Innovative. Open. Agile.

TechFlow, Inc. Response to
Request for Quotation (RFQ)
4QTFHS150004
Agile Delivery Services (ADS I)

Technical Architecture

July 7th, 2015

TABLE OF CONTENTS

1.0 Overview 1
2.0 Hardware Architecture 1
3.0 Software Architecture 1
3.1 External 1
3.2 Infrastructure 2
3.3 Application 2
3.4 Entities 3
3.5 Request Lifecycle 3
4.0 REST Services 4
5.0 Testing 4
5.1 Test Driven Development 4
5.2 Test Tools 5
5.3 Test Coverage 5
5.4 Testability 6
5.5 Infrastructure Testing 7
6.0 Open Source Technology Stack 8
7.0 Continuous Integration and Deployment 8
8.0 Configuration Management 9
9.0 Documentation 9

LIST OF FIGURES

Figure 1: Hardware Architecture 1
Figure 2: Software Architecture 2
Figure 3: Request Lifecycle 3
Figure 4: Test Driven Development Process 5
Figure 5: Jenkins Test Coverage Report 6
Figure 6: SonarQube Test Coverage Report 6
Figure 7: Test Architecture 7
Figure 8: Test Process 7
Figure 9: Technology Stack 8
Figure 10: Automated Deployment 9

1.0 Overview

The purpose of this document is to provide the general architectural and design approach to the project.

2.0 Hardware Architecture

The application was deployed on an IaaS infrastructure using Amazon Web Services (AWS) as a provider. As you can see in Figure 1, we used two virtual private cloud's (VPCs). The application VPC hosts the Docker container of the prototype application. The management VPC contains the continuous integration and deployment system.

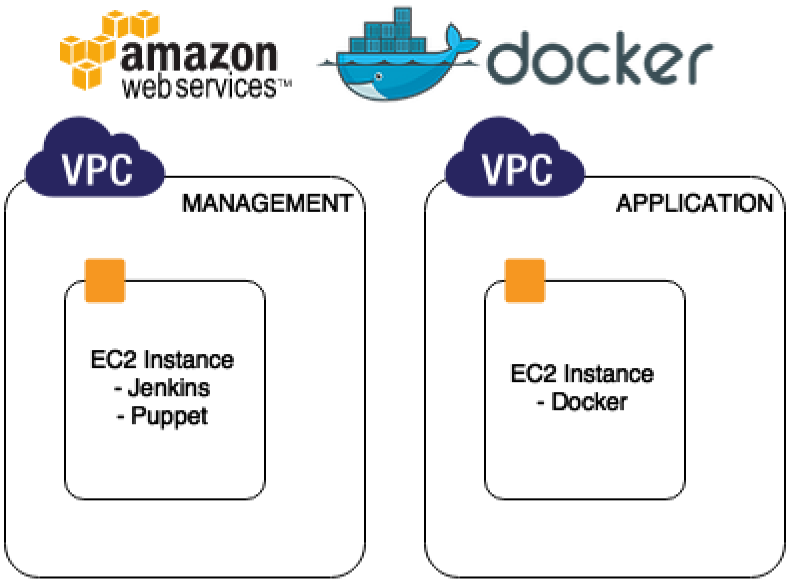


Figure 1: Hardware Architecture

3.0 Software Architecture

Flexibility and testability are key features of an agile software architecture. It should be easy to add new features, refactor, and test the software.

Figure 2 below show how the software is divided into four layers: external, infrastructure, application, and entities. The application API sits between the infrastructure and application layers and provides a public contract for the layers to communicate. To achieve flexibility and have low coupling it is important that the infrastructure and application be isolated and that both should be designed such that neither will know about the internals of the other.

3.1 External

The External Layer covers things like the database or cloud services that are not directly a part of the application.

3.2 Infrastructure

Application frameworks reside in the infrastructure layer. For example, we use Spring MVC to provide RESTful web services. This layer is kept as simple as possible. We want it to be possible to easily swap out the infrastructure layer. If we switch databases it should only be required to add a new database repository. The application layer should remain unchanged.

