OpenLMIS Documentation

Release 3.0

VillageReach

Jun 29, 2018

Contents

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 Developer docs

As of OpenLMIS v3, the architecture has transitioned to (micro) services fulfilling RESTful (HTTP) API requests from a modularized Reference UI. Extension mechanisms in addition to microservices and UI modules further allow for components of the architecture to be customized without the need for the community to fork the code base:

- Extension Points & Modules - allows Service functionality to be modified
- Extra Data - allows for extensions to store data with existing components

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

1.1.1 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",
    "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 automated test strategy for OpenLMIS.

Test Strategy

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.

Types of Tests

The following test categories have been identified for use in OpenLMIS. As illustrated in this great 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 code-author during 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 - I can run just a portion of the test suite

- Where: Reside inside a service, next to unit under test. Generally able to access package-private scope
- Why: to test fundamental pieces/functionality, helps guide and document design and refactors, protects against regression

Unit Test Examples

- **Every single test should be independent and isolated. 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);
}

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

- **Every unit test should have 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);
}
```

- **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());
}
```

- **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"})
public class MyServiceTest implements ApplicationContextAware
{

  @Autowired
  MyService service;
  ...
    @Override
    public void setApplicationContext(ApplicationContext context) throws␣
↪BeansException
    {
```

(continues on next page)

(continued from previous page)

```
        // something with the context here
    }
}
```

- **Test method 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() {
  ...
}
```

- **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 intializing and cleaning each global state between test runs.**

  DO:

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

@Before
public void init() {
  age = 18;
  when(ageService.isAdult(age)).thenReturn(true);
}

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

- **Test should run fast. When we have hundreds of tests we just don't want to wait several minutes till 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: Code author during implementation
- What: Test basic operation of a 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 build to not slow development. Will be run in CI on every change.
- Where: Reside inside a service, separated from other types of tests/code.
- Why: Ensures that the basic pathways to a service's external run-time dependancies work. e.g. that a db schema supports the ORM, or a non-responsive service call is gracefully handled.

For testing controllers, they are divided up into unit and integration tests. The controller unit tests will be testing the logic in the controller, while the integration tests will be mostly testing 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: Code author during implementation
- What: Test more complex operations in a 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 build to not slow development. Will be run in CI on every change.
- Where: Reside inside a service, separated from other types of tests/code.
- Why: Tests interactions between components in a service are working as expected.

These are not integration tests, which strictly test the integration between the service and an external dependency. These test the interactions between components in a service are working correctly. While integration tests just test the basic pathways are working, component tests verify that, based on input, the output matches what is expected.

These 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: Code author during implementation, with input from BA/QA.
- What: Enforces contracts between and to services.
- When: Ran in CI.
- Where: Reside inside separate repository: openlmis-contract-tests.
- Why: Tests multiple services working together, testing contracts that a Service both provides as well as the requirements a dependant has.

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

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

End-to-End Tests

- Who: QA / developer with input from BA.
- What: Typical/core business scenarios.
- When: Ran in CI.
- Where: Resides in seperate repository.
- Why: Ensures all the pieces are working together to carry-out a business scenario. Helps ensure end-users can achieve their goals.

Testing services dependent on external APIs

OpenLMIS is using WireMock for mocking web services. An example integration test can be found here: https://github.com/OpenLMIS/openlmis-example/blob/master/src/test/java/org/openlmis/example/WeatherServiceTest.java

The stub mappings which are served by WireMock's HTTP server are placed 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

- 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"
    },
    "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"]
      }

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

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
```

1.1.2 Component Readme's

OpenLMIS Requisition Service

This repository holds the files for the OpenLMIS Requisition Independent Service.

Prerequisites

- Docker 1.11+
- Docker Compose 1.6+

Quick Start

1. Fork/clone this repository from GitHub.

```
git clone https://github.com/OpenLMIS/openlmis-requisition.git
```

1. Add an environment file called `.env` to the root folder of the project, with the required project settings and credentials. For a starter environment file, you can use this one. e.g.

