OpenLMIS Documentation

Release 3.0

VillageReach

Dec 13, 2018

Contents

1 Contents: 3

2 Links: 159



OpenLMIS (Open Logistics Management Information System) is software for a shared, open source solution for managing medical commodity distribution in low- and middle-income countries. For more information, see OpenLMIS.org.

CHAPTER 1

Contents:

1.1 Release Notes

To download a release, please visit GitHub.

1.1.1 DRAFT 3.5.0 Release Notes -

Status: Pending

3.5.0 is a stable release, and all users of OpenLMIS version 3 are encouraged to adopt it.

Release Notes

The OpenLMIS Community is excited to announce the **3.5.0 release** of OpenLMIS! It is another major milestone in the version 3 re-architecture that allows more functionality to be shared among the community of OpenLMIS implementers. This release was a major milestone in the the Gap Project and adds more functionality to reduce the gap in features between different versions of OpenLMIS. We are excited to announce that four organizations collaboratively worked on the 3.5 release!

For a full list of features and bug-fixes since 3.4.1, see OpenLMIS 3.5.0 Jira tickets.

For information about future planned releases, see the Living Product Roadmap. Pull requests and contributions are welcome.

Compatibility

Unless noted here, all other changes to OpenLMIS 3.x are backwards-compatible. All changes to data or schemas include automated migrations from previous versions back to version 3.0.1. All new or altered functionality is listed in the sections below for New Features and Changes to Existing Functionality.

Upgrading from Older Versions

If you are upgrading to OpenLMIS 3.5.0 from OpenLMIS 3.0.x or 3.1.x (without first upgrading to 3.2.x), please review the 3.2.0 Release Notes for important compatibility information about a required PostgreSQL extension and data migrations.

For information about upgrade paths from OpenLMIS 1 and 2 to version 3, see the 3.0.0 Release Notes.

Download or View on GitHub

OpenLMIS Reference Distribution 3.5.0

Known Bugs

Bug reports are collected in Jira for troubleshooting, analysis and resolution on an ongoing basis. See OpenLMIS 3.5.0 Bugs for the current list of known bugs.

To report a bug, see Reporting Bugs.

New Features

The OpenLMIS community proudly presents the following new features with 3.5.0:

- New in application reporting metrics, visualizations and tooling! To see a in-depth demo of the reporting tooling and capabilities, please reference the OpenLMIS YouTube channel) for a short video. Please note that the data is for development purposes only and not reflective of real data.
- Configuring OpenLMIS to support multiple suppliers for one facility based on specific products.
- Support the mCSD profile for facilities to allow for OpenLMIS to subscribe to a Facility Registry.
- Allow Average Consumption to calculate the order quantities for Stock Based Requisitions, which are requisitions automatically populated by stock transactions recorded in the stock management service.
- A new possible column for requisitions to capture the additional stock needed for new patients, which impacts the calculated order quantity.
- Ability to configure the Packs to Ship column to show up during the approval step to support users in seeing the number of packs to be ordered based on the dispensing unit.
- New integration tests to support the team in releasing faster!

Reference the 3.5 epics for more details.

Changes to Existing Functionality

See all 3.5.0 issues tagged 'UIChange' in Jira.

API Changes

Some APIs have changes to their contracts and/or their request-response data structures. These changes impact developers and systems integrating with OpenLMIS:

- **Fulfillment Service v8.0.0 - fulfillment changelog**

  - Refactored /api/orderFileTemplate API into /api/fileTemplate that persists config for both order and shipment templates.
  - Updated Transfer properties so that it can also be used to persist transfer properties for shipment files imports.

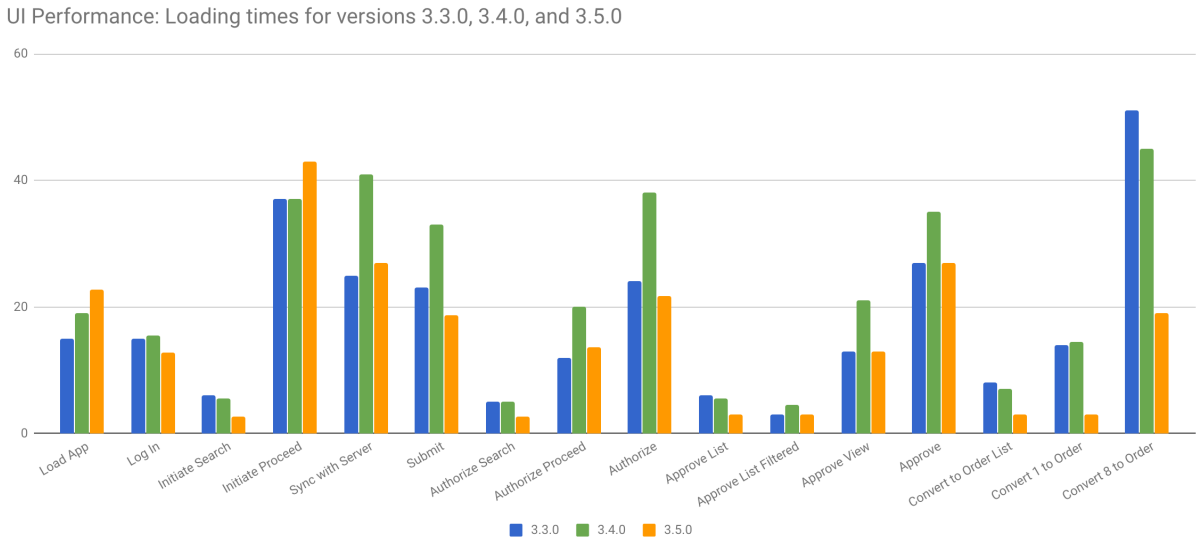
- **Reference Data Service v12.0.0 - ref-data changelog**
  - Removed DELETE /api/processingPeriods/{id} endpoint
  - Removed GET /api/users/{id}/supervisedFacilities endpoint
  - Changed supervisory node structure
  - Removed login restricted from the User model
- **Stock Management Service v4.0.0 - stockmanagement changelog**
  - Can't change type or category for stock card line item reason on update.
- **UI Components Service v7.0.0 - ui-components changelog**
  - Changed syntax for using sort component.

Performance

Overall, the performance of version 3.5.0 improves upon the 3.4.0 release. Significant improvements have been seen in the requisition submission, authorization, approval, and convert to order processes; with slight regressions in the initial load and initiate requisition processes.

OpenLMIS conducted manual performance tests of the same user workflows with the same test data we used in testing v3.2.1 to establish that last-mile performance characteristics have been retained at a minimum. For details on the test results and process, please see this wiki page.

The following chart displays the 3.5.0 UI loading times in seconds for 3.3.0, 3.4.0, and 3.5.0 using the same test data.



Test Coverage

Sam

All Changes by Component

Version 3.4.0 of the Reference Distribution contains updated versions of the components listed below. The Reference Distribution bundles these component together using Docker to create a complete OpenLMIS instance. Each component has its own own public GitHub repository (source code) and DockerHub repository (release image). The Reference Distribution and components are versioned independently; for details see Versioning and Releasing.

Auth Service 4.1.0

Auth CHANGELOG

CCE Service 1.0.2

CCE CHANGELOG

Fulfillment Service 8.0.0

Fulfillment CHANGELOG

Notification Service 4.0.1

Notification CHANGELOG

Reference Data Service 12.0.0

ReferenceData CHANGELOG

Report Service 1.1.2

This service is intended to provide reporting functionality for other components to use. Built-in reports in OpenLMIS 3.4.0 are still powered by their own services. In future releases, they may be migrated to a new version of this centralized report service.

Warning: Developers should take note that the design of this service will be changing with future releases. Developers and implementers are discouraged from using this 1.1.1 version to build additional reports.

Report CHANGELOG

Requisition Service 7.1.0

Requisition CHANGELOG

Stock Management 4.0.0

Stock Management CHANGELOG

Reference UI 5.1.2

The Reference UI is the web-based user interface for the OpenLMIS Reference Distribution. This user interface is a single page web application that is optimized for offline and low-bandwidth environments. The Reference UI is compiled together from module UI modules using Docker compose along with the OpenLMIS dev-ui. UI modules included in the Reference UI are:

Reference Data-UI 5.5.0

ReferenceData-UI CHANGELOG

Auth-UI 6.1.3

Auth-UI CHANGELOG

CCE-UI 1.0.2

CCE-UI CHANGELOG

Fulfillment-UI 6.0.2

Fulfillment-UI CHANGELOG

Report-UI 5.1.0

Report-UI CHANGELOG

Requisition-UI 5.5.0

Requisition-UI CHANGELOG

Stock Management-UI 2.0.2

Stock Management-UI CHANGELOG

UI-Components 7.0.0

UI-Components CHANGELOG

UI-Layout 5.1.2

UI-Layout CHANGELOG

Dev UI 8.1.0

The Dev-UI CHANGLOG

Components with No Changes

The components that have not changed are:

- Service Util
- Logging Service
- Consul-friendly distribution of nginx
- Docker Postgres 9.6-postgis image
- Docker scalyr image

Contributions

Many organizations and individuals around the world have contributed to OpenLMIS version 3 by serving on our committees (Governance, Product and Technical), requesting improvements, suggesting features and writing code and documentation. Please visit our GitHub repos to see the list of individual contributors on the OpenLMIS codebase. If anyone who contributed in GitHub is missing, please contact the Community Manager.

Thanks to the Malawi implementation team who has continued to contribute a number of changes that have global shared benefit.

Further Resources

Please see the Implementer Toolkit on the OpenLMIS website to learn more about best practicies in implementing OpenLMIS. Also, learn more about the OpenLMIS Community and how to get involved!

1.1.2 3.4.1 Patch Release Notes - 10 October 2018

Status: Stable with disclaimer

The 3.4.1 patch release is recommended for users of OpenLMIS version 3.4.0 and includes fixes for the delayed login process when using the Firefox browser and enabling the update of offline requisitions.

Disclaimer: The 3.4.1 Patch release does not contain any known blocking bugs. Full regression testing and manual performance testing was not conducted as part of the patch release.

Patch Release Notes

3.4.1 Patch Release contains the bug fixes for - OLMIS-5235 - OLMIS-5502

For information about future planned releases, see the Living Product Roadmap. Pull requests and contributions are welcome.

Compatibility

Compatible with OpenLMIS 3.4.0

Backwards-Compatible Except As Noted

Unless noted here, all other changes to OpenLMIS 3.x are backwards-compatible. All changes to data or schemas include automated migrations from previous versions back to version 3.0.1. All new or altered functionality is listed in the sections below for New Features and Changes to Existing Functionality.

Upgrading from Older Versions

If you are upgrading to OpenLMIS 3.4.1 from OpenLMIS 3.0.x or 3.1.x (without first upgrading to 3.2.x), please review the 3.2.0 Release Notes for important compatibility information about a required PostgreSQL extension and data migrations.

For information about upgrade paths from OpenLMIS 1 and 2 to version 3, see the 3.0.0 Release Notes.

Download or View on GitHub

OpenLMIS Reference Distribution 3.4.1

Known Bugs

No known additional bugs were included in this patch release. Bug reports are collected in Jira for troubleshooting, analysis and resolution on an ongoing basis. See OpenLMIS 3.4.0 Bugs for the current list of known bugs.

To report a bug, see Reporting Bugs.

New Features

No new features were introduced with this patch release.

Changes to Existing Functionality

Version 3.4.1 contains changes that impact users of existing functionality. Please review these changes which may require informing end-users and/or updating your customizations/extensions:

- OLMIS-5235: Performance issue after login using Firefox
- OLMIS-5502: No 'Update Requisiton' button after offline mode

Performance

No manual performance testing was conducted for this patch release.

Test Coverage

Manual regression tests were conducted using a set of 139 Zephyr tests tracked in Jira. One bug was found and resolved during testing. See the test cycle for all regression test case executions for this patch release: 3.4.1 Patch Release Test Plan and Execution.

Component Version Numbers

Version 3.4.1 of the Reference Distribution contains the following components and versions listed below. The Reference Distribution bundles these components together using Docker to create a complete OpenLMIS instance. Each component has its own own public GitHub repository (source code) and DockerHub repository (release image). The Reference Distribution and components are versioned independently; for details see Versioning and Releasing.

Auth Service 4.0.0

CCE Service 1.0.1

Fulfillment Service 7.0.1

Notification Service 4.0.0

Reference Data Service 11.0.0

Report Service 1.1.1

This service is intended to provide reporting functionality for other components to use. Built-in reports in OpenLMIS 3.4.0 are still powered by their own services. In future releases, they may be migrated to a new version of this centralized report service.

Warning: Developers should take note that the design of this service will be changing with future releases. Developers and implementers are discouraged from using this 1.1.1 version to build additional reports.

Requisition Service 7.0.1

Stock Management 3.1.0

Reference UI 5.1.0

The Reference UI is the web-based user interface for the OpenLMIS Reference Distribution. This user interface is a single page web application that is optimized for offline and low-bandwidth environments. The Reference UI is compiled together from module UI modules using Docker compose along with the OpenLMIS dev-ui. UI modules included in the Reference UI are:

Reference Data-UI 5.4.1

ReferenceData-UI CHANGELOG

Auth-UI 6.1.2

Auth-UI CHANGELOG

CCE-UI 1.0.1

Fulfillment-UI 6.0.1

Report-UI 5.0.6

Requisition-UI 5.5.0

Requisition-UI CHANGELOG

Stock Management-UI 2.0.1

Stock Management-UI CHANGELOG

UI-Components 6.0.1

UI-Components CHANGELOG

UI-Layout 5.1.1

Dev UI v8

1.1.3 3.4.0 Release Notes - 17 August 2018

Status: Stable

3.4.0 is a stable release, and all users of OpenLMIS version 3 are encouraged to adopt it.

Release Notes

The OpenLMIS Community is excited to announce the **3.4.0 release** of OpenLMIS! It is another major milestone in the version 3 re-architecture that allows more functionality to be shared among the community of OpenLMIS implementers. It is also the first release since the Gap Project has started. We are excited to announce that four organizations collaboratively worked on the 3.4 release!

For a full list of features and bug-fixes since 3.3.1, see OpenLMIS 3.4.0 Jira tickets.

For information about future planned releases, see the Living Product Roadmap. Pull requests and contributions are welcome.

Compatibility

Unless noted here, all other changes to OpenLMIS 3.x are backwards-compatible. All changes to data or schemas include automated migrations from previous versions back to version 3.0.1. All new or altered functionality is listed in the sections below for New Features and Changes to Existing Functionality.

Upgrading from Older Versions

If you are upgrading to OpenLMIS 3.4.0 from OpenLMIS 3.0.x or 3.1.x (without first upgrading to 3.2.x), please review the 3.2.0 Release Notes for important compatibility information about a required PostgreSQL extension and data migrations.

For information about upgrade paths from OpenLMIS 1 and 2 to version 3, see the 3.0.0 Release Notes.

Download or View on GitHub

OpenLMIS Reference Distribution 3.4.0

Known Bugs

Bug reports are collected in Jira for troubleshooting, analysis and resolution on an ongoing basis. See OpenLMIS 3.4.0 Bugs for the current list of known bugs.

To report a bug, see Reporting Bugs.

New Features

3.4 offers a wide range of new features and enhancements:

Reporting: The new reporting infrastructure is dockerized and deployed within the OpenLMIS deployment. See the Reporting service for more details.

Requisitions:

- Configurable templates for one-click requisitions based on current stock on hand records within stock management. We call this a stock based requisition.
- Ability to hide and add new products to improve end user usability.
- Ability to convert requisitions without creating an order.
- New R&R column to account for "additional quantities" from new demand.

Orders: New FTP retry feature when orders fail to transmit initially.

User Management:

- New "opting out" feature for email notifications.
- Improving the user profile so that users can manage their own data.
- New reset password options.
- New user profile information (roles and rights).

Notifications: Redesigned to support more personalization and future work on supporting more channels.

Changes to Existing Functionality

See all 3.4.0 issues tagged 'UIChange' in Jira.

API Changes

Some APIs have changes to their contracts and/or their request-response data structures. These changes impact developers and systems integrating with OpenLMIS:

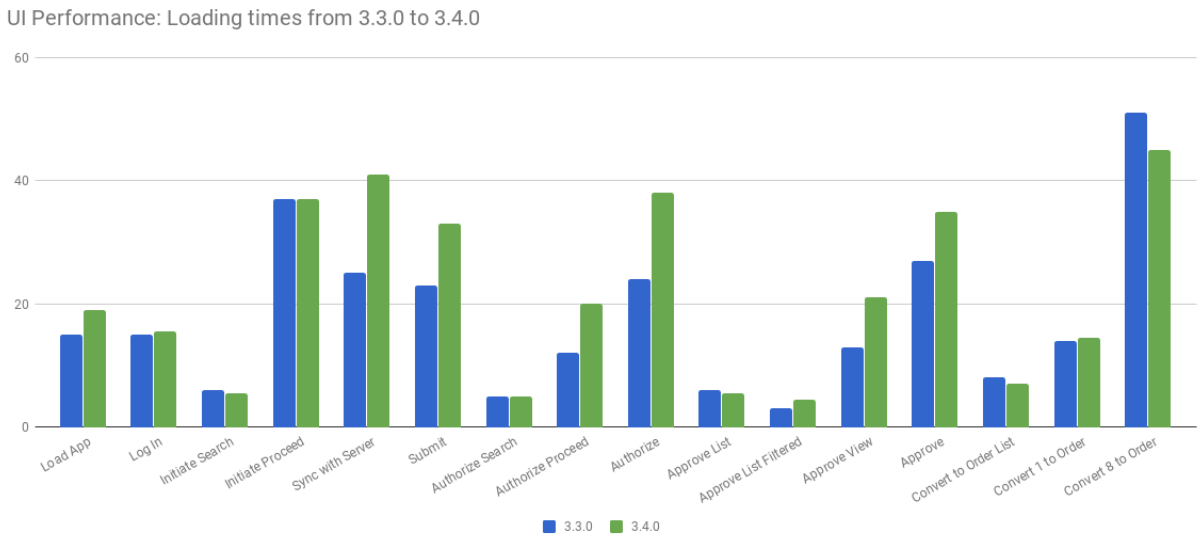
- Authentication Service v4.0.0 - Changes to the user resource structure: auth changelog
- Notification Service v4.0.0 - Changes to the user contact details resource and the /api/notification endpoint: notification changelog
- Reference Data Service v11.0.0 - Changes to the user resource structure: ref-data changelog
- Requisition Service v7.0.0 - Changes to the periodsForInitiate endpoint: requisition changelog
- Stock Management UI Service v3.1.0 - Renamed admin-reason-modal module: stockmanagement changelog
- UI Components Service v6.0.0 - Changed syntax for using datepicker: ui-components changelog
- Dev-UI Service v8.0.0 - Replaced syncTransifex grunt option: dev-ui changelog

Performance

There are minor regressions in the sync, submit, authorize and single approve within the requisition service with slight improvements in convert to order.

OpenLMIS conducted manual performance tests of the same user workflows with the same test data we used in testing v3.2.1 to establish that last-mile performance characteristics have been retained at a minimum. For details on the test results and process, please see this wiki page. For more details about the specific work done to improve performance for 3.4.0, please reference this list of tasks.

The following chart displays the 3.4.0 UI loading times in seconds for both 3.3.1 and 3.4.0 using the same test data.



Test Coverage

OpenLMIS 3.4.0 is the second release using the new Release Candidate process. As part of this process, full manual test cycles were executed for each release candidate published. Any critical or blocker bugs found during the release candidate were resolved in a bug fix cycle with a full manual test cycle executed before releasing the final version

3.4.0. Manual tests were conducted using a set of 142 Zephyr tests tracked in Jira. A total of 34 bugs were found during testing. See the spreadsheet of all test executions for this release: 3.4.0 release test case executions.csv.

All Changes by Component

Version 3.4.0 of the Reference Distribution contains updated versions of the components listed below. The Reference Distribution bundles these component together using Docker to create a complete OpenLMIS instance. Each component has its own own public GitHub repository (source code) and DockerHub repository (release image). The Reference Distribution and components are versioned independently; for details see Versioning and Releasing.

Auth Service 4.0.0

Auth CHANGELOG

CCE Service 1.0.1

CCE CHANGELOG

Fulfillment Service 7.0.1

Fulfillment CHANGELOG

Notification Service 4.0.0

Notification CHANGELOG

Reference Data Service 11.0.0

ReferenceData CHANGELOG

Report Service 1.1.1

This service is intended to provide reporting functionality for other components to use. Built-in reports in OpenLMIS 3.4.0 are still powered by their own services. In future releases, they may be migrated to a new version of this centralized report service.

Warning: Developers should take note that the design of this service will be changing with future releases. Developers and implementers are discouraged from using this 1.1.1 version to build additional reports.

Report CHANGELOG

Requisition Service 7.0.0

Requisition CHANGELOG

Stock Management 3.1.0

Stock Management CHANGELOG

Reference UI 5.1.0

The Reference UI is the web-based user interface for the OpenLMIS Reference Distribution. This user interface is a single page web application that is optimized for offline and low-bandwidth environments. The Reference UI is compiled together from module UI modules using Docker compose along with the OpenLMIS dev-ui. UI modules included in the Reference UI are:

Reference Data-UI 5.4.0

ReferenceData-UI CHANGELOG

Auth-UI 6.1.1

Auth-UI CHANGELOG

CCE-UI 1.0.1

CCE-UI CHANGELOG

Fulfillment-UI 6.0.1

Fulfillment-UI CHANGELOG

Report-UI 5.0.6

Report-UI CHANGELOG

Requisition-UI 5.4.0

Requisition-UI CHANGELOG

Stock Management-UI 3.1.0

Stock Management-UI CHANGELOG

UI-Components 6.0.0

UI-Components CHANGELOG

UI-Layout 5.1.1

UI-Layout CHANGELOG

Dev UI v8

The Dev UI developer tooling has advanced to v8.

Components with No Changes

The components that have not changed are:

- Service Util
- Logging Service
- Consul-friendly distribution of nginx
- Docker Postgres 9.6-postgis image
- Docker scalyr image

Contributions

Many organizations and individuals around the world have contributed to OpenLMIS version 3 by serving on our committees (Governance, Product and Technical), requesting improvements, suggesting features and writing code and documentation. Please visit our GitHub repos to see the list of individual contributors on the OpenLMIS codebase. If anyone who contributed in GitHub is missing, please contact the Community Manager.

Thanks to the Malawi implementation team who has continued to contribute a number of changes that have global shared benefit.

Further Resources

Please see the Implementer Toolkit on the OpenLMIS website to learn more about best practicies in implementing OpenLMIS. Also, learn more about the OpenLMIS Community and how to get involved!

1.1.4 3.3.1 Patch Release Notes - 17 July 2018

Status: Stable with disclaimer

3.3.1 Patch release is recommended for users of OpenLMIS version 3.3.0 because the patch inclues a bug fix for requisition statuses when saved concurrently. Disclaimer: The 3.3.1 Patch release does not contain any known blocking bugs. Full regression testing and manual performance testing was not conducted as part of the patch release.

Patch Release Notes

3.3.1 Patch Release contains the bug fix for - OLMIS-4728.

For information about future planned releases, see the Living Product Roadmap. Pull requests and contributions are welcome.

Compatibility

Compatible with OpenLMIS 3.3.0

Backwards-Compatible Except As Noted

Unless noted here, all other changes to OpenLMIS 3.x are backwards-compatible. All changes to data or schemas include automated migrations from previous versions back to version 3.0.1. All new or altered functionality is listed in the sections below for New Features and Changes to Existing Functionality.

Upgrading from Older Versions

If you are upgrading to OpenLMIS 3.3.0 from OpenLMIS 3.0.x or 3.1.x (without first upgrading to 3.2.x), please review the 3.2.0 Release Notes for important compatibility information about a required PostgreSQL extension and data migrations.

For information about upgrade paths from OpenLMIS 1 and 2 to version 3, see the 3.0.0 Release Notes.

Download or View on GitHub

OpenLMIS Reference Distribution 3.3.1

Known Bugs

No known additional bugs were included in this patch release. Bug reports are collected in Jira for troubleshooting, analysis and resolution on an ongoing basis. See OpenLMIS 3.3.0 Bugs for the current list of known bugs.

To report a bug, see Reporting Bugs.

New Features

No new features were introduced with this patch release.

Changes to Existing Functionality

Version 3.3.1 contains changes that impact users of existing functionality. Please review these changes which may require informing end-users and/or updating your customizations/extensions:

- OLMIS-4728: Requisition's properties can be overwritten when saved concurrently.

Performance

No manual performance testing was conducted for this patch release.

Test Coverage

Manual regression tests were conducted using a set of 30 Zephyr tests tracked in Jira. One bug was found and resolved during testing. See the test cycle for all regression test case executions for this patch release: 3.3.1 Patch Release Test Plan and Execution.

Component Version Numbers

Version 3.3.1 of the Reference Distribution contains the following components and versions listed below. The Reference Distribution bundles these components together using Docker to create a complete OpenLMIS instance. Each component has its own own public GitHub repository (source code) and DockerHub repository (release image). The Reference Distribution and components are versioned independently; for details see Versioning and Releasing.

Auth Service 3.2.0

CCE Service 1.0.0

Fulfillment Service 7.0.0

Notification Service 3.0.5

Reference Data Service 10.0.0

Reference UI 5.0.7

The Reference UI (https://github.com/OpenLMIS/openlmis-reference-ui/) is the web-based user interface for the OpenLMIS Reference Distribution. This user interface is a single page web application that is optimized for offline and low-bandwidth environments. The Reference UI is compiled together from module UI modules using Docker compose along with the OpenLMIS dev-ui. UI modules included in the Reference UI are:

auth-ui 6.1.0

cce-ui 1.0.0

fulfillment-ui 6.0.0

referencedata-ui 5.3.0

report-ui 5.0.5

requisition-ui 6.1.0

stockmanagement-ui 1.1.0

ui-components 5.3.0

ui-layout 5.1.0

Dev UI v7

Report Service 1.0.1

This service is intended to provide reporting functionality for other components to use. It is a 1.0.0 release which is stable for production use, and it powers one built-in report: the Facility Assignment Configuration Errors report (OLMIS-2760).

Additional built-in reports in OpenLMIS 3.3.1 are still powered by their own services. In future releases, they may be migrated to a new version of this centralized report service.

Warning: Developers should take note that the design of this service will be changing with future releases. Developers and implementers are discouraged from using this 1.0.1 version to build additional reports.

Requisition Service 6.0.0

Stock Management 3.0.0

Service Util 3.1.0

1.1.5 3.3.0 Release Notes - 27 April 2018

Status: Stable

3.3.0 is a stable release, and all users of OpenLMIS version 3 are encouraged to adopt it.

Release Notes

The OpenLMIS Community is excited to announce the **3.3.0 release** of OpenLMIS! It is another major milestone in the version 3 re-architecture that allows more functionality to be shared among the community of OpenLMIS implementers.

3.3.0 includes a wide range of new features and functionality. The majority of the features were defined as the Minimal Viable Product (MVP), or minimum feature set, to support countries in managing their immunization supply chain by a group of key immunization stakeholders and OpenLMIS community members. Key features include managing cold chain equipment (CCE) inventory, integrating with a Remote Temperature Monitoring (RTM) platform, calculating reorder amounts based on targets, fulfilling orders, and receiving commodities into inventory based on shipments. See the New Features section for details.

For a full list of features and bug-fixes since 3.2.1, see OpenLMIS 3.3.0 Jira tickets.

For information about future planned releases, see the Living Product Roadmap. Pull requests and contributions are welcome.

Compatibility

The requisition service introduced, OLMIS-3929: View and edit multiple requisition templates per program, which requires a manual data migration explained here.

The fulfillment service has a major release due to the additional features in fulfilling orders within OpenLMIS. Please review the fulfillment service changelog in detail to ensure a clear understanding of the breaking changes.

The reference data service uses new rights associated with the new proof of delivery functionality. Please review the changlog for the Reference data service in detail to ensure a clear understanding of the breaking changes related to rights.

Batch Requisition Approval: The Batch Approval screen, which was improved in OpenLMIS 3.2.1, is still not officially supported. The UI screen is disabled by default. Implementations can override the code in their local customizations in order to use the screen. Further performance improvements are needed before the screen is officially supported. See OLMIS-3182 and the tickets linked to it for details.

Backwards-Compatible Except As Noted

Unless noted here, all other changes to OpenLMIS 3.x are backwards-compatible. All changes to data or schemas include automated migrations from previous versions back to version 3.0.1. All new or altered functionality is listed in the sections below for New Features and Changes to Existing Functionality.

Upgrading from Older Versions

If you are upgrading to OpenLMIS 3.3.0 from OpenLMIS 3.0.x or 3.1.x (without first upgrading to 3.2.x), please review the 3.2.0 Release Notes for important compatibility information about a required PostgreSQL extension and data migrations.

For information about upgrade paths from OpenLMIS 1 and 2 to version 3, see the 3.0.0 Release Notes.

Download or View on GitHub

OpenLMIS Reference Distribution 3.3.0

Known Bugs

Bug reports are collected in Jira for troubleshooting, analysis and resolution on an ongoing basis. See OpenLMIS 3.3.0 Bugs for the current list of known bugs.

To report a bug, see Reporting Bugs.

New Features

OpenLMIS 3.3.0 contains the following features, the majority are specific to the Vaccine Module MVP Features, were completed by the OpenLMIS development team:

- Vaccine stock based requisitions that allow users to populate a requisition based on current stock levels and forecasted demand targets or ideal stock amounts.
- Enhancements to support stock management for vaccines.
- Order fulfillment, sometimes referred to as the process of resupplying supervised facilities. Includes support for configuring some facilities to have orders fulfilled within OpenLMIS and others sending orders to external suppliers like a National Store or third party supplier. Supports using the ideal product model, ordering using commodity types and fulfilling using TradeItems, to enable end-to-end visibility.
- Receiving stock into inventory, using an electronic Proof of Delivery based on the shipment details created in OpenLMIS.
- Forecasting and Estimation features to upload forecasted demand targets and use those targets to calculate reorder amounts.
- Official release of the Cold Chain Equipment (CCE) service and includes a new feature displaying active alerts on specific pieces of equipment inventory using a standards based interoperability with a Remote Temperature Monitoring (RTM) platform.
- Administration screens included assigning requisition templates to facility types within a program, view and create facility types, and manage API keys.
- The analytics infrastructure and DISC indicators were developed and deployed in a new open-source stack. By the 3.3 release, this technology infrastructure is not deployed within our dockerized microservice architecture. We can provide access to the demo environment for showcasing and will focus on deploying in docker for the next release.

The following Pull Requests were contributed by community members:

- Reference Data and Reference Data UI OLMIS-3448
- Reference Data OLMIS-4337
- Requisition OLMIS-4383

Changes to Existing Functionality

Version 3.3.0 contains changes that impact users of existing functionality. Please review these changes which may require informing end-users and/or updating your customizations/extensions:

- OLMIS-3949: The **redesign of emergency requisitions** made large UI and API changes. Emergency requisitions now use a simplified template with limited columns. Please ensure to review all relevant documentation to understand the decision making, which went through the product committee, and major UI changes to alert relevant users.
- OLMIS-3929: View and edit multiple requisition templates per program.
- OLMIS-3166: Add user control for AppCache. Users can see their build number and update their web page application to the latest build.
- OLMIS-3877: UI filter component is consistent across pages.
- OLMIS-4026: Changed table styles to support order fulfillment complexity.

See all 3.3.0 issues tagged 'UIChange' in Jira.

API Changes

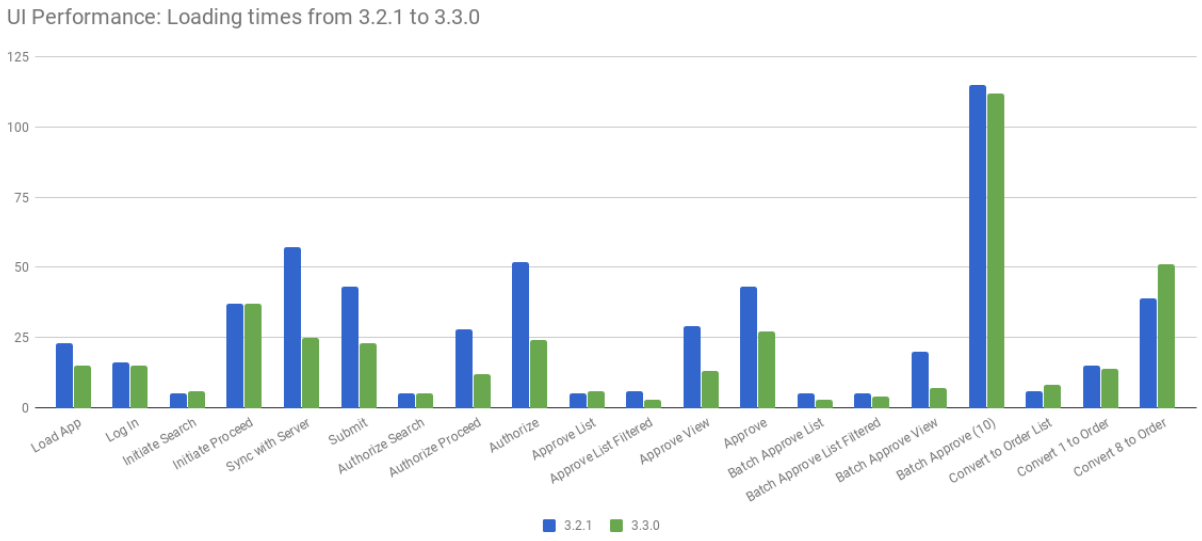
Some APIs have changes to their contracts and/or their request-response data structures. These changes impact developers and systems integrating with OpenLMIS:

- Requisition service has a major release, v6.0.0, due to the redesign of emergency requisitions. See the Requisition changelog for details.
- Fulfillment service has a major release, v7.0.0, due to significant changes in the data model for orders, shipments and proofs of delivery. See the Fulfillment changelog for details.
- Reference data service has a major release, v10.0.0, due to changes for pagination, filtering and rights. See the Reference data changelog for details.
- Stock management service has a major release, v3.0.0, due to significant changes to stock events and physical inventory data. See the Stock management changelog for details.

Performance

OpenLMIS conducted manual performance tests of the same user workflows with the same test data we used in testing v3.2.1 to establish that last-mile performance characteristics have been retained at a minimum. For details on the test results and process, please see this wiki page for details. There are minor improvements in the sync, submit, authorize and single approve within the requisition service. For more details about the specific work done to improve performance for 3.3.0, please reference this list of tasks.

The following chart displays the 3.3.0 UI loading times in seconds for both 3.2.1 and 3.3.0 using the same test data.



Test Coverage

OpenLMIS 3.3.0 is the second release using the new Release Candidate process. As part of this process, a full manual regression test cycle was conducted, and multiple release candidates were published to address critical bugs before releasing the final version 3.3.0.

Manual tests were conducted using a set of 136 Zephyr tests tracked in Jira. A total of 50 bugs were found during testing. The full set of tests were executed on the third Release Candidate (RC3). With previous release candidates

(RC1 and RC2), only the first phase of testing was conducted. See the spreadsheet of all regression test executions for this release: 3.3.0-regression-tests.csv.

OpenLMIS 3.3.0 also includes a large set of automated tests. There are multiple types of tests, including Unit Tests, Integration, Component, Contract and End-to-End. These tests exist in the API services in Java as well as in the JavaScript UI web application. See the Testing Guide.

For OpenLMIS 3.3.0, here are a few key statistics on automated tests:

- There are **2,665 unit tests** in the API services in Java, not including other types of tests nor tests in the Javascript UI application. Sonar counts unit tests on each Java component.
- Test **coverage is over 60%** for all components, both Java and JavaScript, and is over 80% for many components. Sonar tracks test coverage and fails

  quality gates if developers contribute new code with less than 80% coverage.

All of the automated tests, both Java and Javascript tests of all types, are passing as of the time of the release. Any failing test would stop the build and block a release.

Further advances in automated testing are still on the horizon for future releases of OpenLMIS:

- Automated performance tests: There is already an automated test tool that measures the speed of API endpoints with a large set of performance test data. However, not all tests pass and there is not an established baseline for performance/speed of all areas of the system. Achieving this will greatly improve the objective means for tracking and improving performance.
- End-to-end testing: There is already an end-to-end testing toolset. However, coverage is very low. The addition of more end-to-end automated tests can reduce the manual test effort that is currently required for each release. It can help developers identify and fix regressions so the community can move towards a "continuous delivery" release process.

All Changes by Component

Version 3.3.0 of the Reference Distribution contains updated versions of the components listed below. The Reference Distribution bundles these component together using Docker to create a complete OpenLMIS instance. Each component has its own own public GitHub repository (source code) and DockerHub repository (release image). The Reference Distribution and components are versioned independently; for details see Versioning and Releasing.

Auth Service 3.2.0

Source: Auth CHANGELOG

CCE Service 1.0.0

This is the first stable release of openlmis-cce.

Source: CCE CHANGELOG

Fulfillment Service 7.0.0

Source: Fulfillment CHANGELOG

Notification Service 3.0.5

Source: Notification CHANGELOG

Reference Data Service 10.0.0

Source: ReferenceData CHANGELOG

Reference UI 5.0.6

The Reference UI (https://github.com/OpenLMIS/openlmis-reference-ui/) is the web-based user interface for the OpenLMIS Reference Distribution. This user interface is a single page web application that is optimized for offline and low-bandwidth environments. The Reference UI is compiled together from module UI modules using Docker compose along with the OpenLMIS dev-ui. UI modules included in the Reference UI are:

auth-ui 6.1.0

See openlmis-auth-ui CHANGELOG

cce-ui 1.0.0

This is the first stable release of openlmis-cce-ui; it provides CCE inventory management and administration screens that work with the openlmis-cce service APIs.

See: openlmis-cce-ui CHANGELOG

fulfillment-ui 6.0.0

See openlmis-fulfillment-ui CHANGELOG

referencedata-ui 5.3.0

See openlmis-referencedata-ui CHANGELOG

report-ui 5.0.5

See openlmis-report-ui CHANGELOG

requisition-ui 5.3.1

See openlmis-requisition-ui CHANGELOG

stockmanagement-ui 1.1.0

See openlmis-ui-components CHANGELOG

ui-components 5.3.0

See openlmis-ui-components CHANGELOG

ui-layout 5.1.0

See openlmis-ui-layout CHANGELOG

Dev UI v7

The Dev UI developer tooling has advanced to v7.

Report Service 1.0.1

This service is intended to provide reporting functionality for other components to use. It is a 1.0.0 release which is stable for production use, and it powers one built-in report: the Facility Assignment Configuration Errors report (OLMIS-2760).

Additional built-in reports in OpenLMIS 3.3.0 are still powered by their own services. In future releases, they may be migrated to a new version of this centralized report service.

Warning: Developers should take note that the design of this service will be changing with future releases. Developers and implementers are discouraged from using this 1.0.1 version to build additional reports.

Source: Report CHANGELOG

Requisition Service 6.0.0

Source: Requisition CHANGELOG

Stock Management 3.0.0

Source: Stock Management CHANGELOG

Service Util 3.1.0

We now use an updated library for shared Java code called service-util.

Source: Report CHANGELOG

Components with No Changes

Other tooling components have not changed, including: the logging service, the Consul-friendly distribution of nginx, the docker Postgres 9.6-postgis image, and the docker scalyr image.

Contributions

Many organizations and individuals around the world have contributed to OpenLMIS version 3 by serving on our committees (Governance, Product and Technical), requesting improvements, suggesting features and writing code and documentation. Please visit our GitHub repos to see the list of individual contributors on the OpenLMIS codebase. If anyone who contributed in GitHub is missing, please contact the Community Manager.

Thanks to the Malawi implementation team who has continued to contribute a number of changes that have global shared benefit.

Further Resources

We are excited to announce the release of the first iteration of the Implementer Toolkit on the OpenLMIS website. Learn more about the OpenLMIS Community and how to get involved!

1.1.6 3.2.1 Release Notes - 15 November 2017

Status: Stable

3.2.1 is a stable release, and all users of OpenLMIS version 3 are encouraged to adopt it.

Release Notes

The release of 3.2.1 is primarily a **bug-fix** and **performance** release, with over 40 bugs fixed and over 20 other improvements since 3.2.0 including major improvements in performance.