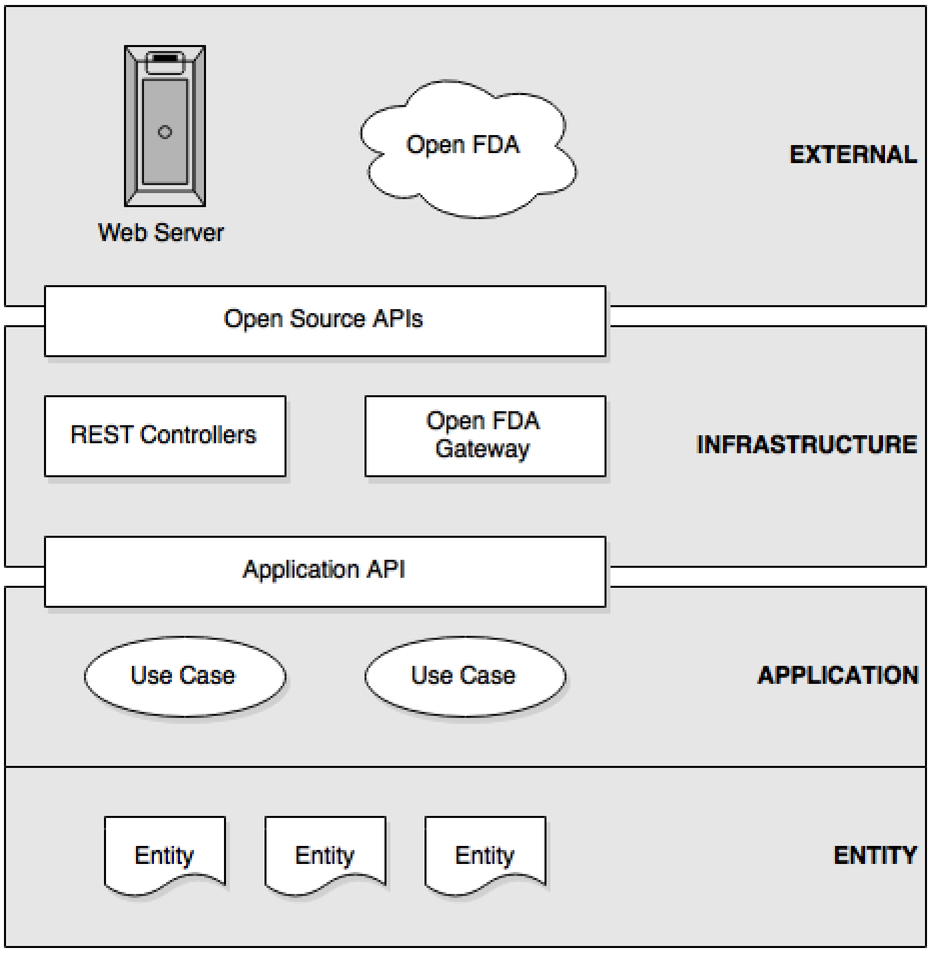


Figure 2: Software Architecture

3.3 Application

The application layer is the "conductor". It implements the use cases and business rules of the software. To achieve this it manipulates the domain model in the entity layer.

For example, if a line item were added to an order it would add the line item to the order entity and then ask the order repository to store the new line item. The application layer doesn't concern itself with where the entity is actually stored. It might be in a database, an XML file, or in memory. It is only important that the repository fulfils the contract as specified by the application API.

3.4 Entities

The entities are objects which store the data manipulated in the Application Layer.

3.5 Request Lifecycle

Figure 3, provides an example of how data passes through the various software layers.