```
cd openlmis-requisition
curl -LO https://raw.githubusercontent.com/OpenLMIS/openlmis-config/master/.env
```

1. Develop w/ Docker by running `docker-compose run --service-ports requisition`. See *Developing w/ Docker*. You should now be in an interactive shell inside the newly created development environment.
2. Run `gradle build` to build. After the build steps finish, you should see 'Build Successful'.
3. Start the service with `gradle bootRun`. Once it is running, you should see 'Started Application in NN seconds'. Your console will not return to a prompt as long as the service is running. The service may write errors and other output to your console.
4. You must authenticate to get a valid `access_token` before you can use the service. Follow the Security instructions to generate a POST request to the authorization server at `http://localhost:8081/`. You can use a tool like Postman to generate the POST. The authorization server will return an `access_token` which you must save for use on requests to this OpenLMIS service. The token will expire with age, so be ready to do this step often.
5. Go to `http://localhost:8080/?access_token=<yourAccessToken>` to see the service name and version. Note: If localhost does not work, the docker container with the service running might not be bridged to your host workstation. In that case, you can determine your Docker IP address by running `docker-machine ip` and then visit `http://<yourDockerIPAddress>:8080/`.
6. Go to `http://localhost:8080/index.html?access_token=<yourAccessToken>` to see the Swagger UI showing the API endpoints. (Click 'default' to expand the list.)
7. Use URLs of the form `http://localhost:8080/api/*?access_token=<yourAccessToken>` to hit the APIs directly.

Stopping the Service

To stop the service (when it is running with `gradle bootRun`) use Control-C.

To clean up unwanted Docker containers, see the Docker Cheat Sheet.

API Definition and Testing

See the API Definition and Testing section in the Example Service README at https://github.com/OpenLMIS/openlmis-example/blob/master/README.md#api.

Building & Testing

See the Building & Testing section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#building.

Security

See the Security section in the Example Service README at https://github.com/OpenLMIS/openlmis-example/blob/master/README.md#security.

Developing with Docker

See the Developing with Docker section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#devdocker.

Development Environment

See the Development Environment section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#devenv.

Build Deployment Image

See the Build Deployment Image section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#buildimage.

Publish to Docker Repository

TODO

Docker's file details

See the Docker's file details section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#dockerfiles.

Running complete application with nginx proxy

See the Running complete application with nginx proxy section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#nginx.

Logging

See the Logging section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#logging.

Internationalization (i18n)

See the Internationalization section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#internationalization.

Debugging

See the Debugging section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#debugging.

Demo Data

You can use a standard data set for demonstration purposes. To do so, first follow the Quick Start until step 3 is done: https://github.com/OpenLMIS/openlmis-requisition/blob/master/README.md#quickstart. Then, before `gradle bootRun`, use `gradle demoDataSeed`. This will generate a sql input file under `./demo-data` directory.

To insert this data into the database, finish the Quick Start steps, and then outside of container's interactive shell, run: `docker exec -i openlmisrequisition_db_1 psql -Upostgres open_lmis < demo-data/input.sql`

Production by Spring Profile

By default when this service is started, it will clean its schema in the database before migrating it. This is meant for use during the normal development cycle. For production data, this obviously is not desired as it would remove all of the production data. To change the default clean & migrate behavior to just be a migrate behavior (which is still desired for production use), we use a Spring Profile named `production`. To use this profile, it must be marked as Active. The easiest way to do so is to add to the .env file:

```
spring_profiles_active=production
```

This will set the similarly named environment variable and limit the profile in use. The expected use-case for this is when this service is deployed through the Reference Distribution.

Environment variables

Environment variables common to all services are listed here: https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#environment-variables

OpenLMIS Fulfillment Service

This repository holds the files for the OpenLMIS Fulfillment Independent Service.

Prerequisites

- Docker 1.11+
- Docker Compose 1.6+

Quick Start

1. Fork/clone this repository from GitHub.

```
git clone https://github.com/OpenLMIS/openlmis-fulfillment.git
```

1. Add an environment file called `.env` to the root folder of the project, with the required project settings and credentials. For a starter environment file, you can use this one. e.g.

