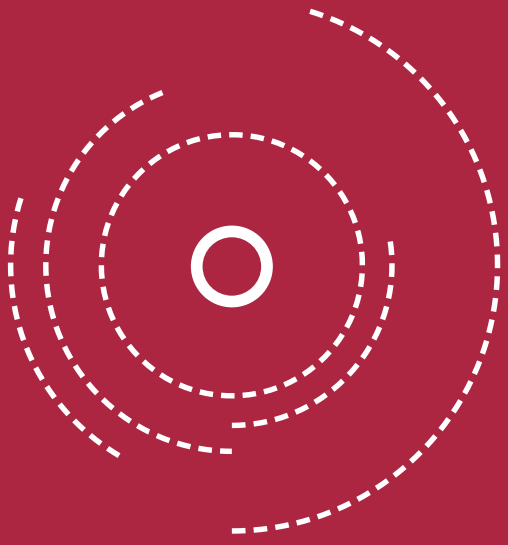
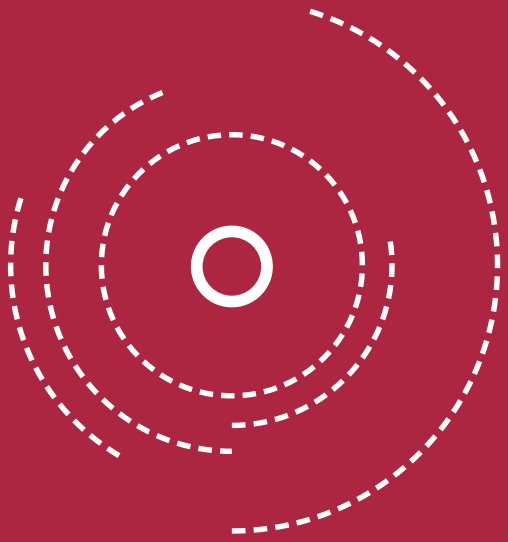


Mojaloop Developer Onboarding

2018-10-29



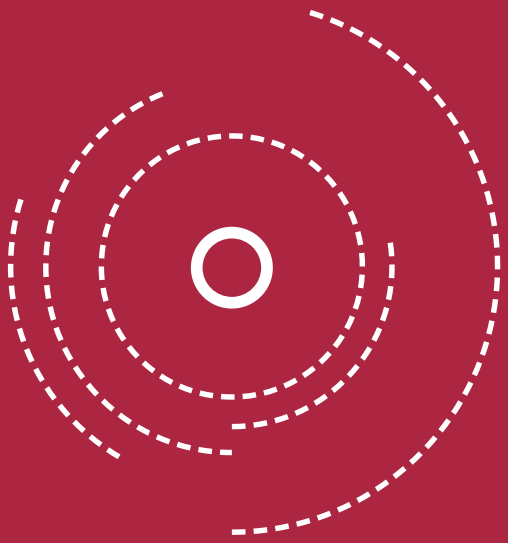


Table of Contents

Appendix

- Resilience Modeling and Anaylysis
- Terminology
- Tools and Process Decisions
- FAQ
- Roadmap

Writing code

- Setting up the development environment'
- Writing Code
- Pragmatic REST Guidelines
- Branching Strategy
- Build and Publish NodeJS
- Code Quality Metrics
- Code Style Guidelines

Software Development Process

- GitHub Labels

Debugging

- Logging Standards

Infrastructure

- Kibana user guide

Onboarding

What is this?

We're sharing code to help create interoperable payments platforms that can be deployed on a national scale. The idea is to make it easier for financial providers to deliver digital financial services to new customers in new markets, especially in the developing world. See our README.md for an overview, or visit mojaloop.io for context and information.

How do I contribute?

Review the Deployment Guide

Review the Mojaloop deployment guide and follow the instructions for deployment: *Deployment and Setup PDF*. For additional information review our onboarding document.

What is the current release?

The last fully tested release is called Phase 2. We took a snapshot of this release in mid-October and the release notes and compoents are as follows:

- central-ledger: Phase 2 Snapshot central-ledger
- central-settlement: Phase 2 Snapshot central-settlement
- ml-api-adapter: Phase 2 Snapshot ml-api-adapter

What work is needed?

Work is tracked as issues in GitHub. You'll see issues there that are open and marked as bugs, stories, or epics. An epic is larger work that contains multiple stories. Anything that is in the backlog and not assigned to someone are things we could use help with. Stories that have owners are in someone's backlog already, though you can always ask about them in the issue or on Slack.

There's a roadmap that shows larger work that people could do or are working on. It has some main initiatives and epics and the order, but lacks dates as this work is community driven. Work is broken down from there into issues in GitHub.