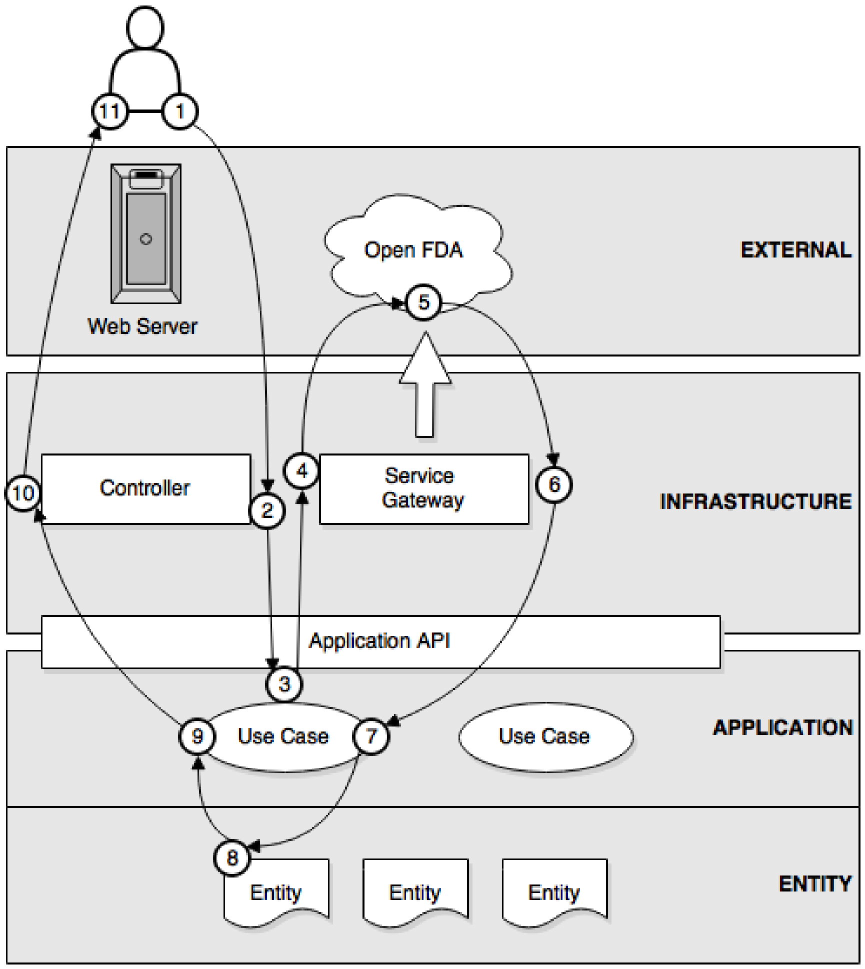


Figure 3: Request Lifecycle

1. Browser sends a request to the web server which is processed by a Spring MVC controller.
2. Spring MVC executes a use case in the application layer.
3. The use case invokes the service gateway interface to request data.
4. The service gateway makes a request to the external REST API.
5. The external REST service returns a JSON representation of the data.
6. The JSON response is unmarshalled to a data transfer object.
7. A domain entity is created from the data transfer object.
8. The domain entity is populated from the data transfer object.
9. The controller is informed of the result.
10. The controller returns a JSON response to the client.

4.0 REST Services

The application consumes the Open FDA Labelling API and the Adverse Events API.

The application provides three APIs which return JSON:

- Drug API - Information about a particular drug
- Event API - Makes several calls to the OpenFDA API to remix it to provide a summary of adverse events
- Suggestion API - Given the partial name of a drug, returns suggestions for autocomplete.

Please refer to the Swagger documentation for further details about these APIs.

5.0 Testing

Testing ensures that the software meets the acceptance criteria defined in the user story. Automated tests reduce the time to fully test the software by making it possible to fully re-test the software on demand.

Bugs can be introduced into software each time the code it changed. However, automated testing allows us to run the full suite of tests each time the code is updated. This is critical to successful agile development, because in order to evolve the architecture changes must be made. Automated tests make the code easier to refactor to meet new requirements. It is also possible to aggressively refactor the software and ensure it still works.

5.1 Test Driven Development

Test driven development is software development technique that improves code coverage. A test is written, which fails because there is no code to back it. Then the minimum amount of code is written to make the test pass. If the test is written after the code it is less obvious if the test covers all of the code.

Our test driven development process, sometimes referred to as the "red-green-refactor" loop is show in Figure 4 below. The test is written (red), and then code is written to make the test pass (green). The code written during the green cycle doesn't need to be well-factored: it only needs to pass the test. After the test passes the code is refactored. Then the cycle continues.

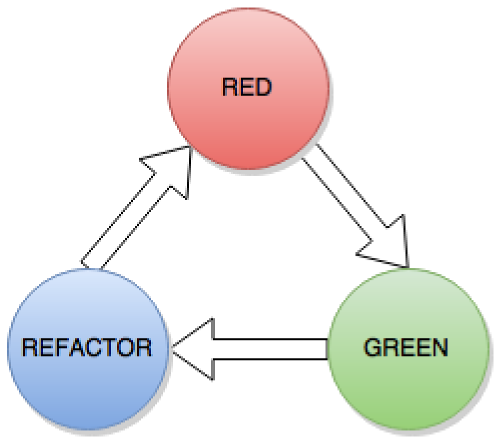


Figure 4: Test Driven Development Process

5.2 Test Tools

- JUnit - A Java based unit test framework.
- Serenity BDD - a automated web integration test tool. It drives a web browser to interact with the website as a user would

5.3 Test Coverage

Jenkins and SonarQube generate reports on the number of unit tests, test status (pass/fail), and test code coverage. The results from these tests are illustrated in Figure 5 and Figure 6.