```
cd openlmis-fulfillment
curl -LO https://raw.githubusercontent.com/OpenLMIS/openlmis-config/master/.env
```

1. Develop w/ Docker by running `docker-compose run --service-ports fulfillment`. See *Developing w/ Docker*. You should now be in an interactive shell inside the newly created development environment.
2. Run `gradle build` to build. After the build steps finish, you should see 'Build Successful'.
3. Start the service with `gradle bootRun`. Once it is running, you should see 'Started Application in NN seconds'. Your console will not return to a prompt as long as the service is running. The service may write errors and other output to your console.
4. You must authenticate to get a valid `access_token` before you can use the service. Follow the Security instructions to generate a POST request to the authorization server at `http://localhost:8081/`. You can use a tool like Postman to generate the POST. The authorization server will return an `access_token` which you must save for use on requests to this OpenLMIS service. The token will expire with age, so be ready to do this step often.
5. Go to `http://localhost:8080/?access_token=<yourAccessToken>` to see the service name and version. Note: If localhost does not work, the docker container with the service running might not be bridged to your host workstation. In that case, you can determine your Docker IP address by running `docker-machine ip` and then visit `http://<yourDockerIPAddress>:8080/`.
6. Go to `http://localhost:8080/index.html?access_token=<yourAccessToken>` to see the Swagger UI showing the API endpoints. (Click 'default' to expand the list.)
7. Use URLs of the form `http://localhost:8080/api/*?access_token=<yourAccessToken>` to hit the APIs directly.

Stopping the Service

To stop the service (when it is running with `gradle bootRun`) use Control-C.

To clean up unwanted Docker containers, see the Docker Cheat Sheet.

API Definition and Testing

See the API Definition and Testing section in the Example Service README at https://github.com/OpenLMIS/openlmis-example/blob/master/README.md#api.

Building & Testing

See the Building & Testing section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#building.

Security

See the Security section in the Example Service README at https://github.com/OpenLMIS/openlmis-example/blob/master/README.md#security.

Developing with Docker

See the Developing with Docker section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#devdocker.

Development Environment

See the Development Environment section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#devenv.

Build Deployment Image

See the Build Deployment Image section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#buildimage.

Publish to Docker Repository

TODO

Docker's file details

See the Docker's file details section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#dockerfiles.

Running complete application with nginx proxy

See the Running complete application with nginx proxy section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#nginx.

Logging

See the Logging section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#logging.

Internationalization (i18n)

See the Internationalization section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#internationalization.

Debugging

See the Debugging section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#debugging.

Demo Data

You can use a standard data set for demonstration purposes. To do so, first follow the Quick Start until step 3 is done: https://github.com/OpenLMIS/openlmis-fulfillment/blob/master/README.md#quickstart. Then, before `gradle bootRun`, use `gradle demoDataSeed`. This will generate a sql input file under `./demo-data` directory.

To insert this data into the database, finish the Quick Start steps, and then outside of container's interactive shell, run: `docker exec -i openlmisfulfillment_db_1 psql -Upostgres open_lmis < demo-data/input.sql`

Production by Spring Profile

By default when this service is started, it will clean its schema in the database before migrating it. This is meant for use during the normal development cycle. For production data, this obviously is not desired as it would remove all of the production data. To change the default clean & migrate behavior to just be a migrate behavior (which is still desired for production use), we use a Spring Profile named `production`. To use this profile, it must be marked as Active. The easiest way to do so is to add to the .env file:

```
spring_profiles_active=production
```

This will set the similarly named environment variable and limit the profile in use. The expected use-case for this is when this service is deployed through the Reference Distribution.

Environment variables

Environment variables common to all services are listed here: https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#environment-variables

If you are using the hauptmedia/proftpd Docker image for testing fulfillment (like we do in our reference distribution), you can use the following variables to set the username and password for that server:

- **FTP_USERNAME** - the username of the FTP user.
- **FTP_PASSWORD** - the password of the FTP user.

Note: the fulfillment service does not use the variables above for setting up any connections - the configuration is kept in the database and managed through the appropriate endpoints.

OpenLMIS Authentication Service

This repository holds the files for OpenLMIS Authentication Independent Service.

Prerequisites

- Docker 1.11+
- Docker Compose 1.6+

Quick Start

1. Fork/clone this repository from GitHub.

```
git clone https://github.com/OpenLMIS/openlmis-auth.git
```

1. Add an environment file called `.env` to the root folder of the project, with the required project settings and credentials. For a starter environment file, you can use this one. e.g.

```
cd openlmis-auth
curl -LO https://raw.githubusercontent.com/OpenLMIS/openlmis-config/master/.env
```

1. Develop w/ Docker by running `docker-compose run --service-ports auth`. See *Developing w/ Docker*.
2. You should now be in an interactive shell inside the newly created development environment, start the Service with: `gradle bootRun`
3. Go to `http://<yourDockerIPAddress>:8080/` to see the service name and version. Note that you can determine yourDockerIPAddress by running `docker-machine ip`.
4. Go to `http://<yourDockerIPAddress>:8080/api?access_token=<access_token_id>` to see the APIs. For additional info about security see the Security section in the Example Service README at https://github.com/OpenLMIS/openlmis-example/blob/master/README.md#security.

Service Design

See the Design document.

API Definition and Testing

See the API Definition and Testing section in the Example Service README at https://github.com/OpenLMIS/openlmis-example/blob/master/README.md#api.

Building & Testing

See the Building & Testing section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#building.

Developing with Docker

See the Developing with Docker section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#devdocker.

Development Environment

See the Development Environment section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#devenv.

Build Deployment Image

See the Build Deployment Image section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#buildimage.

Publish to Docker Repository

TODO

Docker's file details

See the Docker's file details section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#dockerfiles.

Running complete application with nginx proxy

See the Running complete application with nginx proxy section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#nginx.

Logging

See the Logging section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#logging.

Internationalization (i18n)

See the Internationalization section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#internationalization.

Debugging

See the Debugging section in the Service Template README at https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#debugging.

Production by Spring Profile

By default when this service is started, it will clean it's schema in the database before migrating it. This is meant for use during the normal development cycle. For production data, this obviously is not desired as it would remove all of the production data. To change the default clean & migrate behavior to just be a migrate behavior (which is still desired for production use), we use a Spring Profile named `production`. To use this profile, it must be marked as Active. The easiest way to do so is to add to the .env file:

```
spring_profiles_active=production
```

This will set the similarly named environment variable and limit the profile in use. The expected use-case for this is when this service is deployed through the Reference Distribution.

Environment variables

Environment variables common to all services are listed here: https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#environment-variables

The auth service also uses the following variables:

- **TOKEN_DURATION** - The period of inactivity in seconds after which authentication tokens will expire. For example set this to 900 in order to have tokens expire after 15 minutes of inactivity. The default value is 1800 (30 minutes).

OpenLMIS Reference Data Service

This service contains all of the reference data for OpenLMIS 3.x.

Prerequisites

- Docker 1.11+
- Docker Compose 1.6+

Quick Start

1. Fork/clone this repository from GitHub.

```
git clone https://github.com/OpenLMIS/openlmis-referencedata.git
```

1. Add an environment file called `.env` to the root folder of the project, with the required project settings and credentials. For a starter environment file, you can use [this one] (https://github.com/OpenLMIS/openlmis-config/blob/master/.env). e.g.

```
cd openlmis-referencedata
curl -LO https://raw.githubusercontent.com/OpenLMIS/openlmis-config/master/.env
```

1. Develop w/ Docker by running `docker-compose run --service-ports referencedata`. See *Developing w/ Docker*.
2. You should now be in an interactive shell inside the newly created development environment, start the Service with: `gradle bootRun`
3. Go to `http://localhost:8080/` to see the service name and version.
4. Go to `http://localhost:8080/api/` to see the APIs.

Building & Testing

Gradle is our usual build tool. This template includes common tasks that most Services will find useful:

- `clean` to remove build artifacts
- `build` to build all source. `build`, after building sources, also runs unit tests. Build will be successful only if all tests pass.
- `generateMigration -PmigrationName=<yourMigrationName>` to create a "blank" database migration file. The file will be generated under `src/main/resources/db/migration`. Put your migration SQL into it.
- `test` to run unit tests
- `integrationTest` to run integration tests
- `sonarqube` to execute the SonarQube analysis.

The **test results** are shown in the console.

While Gradle is our usual build tool, OpenLMIS v3+ is a collection of Independent Services where each Gradle build produces 1 Service. To help work with these Services, we use Docker to develop, build and publish these.

See *Developing with Docker*.

Developing with Docker

OpenLMIS utilizes Docker to help with development, building, publishing and deployment of OpenLMIS Services. This helps keep development to deployment environments clean, consistent and reproducible and therefore using Docker is recommended for all OpenLMIS projects.

To enable development in Docker, OpenLMIS publishes a couple Docker Images:

- openlmis/dev - for Service development. Includes the JDK & Gradle plus common build tools.
- openlmis/postgres - for quickly standing up a shared PostgreSQL DB.

In addition to these Images, each Service includes Docker Compose instructions to:

- standup a development environment (run Gradle)
- build a lean image of itself suitable for deployment
- publish its deployment image to a Docker Repository

Development Environment

Launches into shell with Gradle & JDK available suitable for building Service. PostgreSQL connected suitable for testing. If you run the Service, it should be available on port 8080.

Before starting the development environment, make sure you have a `.env` file as outlined in the Quick Start instructions.

```
> docker-compose run --service-ports <your-service-name>
$ gradle clean build
$ gradle bootRun
```

Build Deployment Image

The specialized docker-compose.builder.yml is geared toward CI and build servers for automated building, testing and docker image generation of the service.

Before building the deployment image, make sure you have a `.env` file as outlined in the Quick Start instructions.

```
> docker-compose -f docker-compose.builder.yml run builder
> docker-compose -f docker-compose.builder.yml build image
```

Publish to Docker Repository

TODO

Docker's file details

A brief overview of the purpose behind each docker related file

- `Dockerfile`: build a deployment ready image of this service suitable for publishing.
- `docker-compose.yml`: base docker-compose file. Defines the basic composition from the perspective of working on this singular vertical service. These aren't expected to be used in the composition of the Reference Distribution.
- `docker-compose.override.yml`: extends the `docker-compose.yml` base definition to provide for the normal usage of docker-compose inside of a single Service: building a development environment. Wires this Service together with a DB for testing, a gradle cache volume and maps tomcat's port directly to the host.
- `docker-compose.builder.yml`: an alternative docker-compose file suitable for CI type of environments to test & build this Service and generate a publishable/deployment ready Image of the service.
- `docker-compose.prod.yml`: Docker-compose file suitable for production. Contains nginx-proxy image and virtual host configuration of each service.

Running complete application with nginx proxy

1. Enter desired `VIRTUAL_HOST` for each service in the `docker-compose.prod.yml` file.
2. Start up containers