In general, we are looking for example implementations and bug fixes, and project enhancements.

Where do I get help?

Join the Mojaloop Slack Discussions to connect with other developers.

Also checkout the FAQ

Where to I send bugs, questions, and feedback?

For bugs, see Reporting bugs.

Pull Request Process

It's a good idea to ask about major changes on Slack. Submit pull requests which include both the change and the reason for the change. Pull requests will be denied if they violate the Level One Principles

Style guides and templates

- Documentation standards
- Documentation style guide
- Code Style
- Code Quality Metrics

Code of conduct

We use a standard developer code of conduct

Licensing

See License policy

What version?

Naturally, we use semantic versioning, and each repo has it's own version. Update the version appropriately when you submit a pull request. Example: git tag v0.4.0 # update the version.

Additional Information

For additional information please review the frequently asked questions.

Structure of the Documentation

Structure of the Documentation

The Contribute Guide describes how developers collaborate together on the code base.

The Mojaloop **docs** repo details overall architecture of the Level One Project (starting in it's README.md) while each other repo has architecture details specific to it (starting in the README.md for that repo).

- The Mojaloop Documentation and Template Standards describes how the documents are structured.
- The Architecture-Documentation-Guidelines describes types of individual documents and how they are built.
- Documentation Style Guide describes basic style guidelines
- The README-Template is a template for repo overviews

Documentation and Template Standards

Document Scopes

To prevent duplication and allow for easy maintenance, the documentation follows a simple hierarchical pattern:

- Overview - anything that spans the project
- Service - specific to a single service
- Repo - specific to a single component or microservice

More specifics are below.

Overview Documents

Overview documents include the topics, components and or services that span the entire project. The documents should have their own readme file and be located in a corresponding sub-folder in docs repository. Examples or topics and examples that span the entire project are included below:

- Overview
- Scenarios
- Scenario Tests

- Cross-service architecture
- Test strategy
- Physical Architecture
- Overall threat model

Services Documentation

Each of the main services also have a section describing them in the Docs repo. They should follow the template and format specified here in the Architecture-Documentation-Guidelines

- Central Ledger
- Central Services
- DFSP scheme adapter
- Central Settlement

Services docs cover the service architecture, service deployment, configuration, health model, integration tests, and API docs. If there are cross-service standards such as logging also include them here.

For the central services, this is information that is true for all central service. Detail on each service is at the repo level.

Repo Level Documents

Each time you create a new repo, the readme file should provide standard information on the specific component/ service and also make reference to the higher level component or service located in the docs repo. Examples or topics that will be part of a specific repo include:

- Central Rules
- Fraud Services
- ILP Adapter

For these topics, please use the README-Template

API Documentation

- All APIs should be documented in RAML or Swagger, see Architecture-Documentation-Guidelines](Architecture-Documentation-Guidelines.md)
- Use Central Ledger API as a template.

Section Headings

- Do not number headings - for example, "Prepare and Fulfill", not "C - Prepare and Fulfill"
- Follow standards for specific section types as documented in:
- Overview Documents
- Services Documenatation
- Repo Level Documents
- Make sure section headings (#) match the heading to which they correspond in the comprehensive PDF (built from the dactyl config file)
- Do not include the word "documentation" in headings

Retrievability

- For sections that contain many subsections of endpoints or methods, provide a table of contents at the beginning of the section
- Don't say the word project; use component, microservice, interfaces, etc

Language

Instead of the word "project," use a specific noun such as component, microservice, or interface.

Procedures

- Introduce procedures with H3 (###) or H4 (####) headers (not H2 (##)).
- Do not use numbers in procedure section headings.
- Use ordered-list tagging for procedure steps. For example:
- Step 1
- Step 2
- Step 2

Use this template for the README file when you create a component or a service

This should be created in a sub-folder in the docs repo, such as an architecture overview, user flow, and resilience modeling.

File Formats

Mojaloop documentation uses GitHub markdown (md) files. Formatting is limited to GitHub compatible MD. You can use any editor to create the files including the built-in GitHub editors or an external online tool like StackEdit.IO or a download like Haroopad, but be aware that some external tools, like StackEdit.IO, support a superset of GitHub MD and we are limiting the docs to just the GitHub MD. Pandoc is a good tool for coverting docs to and from MD format.

Files can be viewed, reviewed, commented on, and edited directly in GitHub. One exception are diagrams, which will use the tool Draw.IO. Architecture drawings saved as PNG files that are linked in mark down files. Syntax example: [Display Text](./images/diagram.png). Diagrams are stored in GitHub their both their a SVG format as well as PNG. PNG is used for display,but SVG is the master format used for editing. Diagrams should be referenced using a relative path like ./wiki/diagram.png, not the full path such as https://github.com/Mojaloop/Docs/blob/master/Wiki/diagram.png. For larger diagrams, put a blank line above them to allow them to be centered properly upon export.