This release does include some new features; see the New Features section below.

See the Living Product Roadmap for information about future planned releases. Pull requests and contributions are welcome.

Compatibility

Important! Stock Management data migration: OpenLMIS 3.2.1 introduces a new constraint that forces the adjustment reasons to be unique within each requisition line item. This means that it will no longer be possible to have two "expired" adjustments in a single product, eg. Expired: 20 and Expired: 30. It will still be possible to have different adjustment reasons, eg. Expired: 20 and Lost: 30. The UI does not allow users to add the same adjustment reason twice starting with OpenLMIS 3.2.1. Users should now provide a total value for a given adjustment reason.

Due to this change, it is necessary for any existing OpenLMIS implementations to migrate their stock adjustments data to merge any duplicates. Implementations can do this manually before upgrading to 3.2.1, otherwise OpenLMIS 3.2.1 will apply a default migration automatically. The default migration automatically merge the duplicates by adding together the quantities from the same adjustment reasons in each requisition line item. For instance, if a line item had two adjustments with the same reason (Expired: 20 and Expired: 30), this will be replaced by a single adjustment with the total (Expired: 50). We highly recommend that all implementations review their duplicate stock adjustments manually and determine how they should be merged prior to upgrading to 3.2.1. The default migration may not be valid for all the cases that can occur in real-world data.

Batch Requisition Approval: During work on OpenLMIS 3.2.1, further improvements to the Batch Approval screen were made, but the feature is still not officially supported. The UI screen is disabled. Implementations can override the code in their local customizations in order to use the screen. Further changes to the screen are expected in future releases before it is officially supported. See OLMIS-3182 for more info.

Backwards-Compatible Except As Noted

Unless noted here, all other changes to OpenLMIS 3.x are backwards-compatible. All changes to data or schemas include automated migrations from previous versions back to version 3.0.1. All new or altered functionality is listed in the sections below for New Features and Changes to Existing Functionality.

Upgrading from Older Versions

If you are upgrading to OpenLMIS 3.2.1 from OpenLMIS 3.0.x or 3.1.x, please review the 3.2.0 Release Notes for important compatibility information.

For information about upgrade paths from OpenLMIS 1 and 2 to version 3, see the 3.0.0 Release Notes.

Download or View on GitHub

OpenLMIS Reference Distribution 3.2.1

Known Bugs

Bug reports are collected in Jira for troubleshooting, analysis and resolution. See OpenLMIS 3.2.1 Bugs.

To report a bug, see Reporting Bugs.

New Features

OpenLMIS 3.2.1 contains these new features:

- Facility administration screens now support adding and editing facilities.
- User administration screens now provide filtering and more password reset options.
- Demo data is significantly expanded, including for use in contract tests and performance tests.
- Vaccine MVP features including Ideal Stock Amount (ISA) management, printing of physical inventory counts and additional work in Cold Chain Equipment (CCE) tracking (CCE features are released in a Beta version which is not included in the 3.2.1 release).
- Contributions from the Malawi implementation, including a new Extension Point for customizing Order Numbers and deleting previously skipped requisitions.

Changes to Existing Functionality

Version 3.2.1 contains changes that impact users of existing functionality. Please review these changes which may require informing end-users and/or updating your customizations/extensions:

- OLMIS-3233: Ability to delete previously skipped Requisitions.
- OLMIS-3076: DataIntegrityViolationException when trying to remove previous requisition / Average Period Consumption should not calculate using Emergency requisition data. This change updates the rules about when it is possible to delete older requisitions. It also changes how newer requisitions use past data to compute the Average Period Consumption.
- OLMIS-3246: Ability to hide special reasons from Total Losses and Adjustments. This feature provides a new configuration option so that administrators can hide selected reasons from end-users.

- OLMIS-3221 and OLMIS-3222: View Orders filtering by period start and end dates.
- OLMIS-2700: View Requisition enhancements. This includes new sort order controls and makes the Date Initiated visible in the table.
- OLMIS-3449: Explanation field on Non-Full Supply is no longer mandatory.

See all 3.2.1 issues tagged 'UIChange' in Jira.

API Changes

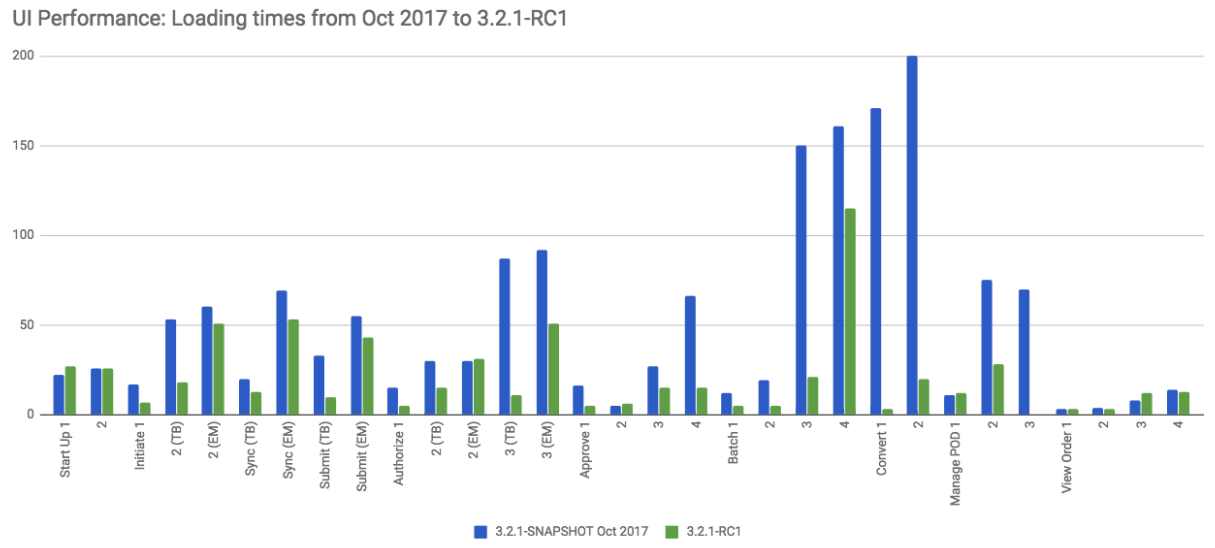
Some APIs have changes to their contracts and/or their request-response data structures. These changes impact developers and systems integrating with OpenLMIS:

- OLMIS-3254: Unrestrict GET operations on certain reference data resources. This makes certain information (EG, lists of all facilities and orderables) available for any user with a valid login token.
- OLMIS-3116: User DTO now returns home facility UUID instead of Facility object.
- OLMIS-3105: User DTO now returns UUIDs instead of codes for role assignments.
- OLMIS-3293: Paginate search facilityTypeApprovedProducts and made endpoint RESTful.
- OLMIS-2732: Stock Management Physical Inventory API was redesigned to be RESTful (during work on this ticket for print support).

Performance Improvements

Targeted performance improvements were made in the RESTful API services as well as in the UI application. The improvements were chosen based on testing using a new performance data set and by manually testing with simulated conditions (EG, network set to Slow 3G).

This chart shows a side-by-side comparison of the loading times for different actions in the UI in version 3.2.1 (green) compared to testing done in early October 2017 before improvements (blue).



These loading times are measured from the UI app with network set to Slow 3G and CPU throttled. The data was gathered manually by timing the application while running the new performance data set.

Top Areas Improved in 3.2.1:

- Convert to Order has dramatically improved loading times (now under 20 seconds): OLMIS-3318 and OLMIS-3320.
- Requisition Approve is significantly faster (now under 15 seconds): OLMIS-3346.
- Requisition Initiate is faster. OLMIS-3332 and OLMIS-3322.
- Requisition Submit and Authorize are also faster (improved by those same tickets).
- Batch Approve performs better scrolling through large numbers of products.

For more info about the data and results, see: https://openlmis.atlassian.net/wiki/spaces/OP/pages/116949318/Performance+Metrics

Test Coverage

OpenLMIS 3.2.1 is the first release using the new Release Candidate process. As part of this process, a full manual regression test cycle was conducted, and multiple release candidates were published to address critical bugs before releasing the final version 3.2.1.

Manual tests were conducted using a set of 110 Zephyr tests tracked in Jira. A total of 34 bugs were found during testing. The full set of 110 tests was executed on the first Release Candidate (RC1). With subsequent release candidates (RC2 and RC3), a smaller set of tests were re-executed based on which components were changed. In total, 34 bugs were found from all rounds of manual testing for 3.2.1. See a spreadsheet of all regression test executions for this release: 3.2.1-regression-tests.csv.

The automated tests (unit tests, integration tests, and contract tests) were 100% passing at the time of the 3.2.1 release. Automated test coverage is tracked in Sonar.

All Changes by Component

Version 3.2.1 of the Reference Distribution contains updated versions of the components listed below. The Reference Distribution bundles these component together using Docker to create a complete OpenLMIS instance. Each component has its own own public GitHub repository (source code) and DockerHub repository (release image). The Reference Distribution and components are versioned independently; for details see Versioning and Releasing.

Auth Service 3.1.1

Bug fixes added in a backwards-compatible manner:

- OLMIS-3119: Fixed issue with TOKEN_DURATION variable being ingored, which in reality was an issue with set up of the Spring context and autowiring not working as expected.
- OLMIS-3357: Reset email will not be sent when user is created or updated.

Source: Auth CHANGELOG

CCE Service 1.0.0-beta

This component is a **beta** of new Cold Chain Equipment functionality to support Vaccines in medical supply chains. This API service component has an accompanying beta CCE UI component.

For details, see the functional documentation: Cold Chain Equipment Management.

Warning: This is a beta component, and is not yet intended for production use. APIs and functionality are still subject to change until the official release.

Fulfillment Service 6.1.0

New functionality added in a backwards-compatible manner:

- OLMIS-3221: Added period start and end dates parameters to the order search endpoint.

Improvements added in a backwards-compatible manner:

- OLMIS-3112: Added OrderNumberGenerator extension point. Changed the default implementation to provide 8 character, base36 order numbers.

Source: Fulfillment CHANGELOG

Notification Service 3.0.4

Bug fixes, security and performance improvements (backwards-compatible):

- OLMIS-3394: Added notification request validator. From, to, subject and content fields are required, and if one of them will be empty the endpoint will return response with 400 status code and error message.

Source: Notification CHANGELOG

Reference Data Service 9.0.0

Breaking changes:

- OLMIS-3116: User DTO now returns home facility UUID instead of Facility object.
- OLMIS-3105: User DTO now returns UUIDs instead of codes for role assignments.
- OLMIS-3293: Paginate search facilityTypeApprovedProducts and made endpoint RESTful.

New functionality added in a backwards-compatible manner:

- OLMIS-2892: Added ideal stock amounts model.
- OLMIS-2966: Create User Rights for Managing Ideal Stock Amounts.
- OLMIS-3227: Added GET Ideal Stock Amounts endpoint with download csv functionality.
- OLMIS-3022: Refresh right assignments on role-based access control (RBAC) structural changes.
- OLMIS-3263: Added new ISA dto with links to nested objects.
- OLMIS-396: Added ISA upload endpoint.
- OLMIS-3200: Designed and added new demo data for EPI (Vaccines) program.
- OLMIS-3254: Un-restrict most GET APIs for most resources.
- OLMIS-3351: Added search by ids to /api/facilities endpoint.
- OLMIS-3512: Added code validation for supervisory node create and update endpoints.

Bug fixes, security and performance improvements, also backwards-compatible:

- OLMIS-2857: Refactored user search repository method to user database pagination and sorting.
- OLMIS-2913: add DIVO user and assign to Inventory Manager role for SN1 and SN2.
- OLMIS-3146: added PROGRAMS_MANAGE right and enforce it on CUD endpoints.
- OLMIS-3209: Fixed problem with parsing orderable DTO when it contains several program orderables.
- OLMIS-3290: Fixed searching Orderables by code and name.

- OLMIS-3291: Fixed searching RequisitionGroups by supervisoryNode.
- OLMIS-3346: Decreased number of database calls to retrieve Facility Type Approved Products.

Source: ReferenceData CHANGELOG

Reference UI 5.0.4

The Reference UI (https://github.com/OpenLMIS/openlmis-reference-ui/) is the web-based user interface for the OpenLMIS Reference Distribution. This user interface is a single page web application that is optimized for offline and low-bandwidth environments. The Reference UI is compiled together from module UI modules using Docker compose along with the OpenLMIS dev-ui. UI modules included in the Reference UI are:

auth-ui 6.0.0

New functionality:

- OLMIS-2956: Simplified login and authorization services by removing "user rights" functionality and moving to openlmis-referencedata-ui.

New functionality added in backwards-compatiable manner:

- OLMIS-3141: After user resets their password, they are redirected to the login screen.
- OLMIS-3283: Added a "Show password" option on password reset screen.

Bug fixes which are backwards-compatible:

- OLMIS-3140: Added loading icon on forgot password modal.

Improvements:

- Updated dev-ui version to 6.

See openlmis-auth-ui CHANGELOG

cce-ui 1.0.0-beta

Beta release of CCE UI. See CCE service component above for more info.

fulfillment-ui 5.1.0

New functionality added in a backwards-compatible manner:

- OLMIS-3222: Added period start and end dates parameters to the order view screen

Bug fixes:

- OLMIS-3159: Fixed facility select loosing state no POD manage page.
- OLMIS-3285: Fixed broken pagination on Manage Proofs of Delivery page.
- OLMIS-3540: Now Manage POD displays items with IN_ROUTE status.

Improvements:

- Updated dev-ui version to 6.

See openlmis-fulfillment-ui CHANGELOG

referencedata-ui 5.2.2

New features:

- OLMIS-3153: Added facilityOperatorsService for communicating with the facilityOperators endpoints
- Extended facilityService with the ability to save facility
- OLMIS-3154: Changed facility view to edit screen.
- OLMIS-3228: Create Download Current ISA Values page.
- OLMIS-2217: Added ability to send reset password email.
- OLMIS-396: Added upload functionality to manage ISA screen.

Improvements:

- OLMIS-2857: Added username filter to user list screen.
- OLMIS-3283: Added a "Show password" option on password reset screen.
- OLMIS-3296: Reworked facility-program select component to use cached rograms, minimal facilities and permission strings.
- Updated dev-ui version to 6.

Bug fixes:

- Added openlmis-offline as a dependency to the referencedata-program module.
- OLMIS-3291: Fixed incorrect state name.
- OLMIS-3499: Fixed changing username in title header.

See openlmis-referencedata-ui CHANGELOG

report-ui 5.0.4

Improvements:

- Updated dev-ui version to 6.

See openlmis-report-ui CHANGELOG

requisition-ui 5.2.0

Improvements:

- OLMIS-2956: Removed UserRightFactory from requisition-initiate module, and replaced with permissionService.
- OLMIS-3294: Added loading modal after the approval is finished.
- OLMIS-2700: Added date initiated column and sorting to the View Requisitions table. Removed date authorized and date approved.
- OLMIS-3181: Added front-end validation to the requisition batch approval screen.
- OLMIS-3233: Added ability to delete requisitions with "skipped" status.
- OLMIS-3246: Added 'show' field to reason assignments.
- OLMIS-3471: Explanation field on Non Full supply tab is no longer mandatory.

- OLMIS-3318: Added requisitions caching to the Convert to Order screen.
- Updated dev-ui version to 6.

Bug fixes:

- OLMIS-3151: Fixed automatically resolving mathematical error with adjustments.
- OLMIS-3255: Fixed auto-select the "Supplying facility" on Requisition Convert to Order.
- OLMIS-3296: Reworked facility-program select component to use cached programs, minimal facilities and permission strings.
- OLMIS-3322: Added storing initiated requisition in offline cache.

See openlmis-requisition-ui CHANGELOG

stockmanagement-ui 1.0.1

New functionality that are backwards-compatible:

- OLMIS-2732: Print submitted physical inventory.

Improvements:

- OLMIS-3246: Added support for hidden stock adjustment reasons.
- OLMIS-3296: Reworked facility-program select component to use cached rograms, minimal facilities and permission strings.
- Updated dev-ui version to 6.

See openlmis-ui-components CHANGELOG

ui-components 5.2.0

ui-components 5.2.0 contains significant new functionality including virtual table scrolling for improved performance of large tables, a new sort control, PouchDB support, improved Offline detection and much more.

New functionality added in a backwards-compatible manner:

- OLMIS-3182: Added openlmis-table-pane that implements high performance table rendering for large data tables.
- OLMIS-2655: Added sort control component.
- OLMIS-3462: Added debounce option for inputs.
- OLMIS-3199: Added PouchDB.

New functionality:

- Added modalStateProvider to ease modal state defining

Bug fixes:

- OLMIS-3248: Added missing message for number validation.
- OLMIS-3170: Fixed auto resize input controls.
- OLMIS-3500: Fixed a bug with background changing color when scrolling.

Improvements:

- OLMIS-3114: Improved table keyboard accessibility. Made table scroll if focused cell is off screen. Wrapped checkboxes in table cells automatically if they don't have a label.
- Modals now have backdrop and escape close actions disabled by default. Can by overridden by adding 'backdrop' and 'static' properties to the dialog definition.
- Extended stateTrackerService with the ability to override previous state parameters and pass state options.
- Updated dev-ui version to 6.
- OLMIS-3359: Improved the way offline is detected.

See openlmis-ui-components CHANGELOG

ui-layout:5.0.3

New features:

- OLMIS-2956: Added loadingService with $stateChangeStart interceptor

Improvements:

- OLMIS-3303: Added warning for users with Javascript disabled
- Updated dev-ui version to 6.

See openlmis-ui-layout CHANGELOG

Dev UI

The Dev UI developer tooling has advanced to v6.

Report Service 1.0.0

This new service is intended to provide reporting functionality for other components to use. It is a 1.0.0 release which is stable for production use, and it powers one built-in report: the Facility Assignment Configuration Errors report (OLMIS-2760).

Additional built-in reports in OpenLMIS 3.2.1 are still powered by their own services. In future releases, they may be migrated to a new version of this centralized report service.

Warning: Developers should take note that the design of this service will be changing with future releases. Developers and implementers are discouraged from using this 1.0.0 version to build additional reports.

Changes since Report Service 1.0.0-beta:

- OLMIS-3116: Change user home facility from Facility DTO to UUID

Requisition Service 5.1.0

Improvements:

- OLMIS-3544: Added sort to requisition search endpoint.
- OLMIS-3246: Added support for hidden stock adjustment reasons. Also added validations to ensure all special reasons configured for Requisition service to use are valid reasons.
- OLMIS-3233: Added ability to delete requisitions with Skipped status.

- OLMIS-3351: Improve performance of batch retrieveAll.

Bug fixes added in a backwards-compatible manner:

- OLMIS-3126: Fix unable to batch save when skip is disabled in Requisition Template.
- OLMIS-3215: Do not allow for status change (submit/authorize/approve) when period end after today.
- OLMIS-3076: Exclude emergency from previous requisitions, remove regular requisition only if it is newest.
- OLMIS-3320: Improved requisitions for convert endpoint performance.
- OLMIS-3404: Added validation for sending reasons in line item adjustments that are not present on available reason list in requisition.

Improve demo data:

- OLMIS-3202: Modified requisition template for EM program to match Malawi example columns.

Source: Requisition CHANGELOG

Stock Management 2.0.0

Contract breaking changes:

- OLMIS-2732: Print submitted physical inventory. During work on this ticket physical inventory API was redesigned to be RESTful.

New functionality that are backwards-compatible:

- OLMIS-3246: Add ability to configure hidden stock adjustment reasons. Updated demo data. Also impacts Requisition and UI.

Bug fixes, security and performance improvements, also backwards-compatible:

- OLMIS-3148: Added missing messages for error keys
- OLMIS-3346: Increase performance of POST /stockEvents endpoint by reducing db calls and use lazy-loading in the stock event process context. Also changed logic for notification of stockout to asynchronous.

Source: Stock Management CHANGELOG

Components with No Changes

Other tooling components have not changed, including: the logging service, the Consul-friendly distribution of nginx, the docker Postgres 9.6-postgis image, the docker rsyslog image, the docker scalyr image, and a library for shared Java code called service-util.

Contributions

Thanks to the Malawi implementation team who has contributed a number of pull requests to add functionality and customization in ways that have global shared benefit.

For a detailed list of contributors, see the Release Notes for OpenLMIS 3.2.0, 3.1.0 and 3.0.0.

Further Resources

Learn more about the OpenLMIS Community and how to get involved!

1.1.7 3.2.0 Release Notes - 1 September 2017

Status: Stable

3.2.0 is a stable release, and all users of OpenLMIS version 3 are encouraged to adopt it.

Release Notes

The OpenLMIS Community is excited to announce the **3.2.0 release** of OpenLMIS!

This release represents another major milestone in the version 3 series, which is the result of a software re-architecture that allows more functionality to be shared among the community of OpenLMIS users.

3.2.0 includes new features in **stock management**, new **administrative screens**, targeted **performance improvements** and a beta version of the **Cold Chain Equipment (CCE)** service. It also contains contributions in the form of pull requests from the **Malawi implementation**, a national implementation that is now live on OpenLMIS version 3.

3.2.0 represents the first milestone towards the Vaccines MVP feature set.

After 3.2.0, there are further planned milestone releases and patch releases that will add more features to support Vaccine/EPI programs and continue making OpenLMIS a full-featured electronic logistics management information system (LMIS). Please reference the Living Product Roadmap for the upcoming release priorities. Patch releases will continue to include bug fixes, performance improvements, and pull requests are welcomed.

Compatibility

Important: If you are upgrading to 3.2.0 and using your own database solution (i.e. Amazon RDS), and not the Postgres image in the Reference Distribution, please make sure you have the Postgres "uuid-ossp" extension installed. If you are using the Postgres image from the Reference Distribution, then this extension will be installed for you once you pull the latest image from DockerHub. For more information about this change, please see the Postgres section, and OLMIS-2681 under the Requisition Service section.

Important: 3.2.0 requires a data load script that must be run once in order to properly upgrade from an older version 3 to 3.2.0. To run this script, add `refresh-db` to your Spring profile. An example: `export spring_profiles_active="refresh-db"`. You only need to run it the first time you start the server after upgrading to 3.2.0. For more information about this change, please see OLMIS-2811 under the Reference Data Service section.

Important: 3.2.0 contains a data migration script that must be applied in order to upgrade from older version 3 to 3.2.0. This migration has its own GitHub repo and Docker image. See Adjustment Reason Migration. If you are upgrading from any previous version of 3 to 3.2.0, see the README file which has specific instructions to apply this migration. For background on this migration, see Connecting Stock and Requisition Services.

Important: Requisition Service now requires use of the Stock Management service. Data collected on requisition forms uses adjustment reasons from the Stock service and submits data to stock cards. Certain columns on the Requisition Template are now required. See Requisition Template Column Dependencies and Calculations as well as more details in the Requisition component below.

All other changes are backwards-compatible. Any changes to data or schemas include automated migrations from previous versions back to version 3.0.1. All new or altered functionality is listed in the sections below for New Features and Changes to Existing Functionality.

For background information on OpenLMIS version 3's micro-service architecture, extensions/customizations, and upgrade paths for OpenLMIS versions 1 and 2, see the 3.0.0 Release Notes.

Download or View on GitHub

OpenLMIS Reference Distribution 3.2.0

New Features

This is a new section to flag all the new features.

- **Stock Management**: is not an official release and added a notification and new support for recording VVM status.
- **Administrative Screens**: view supply lines, geogrphic zones, requisition groups and program settings.
- beta version of the new **Cold Chain Equipment (CCE)** service: which includes the support to upload a catalog of cold chain equipment, add equpiment inventory (from the catalog) to facilities, and manually update the functional status of that equipment. Review the wiki for details on the upcoming features.
- **Performance**: targeted improvements were made based on the first implementation's use and results. Improvements were made in server response times, which impacts load time, and memory utilization. In addition, new tooling was introduced to provide the ability to track performance improvements and bottlenecks.
- Reference data
- Report service is now a separate component (see Report component below)

Changes to Existing Functionality

Version 3.2.0 contains changes that impact users of existing functionality. Please review these changes which may require informing end-users and/or updating your customizations/extensions:

- Requisition Service now uses Stock Management to handle adjustment reasons and to store stock data in stock cards. This change does not alter end-user functionality in Requisitions, but it does allow users with Stock Management rights to begin viewing stock cards with data populated from requisitions. This change also requires a data migration script to upgrade older version 3 systems to 3.2.0. (See Requisition component below.)

API Changes

Some APIs have changes to their contracts and/or their request-response data structures. These changes impact developers and systems integrating with OpenLMIS:

- Auth Service uses Authorization header instead of access_token (see Auth OLMIS-2871 below)
- Fulfillment Service and Requisition Service changed some dates from ZonedDateTime to LocalDate (see OLMIS-2898 below)
- ReferenceData contains changes to Facility search and Geographic Search APIs (see component below)
- Requisition Service now requires use of the Stock Management service and connects to Stock service to handle adjustment reasons and store data on stock cards (see Requisition component)
- Configuration settings endpoints (/api/settings) are no longer available; use environment variables to configure the application (see OLMIS-2612 below)
- postgres database now requires one additional extension: uuid. It is already included in the postgres component (see postgres component below), but those hosting on Amazon AWS RDS will need to add the extension.

All Changes by Component

Version 3.2.0 of the Reference Distribution contains updated versions of the components listed below. The Reference Distribution bundles these component together using Docker to create a complete OpenLMIS instance. Each component has its own own public GitHub repository (source code) and DockerHub repository (release image). The Reference Distribution and components are versioned independently; for details see Versioning and Releasing.

Auth Service 3.1.0

Improvements which are backwards-compatible:

- OLMIS-1498: The service will now fetch list of available services from consul, and update OAuth2 resources dynamically when a new service is registered or de-registered. Those tokens are no longer hard-coded.
- OLMIS-2866: The service will no longer used self-contained user roles (USER, ADMIN), and depend solely on referencedata's roles for user management.
- OLMIS-2871: The service now uses an Authorization header instead of an access_token request parameter when communicating with other services.

Source: Auth CHANGELOG

CCE Service 1.0.0-beta

This component is a **beta** of new Cold Chain Equipment functionality to support Vaccines in medical supply chains. This API service component has an accompanying beta CCE UI component.

CCE 1.0.0-beta includes many new features:

- Create and update a cold chain equipment catalog
- Add equipment inventory to facilities
- Update the functional status of equipment inventory

For details, see the functional documentation: Cold Chain Equipment Management

Warning: This is a beta component, and is not yet intended for production use. APIs and functionality are still subject to change until the official release.

Fulfillment Service 6.0.0

Contract breaking changes:

- OLMIS-2898: Changed POD receivedDate from ZonedDateTime to LocalDate.

New functionality added in a backwards-compatible manner:

- OLMIS-2724: Added an endpoint for retrieving all the available, distinct requesting facilities.

Bug fixes and improvements (backwards-compatible):

- OLMIS-2871: The service now uses an Authorization header instead of an access_token request parameter when communicating with other services.
- OLMIS-3059: The search orders endpoint now sorts the orders by created date property (most recent first).

Source: Fulfillment CHANGELOG

nginx v4

Improves stability and reliability of the application when individual services stop and start in their lifecycle. Also performance is improved by reducing latency under load between nginx and Services through configuration tuning.

- OLMIS-2840: Allow services to stop and start without crashing consul-template.
- OLMIS-2957: Reduce nginx latency.

Notification Service 3.1.0

Bug fixes, security and performance improvements (backwards-compatible):

- OLMIS-2871: The service now uses an Authorization header instead of an access_token request parameter when communicating with other services.

Source: Notification CHANGELOG

Postgres 9.6-postgis

The postgres image in OpenLMIS 3.2.0 has changed slightly to include the **uuid-ossp** extension, in order to randomly generate UUIDs in SQL (this new requirement was introduced in OLMIS-2681). Because the change is minor and does not change the version of Postgres, we have released an updated image with the same version number (9.6-postgis). When using the 3.2.0 release, as long as you use `docker-compose pull`, it will pull the correct version of the postgres image.

Reference Data Service 8.0.0

Breaking changes:

- OLMIS-2709: Facility search now returns smaller objects.
- OLMIS-2698: Geographic Zone search endpoint now is paginated and accepts POST requests, also has new parameters: name and code.

New functionality added in a backwards-compatible manner:

- OLMIS-2609: Created rights to manage CCE and assigned to system administrator.
- OLMIS-2610: Added CCE Inventory View/Edit rights, added demo data for those rights.
- OLMIS-2696: Added search requisition groups endpoint.
- OLMIS-2780: Added endpoint for getting all facilities with minimal representation.
- Introduced JaVers to all domain entities. Also each domain entity has endpoint to get the audit information.
- OLMIS-3023: Added enableDatePhysicalStockCountCompleted field to program settings.
- OLMIS-2619: Added CCE Manager role and assigned CCE Manager and Inventory Manager roles to new user ccemanager.
- OLMIS-2811: Added API endpoint for user's permission strings.
- OLMIS-2885: Added ETag support for programs and facilities endpoints.

Bug fixes, security and performance improvements, also backwards-compatible:

- OLMIS-2871: The service now uses an Authorization header instead of an access_token request parameter when communicating with other services.

- OLMIS-2534: Fixed potential huge performance issue.
- OLMIS-2716: Set productCode field in Orderable as unique.

Source: ReferenceData CHANGELOG

Reference UI 5.0.3

The Reference UI bundles the following UI components together using Docker images specified in its compose file.

auth-ui 5.0.3

New functionality added in backwards-compatiable manner:

- OLMIS-3085: Added standard login and logout events.

Bug fixes and security updates:

- OLMIS-3124: Removed openlmis-download directive and moved it to openlmis-ui-components
- MW-348: Added loading modal while logging in.
- OLMIS-2871: Made the component use an Authorization header instead of an access_token request parameter when calls to the backend are made.
- OLMIS-2867: Added message when user tries to log in while offline.

See openlmis-auth-ui CHANGELOG

cce-ui 1.0.0-beta

Beta release of CCE UI. See CCE service component below for more info.

fulfillment-ui 5.0.3

Bug fixes:

- OLMIS-2837: Fixed filtering on the manage POD page.
- OLMIS-2724: Fixed broken requesting facility filter select on Order View.

See openlmis-fulfillment-ui CHANGELOG

referencedata-ui 5.2.1

Improvements:

- OLMIS-2780: User form now uses minimal facilities endpoint.

New functionality added in a backwards-compatible manner:

- OLMIS-3085: Made minimal facility list download and cache when user logs in.
- OLMIS-2696: Added requisition group administration screen.
- OLMIS-2698: Added geographic zone administration screens.
- OLMIS-2853: Added view Supply Lines screen.

- OLMIS-2600: Added view Program Settings screen.

Bug fixes

- OLMIS-2905: User with only POD_MANAGE or ORDERS_MANAGE can now access 'View Orders' page.
- OLMIS-2714: Fixed loading modal closing too soon after saving user.

See openlmis-referencedata-ui CHANGELOG

report-ui 5.0.3

Big fixes:

- OLMIS-2911: Added http method and body to jasper template paramter

See openlmis-report-ui CHANGELOG

requisition-ui 5.1.1

- OLMIS-2797: Updated product-grid error messages to use openlmis-invalid.

New functionality that are not backwards-compatible:

- OLMIS-2833: Add date field to Requisition form

Date physical stock count completed is required for submit and authorize requisition. - OLMIS-3025: Introduced frontend batch-approval functionality. - OLMIS-3023: Added configurable physical stock date field to program settings. - OLMIS-2694: Change Requisition adjustment reasons to come from Requisition object. OpenLMIS Stock Management UI is now connected to Requisition UI.

Improvements:

- OLMIS-2969: Requisitions show saving indicator only when requisition is editable.

Bug fixes:

- OLMIS-2800: Skip column will not be shown in submitted status when user has no authorize right.
- OLMIS-2801: Disabled the 'Add Product' button in the non-full supply screen for users without rights to edit the requisition. Right checks for create/initialize permissions were also fixed.
- OLMIS-2906: "Outdated offline form" error is not appearing in a product grid when requisition is up to date.
- OLMIS-3017: Fixed problem with outdated status messages after Authorize action.

See openlmis-requisition-ui CHANGELOG

stockmanagement-ui 1.0.0

First release of Stock Management UI. See Stock Management service component below for more info.

ui-components 5.1.1

New functionality added in a backwards-compatible manner:

- OLMIS-2978: Made sticky table element animation more performant.
- OLMIS-2573: Re-worked table form error messages to not have multiple focusable elements.

- OLMIS-1693: Added openlmis-invalid and error message documentation.
- OLMIS-249: Datepicker element now allows translating day and month names.
- OLMIS-2817: Added new file input directive.
- OLMIS-3001: Added external url run block, that allows opening external urls.

Bug fixes:

- OLMIS-3088: Re-implemented tab error icon.
- OLMIS-3036: Cleaned up and formalized input-group error message implementation.
- OLMIS-3042: Updated openlmis-invalid and openlmis-popover element compilation to fix popovers from instantly closing.
- OLMIS-2806: Fixed stock adjustment reasons display order not being respected in the UI.

See openlmis-ui-components CHANGELOG

ui-layout:5.0.2

New features:

- OLMIS-2543: Added interceptor for displaying server errors

See openlmis-ui-layout CHANGELOG

Dev UI

The Dev UI developer tooling has advanced to v5.

Report Service 1.0.0

This new service is intended to provide reporting functionality for other components to use. It is a 1.0.0 release which is stable for production use, and it powers one built-in report (the Facility Assignment Configuration Errors report).

Warning: Developers should take note that its design will be changing with future releases. Developers and implementers are discouraged from using this 1.0.0 version to build additional reports.

Current report functionality:

- OLMIS-2760: Facility Assignment Configuration Errors

Additional built-in reports in OpenLMIS 3.2.0 are still powered by their own services. In future releases, they may be migrated to a new version of this centralized report service.

Requisition Service 5.0.0

Contract breaking changes:

- OLMIS-2612: Configuration settings endpoints (/api/settings) are no longer available. Use environment variables to configure the application.
- MW-365: Requisition search endpoints: requisitionsForApproval and requisitionsForConvert will now return smaller basic dtos.

- OLMIS-2833 and OLMIS-3023: Added date physical stock count completed to Requisition; and feature can be turned on and off in Program Settings
- OLMIS-2671: Stock Management service is now required by Requisition
- OLMIS-2694: Changed Requisition adjustment reasons to come from Stock Service
- OLMIS-2898: Requisition search endpoint takes from/to parameters as dates without time part.
- OLMIS-2830: As of this version, Requisition now uses Stock Management as the source for adjustment reasons, moreover it stores snapshots of these available reasons during initiation. **Important**: in order to migrate from older versions, running this migration is required: https://github.com/OpenLMIS/openlmis-adjustment-reason-migration

New functionality added in a backwards-compatible manner:

- OLMIS-2709: Changed ReferenceData facility service search endpoint to use smaller dto.
- The /requisitions/requisitionsForConvert endpoint accepts several sortBy parameters. Data returned by the endpoint will be sorted by those parameters in order of occurrence. By defaults data will be sorted by emergency flag and program name.
- OLMIS-2928: Introduced new batch endpoints, that allow retrieval and approval of several requisitions at once. This also refactored the error handling.

Bug fixes added in a backwards-compatible manner:

- OLMIS-2788: Fixed print requisition.
- OLMIS-2747: Fixed bug preventing user from being able to re-initiate a requisition after being removed, when there's already a requisition for next period.
- OLMIS-2871: The service now uses an Authorization header instead of an access_token request parameter when communicating with other services.
- OLMIS-2534: Fixed potential huge performance issue. The javers log initializer will not retrieve all domain objects at once if a repository implemenets PagingAndSortingRepository
- OLMIS-3008: Add correct error message when trying to convert requisition to an order with approved quantity disabled in the the requisition template.
- OLMIS-2908: Added a unique partial index on requisitions, which prevents creation of requisitions which have the same facility, program and processing period while being a non-emergency requsition. This is now enforced by the database, not only the application logic.
- OLMIS-3019: Removed clearance of beginning balance and price per pack fields from skipped line items while authorizing.
- OLMIS-2911: Added HTTP method parameter to jasper template parameter object.
- OLMIS-2681: Added profiling to requisition search endpoint, also it is using db pagination now.

Source: Requisition CHANGELOG

Stock Management 1.0.0

This is the **first official release** of the new Stock Management service. Its beta version was previously released in Reference Distribution 3.1.0. Since then, the major improvements are:

- OLMIS-2710: Configure VVM use per product
- OLMIS-2654 and OLMIS-2663: Record VVM status with physical stock count and adjustments
- OLMIS-2711: Change Physical Inventory to include reasons for discrepancy

- OLMIS-2834: Requisition form info gets pushed into Stock cards (see more in Requisition component)
- *plus lots of technical work including Flyway migrations, RAML, tests, validations, translations, documentation, and demo data.*

Watch a video demo of the Stock Management functionality: https://www.youtube.com/watch?v=QMcXX3tUTHE (English) or https://www.youtube.com/watch?v=G8BK0izxbnQ (French)

Now that this is an official release, the Stock service is considered stable for production use. Future changes to functionality or APIs will be tracked and documented.

For a list of all commits since 1.0.0-beta, see GitHub commits

Components with No Changes

Other tooling components have not changed, including: the logging service and a library for shared Java code called service-util.

Contributions

Many organizations and individuals around the world have contributed to OpenLMIS version 3 by serving on committees, bringing the community together, and of course writing code and documentation. Below is a list of those who contributed code or documentation into the GitHub repos. If anyone who contributed in GitHub is missing, please contact the Community Manager.

Team Parrot: Paweł Gesek, Paweł Albecki, Nikodem Graczewski, Mateusz Kwiatkowski, Joanna Bebak, Paweł Nawrocki

Team ILL: Chongsun Ahn, Brandon Bowersox-Johnson, Sam Im, Mary Jo Kochendorfer, Ben Leibert, Nick Reid, Josh Zamor

A special thanks to the implementers working in Malawi who contributed features and improvements: Sebastian Brudzinski, Weronika Ciecierska, Łukasz Lewczynski, Klaudia Pałkowska, Ben Leibert, Christine Lenihan.

Since version 3.1.2, we have received 40 pull requests from outside implementers and contributors.

Special thanks to community members: Kaleb Brownlow, Lindabeth Doby, Tenly Snow, Jake Watson, Ashraf Islam, Parambir Gill, and all who attended the Product Committee, Technical Committee and Governance Committee meetings, and the many funders, supporters, implementors, partners, and those working around the world to make medical supply chains work for all people.

Further Resources

View all JIRA Tickets in 3.2.0.

Learn more about the OpenLMIS Community and how to get involved!

For older Release Notes before 3.2.0, see Releases in the OpenLMIS wiki.

For more about OpenLMIS releasing and versioning, see Versioning and Releasing.

1.2 Architecture

As of OpenLMIS v3, the architecture has transitioned to (micro) services fulfilling RESTful (HTTP) API requests. A modularized Reference UI application runs in a browser and uses those APIs to expose functionality to end users.

Other systems and mobile apps may also use the APIs to integrate and provide functionality. A Reporting and Analytics Platform uses a data warehouse strategy to offer visualizations of OpenLMIS data.

Extension mechanisms allow for components of the architecture to be customized without the need for the community to fork the code base:

- UI modules give flexibility in creating new user experiences or changing existing ones
- Extension Points & Modules - allows Service/API functionality to be modified
- Extra Data - allows for extensions to store data with existing components
- Reporting and Analytics Platform - allows for robust reporting solutions to be developed to meet implementation needs

Combined these components allow the OpenLMIS community to customize and contribute to a shared LMIS.

1.2.1 New Service Guidelines

OpenLMIS' Service architecture is centered around the concept of Bounded Contexts. In this pattern we identify Service's by grouping similar *things* (noun) into a Service, and define a clear boundary between that Service and others. Where to draw this line, and decide when to create a new Service or when to contribute to/extend an existing Service can sometimes be difficult to judge.