- Jenkins reports
- SonarQube reports

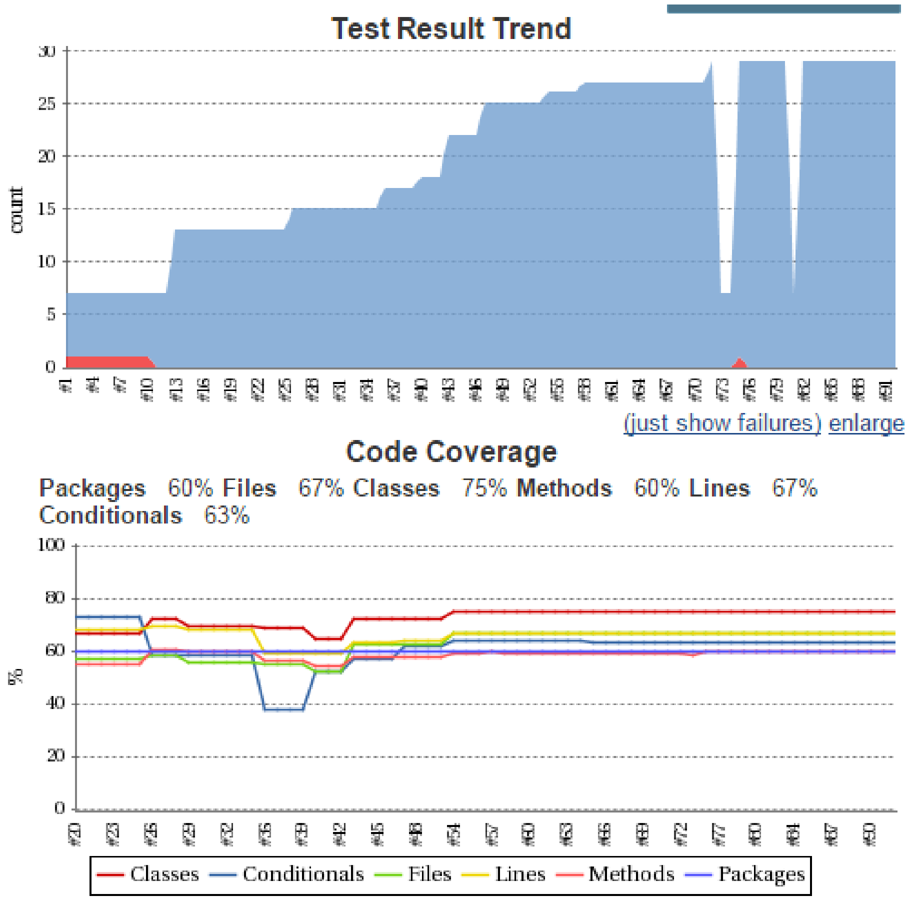
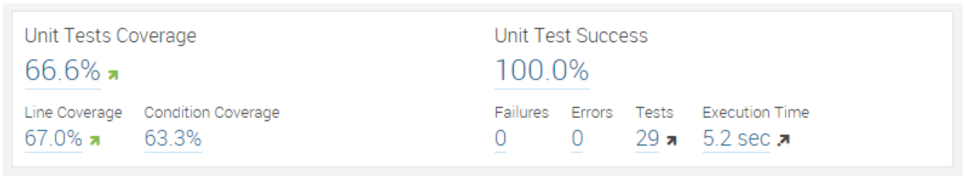


Figure 5: Jenkins Test Coverage Report



Unit Tests Coverage		Unit Test Success			
66.6%		100.0%			
Line Coverage	Condition Coverage	Failures	Errors	Tests	Execution Time
67.0%	63.3%	0	0	29	5.2 sec

Figure 6: SonarQube Test Coverage Report

5.4 Testability

This software was designed to be easy to test, as illustrated in Figure 7. The core application interacts with the infrastructure layer via Java interfaces. The infrastructure can be swapped out with test objects that emulate the expected behavior of the system.