Template Overview

Component/Service Name

Each component should include and introduction paragraph or two explaining what this component or service is for and how it fits into the overall Mojaloop project.

It is expected that this template would not be fully complete at the beginning of development. Design details are expected to emerge in an Agile fashion. However, the building blocks such as key components, high-level interactions, protocols, approaches, and general test strategy need to be thought through due to all the moving parts in the project. Specify enough detail such that another engineer can review the approach and provide feedback, but don't be too detailed that the architecture/design is obsolete in the very short term. Architecture/Design documents are stored as many smaller files rather than one large one. This way individual parts of the design can be worked on separately and in parallel.

In keeping with agile development, documents can be simple diagrams and do not need to be formal UML. Some of the document names below might imply UML, and while that's fine, it is not required. These docs are meant to be living representatives of our understanding. As such, they will likely start out as rough sketches and become more formal over time.

Contents:

Component Diagram

(This shows the sub-components of the service and their relationships. The Level One Project will have at least two levels of design, one overall system level design which shows the big picture, which the main components are and the primary connections. Each service or main component also has a set of design documents, so we can drill from the big picture into the design of a particular service. These design documents might exist in the specific repo.)

User Message/Flow Diagrams

(This set of diagrams shows the positive or "happy" path of the user. Negative and boundary cases are described. A data flow diagram is also used for threat modeling (see below).)

Interfaces

(This shows the inputs and outputs and their types. What services/components call this and what does it call?_

Example: https://github.com/interledger/rfcs/blob/master/0009-simple-payment-setup-protocol/0009-simple-payment-setup-protocol.md.

(All APIs should be documented in Swagger, see Open API Documentation)

Test Strategy

(This section shows how will it be possible to do functional integration testing of these components? Will we be mocking some components or interfaces?)

Security/Threat Model

(This contains data flow diagram of the system showing components and calls. Each call is threat modeled using STRIDE. Also see threat modeling.)

Resilience Model

See RMA

(For background, see resilience modeling and the linked white paper.)

Monitoring/Health Model

Largely contained in the resilience model.

(For background, see Health modeling.)

Use this template for the README file when you create a repository for a new (micro)service or application under the Mojaloop organization.

Also don't forget to provide higher-level documentation in the Docs repo, such as an architecture overview, user flow, and resilience modeling. See Architecture Documentation Guidelines for details.

(Repository name)

(Intro blurb - no more than two sentences explaining what this repo is for)

Contents:

Deployment

(How do you deploy/build/run this code? If this is included as part of a larger package, link the parent package.)

(Example for NPM-published services--

Installation:

1. Install Node.js and npm

2. Configure your npm instance to use the Mojaloop repository.

3. Install the '(package name)' package.

```
npm install (package name)
```

Running the server locally:

```
npm start
```

--end example)

Configuration

(Location of Ansible playbooks that configure this code.)

(Explanation of important config parameters)

API

(If the API is short, summarize it here. Otherwise, link a separate page with the API docs or explain how to build the API docs from the repo.)

Logging

(Where are the logs?)

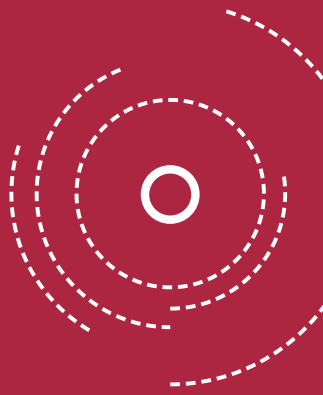
(Explain important things about what gets logged or how to interpret the logs. Use subheaders if necessary.)

Tests

(Explain what's covered, and what's not covered, by the tests. At this level, it's usually just unit tests.)

(Explain how to run the tests)

Local Test Envirnment



devops

Repo for devops, for various scenarios of deploying L1P system, currently in a local setting.

