MifosX Functional Test Strategy

3 June 2014

# High-level system architecture

MifosX Application follows a 3-tier architecture:

* **UI**: The Community UI app, written in AngularJS. Talks to the backend server over REST.
* **Server**: Java app running on Tomcat.
* **Database**: MySQL server cluster (supporting Multi-tenancy).

# Assumptions:

We are assuming that the Functional Testing is being done for the UI application, and the backend server will be independently tested using [Service Tests](https://mifosforge.jira.com/wiki/display/MIFOSX/Integration%2BTests) which hit the services directly. Unit Tests, and Service Tests will cover most business logic scenarios, whereas the UI Functional Tests will be written to check most-frequently used paths from a UI testing perspective.

We will need more discussion on exact scenarios for UI Functional Tests.

# Approach 1: Using cucumber-jvm

## Test Tech-Stack

* Cucumber-jvm / java / apache-logger / test data in json / cucumber-pretty reports
* webdriver to interact with browser
* cxf library to interact with webservices (to validate business functionality by avoiding going through UI layer all the time, and also for data setup - which will be faster by web services instead of UI)

## Sample code:

<https://github.com/anandbagmar/cuke-jvm-sample/tree/mifosx>

Details about the cuke-jvm framework can be found [here](http://essenceoftesting.blogspot.in/2014/04/sample-test-automation-framework-using.html).

## Pros

* Tech-stack learning curve is very simple (test implementation using Java, test specification using cucumber-jvm)
* Mature tech-stack and good support in open-source community

## Cons

* All test implementation will be in Java (even though the UI is developed using AngularJS)

# Approach 2: Using cucumber-protractor

## Test Tech-Stack

* Cucumber.JS - for specifying the intent of the test scenarios
* Programming in Javascript (implementing the Cucumber steps)
* WebDriverJS - for interacting with the browser

## Pros

* Having tests in the same tech stack as development makes Continuous Integration and Continuous Deployment easy.

## Cons

* WebDriverJS (selenium-webdriver to be precise) is based on JavaScript Promises. Promises are cryptic for newbies to JavaScript and it is a difficult learning curve.
* General JavaScript Programming could also be difficult to learn, with the many differences that JavaScript has from other Object oriented languages like Java.

**Assumptions**

* The test framework developers are not proficient in JavaScript.

# Recommended approach:

Given the above findings, we feel ***Approach 1 - using cucumber-jvm with WebDriver*** is a good approach to proceed with. One thing remaining to validate is to embed the cucumber-jvm tests with the community-app code and ensure there is no conflict / un-expected behavior from the same. If this approach is approved, we can do a quick spike on the same.