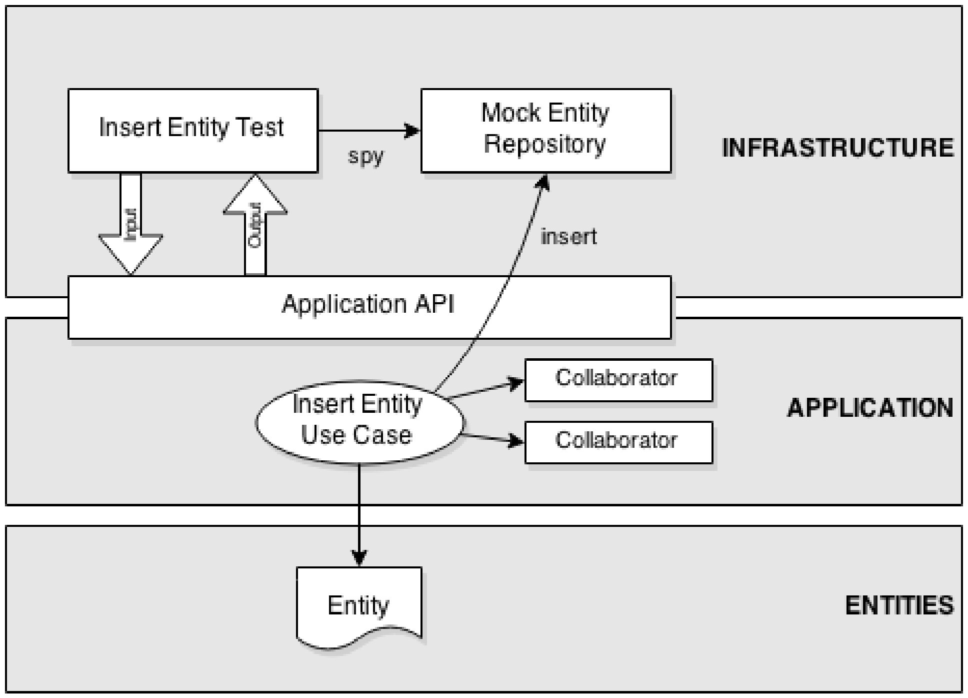


Figure 7: Test Architecture

5.5 Infrastructure Testing

Ideally the infrastructure layer is simple and thin, so it's obvious what it does and doesn't require testing, but when this isn't the case the public API can be stubbed. For example, we can return a "not found" response from the application layer and verify the controller returns a 404 HTTP response code. Our test process is demonstrated in Figure 8 below.

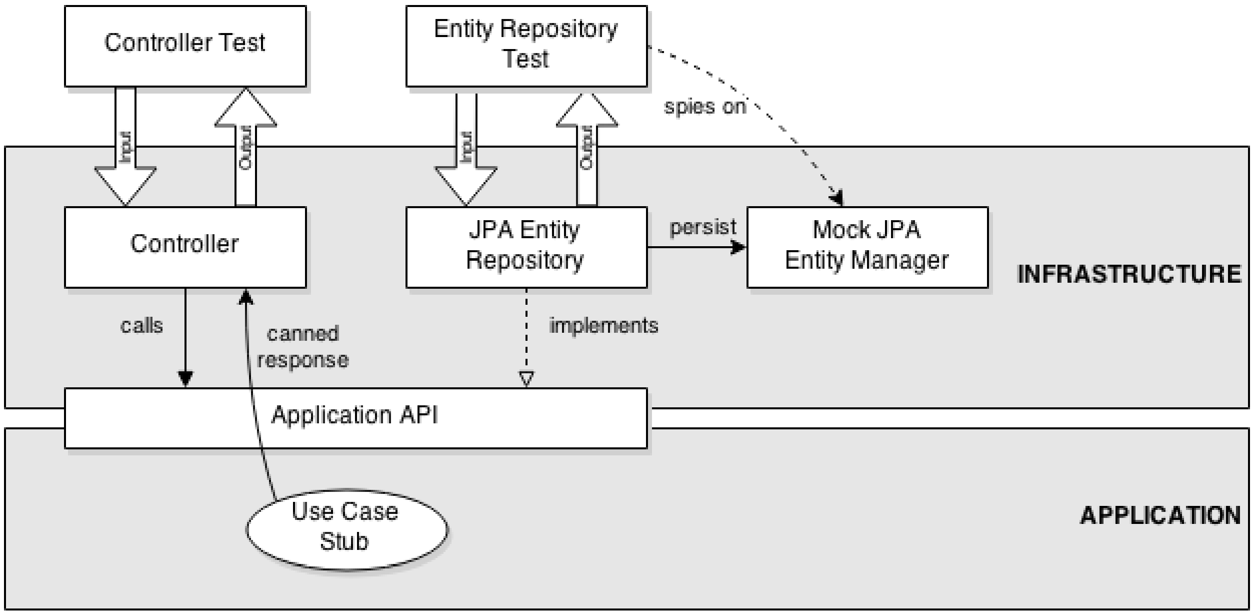


Figure 8: Test Process

6.0 Open Source Technology Stack

1. nginx 1.0.15 - Light-weight HTTP server/proxy
2. Spring Boot 1.2.4 - Standalone Java servlet container.
3. AngularJS 1.3.16 - Front-end MVC framework to create a single page application
4. Twitter Bootstrap 3.3.5 - A popular CSS framework
5. Docker 1.5.0 - Operating system level container-based virtualization
6. OpenJDK 1.8.0 - Java runtime environment and development kit