Supported configurations:

Vargrant - Local install - click here:

Supported Platforms - Mac (Tested on Yosimite, Sierra, and High Sierra) - Linux (Tested on Ubuntu 14.04 and 16.04) - Windows (not supported at this time due to specific shell scripting requirements)

Prequisites and requirements: - HOST OS requirements RAM: minimum 8gb RAM, 12gb RAM required for all components, 16gb RAM recommemded DISK: TBD (Github repo, VM creation, etc.)

```
- Prerequisites:
    Vagrant (Version TBD)
    VirtualBox (version tbd)
    Boto?
```

Cloud - Remote install - click here : - AWS - Native support and can be run end-to-end via the interop-devops install - Azure - Manual install can be performed (requires ssh access to the host with sudo capabilities)

```
- AWS requirements:
   Security
   Network/ports
   AMIs
   Keys

- Prerequisites:
   Ansible (version TBD...assuming 2.4 minimum based on the
functions utilized)
   Boto?
```

Manual and automated Testing Strategy

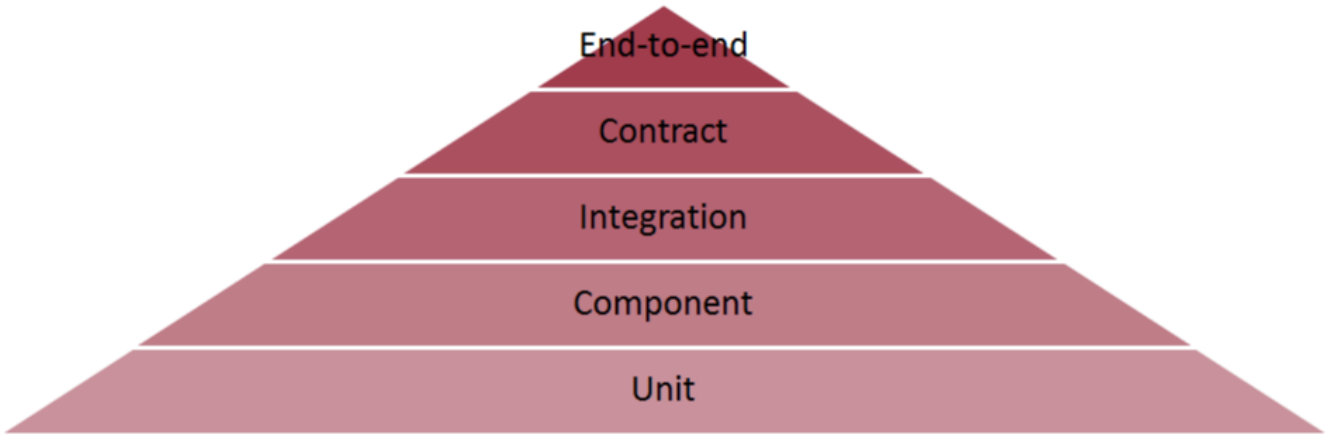
Regression tests are meant to cover use cases that we want to repeatedly test. These include epic and story acceptance criteria, bugs that made it beyond a sprint or main pull request, and code functionality.

We break regression tests into several types:

- **Unit** - tests the smallest functionality of the system, typically by forcing an event or state change. They test only one behavior per test. Often a unit tests validates a single path through a method or function. Stubs, dependency injection, and extract and override are often used to isolate the part under test. Unit tests do NOT test multiple components or include external dependencies like a database. These tests run very quickly (milliseconds each) and should get high code coverage. Mocks can be used to test communication with other components. Unit test code follows the Arrange, Act, Assert (AAA) pattern, and unit test names follow a good naming convention like unit_should_when (ex: Hand_ShouldHaveNoCards_WhenItsCreated). It should always be possible to know exactly what the unit test does from it's title.
- **Component** - testing the interaction between two or more components. Component tests are almost always built from a model of the system. Most systems can be modeled with stateless pair-combinations of the inputs and outputs where the test oracle implements a chain of responsibility pattern going from worst (negative) to best (positive) cases. More complicated systems may be modeled with finite state engines or petri-nets. With microservices, the component tests are all within the microservice and don't cross boundaries to external services. Component tests use much of the same stubs and code used by the unit tests. These test also run quickly.
- **Integration** - Verifies the communication between two services. For example: between the business logic and the persistence service. Because the contracts have already been tested, there are many fewer integration tests. These cover simple positive cases, verifying errors are returned, and performance.
- **Contract** - These are acceptance tests for the contract or interface and are a type of compatibility test. These tests are most often written when different teams or groups are on either sides of the interface. The tests are written by the consumer(s) of the interface, not the