```
> docker-compose -f docker-compose.yml -f docker-compose.prod.yml up
```

1. The application should be available at port 80.

Logging

Logging is implemented using SLF4J in the code, Logback in Spring Boot, and routed to an external Syslog server. There is a default configuration XML (logback.xml) in the resources folder. To configure the log level for the development environment, simply modify the logback.xml to suit your needs.

Configuring log level for a production environment is a bit more complex, as the code has already been packaged into a Spring Boot jar file. However, the default log configuration XML can be overridden by setting the Spring Boot logging.config property to an external logback.xml when the jar is executed. The container needs to be run with a

JAVA_OPTS environment variable set to a logback.xml location, and with a volume with the logback.xml mounted to that location. Some docker compose instructions have been provided to demonstrate this.

1. Build the deployment image. (See *Build Deployment Image*)
2. Get a logback.xml file and modify it to suit your log level configuration.
3. Modify `docker-compose.builder.yml` to point to your logback.xml location. a. Under `volumes`, where it shows two logback.xml locations separated by a colon, change the location before the colon.
4. Run the command below.

```
> docker-compose -f docker-compose.builder.yml run --service-ports template-service
```

Internationalization (i18n)

Internationalization is implemented by the definition of two beans found in the Application class, localeResolver and messageSource. (Alternatively, they could be defined in an application context XML file.) The localeResolver determines the locale, using a cookie named `lang` in the request, with `en` (for English) as the default. The messageSource determines where to find the message files.

Note there is a custom message source interface, ExposedMessageSource, with a corresponding class ExposedMessageSourceImpl. These provide a method to get all the messages in a locale-specific message file.

See the MessageController class for examples on how to get messages.

Additionally, Transifex has been integrated into the development and build process. In order to sync with the project's resources in Transifex, you must provide values for the following keys: `TRANSIFEX_USER`, `TRANSIFEX_PASSWORD`.

For the development environment in Docker, you can sync with Transifex by running the `sync_transifex.sh` script. This will upload your source messages file to the Transifex project and download translated messages files.

The build process has syncing with Transifex seamlessly built-in.

Debugging

To debug the Spring Boot application, use the `--debug-jvm` option.

```
$ gradle bootRun --debug-jvm
```

This will enable debugging for the application, listening on port 5005, which the container has exposed. Note that the process starts suspended, so the application will not start up until the debugger has connected.

Demo Data

You can use a standard data set for demonstration purposes. To do so, first follow the Quick Start until step 3 is done: https://github.com/OpenLMIS/openlmis-referencedata/blob/master/README.md#quickstart. Then, before `gradle bootRun`, use `gradle demoDataSeed`. This will generate a sql input file under `./demo-data` directory.

To insert this data into the database, finish the Quick Start steps, and then outside of container's interactive shell, run: `docker exec -i openlmisreferencedata_db_1 psql -Upostgres open_lmis < demo-data/input.sql`

Production by Spring Profile

By default when this service is started, it will clean its schema in the database before migrating it. This is meant for use during the normal development cycle. For production data, this obviously is not desired as it would remove all of the production data. To change the default clean & migrate behavior to just be a migrate behavior (which is still desired for production use), we use a Spring Profile named `production`. To use this profile, it must be marked as Active. The easiest way to do so is to add to the .env file:

```
spring_profiles_active=production
```

This will set the similarly named environment variable and limit the profile in use. The expected use-case for this is when this service is deployed through the Reference Distribution.

Environment variables

Environment variables common to all services are listed here: https://github.com/OpenLMIS/openlmis-template-service/blob/master/README.md#environment-variables

OpenLMIS Service Template

This template is meant to be a starting point for developing a new OpenLMIS 3.x Independent Service.

Prerequisites

- Docker 1.11+
- Docker Compose 1.6+

All other dependencies, such as Java, are delivered automatically via the Docker image. It is unnecessary to install them locally to run the service, though often helpful to do so for the sake of development. See the *Tech* section of openlmis/dev for a list of these optional dependencies.

Quick Start

1. Fork/clone this repository from GitHub.

```
git clone https://github.com/OpenLMIS/openlmis-template-service.git <openlmis-your-
↪service-name>
```

1. Respectively change all instances of `openlmis-template-service` and `template-service` within the project to `openlmis-your-service-name` and `your-service-name`.
2. Change all instances of the default version number ("0.0.1") in the project to your version number.
3. Change the gradle build file to add any dependencies (e.g. JPA, PostgreSQL).
4. Add Java code to the template.
5. Add an environment file called `.env` to the root folder of the project, with the required project settings and credentials. For a starter environment file, you can use this one. e.g.