A quick set of guidelines for a OpenLMIS Service:

- A Service owns its data. For example the Requisition Service owns all the data that pertains to a Requisition and moving it through the workflow. It depends on information to help it along: facilities, programs, user's, etc. While these *things* are needed for a Requisition, they aren't inherently a Requisition's *things*. The Requisition service owns Requisition *things*: Requisitions and their Line Items, Requisition Templates, etc. It coordinates with other OpenLMIS Service's to obtain references of those other *things* it needs, that it doesn't own.
- A Service owns transactions. Operations on a Service's *things* almost always occur within a transaction. We read the state of a Requisition or write new state about that Requisition. Other Service's may become involved, however the transaction as it appears to the User is owned by the original Service.
- Service's backing data stores (usually relational databases) do not know about one-another. Only Service's know about other Services. Because of this it's the responsibility of the Services for maintaining referential integrity, as Foreign Key's can't cross Services's databases.

When considering creating a new Service, consider if that Service really owns its own *things*, and should be implemented as an OpenLMIS Service, or if instead the functionality needed is a re-use of existing *things* in a new way, in which case a contribution/extension should be made to an existing OpenLMIS Service. OpenLMIS does not follow Serverless architecture at this time.

1.2.2 Docker

Docker Engine and Docker Compose is utilized throughout the tech stack to provide consistent builds, quicken environment setup and ensure that there are clean boundaries between components. Each deployable component is versioned and published as a Docker Image to the public Docker Hub. From this repository of ready-to-run images on Docker Hub anyone may pull the image down to run the component.

Development environments are typically started by running a single Service or UI module's development docker compose. Using docker compose allows the component's author to specify the tooling and test runtime (e.g. PostgreSQL) that's needed to compile, test and build and package the production docker image that all implementation's are intended to use.

After a production docker image is produced, docker compose is used once again in the Reference Distribution to combine the desired deployment images with the needed configuration to produce an OpenLMIS deployment.

1.3 Components

OpenLMIS v3 uses a micro-services *Architecture* with different services each providing different APIs.

Each component below has its own Git repository, API docs and ERD. Many services below also have a corresponding UI component (e.g. Auth UI, Requisition UI). The Reference UI builds all of these UI components together into one web application.

1.3.1 Logging into the Live Documentation

The live documentation links below connect directly to our API Console docs on our CI server. To use the API you'll first need to get an access token from the Auth service, and then you'll need to give that token when using one of the RESTful operations.

Obtaining an access token:

1. Go to the Auth service's POST /api/oauth/token
2. Click `Try it` in the top right of the tab
3. In the Authentication section, enter username `user-client` and password `changeme`
4. In the Query Parameters section, enter username `administrator` and password `password`
5. Click `Authorize` under `password`
6. Enter the username `administrator` and password `password`
7. Click `Post`
8. In the `Response` box, copy the UUID value of the `access_token`. e.g. `"access_token": "a93bcab7-aaf5-43fe-9301-76c526698898"` copy `a93bcab7-aaf5-43fe-9301-76c526698898` to use later
9. Use the Authorization Token you just copied with every request.
    - In the live documentation using `Try It`, type `bearer` followed by the `access_token` you copied earlier into the `Authorization` header. e.g. `bearer a93bcab7-aaf5-43fe-9301-76c526698898`
    - Alternatively, in any other HTTP request tool (e.g. Postman) you may append it in the query parameters using the `access_token` field. e.g. `GET https://test.openlmis.org/api/facilities?access_token=a93bcab7-aaf5-43fe-9301-76c526698898`

1.3.2 Auth Service

Auth Service provides RESTful API endpoints for Authentication and Authorization. It holds user security credentials, handles password resets, and also manages API keys. It uses OAuth2. The Auth Service works with the Reference Data service to handle role-based access controls. (See the Auth Service README for details.)

- Auth Service GitHub repo
- Auth Service README
- Auth Service Design
- Auth Service ERD
- Live Documentation for Auth API
- Static Documentation for Auth API

1.3.3 CCE Service

The Cold Chain Equipment (CCE) Service provides RESTful API endpoints for managing a CCE catalog, inventory (tracking equipment at locations) and functional status. The catalog can use the WHO PQS.

- CCE Service GitHub repo
- CCE Service README
- CCE ERD
- Live Documentation for CCE API
- Static Documentation for CCE API

1.3.4 Fulfillment Service

Fulfillment Service provides RESTful API endpoints for orders, shipments, and proofs of delivery. It supports fulfillment within OpenLMIS as well as external fulfillment using external ERP warehouse systems.

- Fulfillment Service GitHub repo
- Fulfillment Service README
- Fulfillment ERD
- Live Documentation for Fulfillment API
- Static Documentation for Fulfillment API

1.3.5 HAPI FHIR Service

The HAPI FHIR Service provides RESTful API endpoints for FHIR locations. It supports keeping OpenLMIS facility data in sync with external facility registries through FHIR.

- HAPI FHIR Service GitHub repo
- HAPI FHIR Service README
- Static Documentation for HAPI FHIR API

1.3.6 Notification Service

The Notification Service provides RESTful API endpoints that allow other OpenLMIS services to send email notifications to users. The Notification Service does not provide a web UI.

- Notification Service GitHub repo
- Notification Service README
- Notification ERD
- Live Documentation for Notification API
- Static Documentation for Notification API

1.3.7 Reference Data Service

The Reference Data Service provides RESTful API endpoints that provide master lists of reference data including users, facilities, programs, products, schedules, and more. Most other OpenLMIS services depend on Reference Data Service. Many of these master lists can be loaded into OpenLMIS in bulk using the Reference Data Seed Tool or can be added and edited individually using the Reference Data Service APIs.

- Reference Data Service GitHub repo
- Reference Data Service README
- Reference Data ERD
- Live Documentation for Reference Data API
- Static Documentation for Reference Data API

1.3.8 Reference UI

The OpenLMIS Reference UI is a single page application that is compiled from multiple UI repositories. The Reference UI is similar to the OpenLMIS-Ref-Distro, in that it's an example deployment for implementers to use.

Learn about the Reference UI:

- OpenLMIS UI Overview describes the UI architecture and tooling
- UI Styleguide shows examples and best practices for many re-usable components
- Dev UI documents the build process and commands used by all UI components

Coding and Customizing the UI:

- UI Extension Guide
- UI *Conventions*
- Javascript Documentation

UI Repositories:

- Reference UI puts all the UI repositories into one single page application (Reference UI GitHub repo)
- Dev UI provides the build tools and commands. All other UI repositories use these build tools by including Dev UI as a base image in docker-compose. (Dev UI GitHub repo)
- UI Components is where OpenLMIS reusable components are defined along with base CSS styles (UI Components GitHub repo)
- Auth UI connects the OpenLMIS UI to the OpenLMIS Auth Service and handles all authentication details so other UI repositories don't have to (Auth UI GitHub repo)
- UI Layout defines UI layouts and page architecture used in the OpenLMIS UI (UI Layout GitHub repo)
- Reference Data UI adds administration screens for objects defined in the OpenLMIS Reference Data Service (Reference Data UI GitHub repo)
- Stock Management UI adds screens to interact with the OpenLMIS Stock Management Service (Stock Management UI GitHub repo)
- Fulfillment UI adds screens to connect to the OpenLMIS Fulfillment Service (Fulfillment UI GitHub repo)
- CCE UI adds screens for the OpenLMIS CCE Service. (CCE UI GitHub repo)
- Requisition UI adds screens to support the OpenLMIS Requisition Service (Requisition UI GitHub repo)

- Report UI adds screens to interact with OpenLMIS Report Service (Report UI GitHub repo)

1.3.9 Report Service

The Report Service provides RESTful API endpoints for generating printed / banded reports. It owns report storage, generation (including in PDF format), and seeding rights that users may be given.

- Report Service GitHub repo
- Report Service README
- Report ERD
- Live Documentation for Report API
- Static Documentation for Report API

1.3.10 Requisition Service

The Requisition Service provides RESTful API endpoints for a robust requisition workflow used in pull-based supply chains for requesting more stock on a schedule through an administrative hierarchy. Requisitions are initiated, filled out, submitted, and approved based on configuration. Requisition Templates control what information is collected on the Requisition form for different programs and facilities.

- Requisition Service GitHub repo
- Requisition Service README
- Requisition ERD
- Live Documentation for Requisition API
- Static Documentation for Requisition API

1.3.11 Stock Management Service

The Stock Management Service provides RESTful API endpoints for creating electronic stock cards and recording stock transactions over time.

- Stock Management Service GitHub repo
- Stock Management Service README
- Stock Management ERD
- Live Documentation for Stock Management API
- Static Documentation for Stock Management API

1.3.12 Reporting and Analytics Platform

OpenLMIS includes a reporting and analytics platform that extracts the data from each microservice, streams it to a data warehouse and provides a scalable reporting and dashboard interface. This reporting platform is made of multiple open source components, Apache Nifi, Apache Kafka, Druid and Apache SuperSet. This section provides an overview of each of the components of the reporting and analytics platform.

Nifi

NiFi is used for pulling data from OpenLMIS's APIs, merging data from the APIs into a single schema, and transforming the data into a format that's easy to query in Druid. Currently, NiFi blends data from the stockCardSummaries API and the referenceData API. It splits stock cards into line items and merges reference data with those line items, to have a single schema where stock card transactions (line items) contain detailed reference data like facility name, commodity type name, etc. instead of the reference data ids that natively live on the transaction in the stock management module. NiFi functions like an assembly line, where data moves from "processor" to processor throughout the "flow file."

Kafka

Kafka is used for stream processing and passing the data from NiFi to Druid. It works on a publish-subscribe model, similar to how message queues in an enterprise messaging systems work. Kafka is run on a cluster on one or more servers. A Kafka cluster stores streams of "records" in categories called "topics." A record consists of three parts: a key, a value, and a timestamp. A Kafka topic receives the transformed transaction from NiFi and publishes it to the Druid "supervisor." The Druid supervisor is always listening for updates from Kafka, and indexes the data immediately.

Druid

Druid is a distributed column-oriented OLAP database that the reporting stack uses for data storage and querying. Druid is purpose-built for querying streaming data sets at scale. Each set of data is called a "data source." JSON is the default language used for querying in Druid and is what the DISC indicators use. Druid also includes support for SQL using Apache Calcite, although this is not yet something we've explored. You can find documentation on querying in Druid using JSON here.

Superset

Superset is the visualization layer of the reporting stack and is used to create self-service dashboards on the data in Druid. It's very closely integrated with Druid, and will detect the schema for each data source and the data therein. "Dimensions" are akin to columns within a relational database, and "metrics" are calculations performed on those dimensions - e.g. count distinct, sum, min, max. Typically "metrics" are written off of numeric dimensions, with the exception of count distinct. Superset is the UI in which we write JSON queries for Druid to calculate metrics that are more sophisticated than the basic types outlined above.

Slices are individual visualizations and can be listed by clicking on the Charts tab along the top. Each slice has a visualization type, a data source, and one or more metrics and dimensions that you want to display. Superset supports the development of custom visualization types if it's not included in the default list provided by Apache.

A dashboard is an assembly of slices onto a single page. Filters can be applied at the dashboard-level, and filter all slices sharing the filter's data source to the specified dimension. Filters can also be used to manipulate date ranges. With proper security (more information below), users can save custom private or public versions of dashboards, and drill into a particular slice to modify it and construct an ad hoc visualization.

Security is handled via User Roles and Users. A User is a distinct login with a password, and is tied to an email address. There can only be one User per email address. A User Role is the list of actions that a User can do in Superset. Superset contains three User Roles by default, but they can be customized by duplicating the defaults and adding or removing permissions.

- Gamma - a view-only user who can save private views of dashboards and slices
- Alpha - a power user who is able to view all data sources, and create public dashboards and slices
- Admin - administrator with all access

1.4 Contributing

OpenLMIS is an open source community which appreciates the work of its contributors. Through contribution we're able to build a knowledgable community and make a wider impact than we would apart.

Contributing takes work so these guides aim to make that work clear and manageable:

1.4.1 Contributing to OpenLMIS

By contributing to OpenLMIS, you can help bring life-saving medicines to low- and middle-income countries. The OpenLMIS community welcomes open source contributions. Before you get started, take a moment to review this Contribution Guide, get to know the community and join in on the developer forum.

The sections below describe all kinds of contributions, from bug reports to contributing code and translations.

Reporting Bugs

The OpenLMIS community uses JIRA for tracking bugs. All bugs must be submitted to the OLMIS project to be reviewed or worked on. This system helps track current and historical bugs, what work has been done, and so on. Reporting a bug with this tool is the best way to get the bug fixed quickly and correctly.

Before you report a bug

- Search to see if the same bug or a similar one has already been reported. If one already exists, it saves you time in reporting it again and the community from investigating it twice. You can add comments or explain what you are experiencing or advocate for making this bug a high priority to fix quickly.
- If the bug exists but has been closed, check to see which version of OpenLMIS it was fixed on (the Fix Version in JIRA) and which version you are using. If it is fixed in a newer version, you may want to upgrade. If you cannot upgrade, you may need to ask on the technical forums.
- If the bug does not appear to be fixed, you can add a comment to ask to re-open the bug report or file a new one.

Reporting a new bug

Fixing bugs is a time-intensive process. To speed things along and assist in fixing the bug, it greatly helps to send in a complete and detailed bug report. These steps can help that along:

1. First, make sure you search for the bug in the current OpenLMIS backlog! It takes a lot of work to report and investigate bug reports, so please do this first (as described in the section Before You Report a Bug above).
2. Create a bug in the OpenLMIS Jira Project. Include the following information in the ticket:
   (a) *Type*: Select "bug"
   (b) *Status*: Leave as "ROADMAP". The OpenLMIS team will update the status to "TO DO" once the ticket is ready for work and reproduced.
   (c) *Description*: Write a clear and concise explanation of what you entered and what you saw, as well as what you thought you should see from OpenLMIS. Include the detailed steps, such as the Steps in the example below, that someone unfamiliar with the bug can use to recreate it. Make sure this bug occurs more than once, perhaps on a different personal computer or web browsers. Indicate the web browser (e.g. Firefox), version (e.g. v48), OpenLMIS version, as well as any custom modifications made. Include any time sensitivities or information of impact to support the team in prioritizing the bug.

(d) *Priority*: Indicate the priority level based on the guidence below in the Prioritizing Bugs section. The priority may be updated later by the Product Manager upon grooming and scheduling for work.

(e) *Affects Version/s*: Indicate what version of the reference distribution the bug was found in.

(f) *Component*: If you know which service is impacted by the bug, please include. If not, leave it blank.

(g) *Attachments*: Attach any relevant screen shots, videos or documents that will help the team understand and reproduce the bug.

3. If applicable, include any error message text, a screenshot, stack trace, or logging output in the *Description* or *Attachments*.
4. If possible and relevant, a sample or view of the database - though don't post sensitive information in public.

Once the bug is submitted, the OpenLMIS team will review the bugs prior to the next sprint cycle. Bugs will be prioritized and scheduled for work based on priority, resources, and implementation needs. Follow the ticket in Jira for updates on status and completion. Each release includes a list of bugs fixed.

Prioritizing Bugs

Each bug submission should include an initial prioritization form the reporter. Please follow the guidelines below for the initial prioritization.

- **Blocker**: Cannot execute function (cannot click button, button does not exist, cannot complete action when button is clicked). Cannot complete expected action (does not match expected results for the test case). No error message when there is an error. OpenLMIS will not release with this bug.
- **Critical**: Error message is unactionable by the user, and user cannot complete next action (500 server error message). Search results provided do not match expected results based on data. Poor UI performance or accessibility (user cannot tab to column or use keyboard to complete action). OpenLMIS should not release with this bug.
- **Major**: Performance related (slow response time). Major asthetic issue (See UI Styleguide for reference). Incorrect filtering, but doesn't block users from completing tasks and executing functionality. Wrong user error message (user does not know how to proceed based on the error message provided).
- **Minor**: Aesthetics (spacing is wrong, alignment is wrong, see UI Styleguide). Message key is wrong. Console errors. Service giving the wrong error between services.
- **Trivial**: Anything else.

When the bug is groomed and scheduled for work, the Product Manager will set the final priority level. See Backlog Grooming for details on the scheduling of work.

Ticket exemplars

The bugs listed are examples of bugs for their priorities. When completing exploratory testing and bugs are found, consider these bugs as references for how to record and prioritize bugs.

Blocker:

- OLMIS-3983 - Cannot access offline requisition
- OLMIS-3509 - No error message when create user fails
- OLMIS-3501 - I can select a program that is not supported by My Facility on the Create/Initiate page
- OLMIS-2980 - Broken translations for Stock Management Adjustments page titles
- OLMIS-3508 - Batch approval sticky rows do not respond to scrolling

Critical:

- OLMIS-4076 - It's possible to submit a requisition twice: duplicate status changes
- OLMIS-3500 - The black background is only the height of the window

Minor:

- OLMIS-3299 - Insufficient error message when submitting physical inventory with a future date
- OLMIS-2792 - Cannot add Issue for Essential Meds - demo data issue
- OLMIS-3987 - Long text not wrapped properly in modals
- OLMIS-3746 - Source Comments input field not displayed correctly on Firefox

Example Bug Report

```
Requisition is not being saved
OpenLMIS v3.0, Postgres 9.4, Firefox v48, Windows 10

When attempting to save my in-progress Requisition for the Essential Medicines␣
↪program for the reporting period of Jan 2017,
I get an error at the bottom of the screen that says "Whoops something went wrong".

Steps:

1. log in

2. go to Requistions->Create/Authorize

3. Select My Facility (Facility F3020A - Steinbach Hospital)

4. Select Essential Medicines Program

5. Select Regular type

6. Click Create for the Jan 2017 period

7. Fill in some basic requested items, or not, it makes no difference in the error

8. Click the Save button in the bottom of the screen

9. See the error in red at the bottom. The error message is "Whoops something went␣
↪wrong".

I expected this to save my Requisition, regardless of completion, so that I may␣
↪resume it later.

Please see attached screenshots and database snapshot.
```

Contributing Code

The OpenLMIS community welcomes code contributions and we encourage you to implement a new feature. Review the following process and guidelines for contributing new features or modification to existing functionality.

Coordinating with the Global Community

In reviewing your proposed contribution, the community promotes features that meet the broad needs of many countries for inclusion in the global codebase. We want to ensure that changes to the shared, global code will not negatively impact existing users and existing implementations. We encourage country-specific customizations to be built using the extension mechanism. Extensions can be shared as open source projects so that other countries might adopt them.

To that end, when considering coding a new feature or modification, please follow these steps to coordinate with the global community:

1. Create an OpenLMIS Jira ticket and include information for the following fields:
   (a) *Type*: Select "New Feature"
   (b) *Status*: Leave as "ROADMAP"
   (c) *Summary*: One line description of the feature
   (d) *Component/s*: If you know which service is impacted by the new feature, please include. If not, leave it blank.
   (e) *Description*: Include the user story and detailed description of the feature. Highlight the end user value. Include user steps and edge cases if applicable. Attach screen shots or diagrams if useful.
   (f) *Affects Version*: Leave it blank.
2. Send an email to the product committee listserv (instructions) with the link to the Jira ticket and any additional information or context about the request. Please review the Global vs. Project-Specific Features wiki for details on how to evaluate if a feature is globally applicable or specific to an implementation. Please clearly indicate any time sensitivities so the product committee is aware and can be responsive.
3. The Product Committee will review the feature request at the next Product Committee meeting and provide feed back or request further clarification. Once the feature request is understood, the Product Committee will evaluate the request.
4. If the request is deemed globally applicable and acceptable for the global codebase, the Product Committee will provide any additional guidence or direction needed in preparation for the Technical Committee review.
5. Once approved by the product committee, we request the implementer to contact the developer forum or contact the Technical Committee to provide a proposed technical design to implement the approved feature. They can help share relevant resources or create any needed extension points (further details below).

Extensibility and Customization

A prime focus of version 3 is enabling extensions and customizations to happen without forking the codebase.

There are multiple ways OpenLMIS can be extended, and lots of documentation and starter code is available:

- The Reference UI supports extension by adding CSS, overriding HTML layouts, adding new screens, or replacing existing screens in the UI application. See the UI Extension Guide.
- The Reference Distribution is a collection of collaborative **Services**, Services may be added in or swapped out to create custom distributions.
- The Services can be extended using **extension points** in the Java code. The core team is eager to add more extension points as they are requested by implementors. For documentation about this extension mechanism, see these 3 READMEs: openlmis-example-extensions README, openlmis-example-extension module README, and openlmis-example service README.
- Extra Data allows for clients to add additional data to RESTful resources so that the internal storage mechanism inside a Service doesn't need to be changed.

- Some features may require both API and UI extensions/customizations. The Technical Committee worked on a Requisition Splitting Extension Scenario that illustrates how multiple extension techniques can be used in parallel.

To learn more about the OpenLMIS extension architecture and use cases, see: https://openlmis.atlassian.net/wiki/x/IYAKAw.

Extension Points

To avoid forking the codebase, the OpenLMIS community is committed to providing **extension points** to enable anyone to customize and extend OpenLMIS. This allows different implementations to share a common global codebase, contribute bug fixes and improvements, and stay up-to-date with each new version as it becomes available.

Extension points are simply hooks in the code that enable some implementations to extend the system with different behavior while maintaining compatibility for others. The Dev Forum or Technical Committee group can help advise how best to do this. They can also serve as a forum to request an extension point.

Developing A New Service

OpenLMIS 3 uses a microservice architecture, so more significant enhancements to the system may be achieved by creating an additional service and adding it in to your OpenLMIS instance. See the Template Service for an example to get started.

What's not accepted

- Code that breaks the build or disables / removes needed tests to pass
  - In case developers expect they may not be able to fix Sonar/tests/build issues within a working day, breaking changes should be reverted.
- Code that doesn't pass our Quality Gate - see the Style Guide and Sonar.
- Code that belongs in an Extension or a New Service
- Code that might break existing implementations - the software can evolve and change, but the community needs to know about it first!

Git, Branching & Pull Requests

The OpenLMIS community employs several code-management techniques to help develop the software, enable contributions, discuss & review and pull the community together. The first is that OpenLMIS code is managed using Git and is always publicly hosted on GitHub. We encourage everyone working on the codebase to take advantage of GitHub's fork and pull-request model to track what's going on.

The pull request should be on a short-lived branch and processed very quickly by reviews towards merging back to the master branch. If there is more that one developer on the same short-lived branch, then that branch is at risk of not being short-lived. It is at risk of being more and more like a release branch under active development, and not short at all.

The branch may have received many commits before the developer initiated the pull request, but it's important to remember about rebasing the changes with the master branch before creating it. Otherwise, conflicts will not allow reviewer to merge the changes to the master branch.

OpenLMIS Jenkins runs build, test and Sonar analysis for all branches and also for pull requests. For both, the developer needs to get the commit reviewed. In case of second of them, code review is happening in the pull request

itself. Build status makes the work easier for developers who create the pull requests and those who are reviewing them, so they save the time and know the code is high quality.

As a developer working on a branch, you have Sonar check your work without that check effecting the Sonar report on the master branch. SonarQube gives developers an opportunity to track the quality of code branches to ensure that only clean, approved code gets merged into master, and when it's not it doesn't mislead them looking at the master branch into thinking the overall project quality has dropped.

While creating pull request Sonar quality gates have to pass. Otherwise, the reviewer wouldn't be able to merge the code to the master branch.

For more about version numbers and releasing, see **versioningReleasing.md**.

The general flow:

1. *Communicate* using JIRA, the wiki, or the developer forum!
2. *Fork* the relevant OpenLMIS project on GitHub
3. *Branch* from the `master` branch to do your work
4. *Commit* early and often to your branch
5. *Re-base* your branch *often* from OpenLMIS `master` branch - keep up to date!
6. Issue a *Pull Request* back to the `master` branch - explain what you did and keep it brief to speed review! Mention the JIRA ticket number (e.g., "OLIMS-34") in the commit and pull request messages to activate the JIRA/GitHub integration.

While developing your code, be sure you follow the Style Guide and keep your contribution specific to doing one thing.

Quality responsibilities

The changes made by the developer should be covered by automated tests which have to follow the Testing Guide. The developers should make sure that all acceptance criteria are met and check whether their changes work before moving the ticket to QA. They should cooperate with the QA person and give them some tips on how to verify changes that are difficult to test. The developer should make the tester aware of potential issues or places that could be affected by their changes.

DO:

- cover changes by automated tests
- cover all acceptance criteria
- initial manual testing
- give important information
- cooperate with the QA
- give advice on how to test difficult changes
- warn about potential issues

DON'T:

- leave changes without automated tests
- skip any of the acceptance criteria
- move changes to QA without any verification
- just assign the ticket to the QA

- snub questions
- leave the QA without any help
- conceal any gaps in the code

Feature Flags

This is a mechanism that can reduce branching code and merging large pull requests when working on some big functionality that is not finished. **Feature Flags** are based on branching the code execution based on status of feature flag. We simply put old working code in one branch while our new implementation is placed in another. This allows us to ommit using git branches and have our code on the master branch. Moreover this new code will be deployed on our tests servers. **Feature Flag** status can be changed on deployment by setting an environment variable with proper name.

Feature Flags can be used in situations like:

- making an controversial change that could break other functionality
- making a potential performance improvement
- working on unfinished functionality on master branch
- marking functionality that can be turned on/off after releasing it (those should be documented) All except for the last case should be rather short lived flags and be removed after few weeks. Those **Feature Flags** give us advantage of possibility to verify changes and logs on test server rather than locally.

Here is an example of implementation for **BATCH_APPROVE_SCREEN feature flag** in our UI code:

- Feature flag constant
- Run method for setting flag to our featureFlagService
- Example of feature flag usage

Here is an example of implementation for **FACILITY_SEARCH_CONJUNCTION feature flag** in our backend code:

- Commit with new feature flag

Automated Testing

OpenLMIS 3 includes new patterns and tools for automated test coverage at all levels. Unit tests continue to be the foundation of our automated testing strategy, as they were in previous versions of OpenLMIS. Version 3 introduces a new focus on integration tests, component tests, and contract tests (using Cucumber). Test coverage for unit and integration tests is being tracked automatically using Sonar. Check the status of test coverage at: http://sonar.openlmis.org/. New code is expected to have test coverage at least as good as the existing code it is touching.

Continuous Integration, Continuous Deployment (CI/CD) and Demo Systems

Continuous Integration and Deployment are heavily used in OpenLMIS. Jenkins is used to automate builds and deployments trigged by code commits. The CI/CD process includes running automated tests, generating ERDs, publishing to Docker Hub, deploying to Test and UAT servers, and more. Furthermore, documentation of these build pipelines allows any OpenLMIS implementation to clone this configuration and employ CI/CD best practices for their own extensions or implementations of OpenLMIS.

See the status of all builds online: http://build.openlmis.org/

Learn more about OpenLMIS CI/CD on the wiki: CI/CD Documentation

Language Translations & Localized Implementations

OpenLMIS 3 has translation keys and strings built into each component, including the API services and UI components. The community is encouraging the contribution of translations using Transifex, a tool to manage the translation process. Because of the micro-service architecture, each component has its own translation file and its own Transifex project.

See the OpenLMIS Transifex projects and the Translations wiki to get started. See also the UI Extenion guide for details about how implementations can override UI strings with their own custom messages.

Licensing

OpenLMIS code is licensed under an open source license to enable everyone contributing to the codebase and the community to benefit collectively. As such all contributions have to be licensed using the OpenLMIS license to be accepted; no exceptions. Licensing code appropriately is simple:

Modifying existing code in a file

- Add your name or your organization's name to the license header. e.g. if it reads `copyright VillageReach`, update it to `copyright VillageReach, <insert name here>`
- Update the copyright year to a range. e.g. if it was 2016, update it to read 2016-2017

Adding new code in a new file

- Copy the license file header template, LICENSE-HEADER, to the top of the new file.
- Add the year and your name or your organization's name to the license header. e.g. if it reads `Copyright © <INSERT YEAR AND COPYRIGHT HOLDER HERE>`, update it to `Copyright © 2017 MyOrganization`

For complete licensing details be sure to reference the LICENSE file that comes with this project.

Feature Roadmap

The Living Roadmap can be found here The backlog can be found here

Suggest a New Feature

The OpenLMIS community welcomes suggestions and requests for new features, functionality or improvements to OpenLMIS. **Please note that suggested new features may or may not be scheduled for work depending on resourcing and value to the community. If this feature is needed for a specific implementation in a timely fashion we suggest the team consider building the feature and contributing it back to core. See the section on Contributing Code above for details.** Follow the steps below so that the community can review, evaluate, and potentially schedule the new feature for work:

1. Create an OpenLMIS Jira ticket and include information for the following fields:
   (a) *Type*: Select "New Feature"
   (b) *Status*: Leave as "ROADMAP"
   (c) *Summary*: One line description of the feature

(d) *Component/s*: If you know which service is impacted by the new feature, please include. If not, leave it blank.

(e) *Description* Include the user story and detailed description of the desired new feature, functionality or improvement. Highlight the end user value. Include user steps and edge cases if applicable. Attach screen shots or diagrams if useful in building a shared understanding of the suggested feature.

(f) *Affects Version*: Leave it blank.

2. Send an email to the product committee listserv (instructions) with the link to the Jira ticket and any additional information or context about the suggested feature and functionality. Please review the Global vs. Project-Specific Features wiki for details on how to evaluate if a feature is globally applicable or specific to an implementation. Please clearly indicate any time sensitivities so the product committee is aware and can be responsive.
3. The Product Committee will review the feature request at the next Product Committee meeting and provide feed back or request further clarification. Once the feature request is understood, the Product Committee will evaluate the request to determine the next steps.
    - The Product Committee will set the priority of the feature and keep the Jira ticket updated with information on scheduling, questions, and if any additional information is needed.
4. Follow the ticket in Jira or attend Product Committee meetings to keep updated on the status of the suggested new feature.

Contributing Documentation

Writing documentation is just as helpful as writing code. See Contribute Documentation.

References

- Developer Documentation (ReadTheDocs) - http://docs.openlmis.org/
- Developer Guide (in the wiki) - https://openlmis.atlassian.net/wiki/display/OP/Developer+Guide
- Architecture Overview (v3) - https://openlmis.atlassian.net/wiki/pages/viewpage.action?pageId=51019809
- API Docs - http://docs.openlmis.org/en/latest/api
- Database ERD Diagrams - http://docs.openlmis.org/en/latest/erd/
- GitHub - https://github.com/OpenLMIS/
- JIRA Issue & Bug Tracking - https://openlmis.atlassian.net/projects/OLMIS/issues
- Wiki - https://openlmis.atlassian.net/wiki/display/OP
- Developer Forum - https://groups.google.com/forum/#!forum/openlmis-dev
- Release Process (using Semantic Versioning) - https://openlmis.atlassian.net/wiki/display/OP/Releases
- OpenLMIS Website - https://openlmis.org

1.4.2 Contribute documentation

This document briefly explains the process of collecting, building and contributing the documentation to OpenLMIS v3.

Build process

The developer documentation for OpenLMISv3 is scattered across various repositories. Moreover, some of the artifacts are dynamically generated, based on the current codebase. All that documentation is collected by a single script. In order to collect a new document to be able to include it in the developer documentation, it must be placed in the *collect-docs.py* script. The documentation is built daily and is triggered by a Jenkins job. It then gets published via ReadTheDocs at http://docs.openlmis.org. The static documentation files and the build configuration is kept on the openlmis-ref-distro repository, in the *docs* directory. It is also possible to rebuild and upload the documentation to Read the Docs manually, by running the *OpenLMIS-documentation* Jenkins job.

Contributing

Depending on the part of the documentation that you wish to contribute to, a specific document in one of the GitHub repositories must be edited. The list below explains where the particular pieces of the documentation are fetched from, in order to be able to locate and edit them.

Developer docs - Services: The documentation for each service is taken from the *README.md* file located on that repository.

Developer docs - Style guide: This is the code style guide, located in the openlmis-template-service in file *STYLE-GUIDE.md*.

Developer docs - Testing guide: This is the document that outlines the strategy and rules for test development. It is located in the openlmis-template-service in *TESTING.md* file.

Developer docs - Error Handling: This document outlines how errors should be managed in Services and how they should be reported through API responses.

ERD schema: The ERD schema for certain services is generated by Jenkins. The static file that links to the schema is located together with the documentation and the schemas itself are built and kept on Jenkins as build artifacts. The link always points to the ERD schema of the latest, successful build.

UI Styleguide: The configuration of the styleguide is located on the openlmis-requisition-refUI. The actual Styleguide is generated by the Jenkins job and uploaded to the gh-pages branch on the same repository.

API documentation: This contains the link to the Swagger documentation for the API endpoints. It is built by the Jenkins job and kept as a build artifact, based on the content of the RAML file. The link always points to the API documentation of the latest successful build.

1.5 Conventions

1.5.1 The License Header

Each java or javascript file in the codebase should be annotated with the proper copyright header. This header should be also applied to singnificant html files.

We use checkstyle to check for it being present in Java files. We also check for it during our Grunt build in javascript files.

The current copyright header format can be found here.

Replace the year and holder with appropriate holder, for example:

```
Copyright © 2017 VillageReach
```

1.5.2 OpenLMIS Community Principles (2015)

The OpenLMIS community principles aims to help contributors to the project create quality contributions by illimunating some of the intentions behind the OpenLMIS principles and influence better design and implementation of OpenLMIS features.

This document is an outcome of the 2015 Community meeting and is copied (with minor modification) from its original source.

Principles

Open Source

> OpenLMIS is offered under an open source license, which means that everyone has the right to use and modify the software without paying a license fee. Changes and additions are made available to the community under the terms of the license via our code contribution process.

OpenLMIS is built and licensed under an Open Source license. In addition to the project being Open Source, OpenLMIS strives to always be available to develop on, build, deploy, use and generally contribute to using similarly licensed technologies. In practice this means that strong preference is given to contributions and their dependant technologies that are licensed similarly. Contributions should aspire to contribute:

- Code and other IP licensed in a compatible license as OpenLMIS. Strong preference is given to the OpenLMIS license for simplicity.
- Dependencies on third-party libraries / tools should also be open source and freely distributable.

Appropriate

> OpenLMIS is designed with a focus on users in low resource and capacity environments. Representatives from these environments are welcomed and valued members of the community and their insights help shape the software.

OpenLMIS is built and used by those in low-resource settings:

- Internet is often slow and intermittent. Features should be designed with these limitations in mind. For example, most work-flows should be optimized for slow internet and even work-flows with periods of non-connectivity.Administrative screens however can often take shortcuts and assume that their users will have better internet connectivity.
- Processes not only vary and need to be configurable by program and implementation, they oftentimes areused in parallel or supplement traditional paper processes. Data collection and forms should strive to be configurable to match the official paper form and be able to restore it historically.
- Screens are often older and come with lower resolutions than the latest and greatest. 800x600 px screens are not uncommon. Additionally, many work-flows that would be used by someone at the last mile will be used by someone with a smaller tablet or even a phone.
- Scalability for OpenLMIS is the capability of use in large hospitals to community health workers nation wide. The workflow from data collection, processing through to report delivery should be designed and implemented for thousands of users with thousands of physical facilities.
- Security is important for OpenLMIS to be trusted to run nation-wide government supply chains to NGO initiatives.A role-based security system contains users to see and do only what is required for their role. Care should be given in designing features and running implementations to keep OpenLMIS secure.

Configurable

OpenLMIS flexibly supports the varied needs of low-resource health supply chains. OpenLMIS strives to be designed so that countries can configure and use the software with minimal training and technical capacity.

Supply chains vary. Reporting requirements, process differences, language, and even the look and feel need to be as configurable as is reasonable for OpenLMIS to continue to deliver on its mission. In order to accomplish this, OpenLMIS contributions need to at a minimum continue to deliver:

- Language - Language tags allow messages/UI/email/API/etc to be translated into many different languages and allows the user to switch the language displayed easily. OpenLMIS has standardized development in English for consistency and supports translation projects as the opportunity arises.
- Dates - Date formatting also varies by locality. As such any date or time printed should allow for custom formatting.
- **Programs** allow for OpenLMIS to configure the vertical supply chains present in many low- and middle-income countries independently. e.g. a Malaria program may collect different data by different people than an HIV/AIDS program.
- Schedules allow for a Program to define regular or even planned irregularity for timing of program related events.Monthly and quarterly are typical examples, however a schedule may have periods where a monthly schedule may have to be extended to a couple months when seasonal monsoons slow transportation networks.
- Variable and often Program-segregated administrative hierarchies are needed to ensure programs can operate independently and reflect the common situation of programs not sharing staff.
- A singular geographic hierarchy is currently in use. Unlike many features of OpenLMIS, this definition is not segregated by Program and is meant to reflect that physical facilities often are part of one official geographic hierarchy. In the future this may need to be Program-segregated. For now utilizing administrative hierarchies can be used instead.
- Replenishment cycles also vary by Program in an implementation. The two standard processes, distribution (push) and allocation (pull), are present and in use in OpenLMIS. These two different types of processes differ by who starts them, their cycle, and also how re-supply calculations/projections are made.

Interoperable

OpenLMIS strives to be interoperable with other systems in a larger health information ecosystem.

Achieving interoperability requires a balance between allowing for flexibility and controlling for consistency.OpenLMIS aims to achieve this by:

- designing for and implementing customizable data storage, processing and reporting that's accessible through published APIs & formats.
- encouraging expansion and customization through modularity.
- maintaining a consistent and robust data-model and reporting interfaces so that a field/column/report means the same thing from implementation to implementation.
- maintaining a consistent look & feel so that using OpenLMIS anywhere always looks and behaves in a predictable manner.

Collaborative

OpenLMIS users benefit from the diversity of perspectives and resources that community members bring to the table, which results in a more flexible and powerful system than what any one organization could create. The community acknowledges that successful country implementation requires close collaboration among partners and stakeholders to ensure success.

- documentation is needed to communicate how to use a contribution and the intention behind it. This can take many different forms and it's left to the contributor to determine and provide appropriate levels of documentation. The community strongly discourages contributions that are light on documentation. It's suggested that documentation is prioritized for: published APIs, designs and code contracts. Additionally documenting the why over the how is oftentimes more useful over a longer period of time.
- sharing code often comes with mis-matched expectations and undesired consequences, so it's not unexpected that development often occurs behind closed-doors until "it's ready". The OpenLMIS project however aims to be *open* so all code that is part of the OpenLMIS project is found in the OpenLMIS organization. The recommended approach to collaborating is documented.
- sharing ideas, work items, roadmaps, feature requests, knowledge bases, etc. is vital to know where the project is going and encourage participation. To that end OpenLMIS encourages all participants to utilize the public forums, chat, project management, and wiki spaces to collaborate. An active list is found at docs.openlmis.org.
- automated testing ensures functionality from developer to developer and implementation to implementation is behaving as expected over time. OpenLMIS doesn't currently define code-coverage targets, however, the project expects that appropriate test coverage is provided with every contribution and highly scrutinizes existing tests. Since testing is so important, calling out the kinds of testing done and not done and *why* can greatly help the review process for contributions.

Supportive

The community acts as stewards for the implementation, configuration, training on, operation, and sustainment of OpenLMIS. The community strives to be knowledge experts on the problems that OpenLMIS attempts to solve.

1.5.3 Non-functional Requirements

Note: WIP

Non-functional requirements (NFR) are those that impose constraints on the design of OpenLMIS. The goal of this document is to ensure the OpenLMIS community has a consistent understanding of how these requirements are discovered, defined and prioritized.

Process

NFR as stated previously impose specific constraints on OpenLMIS. These constraints if defined without careful consideration may result in significant increases to level of effort and a focus taken away from other areas.

When defining NFR we need to:

- Be domain focused: some NFR will be defined in terms of the problem domain. e.g. For a Report and Requisition, we can identify the number of Reports and Requisitions the system will handle. We typically might consider the # of concurrent Requisition's being submitted, the total number the system might process a day/week/year/etc.

- Be context aware: NFR often need to be aware of the context they're being defined in. For OpenLMIS this context might be the user persona combined with the problem domain. So for our Requisition's example above, we might focus on the NFR of the # number Requisitions that a Zonal Warehouse Manager might need to process day/week/year/etc. Further we would do well to add to the context what sort of network connection and device characteristics the user persona might typically be expected to have. In this case we'd add that the Zonal Warehouse Manager might be expected to have a recent Laptop with 90 % 3G Internet availability, whereas a Community Health Worker might only expect to have 30 % 2G Internet availability and use a Android phone 2-3 generations old.
- Focus on defining NFR quantitatively, and testing automatically. e.g. we could define response time needs to be < 500ms, and therefore we can automatically test that our response time meets this requirement.
- Bring the user experience and technical leaders together. A classic example of only involving one group is to ask stakeholders "how often should the application be available?" to which they'd rightly respond "all the time". This 100% availability metric can be exceptionally difficult to obtain and leads to costly investments when in reality some gaps in availability are likely preferred considering the costs. By bringing these groups together we'll get a more reasonable requirement: The application must be available 90% of the time, and the 10% of the time it is not should be scheduled for non-work hours and days in Central Africa Timezone.