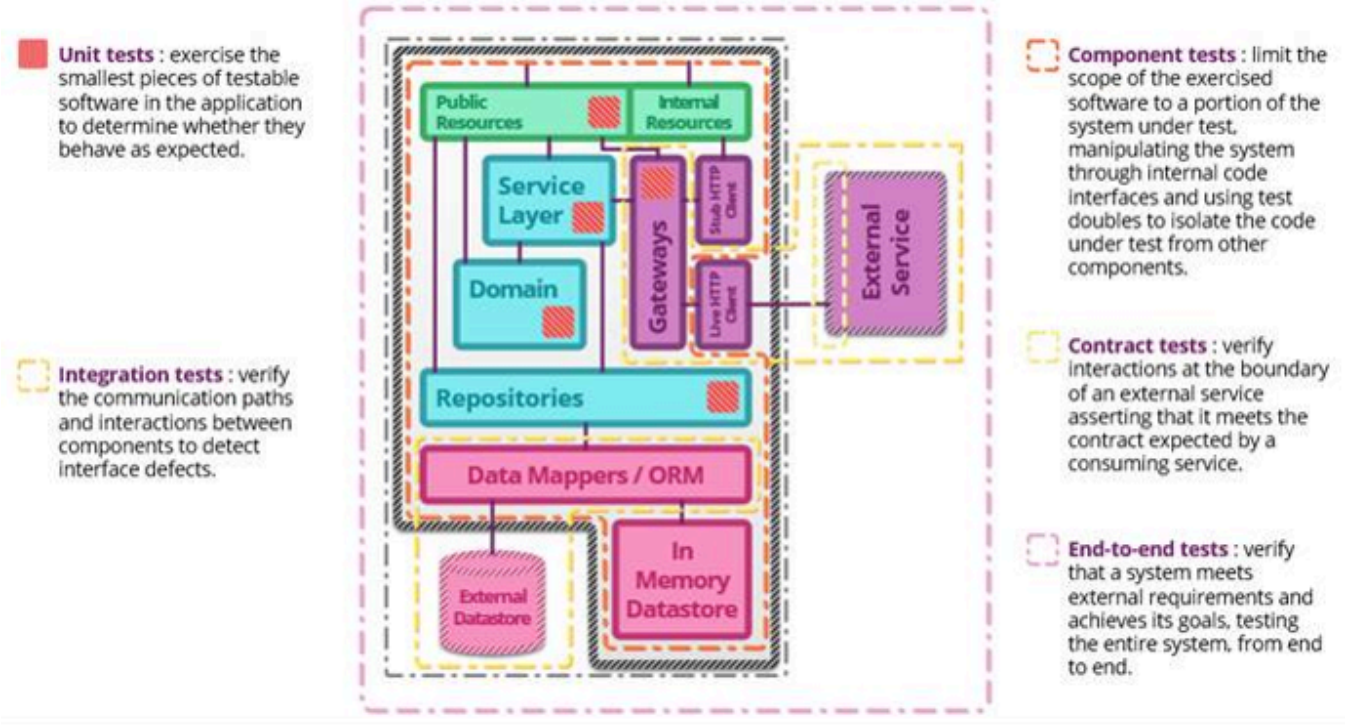
producer. They are run to validate that changes to the interface haven't broken systems that depend on that contract. The contract tests are white box, in that they don't duplicate the producer's unit, component, and integration tests. They often includes performance and scale tests as part of the contract. Contract tests use mocks for the external system and run quickly.

- **End-to-end** - Verifies important user scenarios. These tests often map directly to a use case for an epic. They don't always use UI to drive part of the testing.



By default, regression tests will be automated, especially at the unit, component, contract, and integrations levels. Because we are using microservices as a design pattern, we are using standard microservice testing patterns