```
cd <openlmis-your-service-name>
curl -o .env -L https://raw.githubusercontent.com/OpenLMIS/openlmis-ref-distro/master/
↪settings-sample.env
```

1. Develop w/ Docker by running `docker-compose run --service-ports <your-service-name>`. See *Developing w/ Docker*.
2. You should now be in an interactive shell inside the newly created development environment, start the Service with: `gradle bootRun`
3. Go to `http://<yourDockerIPAddress>:8080/` to see the service name and version. Note that you can determine yourDockerIPAddress by running `docker-machine ip`.
4. Go to `http://<yourDockerIPAddress>:8080/api/` to see the APIs.

Building & Testing

Gradle is our usual build tool. This template includes common tasks that most Services will find useful:

- `clean` to remove build artifacts
- `build` to build all source. `build`, after building sources, also runs unit tests. Build will be successful only if all tests pass.
- `generateMigration -PmigrationName=<yourMigrationName>` to create a "blank" database migration file. The file will be generated under `src/main/resources/db/migration`. Put your migration SQL into it.
- `test` to run unit tests
- `integrationTest` to run integration tests
- `sonarqube` to execute the SonarQube analysis.

The **test results** are shown in the console.

While Gradle is our usual build tool, OpenLMIS v3+ is a collection of Independent Services where each Gradle build produces 1 Service. To help work with these Services, we use Docker to develop, build and publish these.

See *Developing with Docker*.

Developing with Docker

OpenLMIS utilizes Docker to help with development, building, publishing and deployment of OpenLMIS Services. This helps keep development to deployment environments clean, consistent and reproducible and therefore using Docker is recommended for all OpenLMIS projects.

To enable development in Docker, OpenLMIS publishes a couple Docker Images:

- openlmis/dev - for Service development. Includes the JDK & Gradle plus common build tools.
- openlmis/postgres - for quickly standing up a shared PostgreSQL DB

In addition to these Images, each Service includes Docker Compose instructions to:

- standup a development environment (run Gradle)
- build a lean image of itself suitable for deployment
- publish its deployment image to a Docker Repository

Development Environment

Launches into shell with Gradle & JDK available suitable for building Service. PostgreSQL connected suitable for testing. If you run the Service, it should be available on port 8080.

Before starting the development environment, make sure you have a `.env` file as outlined in the Quick Start instructions.

```
> docker-compose run --service-ports <your-service-name>
$ gradle clean build
$ gradle bootRun
```

Build Deployment Image

The specialized docker-compose.builder.yml is geared toward CI and build servers for automated building, testing and docker image generation of the service.

Before building the deployment image, make sure you have a `.env` file as outlined in the Quick Start instructions.

```
> docker-compose -f docker-compose.builder.yml run builder
> docker-compose -f docker-compose.builder.yml build image
```

Publish to Docker Repository

TODO

Docker's file details

A brief overview of the purpose behind each docker related file

- `Dockerfile`: build a deployment ready image of this service suitable for publishing.
- `docker-compose.yml`: base docker-compose file. Defines the basic composition from the perspective of working on this singular vertical service. These aren't expected to be used in the composition of the Reference Distribution.
- `docker-compose.override.yml`: extends the `docker-compose.yml` base definition to provide for the normal usage of docker-compose inside of a single Service: building a development environment. Wires this Service together with a DB for testing, a gradle cache volume and maps tomcat's port directly to the host. More on how this file works: https://docs.docker.com/compose/extends/
- `docker-compose.builder.yml`: an alternative docker-compose file suitable for CI type of environments to test & build this Service and generate a publishable/deployment ready Image of the service.
- `docker-compose.prod.yml`: Docker-compose file suitable for production. Contains nginx-proxy image and virtual host configuration of each service.

Running complete application with nginx proxy

1. Enter desired `VIRTUAL_HOST` for each service in the `docker-compose.prod.yml` file.
2. Start up containers

```
> docker-compose -f docker-compose.yml -f docker-compose.prod.yml up
```

1. The application should be available at port 80.

Logging

Logging is implemented using SLF4J in the code, Logback in Spring Boot, and routed to an external Syslog server. There is a default configuration XML (logback.xml) in the resources folder. To configure the log level for the development environment, simply modify the logback.xml to suit your needs.

Configuring log level for a production environment is a bit more complex, as the code has already been packaged into a Spring Boot jar file. However, the default log configuration XML can be overridden by setting the Spring Boot logging.config property to an external logback.xml when the jar is executed. The container needs to be run with a JAVA_OPTS environment variable set to a logback.xml location, and with a volume with the logback.xml mounted to that location. Some docker compose instructions have been provided to demonstrate this.

1. Build the deployment image. (See *Build Deployment Image*)
2. Get a logback.xml file and modify it to suit your log level configuration.
3. Modify `docker-compose.builder.yml` to point to your logback.xml location. a. Under `volumes`, where it shows two logback.xml locations separated by a colon, change the location before the colon.
4. Run the command below.