In practice for the OpenLMIS community, the two communities that need to agree on NFR are the Product Committee and the Technical Committee.

Auditability

All concepts in OpenLMIS should support an audit log which can track every state change with:

- A time-stamp including timezone of when the change occurred.
- A unique identifier for the user account that made the change. Or if the change was not directly caused by a user that should be clear.
- The before and after states of the change.

Performance

For performance we focus on these measures:

- Response time (<= 500ms)
- Concurrency
- Size (objects or rows) of stored data.
- Size of working data. e.g. off-line or in-process stored locally and / or that a user works with at any given time.
- Render time
- Network latency (typically characterized by 2G or 3G).
- CPU (typically characterized by a fraction of a "mordern" processor).

And we focus on load testing over stress or other types of testing (e.g. stress or endurance). For more on how we test, see Performance Testing and Functional Testing.

Availability

- Offline
- # of 9s

Configuration Management

Security

Data integrity

- ACID

Recoverability

- Drafts
- Backups
- Distributed devices

Interoperability

- mCSD w/ FHIR
- GS1 GTIN and GLN
- GS1 EDI
- OAuth2
- REST w/ JSON

Usability

- Screen size
- Browser
- Affordances

Todo

1. Clarify concurrency in Performance.
2. Clarify network and CPU in performance - currently not given in highly reproducible terms.
3. Clarify render time in performance
4. Clarify the 2 size of data in Performance - how will we define it here across domains? Or use links?
5. Move functional testing into RTD.
6. Fill in Interoperability
7. Fill in Data integrity
8. Fill in etc. . .

1.5.4 Service Conventions

OpenLMIS Service Style Guide

This is a WIP as a style guide for an Independent Service. Clones of this file should reference this definition.

Java

OpenLMIS has adopted the Google Java Styleguide. These checks are *mostly* encoded in Checkstyle and should be enforced for all contributions.

Some additional guidance:

- Try to keep the number of packages to a minimum. An Independent Service's Java code should generally all be in one package under `org.openlmis` (e.g. `org.openlmis.requisition`).
- Sub-packages below that should generally follow layered-architecture conventions; most (if not all) classes should fit in these four: `domain`, `repository`, `service`, `web`. To give specific guidance:
  - Things that do not strictly deal with the domain should NOT go in the `domain` package.
  - Serializers/Deserializers of domain classes should go under `domain`, since they have knowledge of domain object details.
  - DTO classes, belonging to serialization/deserialization for endpoints, should go under `web`.
  - Exception classes should go with the classes that throw the exception.
  - We do not want separate sub-packages called `exception`, `dto`, `serializer` for these purposes.
- When wanting to convert a domain object to/from a DTO, define Exporter/Importer interfaces for the domain object, and export/import methods in the domain that use the interface methods. Then create a DTO class that implements the interface methods. (See Right and RightDto for details.)
  - Additionally, when Exporter/Importer interfaces reference relationships to other domain objects, their Exporter/Importer interfaces should also be used, not DTOs. (See example.)
- Even though the no-argument constructor is required by Hibernate for entity objects, do not use it for object construction (you can set access modifier to `private`); use provided constructors or static factory methods. If one does not exist, create one using common sense parameters.

RESTful Interface Design & Documentation

Designing and documenting

Note: many of these guidelines come from Best Practices for Designing a Pragmatic RESTful API.

- Result filtering, sorting and searching should be done by query parameters. Details
- Return a resource representation after a create/update. Details
- Use camelCase (vs. snake_case) for names, since we are using Java and JSON. Details
- Don't use response envelopes as default (if not using Spring Data REST). Details
- Use JSON encoded bodies for create/update. Details
- Use a clear and consistent error payload. Details

- Use the HTTP status codes effectively. Details
- Resource names should be pluralized and consistent. e.g. prefer `requisitions`, never `requisition`.
- Resource representations should use the following naming and patterns:
  - **Essential**: representations which can be no shorter. Typically this is an id and a code. Useful most commonly when the resource is a collection, e.g. `/api/facilities`.
  - **Normal**: representations which typically are returned when asking about a specific resource. e.g. `/api/facilities/{id}`. Normal representations define the normal transactional boundary of that resource, and *do not* include representations of other resources.
  - **Optional**: a representation that builds off of the resource's **essential** representation, allowing for the client to ask for additional fields to be returned by specifying a `fields` query parameter. The support for these representations is completely, as the name implies, optional for a resource to provide. Details
  - **Expanded**: a representation which is in part, not very RESTful. This representation allows for other, related, resources to be included in the response by way of the `expand` query parameter. Support for these representations is also optional, and in part somewhat discouraged. Details
- A PUT on a single resource (e.g. PUT /facilities/{id}) is not strictly an update; if the resource does not exist, one should be created using the specified identity (assuming the identity is a valid UUID).
- Exceptions, being thrown in exceptional circumstances (according to *Effective Java* by Joshua Bloch), should return 500-level HTTP codes from REST calls.
- Not all domain objects in the services need to be exposed as REST resources. Care should be taken to design the endpoints in a way that makes sense for clients. Examples:
  - `RoleAssignments` are managed under the users resource. Clients just care that users have roles; they do not care about the mapping.
  - `RequisitionGroupProgramSchedules` are managed under the requisitionGroups resource. Clients just care that requisition groups have schedules (based on program).
- RESTful endpoints that simply wish to return a JSON value (boolean, number, string) should wrap that value in a JSON object, with the value assigned to the property "result". (e.g. `{ "result": true }`)
  - Note: this is to ensure compliance with all JSON parsers, especially ones that adhere to RFC4627, which do not consider JSON values to be valid JSON. See the discussion here.
- When giving names to resources in the APIs, if it is a UUID, its name should have a suffix of "Id" to show that. (e.g. `/api/users/{userId}/fulfillmentFacilities` has query parameter `rightId` to get by right UUID.)
- If you are implementing HTTP caching for an API and the response is a DTO, make sure the DTO implements equals() and hashCode() using all its exposed properties. This is because of potential confusion of a property change without a change of ETag.

We use RAML (0.8) to document our RESTful APIs, which are then converted into HTML for static API documentation or Swagger UI for live documentation. Some guidelines for defining APIs in RAML:

- JSON schemas for the RAML should be defined in a separate JSON file, and placed in a `schemas` subfolder in relation to the RAML file. These JSON schema files would then be referenced in the RAML file like this (using role as an example):

```
- role: !include schemas/role.json

- roleArray: |
  {
    "type": "array",
```

(continues on next page)

(continued from previous page)

```
    "items": { "type": "object", "$ref": "schemas/role.json" }
  }
```

  - (Note: this practice has been established because RAML 0.8 cannot define an array of a JSON schema for a request/response body (details). If the project moves to the RAML 1.0 spec and our RAML testing tool adds support for RAML 1.0, this practice might be revised.)

Pagination

Many of the GET endpoints that return *collections* should be paginated at the API level. We use the following guidelines for RESTful JSON pagination:

- Pagination options are done by *query* paramaters. i.e. use `/api/someResources?page=2` and not `/api/someResources/page/2`.
- When an endpoint is paginated, and the pagination options are *not* given, then we return the full collection. i.e. a single page with every possible instance of that resource. It's therefore up to the client to use collection endpoints responsibly and not over-load the backend.
- A paginated resource that has no items returns a single page, with it's `content` attribute as empty.
- Resource's which only ever return a single identified item are *not* paginated.
- For Java Service's the query parameters should be defined by a Pageable and the response should be a Page.

Example Request (note that page is zero-based):

```
GET /api/requisitions/search?page=0&size=5&access_token=<sometoken>
```

Example Response:

```
{
  "content": [
    {
    ...
    }
  ],
  "totalElements": 13,
  "totalPages": 3,
  "last": false,
  "numberOfElements": 5,
  "first": true,
  "sort": null,
  "size": 5,
  "number": 0
}
```

Postgres Database

For guidelines on how to write schema migrations using Flyway, see Writing Schema Migrations (Using Flyway).

- Each Independent Service should store its tables in its own schema. The convention is to use the Service's name as the schema. e.g. The Requisition Service uses the `requisition` schema
- Tables, Columns, constraints etc should be all lower case.
- Table names should be pluralized. This is to avoid *most* used words. e.g. orders instead of order

- Table names with multiple words should be snake_case.
- Column names with multiple words should be merged together. e.g. `getFirstName()` would map to `firstname`
- Columns of type uuid should end in 'id', including foreign keys.

RBAC (Roles & Rights) Naming Conventions

- Names for rights in the system should follow a RESOURCE_ACTION pattern and should be all uppercase, e.g. REQUISITION_CREATE, or FACILITIES_MANAGE. This is so all of the rights of a certain resource can be ordered together (REQUISITION_CREATE, REQUISITION_AUTHORIZE, etc.).

i18n (Localization)

Transifex and the Build Process

OpenLMIS v3 uses Transifex for translating message strings so that the product can be used in multiple languages. The build process of each OpenLMIS service contains a step to sync message property files with a corresponding Transifex project. Care should be taken when managing keys in these files and pushing them to Transifex.

- If message keys are added to the property file, they will be added to the Transifex project, where they are now available to be translated.
- If message keys or strings are modified in the property file, any translations for them will be lost and have to be re-translated.
- If message keys are removed in the property file, they will be removed from the Transifex project. If they are re-added later, any translations for them will be lost and have to be re-translated.

Naming Conventions

These naming conventions will be applicable for the messages property files.

- Keys for the messages property files should follow a hierarchy. However, since there is no official hierarchy support for property files, keys should follow a naming convention of most to least significant.
- Key hierarchy should be delimited with a period (.).
- The first portion of the key should be the name of the Independent Service.
- The second portion of the key should indicate the type of message; error for error messages, message for anything not an error.
- The third and following portions will further describe the key.
- Portions of keys that don't have hierarchy, e.g. `a.b.code.invalidLength` and `a.b.code.invalidFormat`, should use camelCase.
- Keys should not include hyphens or other punctuation.

Examples:

- `requisition.error.product.code.invalid` - an alternative could be `requisition.error.productCode.invalid` if code is not a sub-section of product.
- `requisition.message.requisition.created` - requisition successfully created.
- `referenceData.error.facility.notFound` - facility not found.

Note: UI-related keys (labels, buttons, etc.) are not addressed here, as they would be owned by the UI, and not the Independent Service.

Testing

See the Testing Guide.

Docker

Everything deployed in the reference distribution needs to be a Docker container. Official OpenLMIS containers are made from their respective containers that are published for all to see on our Docker Hub.

- Dockerfile (Image) best practices
- Keep Images portable & one-command focused. You should be comfortable publishing these images publicly and openly to the DockerHub.
- Keep Containers ephemeral. You shouldn't have to worry about throwing one away and starting a new one.
- Utilize docker compose to launch containers as services and map resources
- An OpenLMIS Service should be published in one image found on Docker Hub
- Services and Infrastructure that the OpenLMIS tech committee owns are published under the "openlmis" namespace of docker and on the Docker Hub.
- Avoid Docker Host Mounting, as this doesn't work well when deploying to remote hosts (e.g. in CI/CD)

Gradle Build

Pertaining to the build process performed by Gradle.

- Anything generated by the Gradle build process should go under the `build` folder (nothing generated should be in the `src` folder).

Logging

Each Service includes the SLF4J library for generating logging messages. Each Service should be forwarding these log statements to a remote logging container. The Service's logging configuration should indicate the name of the service the logging statement comes from and should be in UTC.

What generally should be logged:

- DEBUG - should be used to provide more information to developers attempting to debug what happened. e.g. bad user input, constraint violations, etc
- INFO - to log processing progress. If the progress is for a developer to understand what went wrong, use DEBUG. This tends to be more useful for performance monitoring and remote production debugging after a client's installation has failed.

Less used:

- FATAL - is reserved for programming errors or system conditions that resulted in the application (Service) terminating. Developers should not be using this directly, and instead use ERROR.

- ERROR - is reserved for programming conditions or system conditions that would have resulted in the Service terminating, however some safety oriented code caught the condition and made it safe. This should be reserved for a global Service level handler that will convert all Exceptions into a HTTP 5xx level exception.

Audit Logging

OpenLMIS aims to create a detailed audit log for most all actions that occur within the system. In practice this means that as a community we want all RESTful Resources (e.g. `/api/facilities/{id}`) to also have a full audit log for every change (e.g. `/api/facilities/{id}/auditLog`) and for that audit log to be accessible to the user in a consistent manner.

A few special notes:

- When a resource has line items (e.g. Requisition, Order, PoD, Stock Card, etc), the line item would not have its own REST Resource, in that case if changes are made to a line item, those changes need to be surfaced in the lint item's parent. For example, if a change is made to a Requisition Line Item, then the audit log for that change is available in the audit log for the Requisition, as one can't retrieve through the API the single line item.
- There are a few cases where audit logs may not be required by default. These cases typically involve the resource being very transient in nature: short drafts, created Searches, etc. When this is in question, explore the requirements for how long the resource needs to exist and if it forms part of the system of record in the supply chain.

Most Services use JaVers to log changes to Resources. The audits logs for individual Resources should be exposed via endpoints which look as follows:

```
/api/someResources/{id}/auditLog
```

Just as with other paginated endpoints, these requests may be filtered via *page* and *size* query paramaters: `/api/someResources?page=0&size=10`

The returned log may additionally be filtered by *author* and *changedPropertyName* query paramaters. The later specifies that only changes made by a given user should be returned, whereas the later dictates that only changes related to the named property should be shown.

Each `/api/someResources/{id}/auditLog` endpoint should return a 404 error if and only if the specified {id} does not exist. In cases where the resource id exists but lacks an associated audit log, an empty array representing the empty audit should be returned.

Within production services, the response bodies returned by these endpoints should correspond to the JSON schema defined by *auditLogEntryArray* within */resources/api-definition.yaml*. It is recognized and accepted that this differs from the schema intended for use by other collections throughout the system. Specifically, whereas other collections which support paginated requests are expected to return pagination-related metadata (eg: "totalElements," "totalPages") within their response bodies, the responses proffered by /auditLog endpoints do not retur pagination related data.

Testing Guide

This guide is intended to layout the general test strategy for OpenLMIS.

OpenLMIS, like many software projects, relies on testing to guide development and prevent regressions. To effect this, we've adopted a standard set of tools to write and execute our tests, and categorize them to understand what types of tests we have, who writes them, when they're written, run, and where they live.

The objective of manual and automatic tests is to catch bugs as quickly as possible after code is committed to make it less time-consuming to fix. We plan to automate tests as much as possible to ensure better control of product quality.

All tickets should be tested manually by the QA Team. Selected tests will be automated at the API level (by Cucumber) and the UI level (i.e. by Selenium).

The document describes the following issues:

- Manual tests;
- Manual testing standards: The UI style guide compatibility, translations and e.g. performance standards;
- UI testing: Includes a list of types of supported devices/browsers prioritized for manual testing, and the strategy for testing the UI;
- Responsibility;
- Testing workflow: Describes the workflow for manual, automated testing and regression testing, and contains guidance on how to report bugs;
- Regression testing;
- Test environments and updating test data;
- Performance tests;
- Automated tests;
- Tools: What tools are used for testing OpenLMIS.

Manual Tests

Manual tests should:

- Cover edge cases rather than happy paths:
  - Changing the column order in the requisition template should not affect existing requisitions (i.e. change the order of the requisition columns);
  - Disabling/enabling requisition columns should not affect existing requisitions;
  - Blank mandatory fields;
  - Deactivation of different entities (programs, facilities etc.);
  - Running out of periods for the requisition workflow;
- Verify email messages are sent (until there are appropriate automated patterns for this):
  - Emails to the storeroom manager after requisition status changes;
- Cover checking reports (until we have an automated pattern):
  - Printing requisitions;
  - Stock-Based requisition reports;
- Not check whether a modal or a notification contains an exact text, but rather verify if it gives the user all important information and context:
  - "An error message implying the orderable was not found is displayed to the user", instead of checking the exact message (The following error message is displayed to the user: "Orderable with code XYZ was not found");
  - "A question asking about changing requisition status to Submitted", instead of checking the exact message (The following error message is displayed to the user: "Are you sure you want to submit this R&R?");
- Not check any UI-specific details:

  - The order of the columns in a table;
  - Exact label names;
  - Exact placement of inputs;
  - Colors of elements if they are not significant (not notifications, buttons or validations).

Manual Testing Standards

Testing Standards:

- UI guidelines;
- Internationalization;
- Form entry and errors;
- Loading performance;
- Offline performance;
- Role based access control (RBAC);
- Exception scenarios.

UI Testing

This section includes a list of types of supported devices/browsers prioritized for manual testing.

Past versions of OpenLMIS have officially supported Firefox. For OpenLMIS 3.0, we are prioritizing the support of Chrome because of global trends (e.g. see Mozambique Stats), its developer tools and auto-updating nature.

For the testing of OpenLMIS, our browser version priorities are:

1. Chrome: The newest version;
2. Firefox: The newest version.

The next most widely-used browser version is IE 11 but we don't recommend testing and bug fixes specifically for any Internet Explorer compatibility in OpenLMIS.

The operating systems on which we should test are:

1. Windows 7 (by far the most-widely-used in Mozambique, Zambia, Benin and globally);
2. Windows 10.

Note: The QA team performs some tests with the use of Linux (Ubuntu) workstations. This is fine for API tests but Linux is not a priority environment for UI testing and for final testing of OpenLMIS. It's important to test the UI using Chrome and Firefox on Windows 7 and Windows 10.

In other words, OpenLMIS developers and team members may be using Mac and Linux environments. It is fine to report bugs happening on the supported browsers (Chrome and Firefox) on those platforms, but we won't invest QA time in extensive manual testing on Mac or Linux.

We asked different OpenLMIS implementations to share their Google analytics so that we can prioritize browser and device support in the future.

Strategy for Testing UI-specific Elements

- The filter buttons should be checked by E2E tests (no need to test alignment in a separate ticket);
- The sort buttons should be checked by E2E tests (no need to test alignment in a separate ticket);
- We should include checking some component, like the filter button, in the E2E test for a specific screen, rather than create a specific manual test case for all occurrences;
- Checking whether a given screen is accessible for a user with or without proper rights should be checked by E2E tests;
- The situation of UI elements should not be tested;
- Resizing input boxes should be checked by an E2E test;
- Colors of rows in tables should not be tested;
- Redirecting to the login form after token expiration should be checked by E2E tests;
- The product grid/stockmanagement validations should be checked by E2E tests;
- Toggles should be checked by an E2E test;
- There is no need to have a test case for the background of the application;
- The order of the items on the navigation bar should be checked by an E2E test;
- Offline actions should be checked by E2E tests;
- The filter button's color should be checked by an E2E test;
- The pagination component should not be checked separately because it is checked in other E2E tests (e.g. one concerning requisition submission);
- The order of facilities within drop-downs has a unit test, so there is no need to check it manually;
- The auto-save should be checked by an E2E test;
- The table horizontal scrollbar should be tested manually;
- The sticky columns should be tested manually;
- The sticky headers should be tested manually;
- The breadcrumbs should be tested manually;
- The datepickers should be tested manually.

Supported Devices

OpenLMIS 3.0 only officially supports desktop browsers with a pointer (mouse, trackpad, etc.). The UI will not necessarily support touch interfaces without a mouse pointer, such as iPad or other tablets. For now, we do not need to conduct tests or file bugs for tablets, smart watches or other devices.

Screen Size

We suggest testing with the most popular screen sizes:

1. 1000 x 600 (this is a popular resolution for older desktop screen sizes; it is the 16:9 equivalent of the age-old 1024x768 size);
2. 1300 x 975 (this is a popular resolution for newer laptop or desktop screens).

The UI should work on screens within that range of sizes. The screen size can be simulated in any browser by changing the size of the browser window or with the use of the Chrome developer tools.

Responsibility

Developers (all teams):

- Writing unit tests for all code (we want test coverage to reach 80%-100%);
- Writing integration and component tests as assigned in tickets or as a part of the acceptance criteria.

Communication

QA meetings are scheduled to discuss testing-related topics, and the daily communication is held on the #QA slack channel for OpenLMIS. The meeting notes are maintained on the QA Weekly meeting notes page.

Testing Workflow within a Sprint

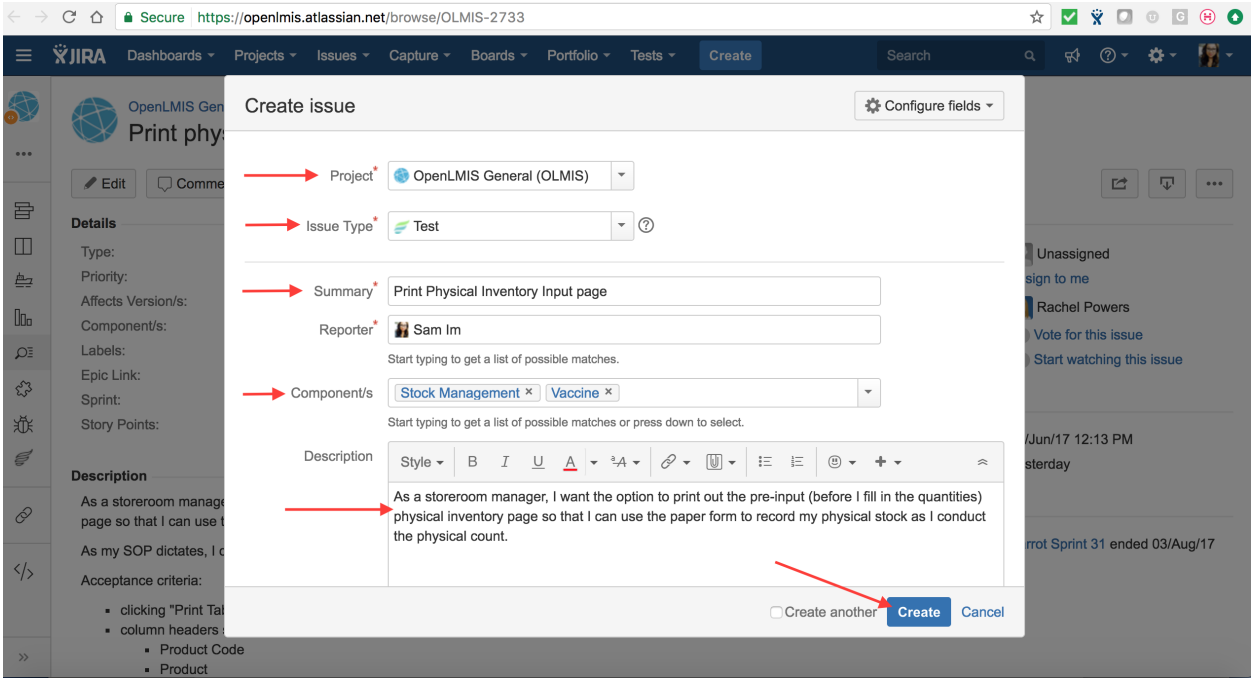
The testing process in the OpenLMIS project is divided into three areas: Manual testing, automated testing and regression testing. This section covers manual and regression testing performed with the use of Zephyr test cases and test cycles.

Step 1: Test Cases

When a new feature ticket that does not concern the API is assigned to a sprint, the QA lead or the tester has to create a test case and link it to the Jira ticket. This can happen in parallel to the development of the ticket. Test cases have to be created for each JIRA ticket that is assigned to a sprint and has to be tested manually; one mustn't create test cases for API changes, as contract tests are used for their verification. It's advised to create test cases before a sprint begins, but this can also be done once a sprint has started. Note that sometimes a given feature might be interrelated with the already-existing ones, which increases the risk of regression. In such situations, the developer working on the ticket should recommend the test cases that should be run in parallel to the testing of a given feature. If this proves impossible, they should inform the tester on the possible influence of the changes on other parts of the system.

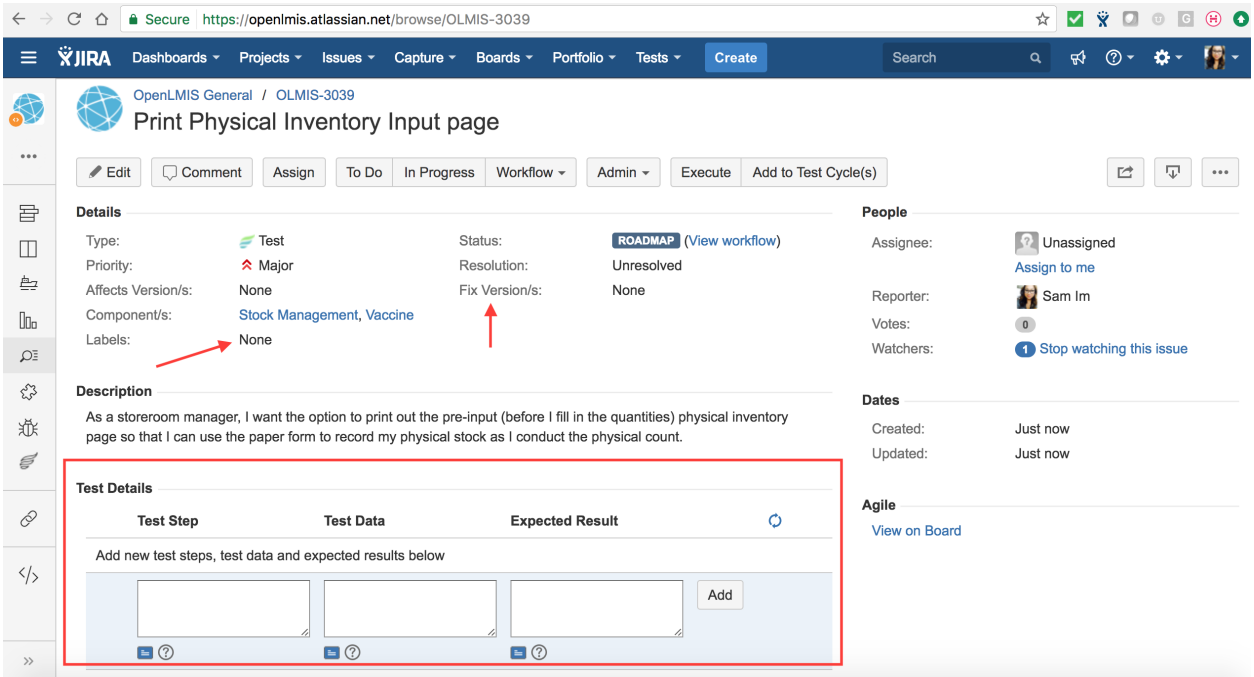
Each new test case has to be reviewed (i.e. its content should be assessed; it should also be determined whether it concerns a feature that is vital to the system – if so, it has to be assigned the "CoreRegression" label), and thus its status should be "QA". After the review, the test case's status should be changed to "To Do". Every 2 or 3 sprints, the test cases concerning the features that had been impacted by the most-recent changes should be reviewed, so as to ensure that they are still valid and that the test steps they include are not outdated.

In JIRA, click on "Create". Select the project, in this case the project is "OpenLMIS General". Then, select the "Issue Type" as Test. The "Summary", "Components", and "Description" can be copied from the ticket to keep consistency, or help in rewriting the test case in the next steps. Note that every valid test case has to have the "To Do" status and "Minor" priority. The test case's name should be in accordance with the following convention: "Screen: Tested feature" (e.g. "Users: Searching for users"). After entering all data, click the "Create" button and a JIRA notification will pop up with the test case's number.

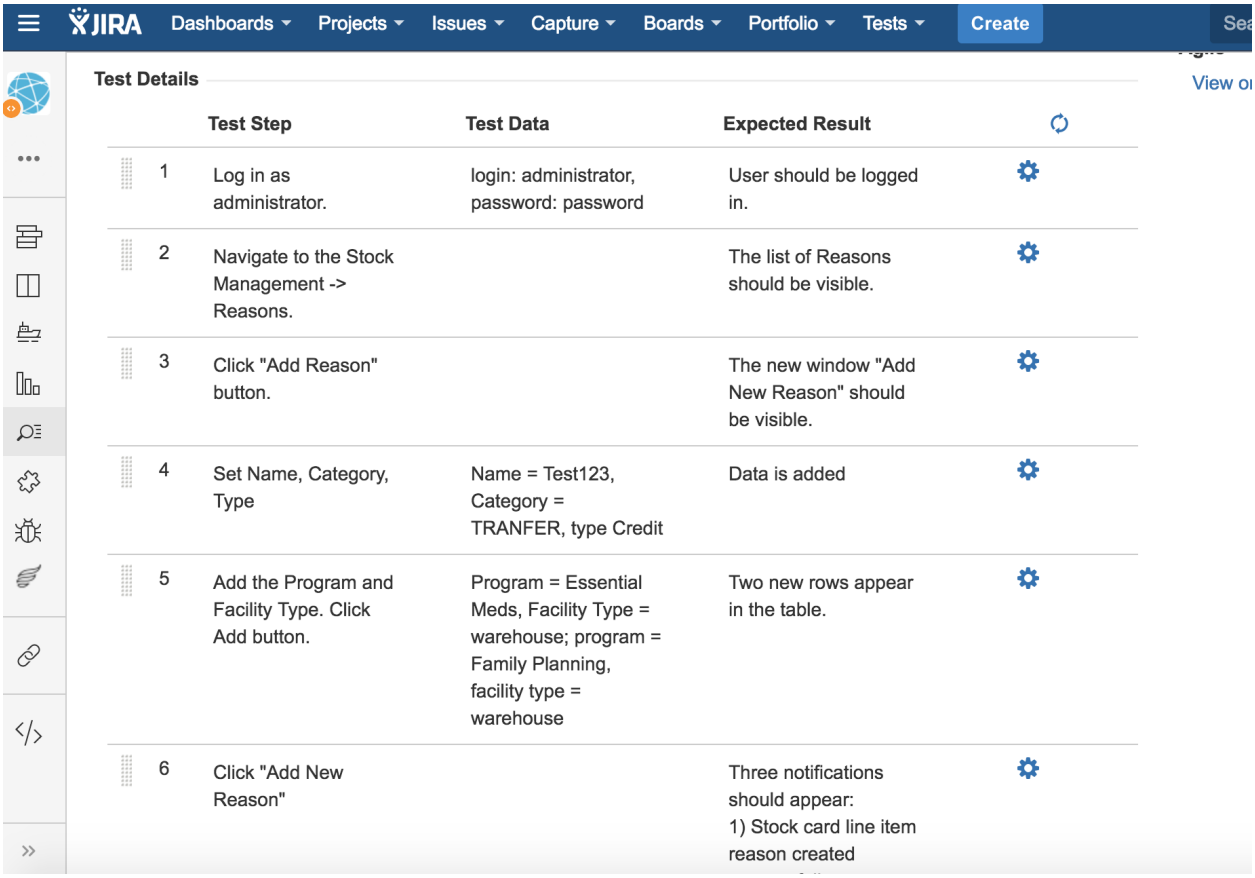


Click the test case's number, and it will bring you to the page enabling one to add the test steps and make required associations. Labels help when querying for test cases to add to regression test cycles. For the example below, a suggested label would be "Stock Management" or "Vaccines". When the tested ticket concerns UI changes and contains a mock-up, the mock-up should be added as an attachment to the ticket with the test case.

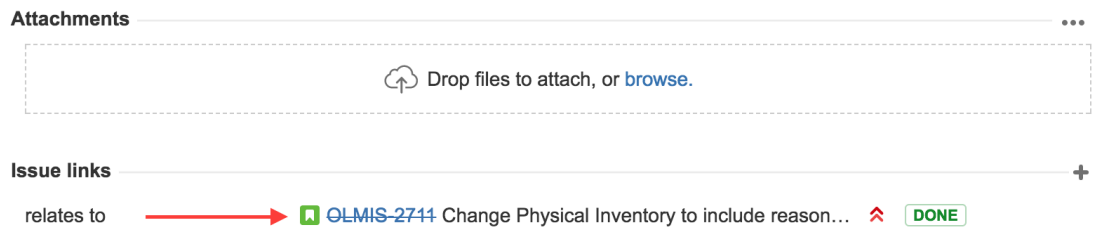
The fix version/s is an association that organizes the test cases. This is used to report test case execution on the Test Metrics Dashboard. If the fix version is not known at the time of creating the test, select "Unscheduled". When the JIRA ticket is assigned to a sprint, the test case must be updated with a correct fix version or it will not appear on the Test Metrics Dashboard. The test steps provide a step by step guide to any tester on how to complete proper validation of the JIRA ticket. The steps should explain in enough detail the actions to take, as well as the expected outcome of those steps.

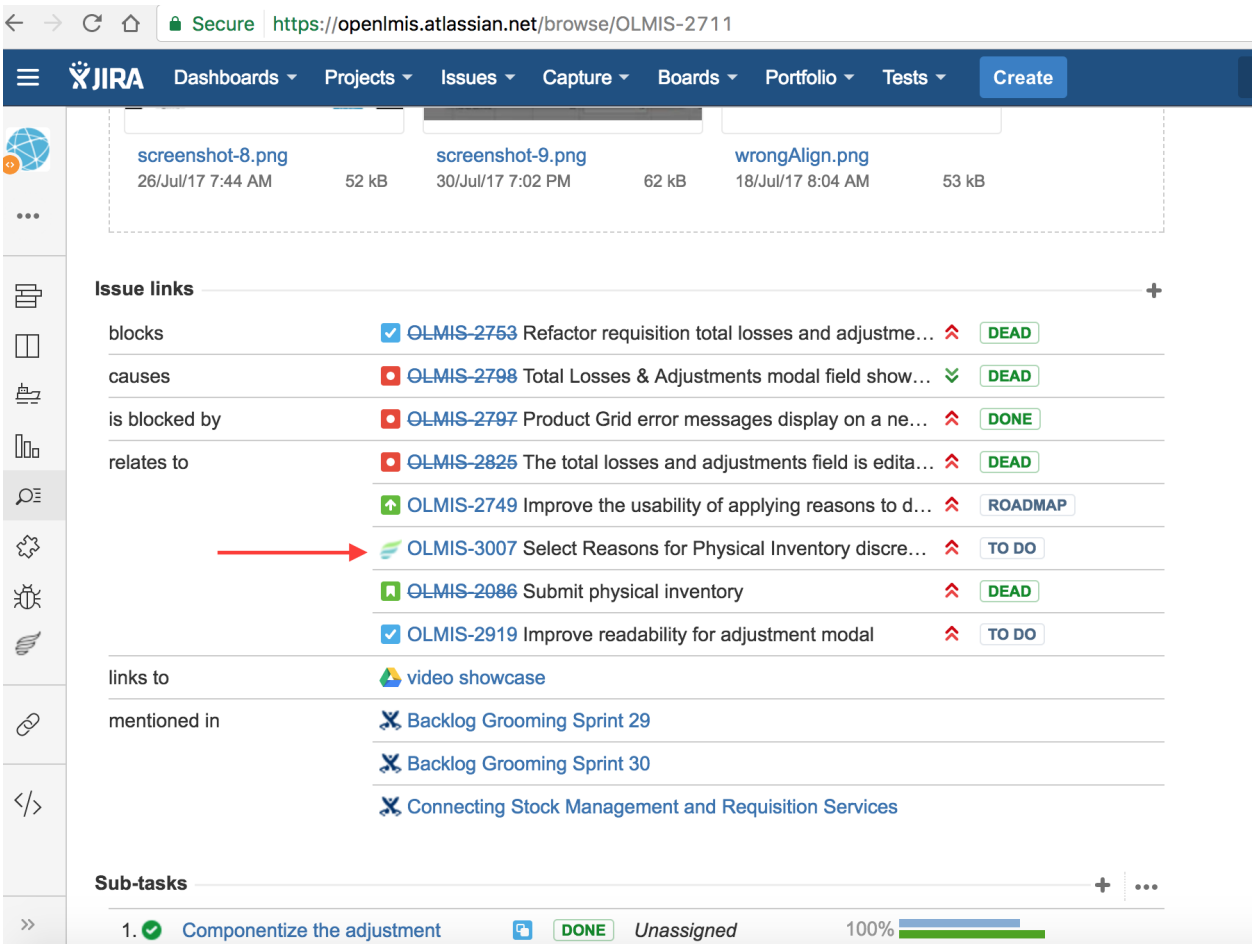


Here is an example of some test steps:



Once the test case had been created, it needs to be associated with the JIRA ticket, as in the example below:





Test Case Best Practices

Creating a Test Case for an Edge Case or Unhappy Path Scenario

One needs to add several types of details in the test case's description. These include pre-conditions, i.e. conditions that need to be met before the test case can be executed (e.g. "At least one approved requisition needs to be created in the system"; "Logging into the application"), the acceptance criteria from the ticket, a description of the scenario/workflow (i.e. whether it is a happy-path one or an edge case, and what the workflow looks like; e.g. "User X authorizes a requisition, User Y rejects it, User X authorizes the requisition again", etc.), and of the test case itself (e.g. "Tests whether it is possible to edit an approved requisition"; "Tests whether authorized users can approve requisitions"). In order to facilitate the execution of test cases concerning requisitions, one has to include the link to the Requisition States and Workflow diagram in the pre-conditions of such test cases.

If possible, one should not include too detailed data in the test case, e.g. user, facility or program names, as they may vary, depending on the implementation. So one should write e.g.: "Choose any program (e.g. Family Planning)"; "Knowing the credentials of any Stock Manager (e.g. srmanager2)"; "Knowing the credentials of any user authorized to approve requisitions (e.g. administrator)", instead of: "Choose the Family Planning program" or "Knowing the password of srmanager2". Information on user names, roles and rights is available here. Providing example test data can be especially helpful for users who are not very familiar with the system. One also has to remember to include the test data that are indispensable for the execution of the test case in the "Test Data" column for a given step. In principle, all data that are necessary to execute the test case should be included in it (e.g. mock-ups): One should write the test case in such a way that it's not necessary to go to the tested ticket in order to test it.

Ideally, a test case should contain up to 40 steps at most. One can usually diminish their number by describing test actions in a more general manner, e.g.: "Approve the requisition", instead of writing: "Go to Requisitions > Approve", and describing the rest of the actions necessary for the achievement of the desired goal. Adding suitable pre-conditions, such as: "Initiating, submitting and authorizing a requisition", instead of adding steps describing all of these actions in detail, also results in a shorter test case. Ideally, one should include all actions that are not verifying a given feature/bug fix in the pre-conditions. If it is not possible to keep the test case within the above-mentioned limit, one should consider creating more than one for a given ticket. Sometimes, splitting the testing of the ticket into more than one test case will prove impossible, though.

If this won't result in a too long test case, one should include both the test steps describing positive testing (happy path scenario) and those concerning negative testing (edge case/unhappy path scenario) in one test case. A happy path scenario or positive testing consists in using a given feature in the default, most usual way, e.g. when there is a form with required and optional fields, one can first complete all of them or only the required ones and check what happens when one tries to submit it. An edge case or negative testing will consist, in this example, in checking what happens when trying to save the form when one or more of the required fields are blank. It is advisable to test the happy path first and then, the edge cases, as the happy path is the most likely scenario and will be most-frequently followed by users. Edge cases are the less usual and, frequently, not obvious scenarios, which are not likely to occur often but still need to be tested in order to prevent failures in the application from happening. They are taken into account because of software complexity (many possible variants of its utilization and thus situations that might occur), and because users, like all people, vary and can make use of the application in different ways.

If it proves impossible to contain all workflows in one test case, the happy path and the edge case(s) have to be described in separate ones, e.g. there should be one test case describing the testing of the happy path and one for each edge case. One also has to write separate test cases if the happy path and the edge case(s) contain contradictory steps or pre-conditions; e.g. the former concerns approving a newly-created requisition and the latter e.g. approving an already-existing old one, not meeting the currently-tested, new requirement.

Finally, one needs to remember that also the expected results have to be specific but not too specific: For instance, it is not recommended to provide exact text of messages and notifications. As an example of this, one shouldn't write e.g.: "The <<Requisition has been deleted!>> notification should appear". Instead, one should write: "The notification that the requisition had been deleted should appear".