Success measures are covered in Code Quality Metrics



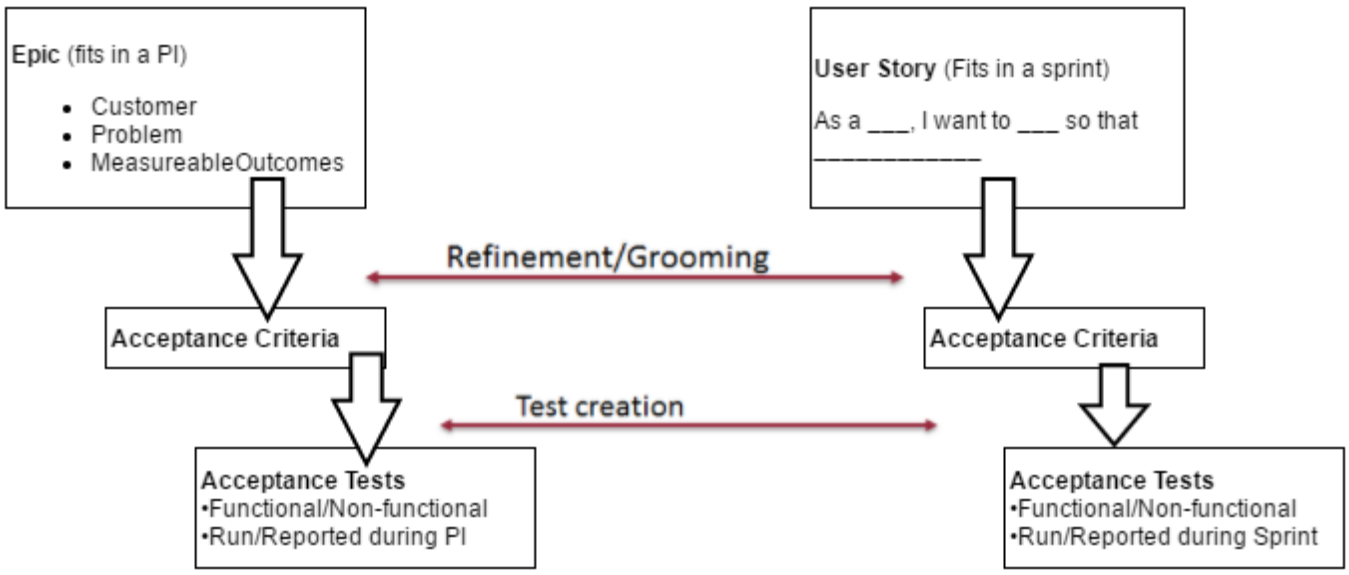
Tests remain manual only in a few situations: ***In-sprint testing*** - *during the sprint, when a change is first created, you might test in manually while automation is being developed. This is a good time to use exploratory testing tours* **Difficult to automate scenarios** - some tests are very hard to automate and these can remain manual, however, when this occurs, think if there may be a better way to write the code to make the testing automatable. ***Very low priority scenarios*** - *Some tests a very low priority and thus don't need to be repeated often.* **UI validation** - when a change to the UI is made, the best way to test it is to see it. With good MVVM design patterns, there should be no business logic in the UI to test, so the UI is just a thin presentation layer. Given that, contract testing below the UI covers most UI functional testing. The UI itself is rarely automated.

For now, manual tests are documented in text files. They include type and components covered so that a subset can be run. Manual test execution is tracked in a spreadsheet.

Most boundary and negative testing is done below the integration level and so run quickly. End-to-end tests don't need to duplicate that testing and so mostly cover positive cases.

Tests are as close to the code they test as possible. Thus, checking a simple function is done by unit tests on the function, interactions between components is done through component testing, checking compliance to an interface is done through automated contract tests on that interface (which don't duplicate the unit test and component tests).

In the end, we expect the number of unit and contract tests to be 10x or more times the number of component and integration tests which are also 10x or more times the number of end-to-end tests. These numbers aren't a goal, but meant to describe the expected shape of a very flat pyramid.



These tests describe the expected behavior of the the services separate from the interfaces. Should the interfaces change, these behaviors would be expected to continue to work. Each test is described as a simple bulleted sentence that states the test conditions and expected behavior. This has the same information you might expect in a more formal format like Gherkin, but is easier to read and review.

Account Management Tests

Account management is internal to the DFSP service. It is tested and verified via the USSD interface.

Use Cases

Add account

A user can have zero or more accounts. The first account is made when the use signs up with the DFSP (see user management tests). Adding an account will add additional accounts for that user in the same DFSP.

The Account Name and Is Primary (Y/N) are required inputs. When a new account is created it shows up in the switch accounts list.

Currently, the account name is unique within DFSP. You can't create an identical account name for two different users. This is a convenience for the current implementation, and not tested. This restriction can be removed without loss of functionality.

Close account

- Secondary accounts can be removed.
- A non-signatory doesn't have the option to close an account. This is manually verified.
- The primary account can't be closed (gives an error message when you try).
- An account can't be closed if there is money in it. There's an error message when it's attempted.
- The account signatory can close an empty account.

Switch account

- USSD shows the message: "You don't have any other account to switch to" when there is only

one account.
- A user can switch between available accounts when there is more than one.

Primary (default) Account

- When there is more than one account, the primary account is where money will be sent.
- A new account can be made primary when it is created. The previous primary account will become a secondary account.
- A secondary account can be made primary. The previous primary account will become a secondary account.

Additional holders (users)

The original holder for the account is considered the owner. For now, there is no option to change owners. Additonal holders can be signatory or non-signatory.

Add additional holder to account

- Non-signatory can't add a user. There's no option for this and it is manually verified.

- A signatory for the account can add other users to an account as a holder of that account. If a user is added to someone else's account, that user can see the new account in their list of accounts to switch to. They can switch to the account and do things like see the balance.

Remove additional holder from account

- The original owner (first user) of the account can't be removed. (requires remove user from DFSP). This gives an error when attempted.