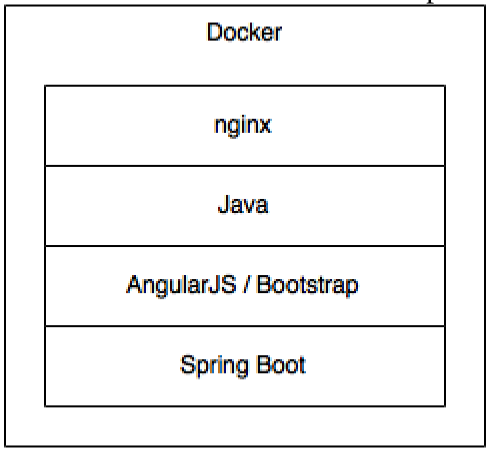


Figure 9: Technology Stack

Additional technologies are listed in LICENSES.md on GitHub.

7.0 Continuous Integration and Deployment

As Illustrated in Figure 10, Jenkins performs builds automatically after source code check-ins to GitHub. A notification is sent to the team chat room. If the unit tests pass, then a post-build task creates a Docker image, deploys it to the application server, and starts up the new container.

Another Jenkins job updates the project's Maven site which includes several reports. The Jenkins reports and SonarQube reports are also updated.

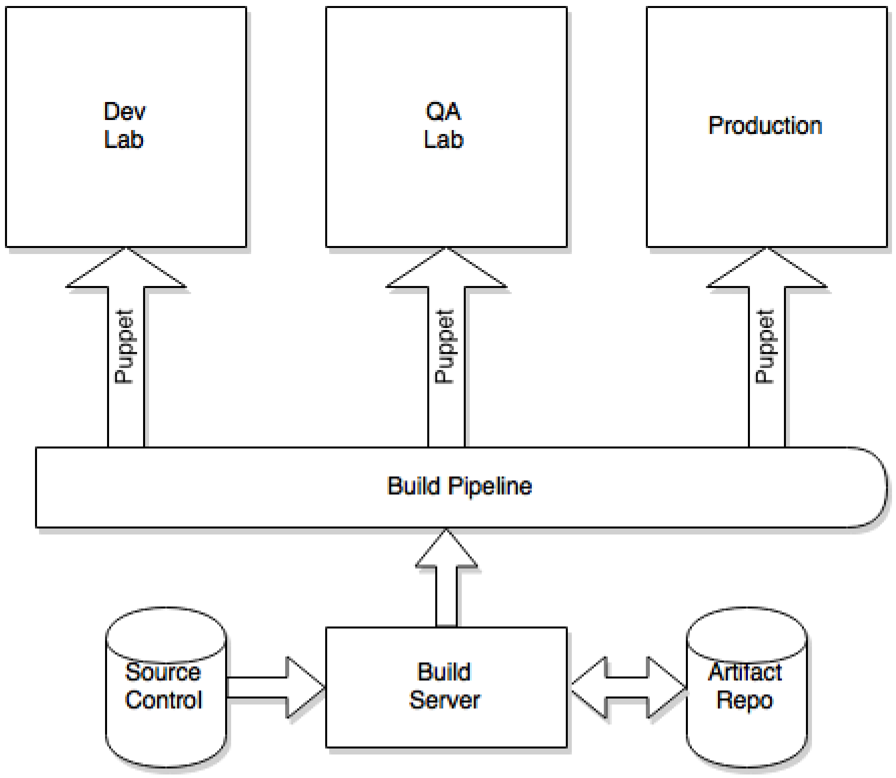


Figure 10: Automated Deployment

8.0 Configuration Management

GitHub is used for version control of Java and Puppet source code, and documentation. Puppet modules are used to manage the state of servers.

9.0 Documentation

Documentation is automatically generated from the source code where possible.

- Javadocs - Java API documentation
- Swagger - REST API documentation