Updating a Test Case for a Bug

When one has to test a bug, one needs to browse through the existing test cases to check whether one that covers the actions performed when testing the bug fix already exists. In virtually all cases, this will be the case. If it is so, one needs to update the test case if necessary and link it to the ticket with the bug. Writing new test cases for bugs is not recommended and has to be avoided. Some bugs are found during the execution of already-existing test cases. In such a situation, the test case will already be linked to the ticket with the bug. One then needs to review it, and if there is a need for it, update it. In most cases, this will not be necessary. Note that sometimes, the bug may in fact not be related to the test case during the execution of which it had been discovered, or it might occur by the end of a given test case, and the preceding steps might not be necessary in its reproduction. In both of these situations, one needs to find the test case that is most likely to cover the feature that the bug concerns and update it accordingly. One also has to keep in mind that test cases have to support and test the functionality, as well as provide a way to ensure that the bug had been fixed.

If the bug wasn't found during the execution of any test case, one still needs to check whether there is one containing steps enabling one to reproduce the bug. In order to do so, one needs to go to "Tests > Search Tests" in the top menu on Jira. Then, one is able to browse through all test cases in the project. Since there might be many of them, it is advisable to use a filter. The first option is to enter word(s) that in one's opinion might occur in the test case in the "Contains text" input field and press "Enter". The second one is clicking on "More" and choosing the criteria. Those that are most likely to prove helpful are "Label" and "Component". One can also use already-existing global filters, which can be found by choosing the "Manage filters" option from the main Jira menu. Then, it'll be possible to search for and use the desired filter(s). They include e.g. "Administration tests" or "Stock Management test cases", and might prove especially useful when searching for test cases.

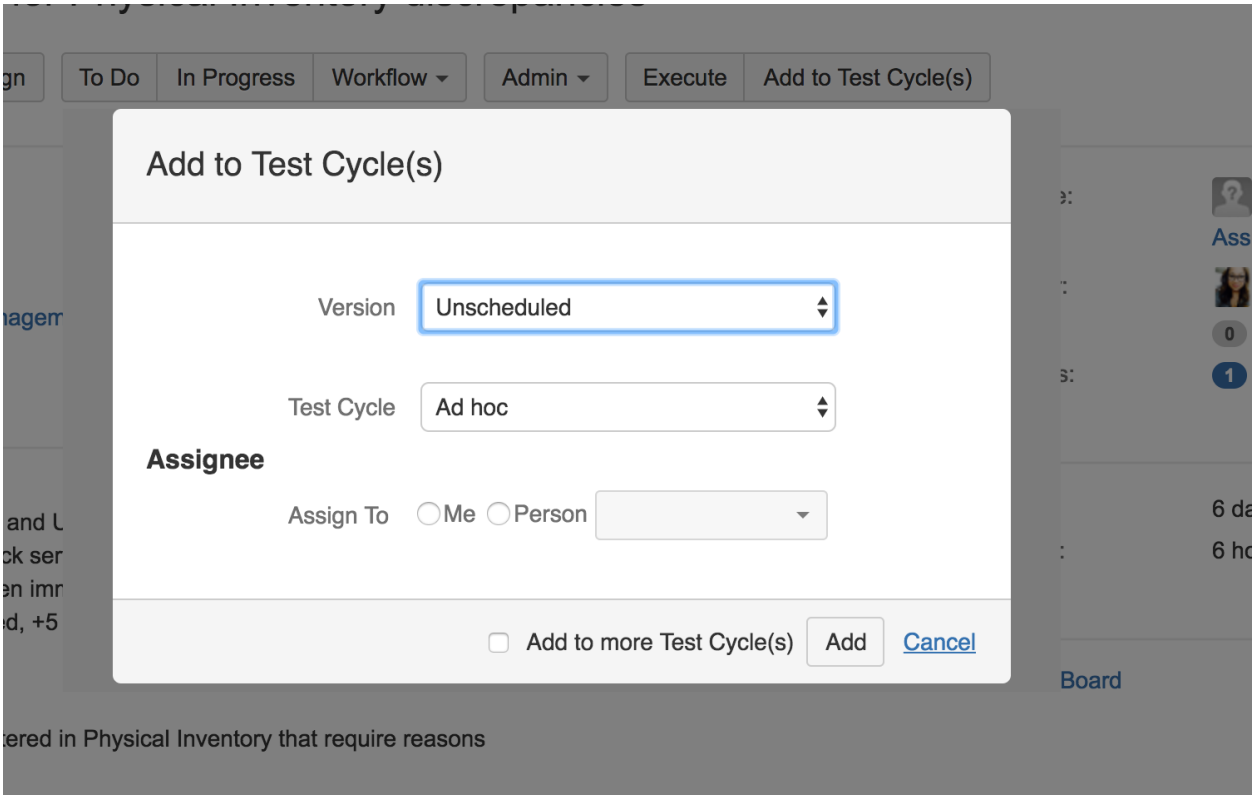
Exploratory Testing

When one is familiarizing oneself with the project or when one already knows the application but there are no tickets currently to test, one can perform exploratory testing. This kind of testing is not related to any ticket. It consists in testing the application without any previous plan, exploring it, in a way. It can be also considered as a form of informal regression testing, as frequently, bugs resulting from regression are found during it. While performing this kind of tests, it is advisable to be as creative as possible – to experiment with testing techniques and test steps, and not to follow the happy path but the edge cases, or to try to find the latter.

Exploratory testing in the UI will focus on testing edge cases and causing errors related to the following categories: Functional issues, UI to server failure, configuration issues, visual inconsistencies, and presentational issues.

Step 2: Test Cycles

Once the test case is written, it needs to be added to a test cycle. Click the "Add to Test Cycle(s)" button, and it will bring you to this screen:



The version should be the version that is associated with the JIRA ticket. The test cycle will be either the current sprint test cycle, or the current feature regression test cycle. If you are executing the test or know who will be executing it, you can assign it here.

Typically, the QA lead or someone designated will create the test cycles for each sprint and they will only need to be linked. If there are no test cycles to select, then these are the fields you must enter to create a new test cycle. The following are two examples of test cycles created for a sprint. The test cycles must have the version, name, description, and environment because they are used in queries for reporting and tracking test metrics.

e.g.

```
* Version - 3.2
* Name - Test Cycle for Sprint 31
* Description - Test Cycle for Sprint 31
```

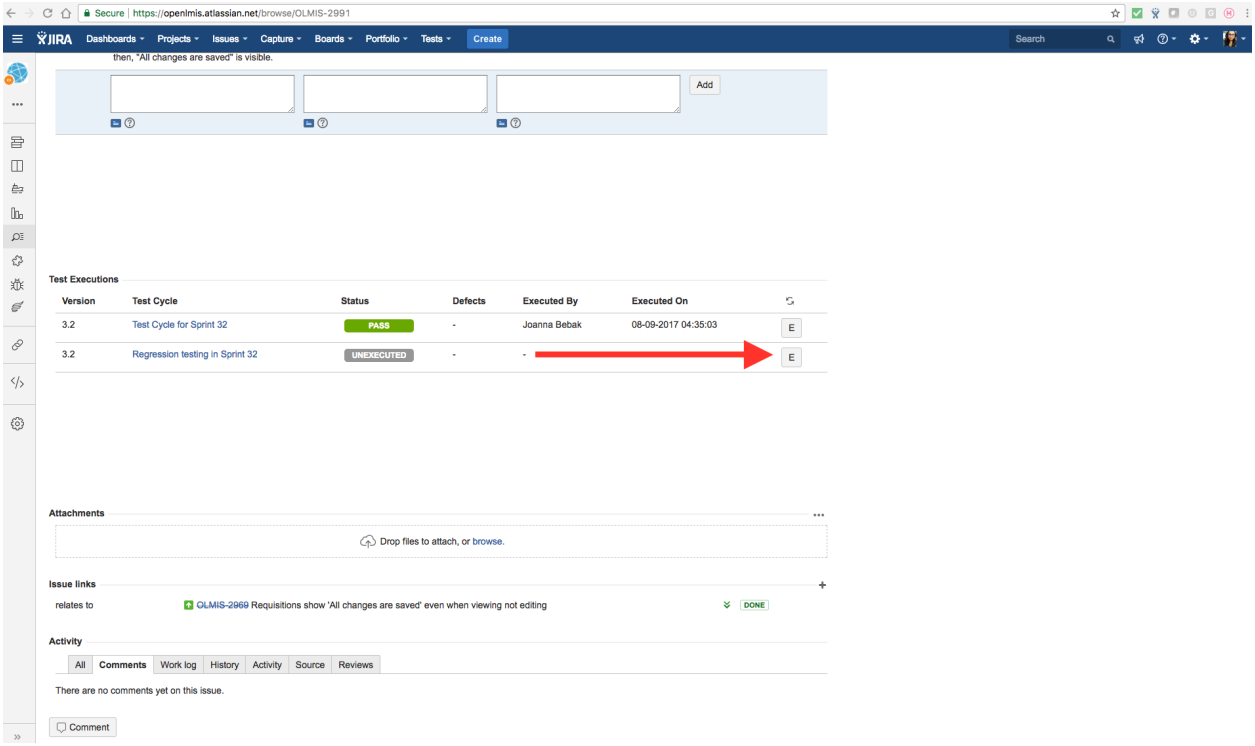
(continues on next page)

(continued from previous page)

```
* Environment - test.openlmis.org
* From (not required) - 2017-07-19
* To (not required) - 2017-08-02
```

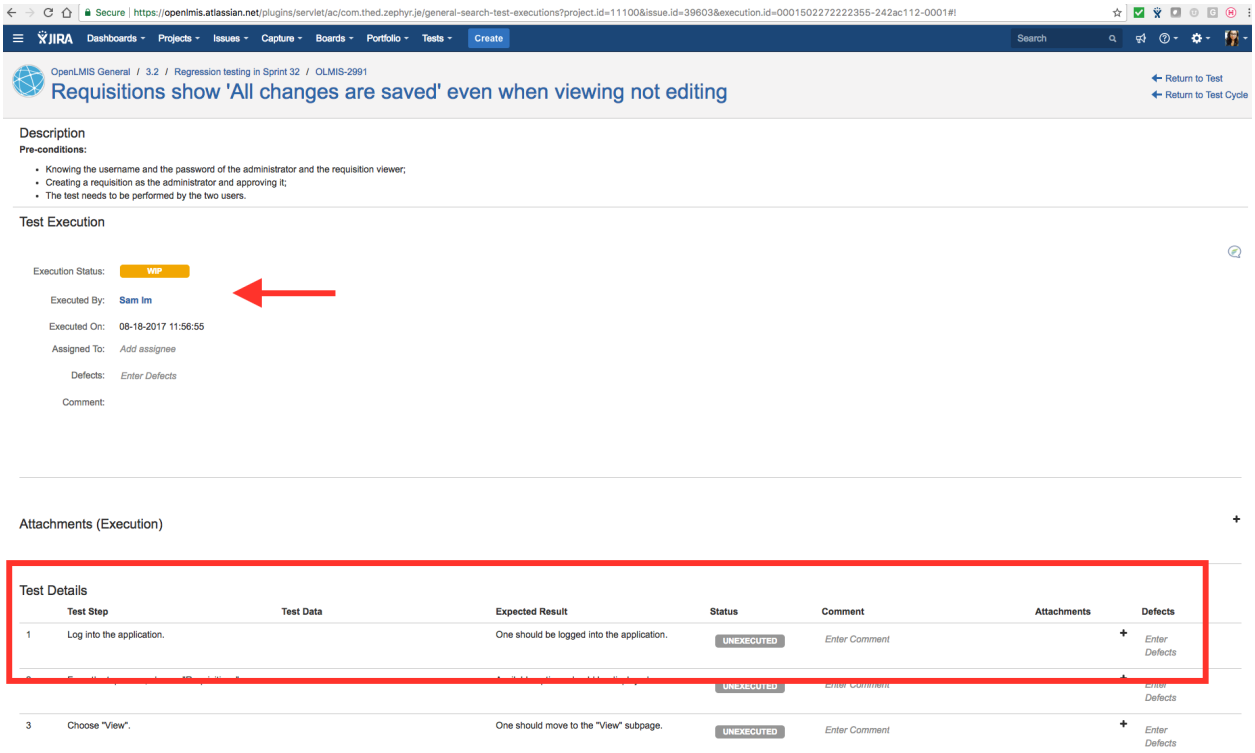
Step 3: Execute Tests within a Test Cycle

Once test cases have been assigned to a test cycle, their execution is tracked in Zephyr. When a test is ready to be executed, open the test case and navigate to the bottom of the page where the "Test Execution" links are shown. Click the "E" button to begin execution.



The "Test Execution" page appears that details the test case and each test step. Select the "WIP" test execution status, which means that the test case is in progress. Zephyr will assign you as the person executing the test and automatically assign the start date and time. Assign the test execution to yourself, and you are ready to begin testing. Each step has "Status", "Comment", "Attachments" and "Defects" fields.

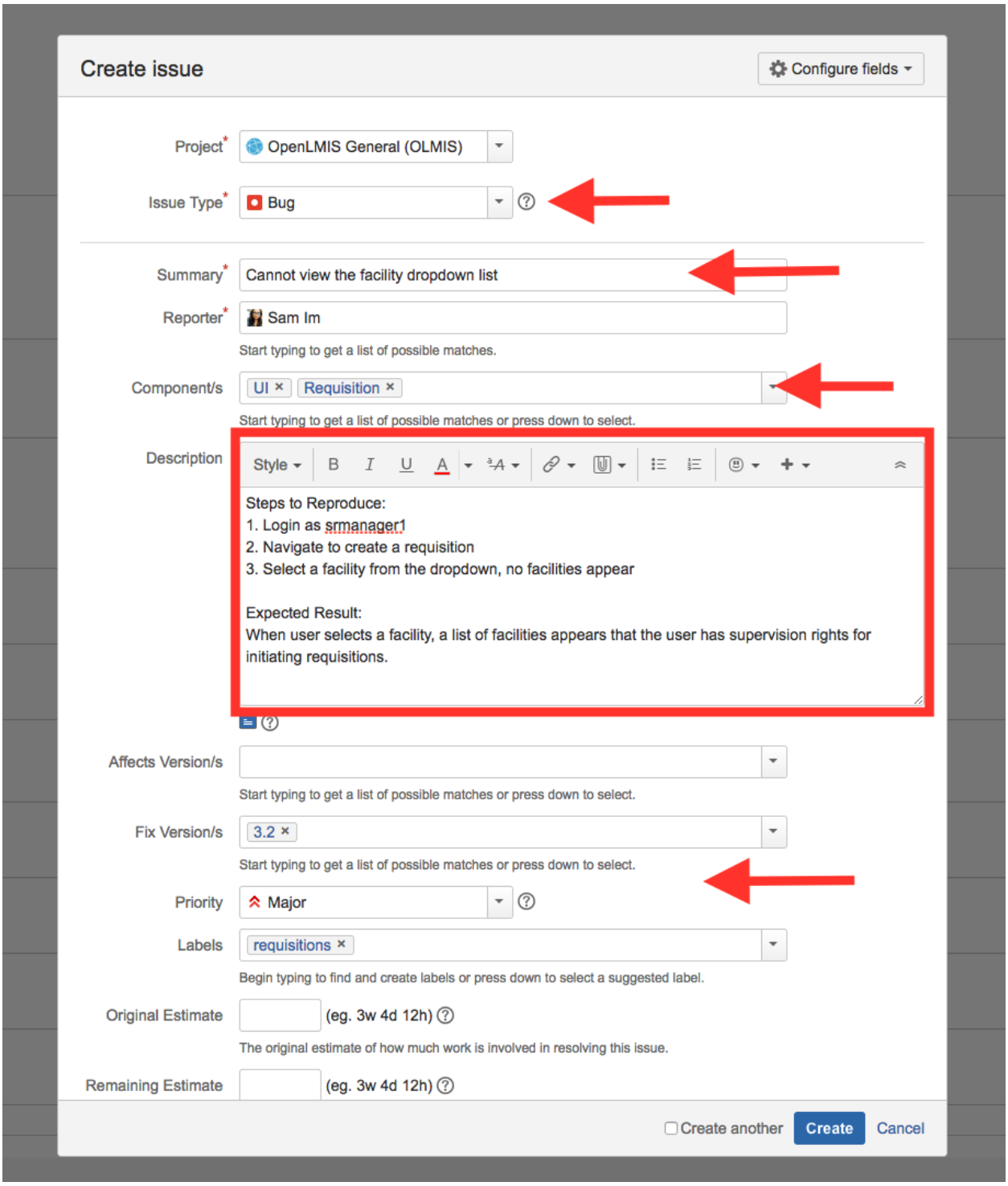
While completing each step, if the expected result matches the actual one, change the status to "Pass". If the step execution does not match expectations, then the status is "Failed". If for some reason the step cannot be executed, such as the system is down, then the status would be "Blocked". Once the test is completed, its status can be updated to reflect whether it passed or failed, and the status of the test execution will be saved in the test case as shown on the above screenshot. This status also appears in the Test Metrics Dashboard.



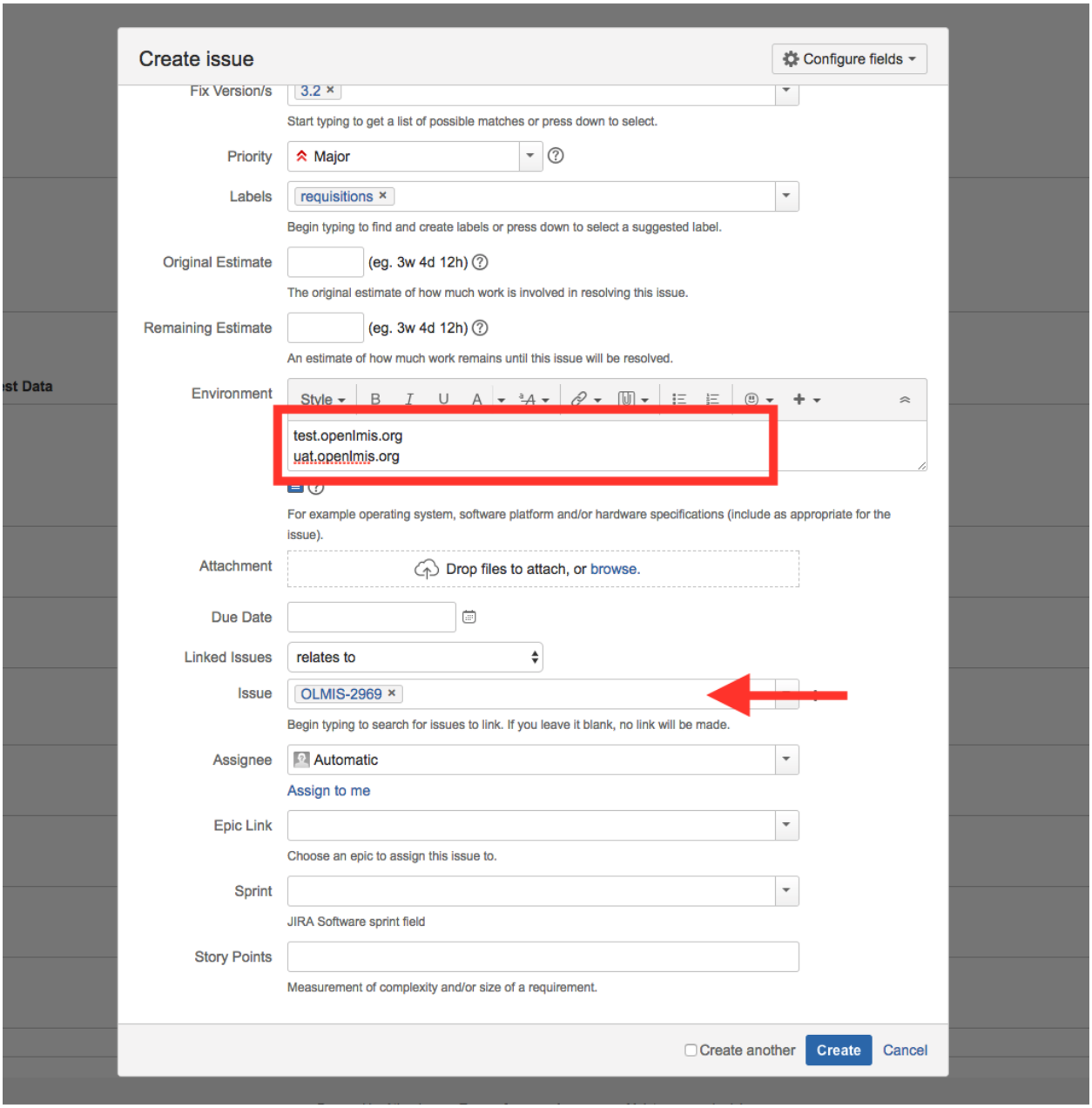
Step 4: Enter Bugs

During testing, if a test step fails and there is a different result, or an error appears that is not expected, then a bug must be entered. Click on the “Defects” column and “Create New Issue”.

The issue type is “Bug”, and the summary, description, priority, environment, and the original linked JIRA ticket should all be added. The summary is a short description of the bug, while the description is a detailed step by step explanation of how to recreate the bug, and what the expected result is per the test case step. It’s helpful to have the test case open in a separate window so that you can copy those details if needed. The environment should be either test or UAT, and you should provide as much detail about the environment that would help the developer when reproducing the bug, such as in which browser you tested.



The new bug should be linked to the JIRA ticket that is linked to the test case.



When a bug is created, it will automatically get sent to the "Roadmap" status. It should stay in this status until it has been triaged and reproduced – only then its status can be changed to "To Do" and when it happens, the bug becomes visible in the product backlog. In summary, each of these steps completes the QA workflow process for a sprint.

Test Coverage: For each sprint's test cycle, the QA lead must assign appropriate test cases so that they can be executed during the sprint. These test cases will be selected based on the tickets assigned to the sprint after the Sprint Planning. The QA lead must determine if test cases are missing for the tickets assigned to the sprint, and create those test cases to ensure complete test coverage. New features require new test cases, while bugs or regression test cycles may rerun existing ones.

Sprint Grooming and Planning Workflow

The tester is responsible for adding test cases to support all testing within the sprint. At the end of each sprint in which regression testing was performed, the QA lead is expected to showcase the results. This should include: Showing the test cycle, the execution metrics, # of added test cases and the final status of bugs found in the sprint (bug tracking).

Bug Tracking

It is important to track the bugs introduced during each sprint. This process helps to identify test scenarios that may need more attention or business process clarification. Bug tracking also helps to identify delays in ticket completion.

- When a test case fails, a bug must be created and linked to the ticket. This bug has to be resolved before the ticket can be marked as done;
- The same test case associated with the sprint's test cycle should be run;
- If a bug is created for the test case, it has to be resolved before the test is executed again in the sprint cycle.

Bug Triage During the Sprint

The bug triage meeting is held twice every sprint, i.e. once a week. Before the meeting, bugs with the "Roadmap" status are attempted to be reproduced and if they still occur, their status is changed to "To Do". If not, they are moved to "Dead". During the meeting, the bugs reported are analyzed in terms of their significance to the project, their final priority is set and the acceptance criteria are updated.

Manual Testing Workflow

When the developer finishes implementing the ticket, he/she should change its status from "In Progress" to "QA". The tester should then check whether all acceptance criteria are correct and up-to-date.

The main manual testing workflow consist of the following steps:

1. The tester starts testing when the ticket is moved to the "QA" column on the Jira sprint board, or when the ticket's status changes to "QA". There is no automatic notification for this until the ticket is assigned, so the tester has to check manually or the developer has to notify the tester.
2. The tester deploys changes to the test environment (uat.openlmis.org).
3. The tester creates a test case, or executes an existing one that is linked to the ticket. The tester should test all acceptance criteria in the ticket. The test case has to be associated with the current sprint test cycle.
4. If bugs are found, the tester should change the ticket's status to "In Progress" and assign it to a proper developer.
    - When a bug is created, it has to be linked to the ticket;
    - The tester notifies the developer that the ticket was moved back to "In Progress".
5. When everything works properly, the tester should change the status to "Done" and assign the ticket to the developer who worked on it.

During the testing process, the tester should write comments in the ticket about the tested features and add the screenshots depicting the noticed issues.

Workflow Between the QA Teams

To improve the testing process, testers should help each other:

- If one of the team members doesn't have anything to do in their own team, they should assist other teams in testing;
- If any tester needs help with testing, they can report it on the QA Slack channel.

Ticket Workflow and Prioritization

In the OpenLMIS project, tickets may have the following statuses:

- "Roadmap": Tickets intended for implementation but not ready for it; e.g. for which the acceptance criteria or other details still need to be defined;
- "To Do": Tickets ready for implementation;
- "In Progress": Tickets currently being implemented or returned to the developer in order to fix bugs;
- "In Review": Tickets in code review;
- "QA": Tickets which should be tested;
- "Done": Tickets which work correctly and are finished.

Ticket priorities are described in "Prioritizing Bugs".

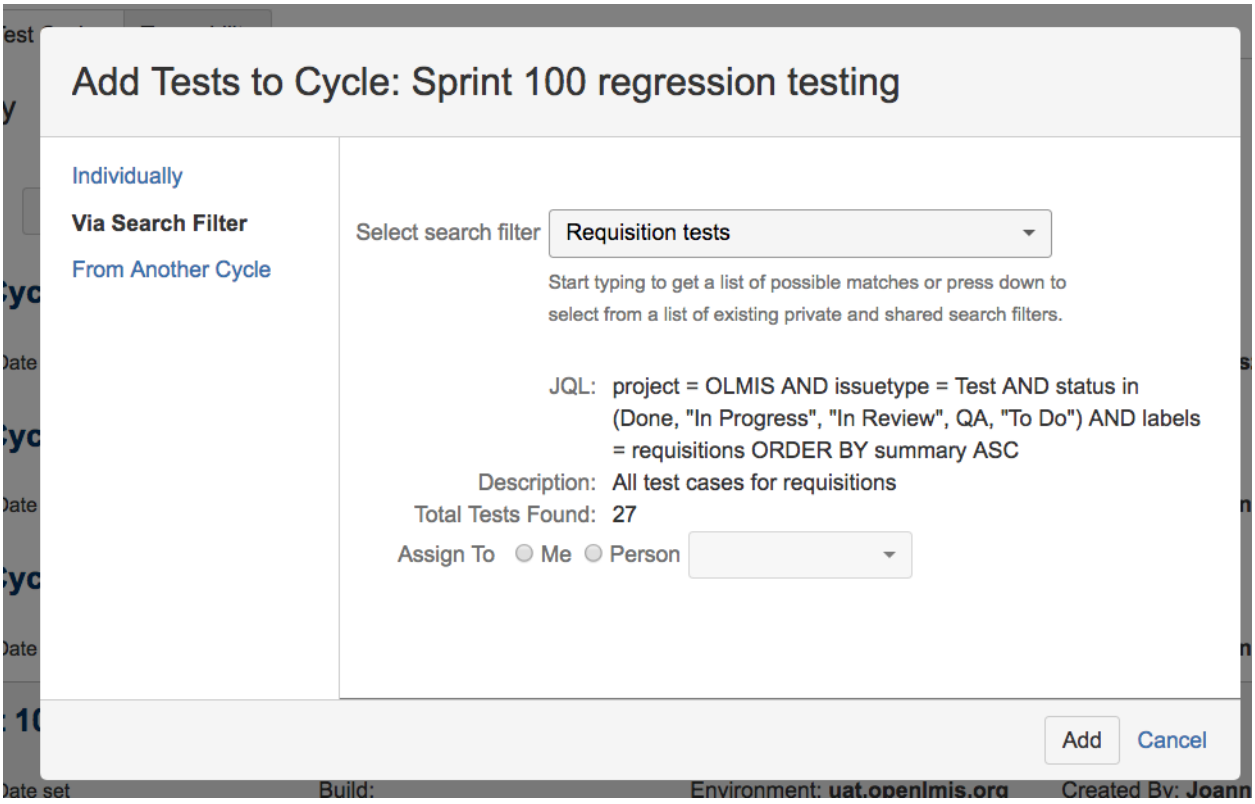
It is very important to start testing tickets with the "Blocker" and "Critical" priority first. When the tester is in the middle of the testing process for a task with lower priority (i.e. "Major", "Minor" or "Trivial"), and a task with higher priority (i.e. "Blocker" or "Critical") returns with changes to the "QA" column, the tester should complete testing the task with lower priority as soon as possible and begin testing the task with higher priority.

Additionally, tickets on the Sprint dashboard in the "QA" column should be prioritized. Tickets with high priority (i.e. "Blocker" or "Critical") should be located at the top of the "QA" column. Every morning, the QA lead should set the tasks in the "QA" column in correct order.

Regression Testing Workflow

Regression testing for OpenLMIS is organized by each of the features (Requisition, Stock Management, Fulfillment, CCE, Administration, etc.) and the test cases are labeled with the feature's name in order to facilitate the planning of regression testing cycles. The tests are managed via the creation of a regression test cycle and assigning it to the sprint. The test cases that are labeled with the feature should include workflow scenarios for the feature, as well as any edge cases that have been identified as required for regression testing.

When creating the regression test cycle, the QA lead will search for test cases with the feature label and assign all of them to the regression test cycle.



Regular manual regression should be executed every second sprint (i.e. once a month). This kind of regression testing is focused on specific services, it doesn't entail testing the entire system. Testers decide what services the regular, focused regression testing is to cover during the QA meetings, and their decision is based on the most-recent changes in the system. For instance, if many changes had been recently introduced in requisitions, the regression testing will consist in the execution of all test cases concerning the requisition service.

Big, full manual regression, consisting in manual tests of the whole system and the execution of all valid test cases, is held one or two weeks before the Release Candidate process. The latter is further described in the "Versioning and Releasing" document.

When bugs are found during regression testing, they are labeled with the "Regression" label. The regression-testing-related bugs are usually considered either "Critical" or "Blocker" ones that must be resolved in the current or in the next sprint. These bugs are also reviewed on the weekly bug triage meeting for completeness, and in order to ensure communication of any concerns to stakeholders.

Workflow for Blocked Tickets

Sometimes during testing, one might come across blocked tickets. This is a situation in which, because of external faults (not related to the content of the task), the tester cannot test a particular ticket. In such a case, one has to change the ticket's status to "In Progress" and assign it to the developer who implemented the ticket. It is important to describe the problem accurately in a comment.

Test Environments

Final tests of all features should be conducted on the UAT server. Some testing may also happen on the test server, in case the UAT server is temporarily unavailable. The UAT server's rebuilding is triggered manually. In order to rebuild

it, and thus perform the test on the latest changes, one has to log into Jenkins and schedule a build of the OpenLMIS-3.x-deploy-to-uat job. One needs to do it when no other jobs are in the build queue, so to to ensure that the server contains the latest changes. After the build is successful, one has to wait for several minutes and then, it is possible to start testing. The test server always contains the latest changes but its use is not normally recommended, as it is very frequently rebuilt (after the rebuilding, all data entered during testing are erased) because it is used for development work. In order to be sure that the server contains the most-recent changes, one has to log into the application and look at the upper-right corner, where information on when the software had last been updated is visible. If only the e.g. "Software last updated on XX/XX/XXXX" text is visible, the software is up to date. As soon as it becomes outdated, information that there are updates available appears next to the text concerning the date of the last update.

Updating Test Data

Before big regression testing, the tester has to update the test users and scenarios to support each of the features that are part of the big regression testing. It is also important to note that several test cases entail changing the permissions of demo-data users. Before executing such a test case, one needs to inform others on the #qa Slack channel that one is going to change these permissions. Having executed the test case, one has to restore the user's permissions to the default ones manually or by scheduling a re-deploy to UAT. In both cases, one has to inform users on #qa that one is changing the user's permissions back to normal. In general, one should avoid changing the permissions of demo-data users if it can be avoided. Instead, one can create a new user and assign suitable rights to them.

Translation Testing

While OpenLMIS 3.0 supports a robust multi-lingual tool set, English is the only language officially provided with the product itself. The translation tools will allow subsequent implementations to easily provide a translation to Portuguese, French, Spanish, etc. Test activities should verify that message keys are being used everywhere on the UI and in error messages so that every string can be translated, with no hard coding. The v3 UI is so similar to v2 that testers could also apply the Portuguese translation strings from v2 to confirm that translations apply everywhere in v3.

Performance Testing

Apart from the usual manual testing, each ticket with the "NeedsPerformanceCheck" label has to undergo performance testing. The instructions on how to perform this kind of tests are available in the "How do I record more test runs?" and "Release testing for 3.3" sections on the "Performance Metrics" page, only one has to use the UAT server and the credentials of users available at that instance. One has to pay attention especially to the CPU throttling (**6x slowdown**) and Network (**Slow 3G**) settings, as they are vital in this kind of testing and render the results of different users comparable.

Types of Automated Tests

The following test categories have been identified for use in OpenLMIS. As illustrated in this slide deck, we expect the effort/number of tests in each category to reflect the test pyramid:

1. *Unit*;
2. *Integration*;
3. *Component*;
4. *Contract*;
5. *End-to-End*.

Unit Tests

- Who: Written by the code's author during the implementation.
- What: The smallest unit (e.g. one piece of a model's behavior, a function, etc.).
- When: At build time, should be fast and targeted, it should be possible to run only a part of the test suite.
- Where: Reside inside a service, next to the unit under test. Generally able to access package-private scope.
- Why: To test fundamental pieces/functionality, help guide and document design and refactors, protect against regression.

Unit Test Examples

- **Every single test should be independent and isolated. The unit test shouldn't depend on another unit test.**

  DO NOT:

```
List<Item> list = new ArrayList<>();

@Test
public void shouldContainOneElementWhenFirstElementisAdded() {
  Item item = new Item();
  list.add(item);
  assertEquals(1, list.size());
}

@Test
public void shouldContainTwoElementsWhenNextElementIsAdded() {
  Item item = new Item();
  list.add(item);
  assertEquals(2, list.size());
}
```

- **One behavior should be tested in just one unit test.**

  DO NOT:

```
@Test
public void shouldNotBeAdultAndShouldNotBeAbleToRunForPresidentWhenAgeBelow18() {
  int age = 17;
  boolean isAdult = ageService.isAdult(age);
  assertFalse(isAdult);

  boolean isAbleToRunForPresident = electionsService.isAbleToRunForPresident(age)
  assertFalse(isAbleToRunForPresident);
}
```

  DO:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  int age = 17;
  boolean isAdult = ageService.isAdult(age);
  assertFalse(isAdult);
}
```

(continues on next page)

(continued from previous page)

```
@Test
public void shouldNotBeAbleToRunForPresidentWhenAgeBelow18() {
  int age = 17;
  boolean isAbleToRunForPresident = electionsService.isAbleToRunForPresident(age)
  assertFalse(isAbleToRunForPresident);
}
```

- **Every unit test should contain at least one assertion.**

  DO NOT:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  int age = 17;
  boolean isAdult = ageService.isAdult(age);
}
```

  DO:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  int age = 17;
  boolean isAdult = ageService.isAdult(age);
  assertFalse(isAdult);
}
```

- **Don't make unnecessary assertions. Don't assert mocked behavior, avoid assertions that check the exact same thing as another unit test.**

  DO NOT:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  int age = 17;
  assertEquals(17, age);

  boolean isAdult = ageService.isAdult(age);
  assertFalse(isAdult);
}
```

- **The unit test has to be independent from external resources (i.e. don't connect with databases or servers).**

  DO NOT:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  String uri = String.format("http://127.0.0.1:8080/age/", HOST, PORT);
  HttpPost httpPost = new HttpPost(uri);
  HttpResponse response = getHttpClient().execute(httpPost);
  assertEquals(HttpStatus.ORDINAL_200_OK, response.getStatusLine().getStatusCode());
}
```

- **The unit test shouldn't test Spring contexts. Integration tests are better for this purpose.**

  DO NOT:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"/services-test-config.xml"})
```

(continues on next page)

(continued from previous page)

```
public class MyServiceTest implements ApplicationContextAware
{

  @Autowired
  MyService service;
  ...
    @Override
    public void setApplicationContext(ApplicationContext context) throws␣
↪BeansException
    {
        // something with the context here
    }
}
```

- **The test method's name should clearly indicate what is being tested and what is the expected output and condition. The "should - when" pattern should be used in the name.**

  DO:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  ...
}
```

  DO NOT:

```
@Test
public void firstTest() {
  ...
}

@Test
public void testIsNotAdult() {
  ...
}
```

- **The unit test should be repeatable: Each run should yield the same result.**

  DO NOT:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  int age = randomGenerator.nextInt(100);
  boolean isAdult = ageService.isAdult(age);
  assertFalse(isAdult);
}
```

- **You should remember about initializing and cleaning each global state between test runs.**

  DO:

```
@Mock
private AgeService ageService;
private age;

@Before
public void init() {
```

(continues on next page)

(continued from previous page)

```
  age = 18;
  when(ageService.isAdult(age)).thenReturn(true);
}

@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  boolean isAdult = ageService.isAdult(age);
  assertTrue(isAdult);
}
```

- **The test should execute fast. When we have hundreds of tests, we don't want to wait several minutes until all tests pass.**

  DO NOT:

```
@Test
public void shouldNotBeAdultWhenAgeBelow18() {
  int age = 17;
  sleep(1000);
  boolean isAdult = ageService.isAdult(age);
  sleep(1000);
  assertFalse(isAdult);
}
```

Integration Tests

- Who: The code's author during the implementation.
- What: Test basic operation of the service to persistent storage or a service to another service. When another service is required, a test double should be used, not the actual service.
- When: As explicitly asked for, these tests are typically slower and therefore need to be kept separate from the build in order not to slow the development down. Are run on the CI on every change.
- Where: Reside inside a service, separated from other types of tests/code.
- Why: Ensure that basic pathways to a service's external run-time dependencies work, e.g. that a db schema supports the ORM, or that a non-responsive service call is gracefully handled.

As far as testing controllers is concerned, tests are divided into unit and integration tests. Controller unit tests test the logic in the controller, and integration tests test serialization/deserialization (and therefore do not need to test all code paths). In both cases, the underlying services and repositories are mocked.

Component Tests

- Who: The code's author during the implementation.
- What: Test more complex operations in the service. When another service is required, a test double should be used, not the actual service.
- When: As explicitly asked for, these tests are typically slower and therefore need to be kept separate from the build to not slow the development down. Will be run on the CI on every change.
- Where: Reside inside a service, separated from other types of tests/code.
- Why: Test whether interactions between components in the service work as expected.

These are not integration tests, which strictly test the integration between the service and an external dependency. These test whether the interactions between the components in the service are correct. While integration tests verify only whether the basic pathways work, component tests verify whether, based on the input, the output matches expectations.

They are not contract tests, which are more oriented towards business requirements, but are more technical in nature. The contract tests will make certain assumptions about components, and these tests make sure those assumptions are tested.

Contract Tests

- Who: The code's author during the implementation, with the BA/QA's input.
- What: Enforce contracts between and to services.
- When: Run on the CI.
- Where: Reside inside a separate repository: openlmis-contract-tests.
- Why: Test multiple services working together, testing contracts that the service both provides, as well as the requirements a dependency has.

Ideally, a single scenario checks a single endpoint. In specific cases, like the requisition workflow, when it's impossible to check something without using other endpoints, it can be omitted.

The main difference between contract and integration tests: In contract tests, all services under test are real, meaning that they will be processing requests and sending responses. Test doubles, mocking, stubbing should not be part of contract tests.

Contract tests should follow the convention below:

- The scenario should be like <*user_name*> *should be able to* <*action_description*>;
- The feature's name should consist of infinitive + noun in plural, e.g. *Creating facility type approved products* or testing the FTAP creating screen/endpoint.

```
@FacilityTypeApprovedProductTests
Feature: Creating facility type approved products

  Scenario: Administrator should be able to create FTAP
    Given I have logged in as admin

    When I try to create a FTAP:
    | orderable.id                           | program.id                             |
↪   | facilityType.id                        | maxPeriodsOfStock | minPeriodsOfStock  |
↪| emergencyOrderPoint |
    | 2400e410-b8dd-4954-b1c0-80d8a8e785fc   | dce17f2e-af3e-40ad-8e00-3496adef44c3 |
↪   | ac1d268b-ce10-455f-bf87-9c667da8f060   | 3                 | 1                  |
↪| 1                    |
    Then I should get response with created FTAP's id
    And I logout
```

Refer to this doc for examples on how to write contract tests.