- A non-signatory can't remove anyone. There is no option to do this and it is manually verified.

- A signatory can remove another user from an account who is not the owner.

Account Permissions

- The owner is a signatory and they have same menus (manually verified). An owner can't be made non-signatory (error).

- A signatory can change another holder, who is not an owner, from signatory to non-signatory and vice-versa.

- A non-signatory holder can't send money or manage accounts. They can send invoices, look at the mini-statement, and account balance (manually verified).

Customer Management Tests

Association of a phone to a customer

A phone has an identifier that the DFSP uses to associate the phone with a customer.

- If a customer connects using a phone with an unassociated identifier then the customer is asked to create an account.
- If the customer connects with a phone that has been associated with a customer, the customer recieves a menu of account actions.

There is currently no way to associate another phone with the same customer.

Add customer

A customer is identified by at least their name, birthdate, and an ID such as their national ID number. Adding a customer requires this data and returns the user number from the central directory service. Adding a customer creates at least one account for that customer in the DFSP.

- If customer already registered at that DFSP, then attemping to add the customer again from another phone (same name, birthdate, and ID) returns an error.

- If the fraud service returns 100, customer isn't added (error).

- A new customer can add themselves to the DFSP. The customer is registered in central directory associated to that DFSP.

- Different customers can be registered with same name and/or birthdate.

- The same customer can be registered with multiple DFSPs, though they will have different customer numbers.

Remove customer

- If the account owner closes the last account (see account management), the customer account closed at DFSP, and no customers can connect to that account. The customer is no longer associated with the DFSP in central directory and the phone will again ask for an account to be created.

Change password

Changing password functionality is not currently implemented [#420]. - Simple passwords are not allowed (all one number, straight runs, too short) [Not implemented: #331] - A customer has a single password for all accounts in a DFSP. This is a convenience for our implementation, and not tested.

Account Types

- There are several types of accounts that may be created: customer, agent, and merchant. A

customer has only one account type and it is set when the customer is added. Currently there is no option to change account types or have different account types for for a single customer.

Merchant accounts

- Merchant can send pending transactions, others can't send pending transactions but can approve them (manual verified).

Agent accounts

- Agents start with two accounts: the main one and the commission account.
- Commission account can't be closed (no option exists, manually verified)
- The commission account can't be made the primary (error)
- Agents have the option to do Cash In and Cash Out transfers, others don't. The commission account can't send money, cash-in, cash-out, or pending transactions (manually verified).

DFSP Management

DFSP's can be registered and put on hold in the central directory. Doing so enables and disables the DFPS from conducting trasfers with the central ledger.

These operations are done through (restful API calls to the central directory)[https://github.com/mojaloop/Docs/blob/master/CentralDirectory/central_directory_endpoints.md]

Add DFSP

Registers the DFSP with the central directory.

Pause DFSP

Not yet implemented. This should cause all calls for that DFSPs users to return "unknown" and all pending transfers for that DFSP to be cancelled at the center. It could be called by a DFSP for itself or a regulator at the center.

Unpause the DFSP

Not yet implemented. Would renew normal operations for the DFSP.

Fee tests

Below we list the equivalence classes that make of the test combinations in the test matrix.

Variations

Fee Source

- Sender fee
- Receiver fee
- Agent cash-out fee
- Agent cash-in fee
- Central fee

Path for transfer

- Cross-DFSP
- Same DFSP (should not apply center fee)

Configure Amount

- Stair-step: flat fee plus percent for range
- Zero

Test Matrix

Using pair combinations of the variations we get a matrix like this:

Path	Source	Receiver	Center	Agent Cash In	Agent Cash Out
Cross-DFSP	0	Stair-step	0	Stair-step	0
Cross-DFSP	Stair-step	0	Stair-step	0	Stair-step
Same-DFSP	Stair-step	Stair-step	0	0	Stair-step
Same-DFSP	0	0	Stair-step	Stair-step	0
*	0	*	Stair-step	0	Stair-step
*	Stair-step	*	0	Stair-step	0
Cross-DFSP	*	*	*	Stair-step	Stair-step
*	*	*	*	0	0

* value doesn't matter

Validations

For each variation verify that the fees can be: - Configured - Shown to the sender (it's enough to check the quote return from the scheme adapter) - Deducted from the transfer - Itemized for settlement (for these last two it's enough to check central ledger)

Pending transacton tests

Merchants are able to send a pending transaction. These show in their pending transaction list till resolved. Anyone can approve or reject a pending transaction sent to them.