```
> docker-compose -f docker-compose.builder.yml run --service-ports template-service
```

Internationalization (i18n)

Internationalization is implemented by the definition of two beans found in the Application class, localeResolver and messageSource. (Alternatively, they could be defined in an application context XML file.) The localeResolver determines the locale, using a cookie named `lang` in the request, with `en` (for English) as the default. The messageSource determines where to find the message files.

Note there is a custom message source interface, ExposedMessageSource, with a corresponding class ExposedMessageSourceImpl. These provide a method to get all the messages in a locale-specific message file.

See the MessageController class for examples on how to get messages.

Additionally, Transifex has been integrated into the development and build process. In order to sync with the project's resources in Transifex, you must provide values for the following keys: `TRANSIFEX_USER`, `TRANSIFEX_PASSWORD`.

For the development environment in Docker, you can sync with Transifex by running the `sync_transifex.sh` script. This will upload your source messages file to the Transifex project and download translated messages files.

The build process has syncing with Transifex seamlessly built-in.

Debugging

To debug the Spring Boot application, use the `--debug-jvm` option.

```
$ gradle bootRun --debug-jvm
```

This will enable debugging for the application, listening on port 5005, which the container has exposed. Note that the process starts suspended, so the application will not start up until the debugger has connected.

Production by Spring Profile

By default when this service is started, it will clean its schema in the database before migrating it. This is meant for use during the normal development cycle. For production data, this obviously is not desired as it would remove all of the production data. To change the default clean & migrate behavior to just be a migrate behavior (which is still desired for production use), we use a Spring Profile named `production`. To use this profile, it must be marked as Active. The easiest way to do so is to add to the .env file:

```
spring_profiles_active=production
```

This will set the similarly named environment variable and limit the profile in use. The expected use-case for this is when this service is deployed through the Reference Distribution.

Demo Data

A basic set of demo data is included with this service, defined under `./demo-data/`. This data may be optionally loaded by using the `demo-data` Spring Profile. Setting this profile may be done by setting the `spring.profiles.active` environment variable.

When building locally from the development environment, you may run:

```
$ export spring_profiles_active=demo-data
$ gradle bootRun
```

To see how to set environment variables through Docker Compose, see the Reference Distribution

Environment variables

The following environment variables are common to our services. They can be set either directly in compose files for images or provided as an environment file. See docker-compose.yml in the reference distribution for example usage. Also take a look at the sample .env file we provide.

- **BASE_URL** - The base url of the OpenLMIS distribution. Will be used in generated links pointing to this distribution, as well as for communication between services. Each service should communicate with others using BASE_URL as the base in order to avoid direct communication, which might not work in more complex deployments. Services should also use this variable if they wish to generate a link to the application. This should be an url, for example: https://example.openlmis.org
- **VIRTUAL_HOST** - This is used by the nginx server as the virtual host under which the services are made avialble. This should be a host, for example: example.openlmis.org
- **CONSUL_HOST** - Identifies the IP address or DNS name of the Consul server. Set this to the host or IP under which the distribution is available and Consul listens for connections. Services should register with Consul under this address. This should be a host or an IP, for example 8.8.8.8.
- **CONSUL_PORT** - The port used by the Consul server - services should use this port to register with Consul. This should be a port number, for example 8500. 8500 is used by default.
- **REQUIRE_SSL** - Whether HTTPS is required. If set to `true`, nginx will redirect all incoming HTTP connections to HTTPS. By default SSL will not be required - either leave it blank or set to `false` if you wish to allow HTTP connections.
- **LOCALE** - Default localized system language. It will be applied to all running services, if this variable is missing default “en” value will be used.

- **CORS_ALLOWED_ORIGINS** - Comma-separated list of origins that are allowed, for example: `https://test.openlmis.org,http://some.external.domain`. `*` allows all origins. Leave empty to disable CORS.
- **CORS_ALLOWED_METHODS** - Comma-separated list of HTTP methods that are allowed for the above origins.

These variables are used by services for their connection to the database (none of these have defaults):

- **DATABASE_URL** - The JDBC url under which the database is accessible. Our services use `jdbc:postgresql://db:5432/open_lmis` for connecting to the PostgreSQL database running in a container.
- **POSTGRES_USER** - The username of the database user that the services should use. This variable is also used by our PostgreSQL container to create a user.
- **POSTGRES_PASSWORD** - The password of the database user that the services should use. This variable is also used by our PostgreSQL container to create a user.

These variables are used by our builds in order to integrate with the Transifex translation management system:

- **TRANSIFEX_USER** - The username to use with Transifex for updating translations.
- **TRANSIFEX_PASSWORD** - The password to use with Transifex for updating translations.

OpenLMIS Requisition Reference UI Module

This repository holds the files for the OpenLMIS Requisition Reference UI Module.

Prerequisites

- Docker 1.11+
- Docker Compose 1.6+

Quick Start

1. Fork/clone this repository from GitHub.