Contract tests should:

- Cover the contract between services:
  - Reasons from Stock Management provided for the requisition service: Stock reasons contract tests;
  - Converting requisitions to orders: Fulfillment contract tests;

  - The dependencies between the user resources in different services: Users contract tests; Contact details contract tests;
  - The information provided to the UI;
- Cover checking the email templates (but appropriate patterns are required for email verification, there are no tests present at the moment);
- Cover checking file upload:
  - ISA values upload.

End-to-End Tests

- Who: Tester/developer with input from the BA.
- What: Typical/core business scenarios.
- When: Run on the CI.
- Where: Reside in separate repository.
- Why: Ensure that all the pieces work together to carry-out a business scenario. Help ensure that end-users can achieve their goals.

A single feature should cover only one (related) UI screen.

End-to-end tests should follow the convention below:

- The scenario should be like *<user_name> should be able to <action_description>*;
- The feature's name should consist of infinitive + noun in plural, e.g. *Adding reasons* for testing the reason adding screen/endpoint.

```
Feature: Adding reasons

    Background:
        Given I have logged with username "administrator" and password "password"
        Given I have navigated to the reason list page

    Scenario: Administrator should be able to add new reason
        When I click on the "Add Reason" button
        Then I should be brought to the reason creation page

        When I select "Family Planning" from the "Program" list
        And I select "Warehouse" from the "Facility Type" list
        And I click on the "Add" button
        Then I should see assignment for "Family Planning" program and "Warehouse"␣
↪facility type

        When I enter "Functional Test Reason" as "Name"
        And I select "Transfer" from the "Category" list
        And I select "Debit" as "Type"
        And I enter "adjustment" as "Tags"
        And I click on the "Add New Reason" button
        Then I should see a successful notification saying "Reason saved successfully"
        And I should see a reason with "Functional Test Reason" name, "Transfer"␣
↪category and "Debit" type inside the table
```

E2E tests should:

- Cover workflow happy paths:

  - Basic requisition workflow;
  - Basic order workflow;
  - Sending stock events.
- Not cover edge cases which require multiple steps from different microservices:
  - Sending data from the requisition to the stockmanagement service.
- Cover checking functionalities dependent on user rights:
  - The presence of elements on the navigation bar;
  - The visibility of action buttons;
  - Editable fields in forms.
- Check UI differences dependent on choosing specific options:
  - How the requisition product grid screen behaves for emergency and regular requisitions;
  - Differences between stock-based and non-stock-based requisitions;
  - Differences between working with home and supervised facility.
- Not check any specific UI details:
  - The order of columns in tables;
  - Exact label names;
  - Exact placement of inputs;
  - Colors of elements if they are not significant.

Testing Services Dependent on External APIs

OpenLMIS uses WireMock to mock web services. An example integration test can be found here.

The stub mappings which are served by the WireMock's HTTP server are available under *src/test/resources/mappings* and *src/test/resources/files*. For instructions on how to create them, please refer to http://wiremock.org/record-playback.html.

Testing Tools

- Cucumber and IntelliJ: For contract tests;
- Browsers: Chrome (the latest version) and Firefox (the latest version);
- REST clients: Postman, API Console and REST Client;
- JIRA: Issue-, task- and bug tracking;
- Zephyr: Test case and test cycle creation;
- Confluence: Project documentation, e.g. various guides and meeting notes;
- Reference Distribution;
- Docker 1.11+;
- Docker Compose 1.6+;
- Spring-boot-starter-test:

  - Spring Boot Test;
  - JUnit;
  - Mockito;
  - Hamcrest;
- WireMock;
- REST Assured;
- raml-tester.

Error Handling Conventions

OpenLMIS would like to follow error handling best practices, this document covers the conventions we'd *like* to see followed in the various OpenLMIS components.

Java and Spring

The Java community has a long-standing debate about the proper use of Exceptions. This section attempts to be pragmatic about the use of exceptions - especially understanding the Spring community's exception handling techniques.

Exceptions in Java are broken down into two categories: those that are recovearable (checked) and those where client code can in no-way recover from the Exception (runtime). OpenLMIS *strongly* discourages the use of checked exceptions, and the following section discusses what is encouraged and why checked exceptions should be avoided.

A pattern for normal error-handling

Normal errors for the purpose of this document are things like input validation or other business logic constraints. There are a number of sources that make the claim that these types of errors are not exceptional (i.e. bad user input is to be expected normally) and therefore Java Exception's shouldn't be used. While that's generally *very* good advice, we will be using runtime exceptions (not checked exceptions) as long as they follow the best practices laid out here.

The reasoning behind this approach is two-fold:

- Runtime exceptions are used when client code *can't* recover from their use. Typically this has been used for the class of programming errors that indicate that the software encountered a completely unexpected programming error for which it should immediately terminate. We expand this definition to include user-input validation and business logic constraints for which further user-action is required. In that case the code can't recover - it has to receive something else before it could ever proceed, and while we don't want the program to terminate, we do want the current execution to cease so that it may pop back to a Controller level component that will convert these exceptions into the relevant (non-500) HTTP response.
- Using Runtime exceptions implies that we *never* write code that catches them. We will use Spring's `@ControllerAdvice` which will catch them for us, but our code should have less "clutter" as it'll be largely devoid of routine error-validation handling.

Effectively using this pattern requires the following rules:

1. The Exception type (class) that's thrown will map one-to-one with an HTTP Status code that we want to return, and this mapping will be true across the Service. e.g. a `throw ValidationException` will always result in the HTTP Status code 400 being returned with the body containing a "nice message" (and not a stacktrace).
2. The exception thrown is a sub-type of `java.lang.RuntimeException`.

3. Client code to a method that returns RuntimeException's should never try to handle the exception. i.e. it should **not** `try {...} catch ...`
4. The only place that these RuntimeExceptions are handled is by a class annotated `@ControllerAdvice` that lives along-side all of the Controllers.
5. If the client code needs to report multiple errors (e.g. multiple issues in validating user input), then that collection of errors needs to be grouped before the exception is thrown.
6. A Handler should never be taking one of our exception types, and returning a HTTP 500 level status. This class is reserved specifically to indicate that a programming error has occurred.Reserving this directly allows for easier searching of the logs for program-crashing type of errors.
7. Handler's should log these exceptions at the DEBUG level. A lower-level such as TRACE could be used, however others such as ERROR, INFO, FATAL, WARN, etc should not.

Example

The exception

```
public class ValidationException extends RuntimeException { ... }
```

A controller which uses the exception

```
@Controller
public class WorkflowController {

  @RequestMapping(...)
  public WorkflowDraft doSomeWorkflow() {
    ...

    if (someError)
      throw new ValidationException(...);

    ...

    return new WorkflowDraft(...);
  }
}
```

The exception handler that's called by Spring should the `WorkflowController` throw `ValidationException`.

```
@ControllerAdvice
public class WorkflowExceptionHandler {
 @ExceptionHandler(ValidationException.class)
 @ResponseStatus(HttpStatus.BAD_REQUEST)
 private Message.LocalizedMessage handleValidationException(ValidationException ve) {
   ...
   logger.debug(ve);
   return ve.getTheLocalizedMessage();
  }
}
```

Exceptions - what we don't want

Lets look at a simple example that is indicative of the sort of code we've been writing using exceptions. This example consists of a web-endpoint that returns a setting for a given key, which hands off the work to an application service layer that uses the key provided to find the given setting.

A controller (HTTP end-point) that is asked to return some setting for a given "key"

```
@RequestMapping(value = "/settings/{key}",  method = RequestMethod.GET)
public ResponseEntity<?> getByKey(@PathVariable(value = "key") String key) {
  try {
    ConfigurationSetting setting = configurationSettingService.getByKey(key);
    return new ResponseEntity<>(setting, HttpStatus.OK);
  } catch (ConfigurationSettingException ex) {
    return new ResponseEntity(HttpStatus.NOT_FOUND);
  }
}
```

The service logic that finds the key and returns it (i.e. configurationSettingService above):

```
public ConfigurationSetting getByKey(String key) throws ConfigurationSettingException
↪{
  ConfigurationSetting setting = configurationSettingRepository.findOne(key);
  if (setting == null) {
    throw new ConfigurationSettingException("Configuration setting '" + key + "' not␣
↪found");
  }
  return setting;
}
```

In this example we see that the expected end-point behavior is to either return the setting asked for and an HTTP 200 (success), or to respond with HTTP 404 - the setting was not found.

This usage of an Exception here is not what we want for a few reasons:

- The Controller directly handles the exception - it has a try-catch block. It should only handle the successful path which is when the exception isn't thrown. We should have a Handler which is `@ControllerAdvice`.
- The exception `ConfigurationSettingException` doesn't add anything - either semantically or functionally. We know that this type of error isn't that there's some type of Configuration Setting problem, but rather that something wasn't found. This could more generically and more accurately be named a `NotFoundException`. It conveys the semantics of the error and one single Handler method for the entire Spring application could handle all `NotFoundExceptions` by returning a HTTP 404.
- It's worth noting that this type of null return is handled well in Java 8's Optional. We would still throw an exception at the Controller so that the Handler could handle the error, however an author of middle-ware code should be aware that they could use Optional instead of throwing an exception on a null immediately. This would be most useful if many errors could occur - i.e. in processing a stream.
- This code is flagged by static analysis tools with the error that this exception should be "Either log or re-throw this exception". A lazy programmer might "correct" this by logging the exception, however this would result in the log being permeated with noise from bad user input - which should be avoided.

How the API responds with validation error messages

What are Validation Error Messages?

In OpenLMIS APIs, validation errors can happen on PUT, POST, DELETE or even GET. When validation or permissions are not accepted by the API, invalid requests should respond with a helpful validation error message. This response has an HTTP response body with a simple JSON object that wraps the message. Different clients may use this message as they wish, and may display it to end-users.

The Goal: We want the APIs to respond with validation error messages in a standard way. This will allow the APIs and the UI components to all be coded and tested against one standard.

When does this pattern apply?

When does this "validation error message" pattern apply? We want to apply this pattern for all of the error situations where we return a HTTP response body with an error message. For more details about which HTTP status codes this aligns with, see the 'HTTP Status Codes' section below.

What do we return on Success?

In general, success responses should not include a validation message of the type specified here. This will eliminate the practice which was done in OpenLMIS v2, EG:

```
PUT /requisitions/75/save.json
Response: HTTP 200 OK
Body: {"success":"R&R saved successfully!"}
```

On success of a PUT or POST, the API should usually return the updated resource with a HTTP 200 OK or HTTP 201 Created response code. On DELETE, if there is nothing appropriate to return, then an empty response body is appropriate with a HTTP 204 No Content response code.

HTTP Status Codes

Success is generally a 2xx HTTP status code and we don't return validation error messages on success. Generally, validation errors are 4xx HTTP status codes (client errors). Also, we don't return these validation error messages for 5xx HTTP status codes (server or network errors). We do not address 5xx errors because OpenLMIS software does not always have control over what the stack returns for 5xx responses (those could come from NGINX or even a load balancer).

Examples below show appropriate use of HTTP 403 and 422 status codes with validation error messages. The OpenLMIS Service Style Guide includes further guidance on HTTP Status Codes that comes from Best Practices for Designing a Pragmatic RESTful API.

Example: Permissions/RBAC

The API does a lot of permission checks in case a user tries to make a request without the needed permissions. For example, a user may try to initiate a requisition at a facility where they don't have permissions. That should generate a HTTP 403 Forbidden response with a JSON body like this:

```
{
  "message" : "Action prohibited because user does not have permission at the facility
↪",
  "messageKey" : "requisition.error.prohibited.noFacilityPermission"
}
```

When creating these error validation messages, we encourage developers to avoid repeating code. It may be appropriate to write a helper class that generates these JSON validation error responses with a simple constructor.

We also don't want developers to spend lots of time authoring wordy messages. It's best to keep the messages short, clear and simple.

Translation/i18n

Message keys are used for translations. Keys should follow our Style Guide i18n Naming Conventions.

The "messageKey" is the key into a property translation file such as a .properties file maintained using Transifex or a similar tool.

The "messageKey" will be used with translation files in order to conduct translation, which we allow and support on the server-side and/or the client-side. Any OpenLMIS instance may configure translation to happen in its services or its clients.

A service will use the "messageKey" to translate responses into a different language server-side in order to respond in the language of choice for that OpenLMIS implementation instance. And/or a client/consumer may use the "messageKey" to translate responses into a language of choice.

The source code where a validation error is handled should have the "messageKey" only. The source code should not have hard-coded message strings in English or any language.

Messages with Placeholders for Translation

Placeholders allow messages to be dynamic. For example, "Action prohibited because user {0} does not have permission {1} at facility {2}".

The Transifex tool appears to support different types of placeholders, such as {0} or %s and %d. In OpenLMIS v2, the MessageService (called the Notification Service in v3) uses placeholders to make email messages translate-able. For an example, see the StatusChangeEventService.

Multiple errors in response

When validation is not accepted, we want to use the top level error message with section below with multiple field errors. Every field error in response should contain message key and message for specific field rejected by validator. Field errors can be nested. Instead of arrays, map should be returned with rejected field name as a key. When field is an element of array, resource identifier should be used as the key, such as UUID or code.

```
{
  "message": "Validation error occurred",
  "messageKey": "requisition.error.validation.fail",
  "fieldErrors": {
    "comment": {
      "message": "Comment is longer than 255 characters and can not be saved",
      "messageKey": "requisition.comment.error.invalidLength"
    },
```

(continues on next page)

(continued from previous page)

```
    "requisitionLineItems": {
      "0c4b5efe-259c-44c9-8969-f157f778ee0f": {
        "stockOnHand": {
          "message": "Stock on hand can not be negative",
          "messageKey": "requisition.error.validation.stockOnHand.cannotBeNegative"
        }
      }
    }
  }
}
```

Future: Arrays of Messages

In the future, we may extend these guidelines to support an array of multiple messages.

Future: Identifying Fields Where Validation Was Not Accepted

In the future, it may also be helpful to extend this to allow the error messages to be associated with a specific piece of data. For example, if a Requisition Validation finds that line item quantities do not add up correctly, it could provide an error message tied to a specific product (line item) and field. Often this kind of validation may be done by the client (such as in the AngularJS UI app), and the client can immediately let the end-user know about a specific field with a validation error.

Future: Including Stack-Traces in Development Mode

In the future, it may be useful to be able to launch the entire application in a debug mode.In this mode errors returned via the API might include a stacktrace or other context normally reserved for the server log. This would be a non-default mode that developers could use to more easily develop the application.

Proposed RAML

```
schemas:
  - localizedErrorResponse: |
      {
        "type": "object",
        "$schema": "http://json-schema.org/draft-04/schema",
        "title": "LocalizedErrorResponse",
        "description": "Localized Error response",
        "properties": {
          "message": { "type": "string", "title": "error message" },
          "messageKey": { "type": "string", "title": "key for translations" },
          "fieldErrors": {
            "type": "object",
            "title": "FieldErrors",
            "description": "Field errors"
          }
        },
        "required": ["messageKey", "message"]
      }
```

(continues on next page)

(continued from previous page)

```
/requisitions:
  /{id}:
    put:
      description: Save a requisition with its line items
      responses:
        403:
        422:
          body:
            application/json:
              schema: errorResponse
```

Service Health

In OpenLMIS' Service Architecture it's important that a Service be able to tell our Service Registry (Consul) when it's ready to accept new work and when it's not. If the service doesn't inform our Service Registry accurately, then new requests for work might be routed to that service from the reverse proxy (Nginx) which won't be fulfilled.

Spring Boot Actuator

In our Spring Boot based services there is a very handy project named Spring Boot Actuator that once enabled turns on a number of useful production features. One of these is the `/health` endpoint.

To make use of this in OpenLMIS v3 architecture we will:

1. Add Spring Boot Actuator to our Service.
2. Enable the `/health` endpoint.
3. Register this endpoint with Consul as a health check.

Adding Spring Boot Actuator to our Service

As simple as adding it as a dependency:

build.gradle:

```
dependencies {
  ...
  compile "org.springframework.boot:spring-boot-starter-actuator"
  ...
}
```

Enabling the `/health` endpoint

May be done through our default configuration:

application.properties:

```
endpoints.enabled=false
endpoints.health.enabled=true
```

Note that we first disable all of the endpoints that Spring Boot Actuator adds to be conservative, we don't need them (yet). Next we ensure that the `/health` endpoint is enabled.

Registering `/health` with Consul (Service Registry)

First we must allow non-authenticated access to this resource:

ResourceServerSercurityConfiguration.java:

```
.antMatchers(
    "/referencedata",
    "/health",
    "/referencedata/docs/**"
).permitAll()
```

Next we need to tell Consul that this endpoint should be used for a health check:

config.json:

```
"service": {
  "Name": "referencedata",
  "Port": 8080,
  "Tags": ["openlmis-service"],
  "check": {
    "interval": "10s",
    "http": "http://HOST:PORT/health"
  }
},
```

This Consul check directive will be registered with Consul, letting Consul know that every 10 seconds it should try this `/health` endpoint and use the HTTP status to determine the Service's availability.

And finally we'll need to ensure that the registration script replaces `HOST` and `PORT` with the correct values when it sends this to Consul:

consul/registration.js:

```
function registerService() {
  service.ID = generateServiceId(service.Name);

  if (service.check) {
    var checkHttp = service.check.http;
    checkHttp = checkHttp.replace("HOST", service.Address);
    checkHttp = checkHttp.replace("PORT", service.Port);
    service.check.http = checkHttp;
  }

  ...
}
```

This commit has the change.

At this point you might be wondering why we left this endpoint unsecured and not mapped to some name which is service specific. After all, every running service will use `/health`. What we did not do however is make this endpoint routable by adding it to our RAML or registering it as a path for Consul. This means that our reverse proxy will never try to take a HTTP request to `/health` and route it to any particular service. Only Consul will know of this endpoint and try to access it through the network at the host and port which the Service registered itself with. No client to our reverse proxy will be able to directly access a Service's health endpoint.

Health and HTTP Status

The Consul check directive is looking for the following HTTP statuses:

- 2xx: Everything is okay, send more requests
- 429: Warning, too many requests. There is a problem, but still send more requests.
- Anything else: failed, not available for servicing requests

The `/health` endpoint naturally fulfills HTTP `200` when the Service is ready and also has the basics of how to report when a service is down (e.g. if the database connection is down the endpoint will return a 5xx level error). This endpoint can do more however. Spring Boot Actuator Health Information has more details about how custom code can be written that modifies the health status returned. This could be especially useful if a Service has a dependancy on another system (e.g. integration with ODK or DHIS2), another Service (e.g. Requisition needs Reference Data) or another piece of infrastructure (e.g. sending emails, SMS, etc).

1.5.5 UI Conventions

See the UI Styleguide for conventions about how components look and function. See the Reference UI section under *Components* to learn about the UI architecture, how to build and extend/customize.

UI Label Conventions

The following document outlines how content, labels and messages should be displayed in the OpenLMIS-UI. This guide presents generalizations for how labels should be written and complex workflows should be organized.

Content Conventions

The following are general stylistic rules for the OpenLMIS-UI, which implementers and developers should keep in mind while crafting content.

Titles

Titles include page titles, report titles, headings within a page (H2, H3, etc), and the subject line of email notifications. Links in the main navigation menu are generally page titles. Most other strings that appear on-screen are Labels, Buttons or others described further below.

Titles should be written so they describe a specific object and state. If there is a state that is being applied to the object in a title, the state is first in the present tense. The first letter of each word in a title should be capitalized, except for the articles of the sentence. Titles do not contain punctuation.

See APA article about title case for more guidance.

Examples Do: “Initiate Requisition” Do Not: “REQUISITION - INITIATE”

Labels

Labels are generally used in form elements to describe the content a user should input. Labels have the first letter of the first word capitalized, and should not have any punctuation such as a colon.

Labels also include table column headers and dividers for sections or categories.

Note: Colons should be added using CSS pseudo-selector, if an implementation requires labels to be formatted with a colon. As a community, we feel that less allows for easier customization.

Example Do: "First name" Do Not: "First Name:"

Buttons

Buttons should be used to refer to a user taking an action on an object, meaning there should always be a specific verb followed by a subject. Buttons have the first letter of each word capitalized and don't have any punctuation.

Example Do: "Search Facilities" Do Not: "SEARCH"

Messages

Messages represent a response from the system to a user. These strings should be written as a command, where the first word is the action that has happened. The first letter of a message is capitalized, but there is no punctuation.

Example Do: "Failed to save user profile" Do Not: "Saving user profile failed."

Confirmations

Confirmations are messages shown to the user to confirm that they actually want to take an action. These messages should address the user directly and be phrased as a single sentence.

Example Do: "Are you sure you want to submit this requisition?" Do Not: "Submitting requisition, are you sure? Please confirm."

Instructions

Instructions might be placed at the top of a form or after a confirmation to clarify the action a user is taking. These should be written as full paragraphs.

Example Do: "Authorize this requisition to send the requisition to the approval workflow." Do Not: "Authorize requisition — send to approval workflow"

Information Architecture

In the context of the OpenLMIS-UI, information architecture refers to how a person finds and edits data by navigating between screens and states. This document provides guidelines used in the OpenLMIS-UI, and while it is preferential to stick to these guidelines, there will be exceptions. *Please document why exceptions have been made.*

The OpenLMIS-UI uses a shallow information architecture, meaning each screen should have a single focused goal for a person managing logistical information. For example, there is an "Approve Requisitions" screen, where the only requisitions that are displayed are requisitions that need to be approved that the current user has permissions to approve. By keeping the information architecture of the OpenLMIS-UI shallow, we hope to provide a user experience that is efficient.

To support our shallow information architecture we:

- Avoid "nested" navigation, meaning we prefer a single long list of pages instead of "folders within folders."
- Use strong defaults, because we don't want to force a user to make lots of choices before getting to work. Ideally a user can navigate to a page and start doing work.

See the OpenLMIS Generic Workflows in Balsamiq for an annotated set of mockups that show and explain these conventions. In addition, see the Mockup Guidelines in the OpenLMIS wiki.

Generic Page Types

The following page types are guidelines for how to discuss the screens and pages that make up workflows that are implemented in the OpenLMIS-UI. Every page type should meet the following rules:

- Each page has a unique URL address
- Each page has a single purpose

List View

A list view is a screen with a paginated list of items from the OpenLMIS Services. A list could be a list of users, products, or orders that need to be fulfilled at a facility.

All list views should:

- Attempt to show the current state of an OpenLMIS Service
- Avoid editing list items directly in the list (editing should be done in a detail or document view)

See the List View in Balsamiq for annotated examples.

Detail View

A detail view most often shows editable details of an item from the proceeding list view. Our recommendation is to show item details inside a model, so a user doesn't lose context of the list.

Detail views should focus on a single set of data or a single action to an object. For example, on the CCE Inventory page, a user is presented with a list view of CCE Inventory items, and from this view there are two separate detail views. The first is a generic view for the history of that CCE Inventory item, while the second is a detail view specifically focused on updating the functional status for the inventory item.

An example mockup for Detail Views is included in the List View in Balsamiq.

Document View

Document views represent a complex item, like a requisition or proof of delivery, and focuses on making these items editable. A document view is generally navigated to from a list view.

Document views should:

- Function when the browser is offline
- Cache all information that is needed on the page so the editing experience is fast and responsive for a user
- Not implement pagination for tables of information, but rather show a long continuous table so the user feels it is a single large document

See the Document View in Balsamiq for annotated examples.

Navigation

In the OpenLMIS-UI, a user generally navigates from screen to screen where:

- Each screen has a unique URL
- The screen is one of our view type (above)

The largest piece of navigation in the OpenLMIS-UI is the header navigation that displays links to specific views and workflows. There are other forms of navigation, like the Program and Facility Navigation (which is detailed below).

Many screens will implement filter and sort controls that will change how information is shown in a view, but doesn't actually represent a navigation change. Currently, in the OpenLMIS-UI it is most common that a list view will implement both a sort and filter control, while a document view will only show a filter control. Filters and sorts are included in the List View in Balsamiq.

Filter Controls

A filter modifies information shown on a page. Filters are always optional, and should never be a primary feature in a screen. If a user is going to accomplish a task, the filter helps the user accomplish the task quicker – but should never be the only way to accomplish the task.

If there is multiple filter criteria, the criterion should be combined using conjunctional logic (ie "AND" not "OR").

If a filter is required, or a primary focus of the entire screen, it should be redesigned to be incorporated into persistent page navigation.

Sort

Sorts refer to how a list of items are ordered within a table or list. Every list should have a sort order presented to the user, so the user can understand how the document is organized.

When the sort order is changed, no items in the list should be removed – unlike a filter control that is only concerned with removing items from a list.

Program and Facility Selection

Many workflows in the OpenLMIS-UI require a user to select both a facility and program they are working in before any data is displayed. This is a form of navigation, but it can be much more complicated than a list of links.

In the OpenLMIS-UI we have created an AngularJS component to keep facility and program selection consistent.

Program and Facility Selection works like this:

- A user is presented with the option of selecting the home facility or selecting one of their supervised facilities
- Home facility is the default selection, unless the user doesn't have a home facility, and then the option should be hidden
- If the user doesn't have supervised facilities, that option is hidden
- If the home facility is selected, the user must then select a program that is supported by that facility
- If the supervised facility option is selected, the user must first select a program, then select a facility that supports that program.

Some list views do not require a user to select both a program and facility, but instead provide an optional filter to help the user drill in on a sub-set of the list. In those cases, the selection rules above don't apply. Ideally, users will only be shown lists of programs and facilities they have access to.

UI Coding Conventions

This document describes the desired formatting to be used within the OpenLMIS-UI repositories. Many of the conventions are adapted from John Papa's Angular V1 styleguide, SMACSS by Jonathan Snook, and Jens Meiert's maintainability guide.

General

The following conventions should be applied to all sections of UI development:

- All indentation should be 4 spaces
- Legacy code should be refactored to meet coding conventions
- No third party libraries should be included in a OpenLMIS-UI repository

File Structure

All file types should be organized together within the `src` directory according to functionality, not file type — the goal is to keep related files together.

Use the following conventions:

- File names are lowercase and dash-separated
- Files in a directory should be as flat as possible (avoid sub-directories)
- If there are more than 12 files in a directory, try to divide files into subdirectories based on functional area

Naming Convention

In general we follow the John-Papa naming conventions. Later sections go into specifics about how to name a specific file type, while this section focusses on general naming and file structure.

Generally, all file names should use the following format `specific-name.file-type.ext` where:

- `specific-name` is a dash-separated name for specific file-type
- `file-type` is the type of object that is being added (ie 'controller', 'service', or 'layout')
- `ext` is the extension of the file (ie '.js', '.scss')

Folder structure should aim to follow the LIFT principal as closely as possible, with a couple extra notes:

- There should only be one *.module.js file per directory hierarchy
- Only consider creating a sub-directory if file names are long and repetitive, such that a sub-directory would improve meaning *Each file type section below has specifics on their naming conventions*

Javascript Guidelines

Almost everything in the OpenLMIS-UI is Javascript. These are general guidelines for how to write and test your code.

General conventions:

- All code should be within an immediately invoked scope
- *ONLY ONE OBJECT PER FILE*
- Variable and function names should be written in camelCase
- All Angular object names should be written in CamelCase

Documentation

To document the OpenLMIS-UI, we are using ngDocs built with grunt-ngdocs. See individual object descriptions for specifics and examples of how to document that object type.

General rules

- any object's exposed methods or variables must be documented with ngDoc
- @ngdoc annotation specifies the type of thing being documented
- as 'Type' in documentation we should use:
- Promise
- Number
- String
- Boolean
- Object
- Event
- Array
- Scope
- in some cases is allowed to use other types i.e. class names like Requisition
- all description blocks should be sentence based, all of sentences should start with uppercase letter and end with '.'
- before and after description block (if there is more content) there should be an empty line
- all docs should be right above the declaration of method/property/component
- when writing param/return section please keep all parts(type, parameter name, description) start at the same column as it is shown in method/property examples below
- please keep the order of all parameters as it is in examples below

General Object Documentation

Regardless of the actual component's type, it should have '@ngdoc service' annotation at the start, unless the specific object documentation says otherwise. There are three annotations that must be present:

- ngdoc definition
- component name
- and description

```
/**
 * @ngdoc service
 * @name module-name.componentName
 *
 * @description
 * Component description.
 */
```

Methods

Methods for all components should have parameters like in the following example:

```
/**
 * @ngdoc method
 * @methodOf module-name.componentName
 * @name methodName
 *
 * @description
 * Method description.
 *
 * @param  {Type} paramsName1 param1 description
 * @param  {Type} paramsName2 (optional) param2 description
 * @return {Type}             returned object description
 */
```

Parameters should only be present when method takes any. The same rule applies to return annotation. If the parameter is not required by method, it should have "(optional)" prefix in the description.

Properties

Properties should be documented in components when they are exposed, i.e. controllers properties declared in 'vm'. Properties should have parameters like in the following example:

```
/**
 * @ngdoc property
 * @propertyOf module-name.componentName
 * @name propertyName
 * @type {Type}
 *
 * @description
 * Property description.
 */
```

Constants

Constants are Javascript variables that won't change but need to be reused between multiple objects within an Angular module. Using constants is important because it becomes possible to track an objects dependencies, rather than use variables set on the global scope.

It's also useful to wrap 3rd party objects and libraries (like jQuery or bootbox) as an Angular constant. This is useful because the dependency is declared on the object. Another useful feature is that if the library or object isn't included, Angular will throw a single verbose error message.

Add rule about when its ok to add a group of constants – if a grouping of values, use a plural name

Conventions:

- All constant variable names should be upper case and use underscores instead of spaces (ie VARIABLE_NAME)
- If a constant is only relevant to a single Angular object, set it as a variable inside the scope, not as an Angular constant
- If the constant value needs to change depending on build variables, format the value like @@VARIABLE_VALUE, and which should be replaced by the grunt build process if there is a matching value
- Wrap 3rd party services as constants, if are not already registered with Angular

Replaced Values

@@ should set own default values

Interceptor

This section is about events and messages, and how to modify them.

HTTP Interceptors are technically factories that have been configured to 'intercept' certain types of requests in Angular and modify their behavior. This is recommended because other Angular objects can use consistent Angular objects, reducing the need to write code that is specialized for our own framework.

Keep all objects in a single file - so its easier to understand the actions that are being taken

The Angular guide to writing HTTP Interceptors is here

General Conventions

- Write interceptors so they only change a request on certain conditions, so other unit tests don't have to be modified for the interceptors conditions
- Don't include HTTP Interceptors in openlmis-core, as the interceptor might be injected into all other unit tests — which could break everything

Unit Testing Conventions

The goal when unit testing an interceptor is to not only test input and output transformation functions, but to also make sure the interceptor is called at an appropriate time.

Javascript Class

Put all direct business logic in a pure javascript class.

Pure javascript classes should only be used to ease the manipulation of data, but unlike factories, these object shouldn't create HTTP connections, and only focus on a single object.

Javascript classes should be injected and used within factories and *some services* services that have complex logic. Modules should be able to extend javascript classes by prototypical inheritance.

Helps with code reusability

Requisition/LineItem is good example

Naming Conventions

SampleName

Classes should be uppercase CamelCased, which represents that they are a class and need to be instantiated like an object (ie `new SampleName()`).

Routes

Routing logic is defined by UI-Router, where a URL path is typically paired with an HTML View and Controller.

Use a factory where possible to keep resolve statements small and testable

General Conventions

- The UI-Router resolve properties are used to ease loading on router
- Routes should define their own views, if their layout is more complicated than a single section

Service

John Papa refers to services as Singletons, which means they should only be used for application information that has a single instance. Examples of this would include the current user, the application's connection state, or the current library of localization messages.

Conventions

- Services should always return an object
- Services shouldn't have their state changed through properties, only method calls

Naming Convention

nameOfService*Service*

Always lowercase camelCase the name of the object. Append 'Service' to the end of the service name so developers will know the object is a service, and changes will be persisted to other controllers.

Unit Testing Conventions

- Keep $httpBackend mock statements close to the specific places they are used (unless the statement is reusable)
- Use Jasmine's spyOn method to mock the methods of other objects that are used
- In some cases mocking an entire AngularJS Service, or a constant, will be required. This is possible by using AngularJS's $provide object within a beforeEach block. This would look like

```
beforeEach(module($provide){
    // mock out a tape recorder service, which is used else where
    this.tape = jasmine.createSpyObj('tape', ['play', 'pause', 'stop', 'rewind']);

    // overwrite an existing service
    var tape = this.tape;
    $provide.service('TapeRecorderService', function(){
        return tape;
    });
});
```

Class

Class is a our custom pattern that was designed to make our codebase easier to migrate to ES6 once we add support for it as it mimics how the ES6 classes looks like.

When to use:

- factories,
- services,
- domain classes that define some logic.

When not to use:

- interceptors,
- domain objects that does not have any logic.

How to define

```
(function() {

    'use strict';

    /**
     * @ngdoc service
     * @name module-name.ClassName
     *
     * @description
     * Example class.
     */
    angular
        .module('module-name')
        .factory('ClassName', ClassName);

    function ClassName() {
```

(continues on next page)

(continued from previous page)

```
        ClassName.prototype.exampleMethod = exampleMethod;

        return ClassName;

        /**
         * @ngdoc method
         * @methodOf module-name.ClassName
         * @name ClassName
         *
         * @description
         * Creates a new instance of the ClassName class.
         *
         * @param  {Object}  parameter the constructor parameter
         * @return {Object}            the object of the ClassName class
         */
        function ClassName(parameter) {
            this.parameter = parameter;
        }

        /**
         * @ngdoc method
         * @methodOf module-name.ClassName
         * @name exampleMethod
         *
         * @description
         * Creates a new instance of the ClassName class.
         *
         * @param  {Object}  parameter the method parameter
         * @return {Object}            the result of the method
         */
        function exampleMethod() {
            //do something
        }
    }

})();
```

Extending classes

In order to extend a class we can use the classExtender factory that deals will basic prototype extension.

```
(function() {

    'use strict';

    /**
     * @ngdoc service
     * @name module-name.ClassName
     *
     * @description
     * Example extending class class.
     */
    angular
        .module('shipment')
```

(continues on next page)

(continued from previous page)

```
        .factory('ExtendingClass', ExtendingClass);

    ExtendingClass.$inject = ['ExtendedClass', 'classExtender'];

    function ExtendingClass(ExtendedClass, classExtender) {

        classExtender.extend(ExtendingClass, ExtendedClass);

        return ExtendingClass;

        function ExtendingClass() {
            //add the following line to call the constructor of the super class
            this.super.apply(this, arguments);
        }
    }

})();
```

Singletons

In order to create a singleton we should still elevate AngularJS dependency injection capabilities. The className object will be shared across the whole application and can be injected using AngularJS dependency injection.

```
(function() {

    'use strict';

    /**
     * @ngdoc service
     * @name module-name.ClassName
     *
     * @description
     * Example class.
     */
    angular
        .module('module-name')
        .factory('ClassName', ClassName);

    function ClassName() {

        ClassName.prototype.exampleMethod = exampleMethod;

        return ClassName;

        /**
         * @ngdoc method
         * @methodOf module-name.ClassName
         * @name ClassName
         *
         * @description
         * Creates a new instance of the ClassName class.
         *
         * @param  {Object}  parameter the constructor parameter
         * @return {Object}            the object of the ClassName class
         */
```

(continues on next page)

(continued from previous page)

```
        function ClassName(parameter) {
            this.parameter = parameter;
        }

        /**
         * @ngdoc method
         * @methodOf module-name.ClassName
         * @name exampleMethod
         *
         * @description
         * Creates a new instance of the ClassName class.
         *
         * @param  {Object}  parameter the method parameter
         * @return {Object}            the result of the method
         */
        function exampleMethod() {
            //do something
        }
    }

})();

(function() {

    'use strict';

    /**
     * @ngdoc service
     * @name module-name.ClassName
     *
     * @description
     * Example class.
     */
    angular
        .module('module-name')
        .factory('className', className);

    className.$inject = ['ClassName'];

    function className(ClassName) {
        return new ClassName();
    }

})();
```

AngularJS Conventions

This document accompanies the UI Coding Conventions. It gives specific guidance for AngularJS modules, controllers, directives, and filters.

Modules

Modules in angular should describe and bind together a small unit of functionality. The OpenLMIS-UI build process should construct larger module units from theses small units.

Documentation

Docs for modules must contain the module name and description. This should be thought of as an overview for the other objects within the module, and where appropriate gives an overview of how the modules fit together.

```
/**
 * @module module-name
 *
 * @description
 * Some module description.
 */
```

Controller

Controllers are all about connecting data and logic from Factories and Services to HTML Views. An ideal controller won't do much more than this, and will be as 'thin' as possible.

Controllers are typically specific in context, so as a rule controllers should never be reused. A controller can be linked to a HTML form, which might be reused in multiple contexts — but that controller most likely wouldn't be applicable in other places.

It is also worth noting that John Papa insists that controllers don't directly manipulate properties in $scope, but rather the ControllerAs syntax should be used which injects the controller into a HTML block's context. The main rationale is that it makes the $scope variables less cluttered, and makes the controller more testable as an object.

Conventions

- Should be only object changing application $state
- Is used in a single context
- Don't use the $scope variable EVER
- Use ControllerAs syntax
- Don't $watch variables, use on-change or refactor to use a directive to watch values

Unit Testing

- Set all items that would be required from a route when the Controller is instantiated
- Mock any services used by the controller

Documentation

The only difference between controllers and other components is the '.controller:' part in the @name annotation. It makes controller documentation appear in controllers section. Be sure to document the methods and properties that the controller exposes.

```
/**
 * @ngdoc controller
 * @name module-name.controller:controllerName
 *
```

(continues on next page)

(continued from previous page)

```
 * @description
 * Controller description.
 */
```

Directive

Directives are pieces of HTML markup that have been extended to do a certain function. *This is the only place where it is reasonable to manipulate the DOM.*

Make distinction between directive and component – components use E tag and isolate scope, directive use C and never isolate scope

Conventions

- Restrict directives to only elements or attributes
- Don't use an isolated scope unless you absolutely have to
- If the directive needs external information, use a controller — don't manipulate data in a link function

Unit Testing

The bit secret when unit testing a directive is to make sure to use the $compile function to return an element that is extended with jQuery. Once you have this object you will be able to interact with the directive by clicking, hovering, or triggering other DOM events.

```
describe('SampleDirective', function() {

    it('gets compiled and shows the selected item name', function($compile,
↪$rootScope) {
        var scope = $rootScope.$new();
        scope['item'] = {
            name: "Sample Title"
        };
        var element = $compile("<sample-directive selected='item'></sample-directive>
↪")(scope);

        expect(element.text()).toBe("Sample Title");
    });

    it('responds to being clicked', function($compile, $rootScope) {
        var element = $compile("<sample-directive selected='item'></sample-directive>
↪")($rootScope.$new());

        // check before the action
        expect(element.text()).toBe("No Title");

        element.click();
        // check to see the results of the action
        // this could also be looking at a spy to see what the values are
        expect(element.text()).toBe("I was clicked");
    });
```

(continues on next page)

(continued from previous page)

```
});
```

Documentation

Directive docs should have well described '@example' section.

Directive docs should always have '@restrict' annotation that takes as a value one of: A, E, C, M or any combination of those. In order to make directive docs appear in directives section there needs to be '.directive:' part in @name annotation.

```
/**
 * @ngdoc directive
 * @restrict A
 * @name module-name.directive:directiveName
 *
 * @description
 * Directive description.
 *
 * @example
 * Short description of how to use it.
 * ```
 *   <div directiveName></div>
 * ```
 * Now you can show how the markup will look like after applying directive code.
 * ```
 * <div directiveName>
 *     <div>something</div>
 * </div>
 * ```
 */
```

Extending a Directive

You can extend a directive by using AngularJS's decorator pattern. Keep in mind that a directive might be applied to multiple places or have multiple directives applied to the same element name.

```
angular.module('my-module')
    .config(extendDirective);

extendDirective.$inject = ['$provide'];
function extendDirective($provide) {

  // NOTE: This method has you put 'Directive' at the end of a directive name
  $provide.decorator('OpenlmisInvalidDirective', directiveDecorator);
}

directiveDecorator.$inject = ['$delegate'];
function directiveDecorator($delegate) {
  var directive = $delegate[0], // directives are returned as an array
      originalLink = directive.link;

  directive.link = function(scope, element, attrs) {
```

(continues on next page)

(continued from previous page)

```
    // do something
    originalLink.apply(directive, arguments); // do the original thing
    // do something after
  }

  return $delegate;
}
```

Filters

Use an AngularJS filter if:

- You need to do complex formatting
- You need to render value in HTML, and it doesn't make sense to include in a controller.

