the medication-management-deployment repository

6. Deliverables

6.1 Deliverables Overview

The developers at HealthSmart Medication Management System understand the value of continuous delivery for release candidates. This approach ensures proper version control and facilitates easy rollback in the event of a failure in a current release. Each stage in the pipeline produces a tangible deliverable, which serves as a key milestone in the overall process. In the next section, deliverables will be defined for each stage.

6.2 Deliverables by Stage

The pipeline consists of three stages: Commit, Acceptance, and Production. There will be a deliverable in the form of a release candidate from every stage. Each stage contributes a refined version of the candidate, ensuring that it meets the necessary quality and integration standards before advancing to the next phase.

6.2.1 Commit Stage Candidate

The first candidate, produced in the commit stage, is the result of a successful code commit that has passed static analysis, unit tests, and initial integration tests. It is then packaged into a container image for further testing.

6.2.2 Acceptance Stage Candidate

The candidate from the commit stage enters the acceptance stage, where it undergoes rigorous end-to-end testing to ensure all components function correctly together, and that each isolated component works properly. If the candidate successfully passes these tests, it is considered quality-verified and ready for production.

6.2.3 Production Stage Candidate

Finally, the production stage delivers the final, deployment-ready candidate. This candidate is wrapped in a deployment object and is prepared for release to the production environment, ensuring that it meets all necessary operational and performance requirements.

Each successful pipeline run results in a progressively validated release candidate, ready for deployment into production, with a clear rollback path if any issues arise at any stage.

6.3. Deliverable Management

To efficiently manage the different deliverables from each stage of the deployment pipeline, it is beneficial to create separate repositories dedicated to handling the output of

each stage. These repositories – application, container, and deployment – serve distinct purposes throughout the pipeline. Below is an overview of each repository's role:

1. Application repository – This repository contains the source code for individual services or components within the HealthSmart Medication Management System. It is where developers push their commits, and it serves as the primary location for version-controlled code. The application repository is the starting point of the pipeline.

2. A container repository – Once the commit has been containerized successfully, the resulting image is stored in the container repository. This marks the first deliverable of the pipeline. The container repository serves as the trigger for the acceptance stage, where the release candidate is further validated to ensure it meets quality standards through integration testing and other checks.

3. Deployment repository – After successfully passing the acceptance stage, the validated release candidate is placed in the deployment repository. At this stage, the release candidate is considered production-ready and has met all quality standards. The deployment repository acts as the staging area for the production deployment. The GitOps Agent continuously monitors this repository for changes and reconciles the actual state with the desired state in the Kubernetes cluster, ensuring that the service is deployed correctly in the production environment.

By utilizing these repositories, the HealthSmart Medication Management System ensures that each stage of the pipeline has a well-defined deliverable that can be tracked, validated, and deployed consistently.