```
git clone https://github.com/OpenLMIS/openlmis-requisition-refUI.git
```

1. Develop w/ Docker by running `docker-compose run --service-ports requisition-ui`.
2. You should now be in an interactive shell inside the newly created development environment, build the project with: `npm install && grunt bower` and then you can build and start it with `grunt build --serve`.
3. Go to `http://localhost:9000/webapp/` to see the login page.

Note: To change the location of where the OpenLMIS-UI attemps to access OpenLMIS, use the command `grunt build --openlmisServerUrl=<openlmis server url> --serve`.

Building & Testing

See the OpenLMIS/dev-ui project for more information on what commands are available, below are the command you might use during a normal work day.

```
// Open docker in an interactive shell
> docker-compose run --service-ports requisition-ui

// Install dependencies
$ npm install
$ grunt bower

// Build and run the UI against a OpenLMIS server
$ grunt build --openlmisServerUrl=<openlmis server url> --serve

// Run unit tests
$ grunt karma:unit

// Run a watch process that will build and test your code
// NOTE: You must change a file at least once before your code is rebuilt
$ grunt watch --openlmisServerUrl=<openlmis server url> --serve
```

Built Artifacts

After the OpenLMIS-UI is built and being served, you can access the following documents:

- `http://localhost:9000/webapp/` The compiled OpenLMIS application
- `http://localhost:9000/docs/` JS Documentation created from the source code
- `http://localhost:9000/styleguide/` KSS Documentation created from the CSS

Build Deployment Image

The specialized docker-compose.builder.yml is geared toward CI and build servers for automated building, testing and docker image generation of the UI module.

```
> docker-compose -f docker-compose.builder.yml run builder
> docker-compose -f docker-compose.builder.yml build image
```

Internationalization (i18n)

Transifex has been integrated into the development and build process. In order to sync with the project's resources in Transifex, you must provide values for the following keys: TRANSIFEX_USER, TRANSIFEX_PASSWORD.

For the development environment in Docker, you can sync with Transifex by running the sync_transifex.sh script. This will upload your source messages file to the Transifex project and download translated messages files.

The build process has syncing with Transifex seamlessly built-in.

1.1.3 Contribute

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. 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! 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. In the 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.
3. 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.
4. The web browser (e.g. Firefox), version (e.g. v48), OpenLMIS version, as well as any custom modifications made.
5. Your priority in fixing this bug
6. If applicable, any error message text, stack trace, or logging output
7. If possible and relevant, a sample or view of the database - though don't post sensitive information in public

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:
```

(continues on next page)

(continued from previous page)

```
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 fix a bug or implement a new feature.

Coordinating with the Global Community

In reviewing contributions, 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:

1. Review your feature idea with the Product Committee. They may help inform you about how other country needs overlap or differ. They may also consider including a new feature in the global codebase using the New Feature Verification Process or reviewing the Global vs. Project-Specific Features wiki.
2. Before modifying or extending core functionality, email the developer forum or contact the Technical Committee. 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 Architecture and 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
- 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.

TODO: More guidance on working on a micro-service team with fork/pull-requests is forthcoming.It's important to communicate your development effort on the dev forum and always work toward the next release.

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.

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.

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: https://openlmis.atlassian.net/wiki/display/OP/Living+Product+Roadmap
The backlog can be found here: https://openlmis.atlassian.net/secure/RapidBoard.jspa?rapidView=46&view=planning.nodetail

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

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 under http://openlmis.readthedocs.io. 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 table below explains where the particular pieces of the documentation are fetched from, in order to be able to locate and edit them.

1.2 ERD

Generated OpenLMIS v3 ERD.

1.2.1 Reference Data ERD

ERD schema of Reference Data service:

- Zip ERD

1.2.2 Requisition ERD

ERD schema of Requisition service:

- Zip ERD

1.2.3 Fulfillment ERD

ERD schema of Fulfillment service:

- Zip ERD

1.2.4 Auth ERD

ERD schema of Auth service:

- Zip ERD

1.3 Style Guide

The OpenLMIS-UI styles and guidelines are documented in the OpenLMIS Styleguide. To get a better idea of how the OpenLMIS-UI application and components work, consult the OpenLMIS-UI javascript documentation.

1.4 API documentation

1.4.1 Auth Service

Security credentials, Authentication and Authorization. Uses OAuth2.

Static Documentation for Auth API

1.4.2 Reference Data Service

Provides the reference data for the rest of the processes: facilities, programs, products, etc.

Static Documentation for Reference Data API

1.4.3 Requisition Service

Requisition (pull) based replenishment process.

Static Documentation for Requisition API

1.4.4 Fulfillment Service

Includes the basics of fulfillment.

Static Documentation for Fulfillment API

1.4.5 Notification Service

Notifying users when their attention is needed.

Static Documentation for Notification API

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)
  - Code Review
  - Code Quality Analysis (SonarQube)
  - CI Server (Jenkins)
  - CD Server
  - UAT Server