Documentation

Filter docs should follow the pattern from example below:

```
/**
 * @ngdoc filter
 * @name module-name.filter:filterName
 *
 * @description
 * Filter description.
 *
 * @param   {Type} input     input description
 * @param   {Type} parameter parameter description
 * @return  {Type}           returned value description
 *
 * @example
 * You could have short description of what example is about etc.
 * ```
 * <div>{{valueToBeFiltered | filterName:parameter}}</div>
 * ```
 */
```

It is a good practice to add example block at the end to make clear how to use it. As for parameters the first one should be describing input of the filter. Please remember of '.filter:' part. It will make sure that this one will appear in filters section.

Unit Testing Guidelines

A unit tests has 3 goals that it should accomplish to test a javascript object:

- Checks success, error, and edge cases
- Tests as few objects as possible
- Demonstrates how an object should be used

With those 3 goals in mind, its important to realize that the variety of AngularJS object types means that the same approach won't work for each and every object. Since the OpenLMIS-UI coding conventions layout patterns for different types of AngularJS objects, it's also possible to illustrate how to unit test objects that follow those conventions.

Check out AngularJS's unit testing guide, its well written and many of out tests follow their styles.

Here are some general rules to keep in mind while writing any unit tests:

- Keep beforeEach statements short and to the point, which will help other's read your statements
- Understand how to use Spies in Jasmine, they can help isolate objects and provide test cases

HTML Markup Guidelines

Less markup is better markup, and semantic markup is the best.

This means we want to avoid creating layout specific markup that defines elements such as columns or icons. Non-semantic markup can be replicated by using CSS to create columns or icons. In some cases a layout might not be possible without CSS styles that are not supported across all of our supported browsers, which is perfectly acceptable.

Here is a common pattern for HTML that you will see used in frameworks like Twitter's Bootstrap (which we also use)

```
<li class="row">
    <div class="col-md-9">
        Item Name
    </div>
    <div class="col-md-3">
        <a href="#" class="btn btn-primary btn-block">
            <i class="icon icon-trash"></i>
            Delete
        </a>
    </div>
</li>
<div class="clearfix"></div>
```

The above markup should be simplified to:

```
<li>
    Item Name
    <button class="trash">Delete</button>
</li>
```

This gives us simpler markup, that could be restyled and reused depending on the context that the HTML section is inserted into. We can recreate the styles applied to the markup with CSS such as:

- A ::before pseudo class to display an icon in the button
- Using float and width properties to correctly display the button
- A ::after pseudo class can replace any 'clearfix' element (which shouldn't exist in our code)

See the UI-Styleguide for examples of how specific elements and components should should be constructed and used.

HTML Views

Angular allows HTML files to have variables and simple logic evaluated within the markup.

A controller that has the same name will be the reference to vm, if the controller is different, don't call it vm

General Conventions

- If there is logic that is more complicated than a single if statement, move that logic to a controller

- Use filters to format variable output — don't format variables in a controller

HTML Form Markup

A goal for the OpenLMIS-UI is to keep business logic separated from styling, which allows for a more testable and extendable platform. Creating data entry forms is generally where logic and styling get tangled together because of the need to show error responses and validation in meaningful ways. AngularJS has built-in features to help foster this type of separation, and OpenLMIS-UI extends AngularJS's features to a basic set of error and validation features.

The goal here is to attempt to keep developers and other implementers from creating their own form submission and validation - which is too easy in Javascript frameworks like AngularJS.

An ideal form in the OpenLMIS-UI would look like this:

```
<form name="exampleForm" ng-submit="doTheThing()">
    <label for="exampleInput">Example</label>
    <input id="exampleInput" name="exampleInput" ng-model="example" required />
    <input type="submit" value="Do Thing" />
</form>
```

This is a good form because:

- There is a name attribute on the form element, which exposes the FormController
- The input has a name attribute, which allow for validation passed to the FormController to be passed back to the correct input
- ng-submit is used rather than ng-click on a button

SASS & CSS Formatting Guidelines

General SASS and CSS conventions:

- Only enter color values in a variables file
- Only enter pixel or point values in a variables file
- Variable names should be lowercase and use dashes instead of spaces (ie: *$sample-variable*)
- Avoid class names in favor of child element selectors where ever possible
- Files should be less than 200 lines long
- CSS class names should be lowercase and use dashes instead of spaces

SMACSS

The CSS styles should reflect the SMACSS CSS methodology, which has 3 main sections — base, layout, and module. SMACSS has other sections and tenants, which are useful, but are not reflected in the OpenLMIS-UI coding conventions.

Base

CSS styles applied directly to elements to create styles that are the same throughout the application.

Layout

CSS styles that are related primarily to layout in a page — think position and margin, not color and padding — these styles should never be mixed with base styles (responsive CSS should only be implemented in layout).

Module

This is a css class that will modify base and layout styles for an element and it's sub-elements.

SASS File-Types

Since SASS pre-processes CSS, there are 3 SCSS file types to be aware of which are processed in a specific order to make sure the build process works correctly.

Variables

A variable file is either named 'variables.scss' or matches '*.variables.scss'

Variables files are the first loaded file type and include any variables that will be used through out the application — *There should be as few of these files as possible*.

The contents of a variables file should only include SASS variables, and output no CSS at any point.

There is no assumed order in which variables files will be included, which means:

- Variable files shouldn't have overlapping variables
- Implement SASS's variable default (!default)

Mixins

A mixin file matches the following pattern *.mixin.scss

Mixins in SASS are reusable functions, which are loaded second in our build process so they can use global variables and be used in any other SCSS file.

There should only be one mixin per file, and the file name should match the function's name, ie: 'simple-function.mixin.scss'

All Other SCSS and CSS Files

All files that match '*.scss' or '*.css' are loaded at the same time in the build process. This means that no single file can easily overwrite another files CSS styles unless the style is more specific or uses `!imporant` — This creates the following conventions:

- Keep CSS selectors as general as possible (to allow others to be more specific)
- Avoid using !important

To keep file sizes small, consider breaking up files according to SMACSS guidelines by adding the type of classes in the file before .scss or .css (ie: `navigation.layout.scss`)

1.5.6 Performance

Performance Testing

OpenLMIS focuses on performance metrics that are typical in web-applications:

- Calls to the server - how many milliseconds does this single operation take, and is the memory usage reasonable.
- Network load - how large are the resources returned from the server. Typically OpenLMIS is designed to work in network-constrained locations, so the size, in bytes, of each resource is important.
- The number of calls the Reference UI makes - again networks being what they, we want to minimize the number of connections that are made to accomplish a user workflow as each connection adds overhead.
- Size of the "working" data set. Here working data is defined as the data that's needed for a user to accomplish a task. Examples are typically Reference Data: # of Products, # of Facilities, # of Users, etc. Though also the # of Requisitions or # of Stock Cards might factor into a user's working data. Since OpenLMIS typically manages countries, it's important that we're efficient in managing country-level data sets.

There are some areas of Performance however that OpenLMIS typically doesn't focus as much on:

- Scaling - typically we're not concerned with tens of thousands of people needing to use the system concurrently. Likewise we don't typically worry yet about surges or dips in user activity requiring more or less resources to serve those users.

Getting Started

OpenLMIS uses Apache JMeter to test RESTful endpoints. We use Taurus, and it's YAML format, to write our test scenarios and generate reports which our CI server can present as an artifact of every successful deployment to our CD test server.

Keeping to our conventions, Taurus is used through a Docker image, with a simple script located at *./performance/test.sh* with tests in the directory *./performance/tests/* of a Service. Any **.yml* file in that test directory will be fed to Taurus to be used against *https://test.openlmis.org*.

Running *test.sh* will place JMeter output as well as Taurus output under *./build/performance-artifacts/*. The file *stats.xml* has the final summary performance metrics. Files of note when developing test scenarios:

- *error-N.jtl* - Contains errors and requests that led to those errors from the HTTP server.
- *JMeter-N.err* - Contains JMeter errors where JMeter didn't understand the test scenario.
- *modified_requests-N.jmx* - Contains the generated JMeter requests (after Taurus generation).
- *kpi-N.jtl* - Individual metrics of a test scenario.

Running in CI

Tests run in a Jenkin's Job that ends in *-performance*. This job is run as part of each Service's build pipeline *that results in a deployment to the test server*.

The reports are presented using Performance Plugin. When looking at this report you'll see:

- A graph that shows all of the endpoints (requests) over time.
- A report for a build which includes an average over time, as well as a table showing KPIs of each request.

A simple Scenario (with authentication)

Nearly all of our RESTful resources require authentication, in this example we'll show a basic test scenario that includes authentication. The syntax and features used here are documented at Taurus' page on the JMeter executer.

```
execution:
  - concurrency: 1
    hold-for: 1m
    scenario: users-get-one
scenarios:
  get-user-token:
    requests:
      - url: ${__P(base-uri)}/api/oauth/token
        method: POST
        label: GetUserToken
        headers:
          Authorization: Basic ${__base64Encode(${__P(basic-auth)})}
        body:
          grant_type: password
          username: ${__P(username)}
          password: ${__P(password)}
        extract-jsonpath:
          access_token:
            jsonpath: $.access_token
  users-get-one:
    requests:
      - include-scenario: get-user-token
      - url: ${__P(base-uri)}/api/users/a337ec45-31a0-4f2b-9b2e-a105c4b669bb
        method: GET
        label: GetAdministratorUser
        headers:
          Authorization: Bearer ${access_token}
```

The *execution* block defines for our test scenario *users-get-one* that runs 1 concurrent user, for one minute. Notice that this definition is for the simplest of test executions - 1 user, run it enough times to get a useful sampling. We use this sort of test execution to first get a sense of what our endpoint's single-user characteristics are.

Next notice that we have two scenarios defined:

1. get-user-token - this is a reusable scenario, which gets a basic user authentication token, and through the *extract-jsonpath* saves it to a variable named *access_token*.
2. users-get-one - this is the test scenario we're primarily interested in: exercise the */api/users/{a specific users uuid}*. We pass the previously obtained *access_token* through the HTTP request's headers.

Summary

- First test the most basic of environments: 1 user, enough times to get useful sample size.
- Re-use the scenario to obtain an access_token using *include-scenario*.
- It's generally OK to use demo-data identifiers (the user's UUID) - though it couples the test to the demo-data, it will provide consistent results.
- Give each request a clear, semantic *label*. This will be used later in pass-fail criteria.

Testing collections

To the simple Scenario we're going to now test the performance of returning a collection of a resource:

```
users-search-one-page:
  requests:
    - include-scenario: get-user-token
    - url: ${__P(base-uri)}/api/users/search?page=1&size=10
      method: POST
      label: GetAUserPageOfTen
      body: '{}'
      headers:
        Authorization: Bearer ${access_token}
        Content-Type: application/json
```

Here we're testing the Users resource by asking for 1 page of 10 users.

Summary

- When testing the performance of collections, the result will be influenced by the number of results returned. Due to this prefer to test a paginated resource, and always ask for a number that exists (i.e. don't ask for 50 when demo-data only has 40).
- Searching often requires a POST, in this case the query parameters must be in the URL.

Testing complex workflows

A complex workflow might be:

1. GET a list of periods for which requisitions may be initiated.
2. Create a new Requisition resource by POSTing with the previously returned periods available.
3. DELETE the previously created Requisition resource, so that we may test again.

```
initiate-requisition:
  requests:
    - url: ${__P(base-uri)}/api/oauth/token
      method: POST
      label: GetUserToken
      headers:
        Authorization: Basic ${__base64Encode(${__P(user-auth)})}
      body:
        grant_type: password
        username: ${__P(username)}
        password: ${__P(password)}
      extract-jsonpath:
        access_token:
          jsonpath: $.access_token
    # program = family planning, facility = comfort health clinic
    - url: ${__P(base-uri)}/api/requisitions/periodsForInitiate?programId=10845cb9-
↪d365-4aaa-badd-b4fa39c6a26a&facilityId=e6799d64-d10d-4011-b8c2-0e4d4a3f65ce&
↪emergency=false
      method: GET
      label: GetPeriodsForInitiate
      headers:
```

(continues on next page)

(continued from previous page)

```
        Authorization: Bearer ${access_token}
      extract-jsonpath:
        periodUuid:
          jsonpath: $.[:1]id
      jsr223:
        script-text: |
          String uuid = vars.get("periodUuid");
          uuid = uuid.replaceAll(/"|\[|\]/, "");
          vars.put("periodUuid", uuid);
    - url: ${__P(base-uri)}/api/requisitions/initiate?program=10845cb9-d365-4aaa-badd-
↪b4fa39c6a26a&facility=e6799d64-d10d-4011-b8c2-0e4d4a3f65ce&suggestedPeriod=$
↪{periodUuid}&emergency=false
      method: POST
      label: InitiateNewRequisition
      headers:
        Authorization: Bearer ${access_token}
        Content-Type: application/json
      extract-jsonpath:
        reqUuid:
          jsonpath: $.id
      jsr223:
        script-text: |
          String uuid = vars.get("reqUuid");
          uuid = uuid.replaceAll(/"|\[|\]/, ""); # remove quotes and []
          vars.put("reqUuid", uuid);
    - url: ${__P(base-uri)}/api/requisitions/${reqUuid}
      method: DELETE
      label: DeleteRequisition
      headers:
        Authorization: Bearer ${access_token}
```

Summary

- When creating a new RESTful resource (e.g. PUT or POST), we may need to clean-up after ourselves in order to run more than one test.
- JSR223 blocks allow us to execute basic Groovy (default). This can be especially useful when you need to clean-up a JSON result from a previous response, such as a UUID, to use in the next request.

Simple stress testing

As mentioned, OpenLMIS performance tests tend to focus first on basic execution environments where we're only testing 1 user interaction at a time. However there is a need to do basic stress testing, especially for endpoints which are used frequently. For example we've seen the authentication resource used repeatedly in all our previous examples. Lets stress test it.

```
modules:
  local:
    sequential: true

execution:
  - concurrency: 10
    hold-for: 2m
```

(continues on next page)

(continued from previous page)

```
    scenario: get-user-token
  - concurrency: 50
    hold-for: 2m
    scenario: get-service-token

scenarios:
  get-user-token:
    requests:
      - url: ${__P(base-uri)}/api/oauth/token
        method: POST
        label: GetUserToken
        headers:
          Authorization: Basic ${__base64Encode(${__P(user-auth)})}
        body:
          grant_type: password
          username: ${__P(username)}
          password: ${__P(password)}
  get-service-token:
    requests:
      - url: ${__P(base-uri)}/api/oauth/token
        method: POST
        label: GetServiceToken
        headers:
          Authorization: Basic ${__base64Encode(${__P(service-auth)})}
        body:
          grant_type: client_credentials
```

Here we've defined 2 tests:

1. Authenticate as if you're a person.
2. Authenticate as if you're another Service (a Service token).

The stress testing here introduces important changes in our *execution* block:

```
- concurrency: 10
  hold-for: 2m
  scenario: get-user-token
```

Instead of defining 1 user, here we'll have 10 concurrent ones. Instead of running the test for 1 minute, we're going to run the test as many times as we can for 2 minutes. For further options see the Taurus' Execution doc.

When stress testing, it's important to remember that too much simply isn't useful, and only slows down the test. Nor do we presently have a test infrastructure in place that allows for tests to originate from multiple hosts.

Summary

- You can define multiple execution definitions for the same scenario, so the first might give us the basic performance characteristics, the second might be a stress test.
- By default the tests defined in the *execution* block are run in parallel. This can be changed to by ran sequential with *sequential: true*.
- Choose a reasonable number of concurrent users. Typically less than a dozen is enough.
- Choose a reasonable time to hold the test for. Typically 1-2 minutes is enough, and no more than 5 minutes unless justifiable.

- Remember that we don't have a performance testing infrastructure in place that can concurrently send requests to our application from multiple hosts. OpenLMIS performance testing typically only requires the most basic stress testing.

Testing file uploads

In this short example we're going to send a request to the catalog items endpoint and upload some items as a CSV file.

```
upload-catalog-items:
  requests:
    - include-scenario: get-user-token
    - url: ${__P(base-uri)}/api/catalogItems?format=csv
      method: POST
      label: UploadCatalogItems
      headers:
        Authorization: Bearer ${access_token}
      upload-files:
        - param: file
          path: /tmp/artifacts/catalog_items.csv
```

Summary

- When uploading a file we don't have to worry about setting correct content header as Taurus take care of it on its own when using upload-files block. This behavior is described in the HTTP Requests of the Taurus user manual

Pass-fail criteria

With the above tests defined, we can now write pass-fail criteria. This is especially useful if we want our test to fail when the performance is less than what we've defined.

```
reporting:
    - module: passfail
      criteria:
        - avg-rt of GetUserToken>300ms, continue as failed
        - avg-rt of GetServiceToken>300ms, continue as failed
```

This allows us to fail the test if the average response time for either of the two tests was greater than 300ms. See the *Taurus Passfail doc* for more.

Summary

- Write the pass-fail criteria within the test definition.

Performance Acceptance Criteria

With Taurus we can now add basic acceptance criteria when working on new issues. For example the acceptance criteria might say:

- the endpoint to retrieve 10 users should complete in 500ms for 90% of users

This would lead us to write a performance test for this new GET operation to retrieve 10 users, and we'd add a pass-fail criteria such as:

```
reporting:
    - module: passfail
      criteria:
        Get 10 Users is too slow: p90 of Get10Users>500ms, continue as failed
```

Read the Taurus Passfail doc for more.

Next Steps (WIP)

We've covered basic performance testing, stress testing, and pass-fail criteria. Next we'll be adding:

- Loading performance-oriented data sets (e.g. what happens to these requests when there are 10,000 products).
- Using Selenium to mimic browser interactions, to give us:
  - How many http requests does a page incur.
  - Network payload size.
- Failing deployments based on performance results.

Performance Data

Performance data in OpenLMIS is meant to be data that helps us answer questions such as:

- What happens to the server and the operations it provides when there are 10,000 orderables, users, facilities, requisitions, etc?
- What happens when all that data is being used by many concurrent users?
- What's the impact on network performance, especially for those in low resource environments?
- What sort of deployment topology works best for typical implementations?
- Does the UI (and possibly other clients) display large sets of data well?

Some basic characteristics of performance data:

- there is a lot of it
- it doesn't have to look nice or make that much sense to domain experts (e.g. a Vaccine could be randomly generated to be ordered through the essential meds Program, and that's okay). Lorem ipsum and random numbers are just fine here.
- it must be deployable in a deployment topology that is as close to a production setup as possible. After all it's for performance testing, and performance testing on a local laptop doesn't tell us (much) about anything a production server running in the cloud would experience.

Where is performance data located?

Performance data is stored in Git within each Service that defines it, much like demo-data. In fact in most cases Performance Data builds off of demo-data, and so a Service should be able to load performance data or demo-data in very similar ways.

How to load performance data

Like demo-data, performance data is an optional set of data that may be loaded when the Service *starts*. To do this a Service should load performance data, likely after any demo-data, by looking for the profiles set in the environment variable `spring.profiles.active`. If this environment variable contains the string `performance-data`, then the service should load this data before it's operational for use.

How to create and manage performance data

Performance data is generated with the help of the tool Mockaroo. This tool is used to define schemas which match the Service's tables and it may generate large CSVs which are then stored in the Service in git. CSVs are used as they easily enable the use of foreign key / UUID lookups when a Mockaroo dataset is used (as this Mockaroo dataset video demonstrates). These CSVs are placed in git for the Service to load the data, however if the Service needs new performance data, the database schema changes or something else causes the performance data to need to be updated, the OpenLMIS Mockaroo account should be used to generate a new set, which will then be stored in the Service.

What types of performance data should be created?

Performance data is relatively expensive and tedious to maintain given the questions we're trying to answer. While it's necessary to do so, here are some general guidelines for what to spend time generating, and what not to:

Do

- Generate performance data that will allows performance tests to reflect country data needs.
- Try to generate data that's more right than random. Random is okay, However if the tool has a sufficiently large set of facilities, or products, use it.
- Respect database constraints, foreign keys, references to IDs in other Services etc
- Keep in mind that some UUIDs need to be *known*. They can't be generated. You'll need to know a few of these key UUIDs (e.g. Program, User, etc) in order to construct useful performance tests.

Don't

- Overcomplicate the data. 1 billion facilities, a trillion requisitions, 1000 programs just aren't anywhere near likely and just take longer to load and more time to maintain. 10k facilities, 100k requisitions, 10 programs are much more representative.
- Similarly, don't generate data when demo-data already has enough. E.g. demo data already has a few Programs, you're time is better spent setting up one of those programs to have 10k facility type approved products than you are generating 100 programs.
- Don't build performance tests off of generated IDs. While Mockaroo makes it easy to build sets of data with referential integrity, using generated IDs hardcoded in performance tests will result in more brittle tests.

Performance Tips

Testing and Profiling

Knowing where to invest time and resources into optimization is always the first step. This document will briefly cover the highlights of two tools which help us determine where we should invest our time, and then we'll dive into specific strategies for making our performance better.

To see how to test HTTP based services see *Performance Testing*.

Profiling

After we've identified that a HTTP operation is slow, there are two simple tools that can help us in understanding why:

- SLF4J Profiler: useful in printing latency meassurements to our log. It's cheap and a bit inaccurate, though quite effective and it works in all production environments.
- VisualVM: perhaps the most well known Java profiling tool can give great information about what the code is doing, however since it needs to connect directly to the JVM running that Service's code, it's better suited for local development environments rather than debugging production servers.

The usefulness of basic profiling metrics from production environments can't be understated. Performance issues rarely occur in local development environments and the people most impacted by slow performance are people using production systems. Just as our performance tests operate against a *Recommended Deployment Topology* that tries to match what most of our customers use, so to is it useful to know how that code is performing in customer implementations. For these reasons this document will focus more on logging performance metrics with SLF4J Profiler rather than VisualVM.

Using SLF4J Profiler in Java code is as simple as:

```
Profiler profiler = new Profiler("GET_ORDERABLES_SEARCH");
profiler.setLogger(XLOGGER); // can be SLF4J Logger or XLogger

profiler.start("CHECK_ADMIN_RIGHT");
rightService.checkAdminRight(ORDERABLES_MANAGE);

profiler.start("ORDERABLE_SERVICE_SEARCH");
Page<Orderable> orderablesPage = orderableService.searchOrderables(queryParams,␣
↪pageable);

profiler.start("ORDERABLE_PAGINATION");
Page<OrderableDto> page = Pagination.getPage(OrderableDto.newInstance(
    orderablesPage.getContent()),
    pageable,
    orderablesPage.getTotalElements());

profiler.stop().log();
```

This will generate log statements that look like the following:

```
2017-07-24T19:49:45+00:00 e2f424e5b617 [nio-8080-exec-5] DEBUG
org.openlmis.referencedata.web.OrderableController #012+ Profiler
[GET_ORDERABLES_SEARCH]#012|
-- elapsed time [CHECK_ADMIN_RIGHT] 1173.997 milliseconds.#012|
-- elapsed time [ORDERABLE_SERVICE_SEARCH] 199.251 milliseconds.#012|
-- elapsed time [ORDERABLE_PAGINATION] 0.255 milliseconds.#012|
-- Total [GET_ORDERABLES_SEARCH] 1373.511 milliseconds.
```

Placed in the Controller for this HTTP operation we can tell:

1. Most of the time for this innvocation is spent checking if the user has a right: more than 1 second.
2. Fetching the entities from the database took about 14% of the time
3. Turning them into DTOs used up *less* than a millisecond.
4. We'd have to look at the Service's access log to find where additional latency is introduced that we can't measure here: serialization, IO overhead, Spring Boot magic, etc

This easily lets us know that improving the performance of the permission check might be well worth the effort. Since this information is in the logs we can also monitor and graph the performance of the data retrievel latency (ORDERABLE_SERVICE_SEARCH) in real-time with a well crafted search on our logs.

SLF4J Profile Conventions

- Use the Profiler in Controller methods for code that's released to production. While in development you can use a Profiler anywhere you wish, it tends to clutter the code and the logs longer term. A few well placed Profiler.start() statements, left in the Controller however, can pay dividends longer term when performance issues need to be diagnosed in implementations.
- Prepend the HTTP operation to the beginning of the name. So `GET_ORDERABLES_SEARCH` and not *ORDERABLES_SEARCH*.
- Prefer all upper-case snake_case. e.g. `GET_ORDERABLES_SEARCH` never `getOrderablesSearch`.
- Be descriptive and strategic in your Profiler.start() names and locations. E.g. use a new Profiler.start() before a block/method that does something unlike the code before it: checking permissions, retrieving data, performing an update, returning a result. Use names that are clear for those who'll be reading the logs in production systems years from now.

Logging

In our service-architecture we have many different components where latency can be introduced and therefore logs we need to examine when diagnosing where time is being spent:

From the top of the stack down:

1. The Amazon ELB: typically the first place a request arrives, there is usually a very minor bit of latency incurred here. ELB logging if turned on is typically logged to S3.
2. Nginx reverse-proxy: Nginx is *the* place for finding HTTP operations. Requests from clients are routed through Nginx to upstream (aka backend) Services, and from service to service. The Nginx access log is the first place to see how long it took to process the request and how much time was spent in an upstream service performing the operation.
3. Service HTTP access log: these (tomcat) access logs are not always prominent however they can be turned on to give an idea of how much time the Service's HTTP server spent serving the request as opposed to how much time was spent transmitting the data. With good network connectivity between Nginx and backend Service (typically localhost), this is rarely an issue, though it can sometimes uncover hidden issues.
4. Service's Profiler statements: these logging statements from Java code are treated like all other Java logging statements and are channeled through our centralised `Rsyslog` container to be aggregated and written to disk (and later picked up by log monitoring service - Scalyr).
5. Database: queries take time, transactions can block, etc. Database logs can uncover both the time specific queries take as well as the actual SQL that's being run in the database. These logs are typically sourced and monitored through the RDS service (and Scalyr).

Lets look at an example of a call seen by Nginx and the Profiler.

Service's Profiler (again):

```
2017-07-24T19:49:45+00:00 e2f424e5b617 [nio-8080-exec-5] DEBUG
org.openlmis.referencedata.web.OrderableController #012+ Profiler
[GET_ORDERABLES_SEARCH]#012|
-- elapsed time [CHECK_ADMIN_RIGHT] 1173.997 milliseconds.#012|
-- elapsed time [ORDERABLE_SERVICE_SEARCH] 199.251 milliseconds.#012|
-- elapsed time [ORDERABLE_PAGINATION] 0.255 milliseconds.#012|
-- Total [GET_ORDERABLES_SEARCH] 1373.511 milliseconds.
```

Nginx access log:

```
10.0.0.238 - - [24/Jul/2017:19:49:45 +0000] "POST /api/orderables/search HTTP/1.1"␣
↪200 18455 "-" "Java/1.8.0_92" 1.401 0.000 1.401 1.401 .
```

Read the Nginx access log format for the details of what these numbers mean. What we can tell comparing these two is that:

- the total time to the user (just for this operation, not a web-page) was 1.4 seconds.
- All of that time was spent by the Reference Data service (because response time == upstream time).
- There is 28ms of latency not accounted for in our Profiler. It could be time spent in serialization of Java objects, Spring Boot overhead, tomcat overhead, network overhead (e.g. we were suffering from a 200ms delay due to a TCP configuration being off previously).
- Our user must be on a fast network connection, as Nginx spent the same time serving the response as it did getting the results from the upstream server. (a bit oversimplified).
- Approx 18.5KB was returned in this Orderables Search.

RESTful representations and the JPA to avoid

Avoid loading entities unnecessarily

Don't load an entity object if you don't have to; use Spring Data JPA `exists()` instead. A good example of this is in the RightService for Reference Data. The `checkAdminRight()` checks for a user when it receives a user-based client token. If the user is checking their own information, they only need to verify the existence of the user, instead of getting the full User info (using findOne()). Spring Data JPA's `CrudRepository` supports this through the method `exists()`.

In Spring Data JPA 1.11's (shipped in Spring Boot 1.5+) *CrudRepository* ships with `exists()` support for more than just the primary key column using Projections.

For example, take this bit of code that was found when searching for Orderables by a Program's code:

```
// find program if given
Program program = null;
if (programCode != null) {
  program = programRepository.findByCode(Code.code(programCode));
  if (program == null) {
    throw new ValidationMessageException(ProgramMessageKeys.ERROR_NOT_FOUND);
  }
}
```

This requires a trip to the database, which will need to pull the entire Program entity, back to the Service which will then turn it into a Java object... which will finally do what we actually wanted and check if the Program is null. Using an exists check, we can write code such as:

```
// find program if given
Code workingProgramCode = Code.code(programCode);
if ( false == workingProgramCode.isBlank()
    && false == programRepository.existsByCode(workingProgramCode) ) {
  throw new ValidationMessageException(ProgramMessageKeys.ERROR_NOT_FOUND);
}
```

The important part here is the use of the repositories `existsByCode(...)`, which is a Spring Data projection. This will avoid pulling the row, avoid turning a row into a Java object, and in general can save upwards of 100ms as well as the extra memory overhead. If the column is indexed (and well indexed), the database may even avoid a trip to disk, which typically can bring this check in under a millisecond.

Use Database Paging

Database paging is vastly more performant and efficient than Java paging or not paging at all. How much more? Before the Orderable's search resource was paged in the database, it was paged in Java. In Java pulling a page of only 10 Orderables out of a set of 10k Orderables took around 20 seconds. After switching to database paging, this same operation took only 2 seconds (10x more performant) and of that 95% of those 2 seconds are spent in an unrelated permission check.

The database paging pattern was established and as of this writing is not well enough adopted. Remember when paging to:

1. Follow the pagination API conventions.
2. Use Spring Data Pageable all the way to the query.
3. Spring Data projection makes this easy (more so in 1.11+). So code like this just works:

   ```
   @Query("SELECT o FROM Orderable o WHERE o.id in ?1")
   Page<Orderable> findAllById(Iterable<UUID> ids, Pageable pageable);
   ```

4. If it's an `EntityManager.createQuery()`, you'll need to run 2 queries: one for a `count()` and one for the (sub) list.
5. If you're a client, *use* the query parameters to page the results - otherwise our convention will be to return the largest page we can to you, which is slower.

Follow the pattern in Orderable search.

Eager Fetching & Lazy Loading

Eager fetching and lazy loading refer to the loading strategy an ORM takes when loading related Entities to the one that you're interested in. When done right, eager fetching can eliminate the N+1 problem in the next section. When done wrong, your service can consume all it's available memory and stall out.

Most often eager loading is not the right strategy to choose, and while Hibernate's default is to always use lazy loading, we should remember that Hibernate uses the JPA recommendation to lazily load all *ToMany relationships and eagerly fetch *ToOne relationships.

Eagerly fetching *ToOne relationships is not wrong, however we can't talk about eager fetching and lazy loading without analyzing what the typical uses of retrieving data/entities is. For that we'll look at the N+1 problem.

N+1 loading

In the simplest terms, N+1 loading occurs when an entity is loaded, related entities are marked as lazily loaded, and then the Java code (service, controller, etc) navigates to the related entity causing the JPA implementation to go load that related entity, which typically is an IO event back to the database. This is especially egregious when the related entity is actually some sort of collection (*ToMany relationship). For each element that's navigated to in the relationship, often another IO call occurs back to the database.

Avoiding N+1 loading is best done through designing for the common case. Take for example a User entity, which has a lazily loaded OneToMany relationship with RoleAssignments. We might think that the common case we should design for is updating a user and their RoleAssignments. If we design for this we'll likely place the full RollAssignment resource in the representation for GET and PUT of a User. Since the relation is lazily loaded we'll incur N+1 loads: 1 for the User and N for the # of RoleAssignments. If we changed the relation to be eagerly fetched, then we'd pull all N RollAssignments when any bit of Java code loaded the User - even if we just needed the User's ID or name.

The simplest solution therefore is to use a lazily loaded relation, and remove the full representations of RoleAssignments from the User resource. After all, updating a User is actually pretty uncommon compared to retrieving a User, or even retriving the User with RoleAssignments to check that user's rights. If we do actually need a User's RoleAssignments, we don't actually want to retrieve them with the User, rather we'll likely want a specific sub Resource of a User for managing their RoleAssignments. This sub-resource would typically look like:

- `/api/users/{id}`
- `/api/users/{id}/roleAssignments`

This would optimize the common case (just load a User to get their name/profile), and provide a seperate resource which could be optimized for pulling that User's RoleAssigments in one trip to the database.

Summary

- Build RESTful resource representations that are shallow: that is don't load more than just the single entity being asked for.
- No FETCH JOINS
- Don't use eager fetching unless it's really safe to do so. It might seem to solve the above problem, but it can go awry quickly. Just use lazy loading.
- During development you can set environment variables to show what SQL is actually being run by Hibernate.

Database JOINs are expensive

Simply put a database join is expensive. While our Services should not denormalize to avoid many joins, we should consider the advice in the *FlattenComplexStructures* section, especially when such a representation is used frequently by other clients.

Indexes

When done right an index can prevent the database from ever having to go to disk - a slow operation. Done wrong and a plethora of indexes can eat up memory and not prevent disk operations.

Some tips (PostgreSQL):

- The primary key is indexed. When you know what you want, using it's primary key, a UUID, is usually the most effecient.

- Foreign keys are not automatically indexed in PostgreSQL, however they almost always should be.
- You almost always want a B-tree index (the default).
- Unique columns are some of the best indicies, when it's not a unique column, keep in mind that low cardinality indexes negatively impact performance
- Don't over-index, each index takes up memory. Choose them based on the common search (i.e. WHERE clause) and prefer to search based on high-cardinality columns with indexes.
- More indexing tips

Flatten complex structures

We should take complex structures that do not change often, flattening and storing them in the database. This would create a higher expense in writes, but improve performance in reads. Since reads would be more common than writes, the trade-off is beneficial overall.

A good example here are the concept of permission strings. The role-based access control (RBAC) for users is complex, with users being assigned to roles potentially by program, facility, both, or neither. However, all of the rights that a user has can be represented by a set of permission strings, with complexity represented in different string formats. Formats as follows:

- RightName - for general rights
- RightName|FacilityUUID - for fulfillment rights
- RightName|FacilityUUID|ProgramUUID - for supervision rights

The different parts of the permission are in different parts of the string, and each part is delimited with a delimiter (pipe symbol in this case).

These strings, or each part of these strings, are saved as rows in a separate table and retrieved directly. This dramatically improves read performance, since we avoid retrieving the complex RBAC hierarchy and manipulating it in the Java code.

See https://groups.google.com/d/msg/openlmis-dev/wKqgpJ2RgBA/uppxJGJiAwAJ for further discussion about permission strings.

HTTP Cache

E-tag and if-none-match

HTTP Caching in a nut-shell is supporting the use of fields in an HTTP header that can help identify if a previous result is no longer valid. This can be very useful for the typical OpenLMIS user that is often in an environment with low network bandwidth.

In our Spring services this can be as simple as:

```
@RequestMapping(value = "/someResource", method = RequestMapping.GET)
public ResponseEntity<SomeEntity> getSomeResource(@PathVariable("id") UUID␣
↪resourceId) {
  ...
  // do work
  ...

  return ResponseEntity
    .ok()
```

(continues on next page)

(continued from previous page)

```
    .eTag(Integer.toString(someResource.hashCode()))
    .body(someResource);
}
```

The key points here are:

- someResource must accurately implement hashCode().
- The Object's hashCode is returned to the HTTP client (browser) in the :code'etag' header.
- On subsequent calls the HTTP client should include the HTTP header *if-none-match* with the previously returned *etag* value. If the etag value is the same, a HTTP 304 is returned, without a body, saving network bandwidth.

This simple implementation won't however save the server from processing the request and generating the `etag` from the Object's hashCode(). If this server operation is particularly expensive, further optimization should be done in the controller to use a field other than the `hashCode()` and to return early:

```
@RequestMapping(value = "/someResource", method = RequestMapping.GET)
public ResponseEntity<SomeEntity> getSomeResource(
  @RequestHeader(value="if-none-match") String ifNoneMatch,
  @PathVariable("id") UUID resourceId) {

  if (false == StringUtils.isBlank(ifNoneMatch)) {
    long versionEtag = NumberUtils.toLong(ifNoneMatch, -1);
    if (someResourceRepo.existsByIdAndVersion(resourceId, versionEtag)) {
      return ResourceEntity
        .ok()
        .etag(ifNoneMatch);
    }
  }

  ...
  // do work
  ...

  return ResponseEntity
    .ok()
    .eTag(Integer.toString(someResource.getVersion())
    .body(someResource);
}
```

The key to the above is using a property of an entity that changes every time the object changes, such as one marked with `@Version`, to use as the resource's etag. By storing the basis of the etag in the database, we can run a query which simply goes and sees if that entity still has that version, and if it does we can return a HTTP 304. The property used here could be anything, so long as we can search for it in a way that saves processing time (hint: a good choice with high-cardinality would be a multi-column index on the id and the version). Another good choice could be a LastModifiedDate.

Cache-control

WIP:

- no-cache
- private
- max-age

Performance

The OpenLMIS-UI is a large application that will be running in a web browser with less RAM and processing power than your computer. This is a fair statement, because if you are reading this, you are probably a developer.

This set of conventions is about detecting, diagnosing, and fixing common performance issues that have been a problem in the OpenLMIS-UI.

Blocking the DOM

Use asynchronous Javascript (promises) so you don't block the thread. This will cause web browers to think the OpenLMIS-UI is crashing, and it will try to close the browser tab.

Memory Leaks

This one is a bit tricky. It's fairly hard to create a memory leak in AngularJS unless you're mixing it with other external libraries that are not based on AngularJS(especially jQuery). Still, there are some things you need to remember while working with it, this article provides some general insight on how to find, fix and avoid memory leaks, for more detailed info I would suggest reading this article(it's awesome!).

Finding memory leaks

I won't lie, finding out if your application has some memory leaks is annoying, and localizing those leaks is even more annoying and can take a lot of time. Google Chrome devtools is incredible tool for doing this. All you need to do is:

1. open you application
2. go to the section you want to check for memory leaks
3. execute the workflow you want to check for memory leaks so any service or cached data won't be shown on the heap snapshot
4. open devtools
5. go to the Profiles tab
6. select Take Heap Snapshot
7. take a snapshot
8. execute the workflow
9. take a snapshot again
10. go to a different state
11. take a snapshot again
12. select the last snapshot
13. now click on the All objects select and choose Objects allocated between Snapshot 1 and Snapshot 2

This will show you the list of all objects, elements and so on, that were created during the workflow and are still residing in the memory. That was the easy part. Now we need to analyze the data we have and this might be quite tricky. We can click on object to see what dependency is retaining them. There is some color coding here that can be useful to you - red for detached elements and yellow for actual code references which you can inspect and see. It takes some time and experience to understand what's going here but it gets easier and easier as you go.

Anti-patterns

Here are some anti-pattern that you should avoid and how to fix them.

Event handlers using scope

Let's look at the following example. We have a simple directive that binds an on click action to the element.

```
(function() {

    'use strict';

    angular
        .module('some-module')
        .directive('someDirective', someDirective);

    function someDirective() {
        var directive = {
            link: link
        };
        return directive;

        function link(scope, element) {

            element.on('click', onClick);

            function onClick() {
                scope.someFlag = true;
            }
        }
    }

})();
```

The problem with this link function is that we've created a closure with context which retains the context, the scope and "then basically everything in the universe" until we unregister the handler from the element. That's right, even after the element is removed from the DOM it will still reside in the memory retained by the closure unless unregister the handler. To do this we need to add a handler for '$destroy' event to the scope object and then unregister the handler from the element. Here's an example how to do it.

```
(function() {

    'use strict';

    angular
        .module('some-module')
        .directive('someDirective', someDirective);

    function someDirective() {
        var directive = {
            link: link
        };
        return directive;

        function link(scope, element) {
```

(continues on next page)

(continued from previous page)

```
            element.on('click', onClick);

            scope.$on('$destroy', function() {

                //this will unregister the this single handler
                element.off('click', onClick);

                //this will unregister all the handlers
                element.off();
            });


            function onClick() {
                scope.someFlag = true;
            }
        }
    }

})();
```

Improper use of the $rootScope.$watch method

$rootScope.$watch can be a powerful tool, but it also requires some experience to use right. It allows the developers to create watchers that live through the whole application life and are only removed when they are explicitly said to unregister or when the application is closed, which may result in a huge memory leaks. Here are some tips on how to use them.

- Use $scope.$watch when possible! If you're using a watcher in a directive, it will have access to the scope object, add the watcher to it! This way we take advantage of AngularJS automatic watcher unregistration when the scope is deleted.
- Avoid using $rootScope.$watch in factories. Don't use it in factories unless you're completely sure what you're doing. Remember to unregister it when it is no longer needed! This takes us to the next bullet point.
- Use them in Services. Watching for current locale can be great example of that. We're using it with service, which is a singleton - it is only created once during application lifetime - and we want to watch for the current locale all the time we rather won't want to stop at any point.
- Unregister it if it is no longer needed. If you're sure you won't be needing that watcher any longer simply unregister it! Here's an example

```
var unregisterWatcher = $rootScope.$watch('someVariable', someMethod);
unregisterWatcher();
```

Using callback functions

Using callback isn't the safest idea either as it can cause some function retention. AngularJS gives us awesome tool to bypass that - promises. They basically gives us the same behavior and are retention-risk free!

1.6 Deployment

Deployment is done currently through Docker and Docker Compose. A living example of deployment scripts and documentation that the OpenLMIS product uses to deploy demo and CD environments is available in the openlmis-deployment repository. Documentation from that repository is listed below:

1.6.1 Recommended Deployment Topology

OpenLMIS uses and therefore recommends that most deployments utilize Amazon Web Services (AWS). However OpenLMIS is in no way tied to only being deployed on AWS.

The basic ingredients of an OpenLMIS deployment are:

- a domain name to reach the installation at (e.g. test.openlmis.org)
- a SSL certificate to make the communication to OpenLMIS secure over the web
- a computer/instance/etc that can run Docker Machine (as well as Compose, etc) with enough bandwidth, processing power, memory and storage to run many (6+) Services and associated utilities
- a computer/instance/etc that can run PostgreSQL for those Services
- credentials with an SMTP server to send emails

In AWS (a region close to your users) this would be:

- A DNS provider for your domain name (e.g. test.openlmis.org). This could be Route 53, however currently OpenLMIS deployments do not utilize this service.
- A SSL certificate from AWS Certificate Manager
- A ELB that can route to/from the OpenLMIS instance and serve the ACM SSL certificate (this becomes more useful when running out of Elastic IPs)
- an EC2 Instance (m4.large - 2vCPU, 8GiB memory, 30GB EBS store)
- an RDS Instance (db.t2 class instances are used below because most OpenLMIS deployments have long periods of inactivity where the t2 class' credits are able to accumulate. Choose another class if your deployment will be actively used for more than half the day):
  - For *local development*, *QA*, and *small private demos*: use Ref Distro's included database or a **db.t2.micro** (though you'll need to increase the max_connections parameter to >150)
  - For *CD*, limited *public demos*, limited *UAT*: **db.t2.medium**
  - For *medium and larger deployments* as well as *production*: **db.t2.large** *or larger* based on need:
    - When reports are frequently run
    - When the number of Products (full and non-full supply) > 500
    - When the number of Requisitions (historical and planned for next 2 processing periods) > 100,000
- a VPC for your EC2 and RDS instances, with appropriate security group - SSH, HTTP, HTTPs, Postgres (limit source to Security Group) at minimum.
- Amazon SES with either the domain (w/DKIM) verified or a specific from-address

For more information on setting this up, see the Provisioning section and also follow the link for backing up/restoring RDS. For more information on how to configure your RDS please visit RDS configuration page.

1.6.2 How to provision a single Docker host in AWS

If case the deployment target is one single host, then swarm is not needed.

In that case, refer to Provision-swarm-With-Elastic-ip.md for step 1, 2, 3, 5. Omit step 4, that should be sufficient to provision a single host deployment environment.

Note: choose **ubuntu** instead of amazon linux distribution. Even though this single host won't be running a swarm, ubuntu is still preferred over amazon linux distribution. Because docker-machine does not support provisioning amazon linux distribution. However, you can manually provision the single host, but then making that host remotely accessible would be tricky and involves a lot of manual steps.

Database

If deploying OpenLMIS with the included Docker Container for Postgres, then no further steps are needed. However this setup is recommended only for development / testing environments and not recommended for production installs.

Test and UAT environments in this repository demonstrate that Postgres could be installed outside of Docker and OpenLMIS services may be pointed to that Postgres server. Test and UAT both use Amazon's RDS service to help manage production-grade database services such as automated patch release updates, rolling backups, snapshots, etc.

Some notes on provisioning an RDS instance for OpenLMIS:

- Test and UAT are both capable of running on economical RDS instances: db.t2.micro
- When choosing a small RDS instance, the max number of connections are set based on an *ideal* number from RDS. OpenLMIS services tend toward using about 10 DB connections per service. Therefore Tests and UAT instances use a Parameter Group named Max Connections that increase this limit to 100. Larger, more expensive, instances likely won't have this limitation.
- RDS instances are in a private VPC and in the same availability zone as their EC2 instance and it's ELB. The security group used should be the same as used for the EC2 instance, though it should limit incoming PostgreSQL connections to only those from the security group.
- Don't forget to update the .env file used to deploy OpenLMIS with the correct Host, username and password settings.

1.6.3 How to provision a docker swarm for deployment in AWS (With ELB)

1. Network setup

Create a VPC for the new swarm cluster.

In it, there should be 2 subnets created(which is needed by AWS ELB later).

Ensure the subnets will assign public ip to EC2 instances automatically, so it's easier to ssh into them later.

2. Create EC2 instances

This step is similar to creating EC2 instances for any other type of purpose.

When creating those instances, make sure to select the VPC created in the previous step.

Mentally mark one of the instances as swarm manager, the rest of them will be regular nodes.

Note: choose **ubuntu** instead of amazon linux distribution. The amazon linux distribution has problems with docker 1.12, the version that has built in support for docker swarm. 1.12 is not available in amazon linux RPM yet. And it also lacks support for aufs, which is recommended by docker.

Make sure to open port 2376 insecurity group, this is the default port that docker-machine uses to provision.

3. Add ssh public key to the newly created EC2 instances

In order to access the EC2 instances, the public key of **the machine from which the provisioning will happen** need to be added to the target machine.

This is done by ssh into the EC2 instances, and then edit `[User Home Dir]/.ssh/authorized_keys` file to add your public key into it.

This will be needed by the `docker-machine create` command later.

4. Create ELB

The reason to create ELB is that AWS has a limit on how many elastic ips each account could have, the default is 5, which could be easily used up.

So in order for the swarm to be available via a constant address, an ELB is created to provide that constant url.

This is also why in the first step, there need to be 2 subnets, it's required by ELB.

When creating the ELB, make sure **TCP port 22 and 2376** are forwarded to the target EC2 instance. 22 is for ssh, 2376 is for docker remote communication. And also, make sure to choose classic ELB instead of the new one. The classic one allows TCP forwarding, the new one only supports http and https.

5. Enable health check

ELB only forwards to a target machine if the target is considered "healthy". And ELB determines the health of a target by pinging it.

So, in the EC2 instance chosen as the swarm manager, start apache2 service at port 80(or any other port you may prefer). Then in ELB settings, set it to ping that port.

For OpenLMIS the Nginx container starts itself automatically after the system is rebooted, so this ensures that **ELB will start forwarding immediately if the instance reboots**.

6. Provision all EC2 instances

Use this command:

```
docker-machine create --driver generic --generic-ip-address=[ELB Url]
--generic-ssh-key ~/.ssh/id_rsa --generic-ssh-user ubuntu name1
```

to provision the swarm manager.

Note: the –driver flag has support for AWS. But the intention to explicitly ***not*** use it is to make sure this provision step could apply to other host environments as well, not just AWS hosted machines.

The –generic-ip-address flag needs to be followed by the ip of ec2 instance(for the swarm manager, it should be the ELB Url).

The –generic-ssh-key flag needs to be followed private key, whose public key pair should have already been added in step 2.

The –generic-ssh-user flag needs to be followed by the user name, in the case of Ubuntu EC2 instances, the default user name is ubuntu.

Lastly, supply a **name** for the docker machine.

Do this **for all the EC2 instances**, to make sure docker is installed on all of them. (When doing this for the none manager nodes, the –generic-ip-address flag should be followed by their public ip that was automatically assigned, since ELB only forwards traffic to the manager node.)

7. Start swarm

Choose one of the EC2 instances as the swarm manager by:

`eval $(docker-machine env [name of the chosen one])` (the name in the [] should be one of the names used in the previous step)

Now your local docker command is pointing at the remote docker daemon, run:

```
docker swarm init
```

Then follow its console output to join the rest of the EC2 instances into the swarm. (it could be done by switching docker-machine env, or by using the -H flag of docker, the former is easier)

Since all the swarm node are in the same VPC, they can talk to each other by private ips which are static inside the VPC. **The swarm will regroup it self and maintain the manager-regular node structure even after EC2 instances are rebooted.**

8. Allow Jenkins to access swarm manager

In order for Jenkins to continuously deploy to the swarm, it needs access to the swarm manager.

In step 3, when the swarm manager EC2 instance was being provisioned. The docker-machine created some certificate files behind the scene.

Those files should be in the machine that the provision command was issued(not the machine that was being provisioned), under:

```
[User Home Dir]/.docker/machine/machines/[name of the swarm manager]
```

Those files need to be copied to jenkins.

In a Jenkins deployment job, **at the start of its build script**, add:

```
export DOCKER_TLS_VERIFY="1"
```

```
export DOCKER_HOST="tcp://[ELB Url that forwards to Swarm manager]"
```

```
export DOCKER_CERT_PATH="[path to the dir that contains certs]"
```

This will make following docker commands use the remote daemon, not the local one.

Now, Jenkins should be able to access and deploy to the swarm.

Node: Jenkins would only need access to the swarm manager, the other nodes are managed by the swarm manager. Jenkins does not need direct access to them.

1.6.4 How to provision a docker swarm for deployment in AWS (With Elastic IP)

1. Create EC2 instances

This step is similar to creating EC2 instances for any other type of purpose.

Mentally mark one of the instances as swarm manager, the rest of them will be regular nodes. Assign an elastic ip to the manager node.

Note: choose **ubuntu** instead of amazon linux distribution. The amazon linux distribution has problems with docker 1.12, the version that has built in support for docker swarm. 1.12 is not available in amazon linux RPM yet. And it also lacks support for aufs, which is recommended by docker.

Make sure to open port 2376, this is the default port that docker-machine uses to provision. And make sure they have auto assigned public ip(not elastic ip) so you can ssh into them.

2. Add ssh public key to the newly created EC2 instances

In order to access the EC2 instances, the public key of **the machine from which the provisioning will happen** need to be added to the target machine.

This is by ssh into the EC2 instances, and then edit `[User Home Dir]/.ssh/authorized_keys` file to add your public key into it.

3. Provision all EC2 instances

With this command:

```
docker-machine create --driver generic --generic-ip-address=*.*.*.*
--generic-ssh-key ~/.ssh/id_rsa --generic-ssh-user ubuntu name1
```

Note: the –driver flag has support for AWS. But the intention to explicitly ***not*** use it is to make sure this provision guide could apply to any host machine, not just AWS hosted machines.

The –generic-ip-address flag needs to be followed by the ip of ec2 instance. For the manager node, use the elastic ip, for the others, use the temp ips assigned by aws.

The –generic-ssh-key flag needs to be followed private key, whose public key pair should have already been added in step 2.

The –generic-ssh-user flag needs to be followed by the user name, in the case of Ubuntu EC2 instances, the default user name is ubuntu.

Lately, supply a **name** for the docker machine.

Do this **for all the EC2 instances**, to make sure docker is installed on all of them.

4. Start swarm

Choose one of the EC2 instances as the swarm manager by:

`eval $(docker-machine env [name of the chosen one])` (the name in the [] should be one of the names used in the previous step)

Now your local docker command is pointing at the remote docker daemon, run:

```
docker swarm init
```

Then follow its console output to join the rest of the EC2 instances into the swarm. (it could be done by switching docker-machine env, or by using the -H flag of docker, the former is easier)

5. Allow Jenkins to access swarm manager

In order for Jenkins to continuously deploy to the swarm, it needs access to the swarm manager.

In step 3, when the swarm manager EC2 instance was being provisioned. The docker-machine created some certificate files behind the scene.

Those files should be in the machine that the provision command was issued(not the machine that was being provisioned), under:

```
[User Home Dir]/.docker/machine/machines/[name of the swarm manager]
```

Those files need to be copied to jenkins(if the provision was done on Jenkins, then there is no need to copy).

In a Jenkins deployment job, **at the start of its build script**, add:

```
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://[ip of the swarm manager]"
export DOCKER_CERT_PATH="[path to the dir that contains certs]"
```

This will make following docker commands use the remote daemon, not the local one.

Now, Jenkins should be able to access and deploy to the swarm.

Node: Jenkins would only need access to the swarm manager, the other nodes are managed by the swarm manager. Jenkins does not need direct access to them.

1.6.5 Deployment Environments

Scripts in this directory are meant to be ran in Jenkins.

Overview

- `shared/` contains scripts for the Jenkins job(s):
  - `init_env.sh` is run in Jenkins to copy the docker environment files (has secure credentials) from `JENKINS_HOME/credentials/` to the current job's workspace
  - `pull_images.sh` always pulls/refreshes the infrastructure images (e.g. db, logs, etc), and then at the end will pull the image for the service that the Jenkins job is attempting to deploy (e.g. requistion, auth, referencedata, etc).
  - `restart.sh` is paramartized by Jenkins to either `keep` or `wipe` volumes (e.g. database and logging volumes). When run this brings the deployed reference distribution down, and then back up. After it's brought up, the `nginx.tmpl` file is copied directly into the running nginx container just started.
  - `nginx.tmpl` is the override of the nginx template for docker and proxying - this is a copy from openlmis-ref-distro. See `restart.sh` for how it's used.
- `test_env` has a compose file which is the Reference distribution, and a script for Jenkins to kick everything off.
- `uat_env` has a compose file which is the Reference distribution, and a script for Jenkins to kick everything off.
- `demo_env` has a compose file which is the latest stable version of the Reference distrubution, and a script for Jenkins to kick everything off.

Local Usage

These scripts **won't** work out of the box in a dev's local machine, to make them work, you need a few files that are present in Jenkins but not in your local clone of this repo:

1. The .env file

   This file is present in Jenkins. It is copied to the workspace of a deployment job(either Jenkins slave or master) every time that job is ran.

This file is **not** included in this repo because the db credentials could be different for different deployment environments. The default .env file that is used during development and CI is open in github, making it not suitable for deployment purposes.

2. The cert files for remotely controlling docker daemon deployment target

   These files should not be included in this public repo for obvious reasons.

   Similar to the .env file, they are also present in Jenkins and copied to a deployment job's workspace(either Jenkins slave or master) every time it is ran.

To get these files, you need to be able to ssh to the Jenkin's host instance.

It's **not** recommended that you connect to the remote deployment environments, however if you have to:

1. pull the remote cert files to a local directory. They are currently located under `JENKINS_HOME/credentials/` in the `Host` directories. `JENKINS_HOME` would currently be `/var/lib/jenkins`.
2. With the above cert files, you could control the remote docker machine by copying the certs to a local `certs/` directory, and then running the following in your shell:

```
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://<path-to-elb-of-docker-host>:2376"
export DOCKER_CERT_PATH="${PWD}/certs"
```

e.g. a current elb path for test is `elb-test-env-swarm-683069932.us-east-1.elb.amazonaws.com`

After this, running docker commands in your shell will be ran against the remote machine. e.g. docker inspect, logs, etc

How to backup persisted data?

if using ref distro's included db container

1. ssh into the docker host that you want, either test env or UAT env. Or use the technique above to connect your docker client to the remote host as needed
2. run this command

   ```
   docker exec -t [PostgresContainerName] /usr/lib/postgresql/9.4/bin/
   pg_dumpall -c -U [DBUserName] > [DumpFileName].sql
   ```

   PostgresContainerName is usually testenv_db_1 or uatenv_db1, you can use `docker ps` to find out. **DBUserName** is the one that was specified in the .env file, it's usually just "postgres". **DumpFileName** is the file name where you want the back up to be stored **in the host machine**.

using Amazon's RDS

RDS provides a number of desirable features that are more ideal for production environments, including automated backups. To backup and restore the OpenLMIS database when using RDS, follow Amazon's documentation: http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_CommonTasks.BackupRestore.html

1.6.6 RDS configuration

This guide assumes a clean RDS instance has just been created.

1. Setting up PostGIS for RDS

   PostGIS is used by some OpenLMIS services to provide better geographical support. Amazon provides a great guide on how to do it under this link. Make sure to execute those instructions in the database containing OpenLMIS schemas, rather than *postgres*.

2. Adding UUID extension on RDS. Some services require the *uuid-ossp* extension in order to randomly generate UUIDs in SQL. In order to ensure OpenLMIS works properly with RDS, you *need* to run the following command to install the extension:

   ```
   CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
   ```

1.7 Versioning and Releasing

1.7.1 Micro-Services are Versioned Independently

OpenLMIS version 3 introduced a micro-services architecture where each component is versioned and released independently. In addition, all the components are packaged together into a **Reference Distribution**. When we refer to OpenLMIS 3.X.Y, we are talking about a release of the Reference Distribution, called the ref-distro in GitHub. The components inside ref-distro 3.X.Y have their own separate version numbers which are listed on the Release Notes.

The components are each semantically versioned, while the ref-distro has "milestone" releases that are conducted roughly quarterly (every 3 months we release 3.2, 3.3, etc). Each ref-distro release includes specific versions of the other components, both service components and UI components.

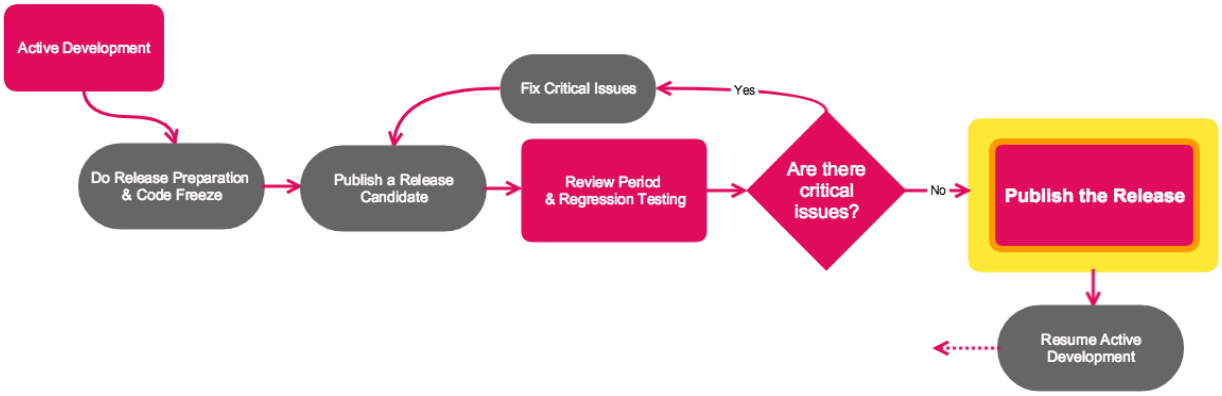
Where We Publish Releases

All OpenLMIS source code is available on GitHub, and the components have separate repositories. Releases are tagged on GitHub for all components as well as the ref-distro. Releases of some components, such as the service components and UI components, are also published to Docker Hub as versioned docker images. In addition, we publish releases of the service utility library to Maven. Moreover, release builds are kept on Jenkins forever by default.

1.7.2 Release Process

Starting with OpenLMIS 3.2.1, each release of the Reference Distribution will go through a Release Candidate process. A Release Candidate will be shared for a Review Period of at least one week to allow for manual regression testing and to allow community review and input. The goal is that we catch and fix issues in order to put out higher-quality releases.

The following diagram illustrates the process, and each step is explained in detail below.



Active Development

- Multiple agile teams develop OpenLMIS services/components and review and incorporate Pull Request contributions
- Microservices architecture provides separation between the numerous components
- Automated test coverage prevents regressions and gives the team the safety net to release often
- Continuous Integration and Deployment (CI/CD) ensures developers get immediate feedback and QA activities can catch issues quickly
- Code is peer reviewed during Jira ticket workflow and in Pull Requests
- Documentation, CHANGELOGs and demo data are kept up-to-date as code development happens in each service/component

Do Release Preparation & Code Freeze

- Verify the pre-requisites, including all automated tests are passing and all CHANGELOGs are up-to-date; see Release Prerequisites below under Rolling a Release
- Conduct a manual regression test cycle 1-2 weeks before the release, if possible
- Begin a Code Freeze: shift agile teams' workloads to bugs and clean-up, rather than committing large new features or breaking changes ("slow down" 1-2 weeks before release)

  Note: Branching is not part of the current process (see 'We Prefer Coordination over Branching' section below), but may be adopted in the future along with CI/CD changes to support more teams working in parallel.
- Write draft Release Notes including sections on 'Compatibility', 'Changes to Existing Functionality', and 'New Features'
- Schedule or timing for releases is documented above and may be discussed and revised by the community

Publish a Release Candidate

- Each component that has any changes since the last release is released and semantically versioned (e.g., openlmis-requisition:6.3.4 or openlmis-newthing:1.0.0-beta)

  Note: Usually, all components are released with the Reference Distribution. Sometimes, due to exceptional requests, the team may release a service/component at another time even when there is not another Reference Distribution release.

- Reference Distribution Release Candidate is released with these components (e.g., openlmis-ref-distro:3.7.0-rc1)

  Note: We archive permanent documentation for every release, but not for every release candidate.

- Share Release Candidate with the OpenLMIS community along with the draft Release Notes and invite testing and feedback

See the 'Rolling a Release' section further below for the specific technical steps to build, tag and publish a release of components and the Reference Distribution.

Review Period

The overall timeline for review period starts when the first Release Candidate is shared and should last at least 1 week, during which time subsequent Release Candidates may be published.

- The community is alerted of the upcoming release candidate date and review period via Slack and the listservs.
- Active Development is paused and the only development work that happens is release-critical bug fixes or work on branches (note: branches are not yet recommended and not supported by CI/CD).
- The team conducts a full manual regression test cycle (including having developers conduct testing) according to the Release Candidate Test Plan. For an example, see the 3.2.1 Regression Test Plan. The test plan is included in the final Release Notes.
- Community members are requested to conduct user acceptance testing to submit bugs and issues with the release candidate. Members can review and leverage the OpenLMIS manual test cases.
- OpenLMIS will run automated performance testing and review results.
- Manual bug reports are submitted in Jira, see the Reporting bugs section for details on how to submit bugs to OpenLMIS. All bugs and issues related to the Release Candidate must be associated with the specific Release Candidate Bugs epic. Bugs can be identified in the code, documentation, and translations.
- A triage team will review and triage all bugs submitted on a daily bases during the review period.

Fix Critical Issues

Are there critical bugs or issues associated with the release candidate? If not, after the first Release Candidate (RC1) OpenLMIS may move directly to a release. Otherwise, OpenLMIS will fix critical issues and publish a new Release Candidate (e.g. RC2).

- Developers fix critical issues in code, documentation, and translations. Only commits for critical issues will be accepted. Other commits will be rejected.
- Every commit is reviewed to determine whether portions or all of the full regression test cycle must be repeated
- And we continue to hold every ticket up to our on-going guidelines and expectations:
  - Every commit is peer reviewed and manually tested, and should include automated test coverage to meet guidelines
  - Every commit must correspond to a Jira ticket and have gone through review and QA steps, and have Zephyr test cases in Jira

Once critical issues are fixed, publish a new Release Candidate and conduct another Review Period.

Publish the Release

When a Release Candidate has gone through a Review Period without any critical issues found, then this release candidate becomes the Golden Master to be published as an official release of OpenLMIS.

- Update the Release Notes to state that this is the official release and include the date
- Release the Reference Distribution; the exact code and components in the Golden Master Release Candidate are tagged as the OpenLMIS Reference Distribution release with a version number tag (e.g. openlmis-ref-distro:3.7.0)
- Share the Release with the OpenLMIS community along with the final Release Notes

After publishing the release, Active Development can resume.

Releasing components outside of a Ref Distro release (draft)

At times OpenLMIS will release stable components outside the process of releasing a new Ref Distro. When a component is released without the Ref Distro it is done on its own - without the benefits of the rigirous release process of the Ref Distro.

Any component may be released at any time. However to release a component, it must pass the following criteria:

- All automated tests of the component must be passing.
- All dependancies must also be co-released and their automated tests passing if a change in the dependancy is needed to successfully release the component.
- The release must be stable - no half-finished features or fixes.
- Since the release of the component is outside of the Ref Distro release process, **implementers should be careful** in taking such releases as they haven't been fully tested in the larger context of the Ref Distro.

Implementation Release Process

A typical OpenLMIS implementation is composed of multiple core OpenLMIS components plus some custom components or extensions, translations and integrations. It is recommended that OpenLMIS implementations follow a similar process as above to receive, review and verify that updates of OpenLMIS perform correctly with their customizations and configuration.

Key differences for implementation releases:

- **Upstream Components**: Implementations treat the OpenLMIS core product as an "upstream" vendor distribution. When a new core Release Candidate or Release are available, they are encouraged to pull the new upstream OpenLMIS components into the implementations CI/CD pipeline and conduct testing and review.
- **Independent Review**: It is critical for the implementation to conduct its own Review Period. It may be a process similar to the diagram above, with multiple Release Candidates for that implementation and with rounds of manual regression testing to ensure that all the components (core + custom) work together correctly.
- **Conduct Testing/UAT on Staging**: Implementations should apply Release Candidates and Releases onto testing/staging environments before production environments. Testing should be conducted on an environment that is a mirror of production (with a recent copy of production data, same server hardware, same networks, etc). There may be a full manual regression test cycle or a shorter smoke test as part of applying a new version onto the production environment. There should also be a set of automated tests and performance tests, similar to the core release process above, but with production data in place to verify performance with the full data set.
- **Follow Best Practices**: When working with a production environment, follow all best practices: schedule a downtime/maintenance window before making any changes; take a full backup of code, configuration and data

at the start of the deployment process; test the new version before re-opening it to production traffic; always have a roll-back plan if issues arise in production that were not caught in previous testing.

1.7.3 Release Numbering

Version 3 components follow the Semantic Versioning standard:

- **Patch** releases with bug fixes, small changes and security patches will come out on an as-needed schedule (1.0.1, 1.0.2, etc). Compatibility with past releases under the Major.Minor is expected.
- **Minor** releases with new functionality will be backwards-compatible (1.1, 1.2, 1.3, etc). Compatibility with past releases under the same Major number is expected.
- **Major** releases would be for non-backwards-compatible API changes. When a new major version of a component is included in a Reference Distribution release, the Release Notes will document any migration or upgrade issues.

The Version 3 Reference Distribution follows a milestone release schedule with quarterly releases. Release Notes for each ref-distro release will include the version numbers of each component included in the distribution. If specific components have moved by a Minor or Major version number, the Release Notes will describe the changes (such as new features or any non-backwards-compatible API changes or migration issues).

Version 2 also followed the semantic versioning standard.

Goals

Predictable versioning is critical to enable multiple country implementations to share a common code base and derive shared value. This is a major goal of the 3.0 Re-Architecture. For example, Country A's implementation might fix a bug or add a new report, they would contribute that code to the open source project, and Country B could use it; and Country B could contribute something that Country A could use. For this to succeed, multiple countries using the OpenLMIS version 3 series must be upgrading to the latest Patch and Minor releases as they become available. Each country shares their bug fixes or new features with the open source community for inclusion in the next release.

Pre-Releases

Starting with version 3, OpenLMIS supports pre-releases following the Semantic Versioning standard.

Currently we suggest the use of **beta** releases. For example, 3.0 Beta is: 3.0.0-beta.

Note: the use of the hyphen consistent with Semantic Versioning. However a pre-release SHOULD NOT use multiple hyphens. See the note in **Modifiers** on why.

Modifiers

Starting with version 3, OpenLMIS utilizes build modifiers to distinguish releases from intermediate or latest builds. Currently supported:

Modifier: SNAPSHOT **Example**: 3.0.0-beta-SNAPSHOT **Use**: The SNAPSHOT modifier distinguishes this build as the latest/cutting edge available. It's intended to be used when the latest changes are being tested by the development team and should not be used in production environments.

Note: that there is a departure with Semantic Versioning in that the (+) signs are not used as a delimiter, rather a hyphen (-) is used. This is due to Docker Hub not supporting the use of plus signs in the tag name.

For discussion on this topic, see this thread. The 3.0.0 semantic versioning and schedule were also discussed at the Product Committee meeting on February 14, 2017.

We Prefer Coordination over Branching

Because each component is independently, semantically versioned, the developers working on that component need to coordinate so they are working towards the same version (their next release).

Each component's repository has a version file (gradle.properties or project.properties) that states which version is currently being developed. By default, we expect components will be working on the master branch towards a Patch release. The developers can coordinate any time they are ready to work on features (for a Minor release).

If developers propose to break with past API compatibility and make a Major release of the component, that should be discussed on the Dev Forum. They should be ready to articulate a clear need, to evaluate other options to avoid breaking backwards-compatibility, and to document a migration path for all existing users of the software. Even if the Dev Forum and component lead decide to release a Major version, we still require automated schema migrations (using Flyway) so existing users will have their data preserved when they upgrade.

Branching in git is discouraged. OpenLMIS does not use git-flow or a branching-based workflow. In our typical workflow, developers are all contributing on the master branch to the next release of their component. If developers need to work on more than one release at the same time, then they could use a branch. For example, if the component is working towards its next Patch, such as 1.0.1-SNAPSHOT, but a developer is ready to work on a big new feature for a future Minor release, that developer may choose to work on a branch. Overall, branching is possible, but we prefer to coordinate to work together towards the same version at the same time, and we don't have a branch-driven workflow as part of our collaboration or release process.

Code Reviews and Pull Requests

We expect all code committed to OpenLMIS receives either a review from a second person or goes through a pull request workflow on GitHub. Generally, the developers who are dedicated to working on OpenLMIS itself have commit access in GitHub. They coordinate in Slack, they plan work using JIRA tickets and sprints, and during their ticket workflow a code review is conducted. Code should include automated tests, and the ticket workflow also includes a human Quality Assurance (QA) step.

Any other developers are invited to contribute to OpenLMIS using Pull Requests in GitHub at any time. This includes developers who are implementing, extending and customizing OpenLMIS for different local needs.

For more about the coding standards and how to contribute, see **contributionGuide.md**.

Future Strategies

As the OpenLMIS version 3 installation base grows, we expect that additional strategies will be needed so that new functionality added to the platform will not be a risk or a barrier for existing users. Feature Toggles is one strategy the technical community is considering.

1.7.4 Rolling a Release

Below is the process used for creating and publishing a release of each component as well as the Reference Distribution (OpenLMIS 3.X.Y).

Goals

What's the purpose of publishing a release? It gives us a specific version of the software for the community to test drive and review. Beta releases will be deployed with demo data to the UAT site uat.openlmis.org. That will be a public, visible URL that will stay the same while stakeholders test drive it. It will also have demo data and will not be automatically wiped and updated each time a new Git commit is made.

Prerequisites

Before you release, make sure the following are in place:

- Demo data and seed data: make sure you have demo data that is sufficient to demonstrate the features of this release. Your demo data might be built into the repositories and used in the build process OR be prepared to run a one-time database load script/command.
- Features are completed for this release and are checked in.
- All automated tests pass.
- Documentation is ready. For components, this is the CHANGELOG.md file, and for the ref-distro this is a Release Notes page in the wiki.

Releasing a Component (or Updating the Version SNAPSHOT)

Each component is always working towards some future release, version X.Y.Z-SNAPSHOT. A component may change what version it is working towards, and when you update the serviceVersion of that component, the other items below need to change.

These steps apply when you change a component's serviceVersion (changing which -SNAPSHOT the codebase is working towards):

- If the component that you are about to release depends on the openlmis-service-util, verify that it uses a stable version of that library. If it uses a snapshot version, a release of openlmis-service-util is required before you can proceed.
- Within the component, set the **serviceVersion** property in the **gradle.properties** file to the new -SNAPSHOT you've chosen.
  - See Step 3 below for details.
- Update **openlmis-ref-distro** to set **docker-compose.yml** to use the new -SNAPSHOT this component is working towards.
  - See Step 5 below for details.
  - Use a commit message that explains your change. EG, "Upgrade to 3.1.0-SNAPSHOT of openlmis-requisition component."
- Update **openlmis-deployment** to set each **docker-compose.yml** file in the deployment/ folder for the relevant environments, probably uat_env/, test_env/, but not demo_env/
  - See Step 7 below for details.
  - Similar to above, please include a helpful commit message. (You do not need to tag this repo because it is only used by Jenkins, not external users.)
- Update **openlmis-contract-tests** to set each **docker-compose...yml** file that includes your component to use the new -SNAPSHOT version.
  - Similar to the previous steps, see the lines under "services:" and change its version to the new snapshot.
  - You do not need to tag this repo. It will be used by Jenkins for subsequent contract test runs.
- (If your component, such as the openlmis-service-util library, publishes to Maven, then other steps will be needed here.)

Patch Releasing a Component

1. Create a hotfix branch that includes 'rel-' prefix and the patch version, e.g. 'rel-10.0.1'
2. Set the **serviceVersion** property in the **gradle.properties** file to the patch version with SNAPSHOT suffix, e.g. 10.0.1-SNAPSHOT.
3. Make sure that Jenkins builds the branch successfully. If needed, run the job with 'contractTestsBranch' parameter set to the branch of **contract-tests** repository containing your ref-distro release, e.g. 'v3.3.0'.
4. Run performance tests
5. If all passes, remove SNAPSHOT suffix from the **serviceVersion** property.
6. Verify that the patch release is available on docker hub.

Releasing the Reference Distribution (openlmis-ref-distro)

When you are ready to create and publish a release (Note that version modifiers should not be used in these steps - e.g. SNAPSHOT):

1. Select a tag name (e.g. '3.3.0') based on the numbering guidelines above.
2. The service utility library should be released prior to the Services. Publishing to the central repository may take some time, so publish at least a few hours before building and publishing the released Services:
   (a) Update the serviceVersion of GitHub's openlmis-service-util
   (b) Check Jenkins built it successfully
   (c) At Nexus Repository Manager, login and navigate to Staging Repositories. In the list scroll until you find **orgopenlmis-NNNN**. This is the staged release.
   (d) Close the repository, if this succeeds, release it. More information.
   (e) Wait 1-2 hours for the released artifact to be available on Maven Central. Search here to check: https://search.maven.org/
   (f) In each OpenLMIS Service's build.gradle, update the dependency version of the library to point to the released version of the library (e.g. drop 'SNAPSHOT')
3. In each service, branch for release. The branch name should start with 'rel-' followed by version of the service, e.g. 'rel-6.0.0'. Set the **serviceVersion** property in the **gradle.properties** file to the next version, which should simply be the version without the SNAPSHOT suffix (e.g. if the serviceVersion was 6.0.0-SNAPSHOT, it should be 6.0.0). Push this to GitHub, then log on to GitHub and create a release tagged with the same tag. Note that GitHub release tags should start with the letter "v", so '6.0.0' would be tagged 'v6.0.0'. Choose the release branch to use as the Target. Also, when you create the version in GitHub check the "This is a pre-release" checkbox if indeed that is true.
   (a) Do this for each service/UI module in the project, including the API services and the UI repos (note: in those repos, the file is called project.properties, not gradle.properties). DON'T update the Reference Distribution yet.
   (b) Do we need a code freeze? We do not need a "code freeze" process. We are branching for release, and everyone can keep committing further work on master as usual. Updates to master will be automatically built and deployed at the Test site, but not the UAT site.
   (c) **Confirm** that your release tags appear in GitHub and in Docker Hub: First, look under the Releases tab of each repository, eg https://github.com/OpenLMIS/openlmis-requisition/releases. Next, look under Tags in each Docker Hub repository. eg https://hub.docker.com/r/openlmis/requisition/tags/ . You'll need to wait for the Jenkins jobs to complete and be successful so give this a few minutes. Note: After tagging each service, you may also want to change the serviceVersion again so that future commits are tagged on Docker

Hub with a different tag. For example, after releasing '6.0.0' you may want to change the serviceVersion to '6.0.1-SNAPSHOT'. You need to coordinate with developers on your component to make sure everyone is working on 'master' branch towards that same next release.

(d) Finally, on Jenkins, identify which build was the one that built and published to Docker/Maven the release. Press the Keep the build forever button.

4. Update **.env** in **openlmis-ref-distro** with the release tag name chosen (e.g '3.3.0')

   (a) For each of the services (and the Reference UI) deployed as the new version on DockerHub, update the version in the **.env** file to the version you're releasing.

   (b) Commit this change and tag the openlmis-ref-distro repo with the release being made. Note: There is consideration underway about using a git branch to coordinate the ref-distro release, to perform this step and the next one.

5. In order to publish the openlmis-ref-distro documentation to ReadTheDocs:

   (a) Edit **collect-docs.py** to change links to pull in specific version tags of README files. In that script, change a line like `urllib.urlretrieve("https://raw.githubusercontent.com/OpenLMIS/openlmis-referencedata/master/README.md", "developer-docs/referencedata.md")` to `urllib.urlretrieve("https://raw.githubusercontent.com/OpenLMIS/openlmis-referencedata/v3.0.0/README.md, "developer-docs/referencedata.md")`

   (b) Edit **index.rst** and the ERD RST files under **docs/source/components** to change links to pull in specific build version numbers of static API documentation and ERD zip files. In those files, change a URL like `http://build.openlmis.org/job/OpenLMIS-auth-pipeline/job/master/lastSuccessfulBuild/artifact/api-definition.html` to `http://build.openlmis.org/job/OpenLMIS-auth-pipeline/job/master/362/artifact/api-definition.html` and `http://build.openlmis.org/job/OpenLMIS-auth-pipeline/job/master/lastSuccessfulBuild/artifact/erd-auth.zip` to `http://build.openlmis.org/job/OpenLMIS-auth-pipeline/job/master/323/artifact/erd-auth.zip`

   (c) To make your new version visible in the "version" dropdown on ReadTheDocs, it has to be set as "active" in the admin settings on readthedocs (admin -> versions -> choose active versions). Once set active the link is displayed on the documentation page (it is also possible to set default version).

6. Create a branch of **openlmis-contract-tests** named by the tag (e.g. v3.3.0) and update **.env** file to include newly released components.

7. Update **.env** in **openlmis-deployment** for the Demo v3 deployment script with the release chosen which is at https://github.com/OpenLMIS/openlmis-deployment/blob/master/deployment/demo_env/.env

   (a) Make sure to coordinate with the OpenLMIS Community Manager when redeploying to Demo v3.

   (b) For each of the services deployed as a the new version on DockerHub, update the version in the **.env** file to the version you're releasing.

   (c) Commit this change. (You do not need to tag this repo because it is only used by Jenkins, not external users.)

8. Kick off the **OpenLMIS-3.x-deploy-to-demo-v3** job on Jenkins

   (a) **Confirm** Demo v3 has each deployed service. e.g. for the auth service: https://demo-v3.openlmis.org/auth check that the **version** is the one chosen.

9. Navigate to demo-v3.openlmis.org and ensure it works

Once all these steps are completed and verified, the release process is complete. At this point you can conduct communication tasks such as sharing the URL and Release Announcement to stakeholders. Congratulations!

CHAPTER 2

Links:

- **Project Management**
  - Issue Tracking & Project Management
  - Wiki
- **Communication**
  - Slack
  - Developer Forum
  - Product Comittee Forum
  - Governance Comittee Forum
- **Development**
  - GitHub
  - DockerHub (Published Docker Images)
  - OSS Sonatype (Maven Publishing)
  - Transifex (translations and localized text)
  - Code Review
  - Code Quality Analysis (SonarQube)
  - CI Server (Jenkins)
  - CD Server
  - UAT Server