Send pending transaction

Assumes the user is a merchant - (x) Send invoice for 0 - (x) Send for valid amount to non-existent customer, get error - Send pending transaction for a valid amount to valid customer

Approve pending transaction

- Approve a transfer, the money and fees are transfered from approver's account. The principle goes to the pending transfer sender.
- (x) Approve a transfer when the amount exceeds the user's balance, get error and transaction is not sent or rejected.

Reject pending transaction

- Reject the proposed transfer. Notification goes back to sender.

Send Money Tests

As part of the Level One principles, the customer must be able to see at least the name of the person or business they are sending their money to and the full cost of the transfer, broken out by principle and total fees, before they approve sending money. Money can only be sent (pushed) not debited (pulled).

Variations

Instead of listing every case, we list the equivalence classes for variations that can be done when sending money. These include positive cases, positive and negative boundary cases, and invalid cases. In all positive cases, the fulfillment should be recorded in the ledgers of both the payee and payer DFSPs and the central ledger. Under no cases should the payment not be represented correctly in all three ledgers, though for some negative cases, the matching will require services to be restarted or connections to be reestablished.

Combinations of some equivalence classes should be tried. In general, negative cases, marked (x), are not combined with other variations unless mentioned and should have an error message.

Destinations

- Payer and Payee are on the same DFSP
- Payer and Payee are on separate DFSPs
- (x) Invalid Destination customer

Customers

Combine with destinations - Same customer - Different customer - Same customer, different account but same DFSP

Test Matrix

This test matrix condenses the positive cases above into two simple tests. Other variations are covered below and in other tests.

Destination	Customer
Same DFSP	Same customer different account
Different DFSP	Same customer but different ID

Amount to send

Amounts don't need to be combined with other variations. - 1 - Some - Exact account balance - (x) More than account balance due to fees - (x) More than account balance

DFSP Limits

Limits to the number of transfers in a day, the maximum account size, or the maximum transfer amount are configuration limits implemented at the DFSP and don't need to be combined with tests of other services.

If there is a limit on the number of transfers, then a cancelled or rejected transfer still counts against a customer limit. Refunds are separate transfers initiated by the DFSP that do not count against a customer limit. Sending to yourself on the same DFSP shouldn't be counted toward any limits.

Configuration: Verify limits can be set to both none and some amount. Changing the limits should be logged to the forensic log.

Set the maximum number of transfers and transaction size low (2) and try: - The maxmimum number of transfers in a day - (x) One more than the maximum number of transfers per day - Send the maximum tansaction size - (x) Send more than the maximum transaction size - exceeding limits to yourself (should work)

States

A transfer can be one of several states. Some of the states have multiple ways they can occur. Each of these variations needs to be tested, but don't need to be combined with other variations.

To be able to test the cancel states we need to be able to hold the sending of a the payment messages in each service till after the timeout. Likewise, to test the rejection states we need to be able to force a rejection from the center or the DFSPs.

Here are the list of possible states:: - Unknown - Preparing and within timeout. This is part of the normal flow, it tested in regular end to end tests. - After timeout, but not notified. This happens when a service is down or a message is dropped and is tested in the resilience tests below. - Cancelled (timeout) - Payer DFSP timeout. After the quote the payer DFSP doesn't send the prepare till it has already timed out. If it attempts to send it anyway, the center should reject the prepare and the sender should show a cancellation to the customer. - Center timeout during prepare. The center sits on the payment till it times out. - Payee DFSP timeout. The receiver sends a cancel notification. - Center

timeout during fulfill. The center acknowledges the fulfill message, but sends a cancellation for the notification to both sender and receiver. - Final Payee timeout. In this state the transfer is already fulfilled. Even if the timeout occurs after reciept by the sender but before the sender ledger handles it, the sender DFSP should process the transfer. - Rejected - Payer DFSP rejects (ex: fraud or insufficient account funds) - Center rejects (ex: insufficient settlement funds) - Payee rejects (ex: fraud) - Fulfilled

See settlement tests below for additional validations.

Thread contention/Sequence errors

- Remove a destination user after the quote but before the transfer is recieved be the destination DFSP. This should result in a rejection of the transfer by the Payee DFSP.

Time skew

- Verify time skew is not relevant by setting each service on different dates and sending money. It is expected that the cross-service logs will show odd times.

Resilience

Despite the failures listed below, no ledger should lose money and the transfer should eventually succeed when the failure is resolved. These are negative tests and not combined with other variations.

- Message failures. The failures occur when messages are dropped or not sent
    - Halt fulfillment notification messages for center
    - Halt fulfillment notification messages for payee DFSP
    - Halt fulfillment notification messages for payer DFSP. Transfer should go through due to retries when the connection is re-established.
- Service failures. In each case the payment should complete afer the service is restarted.

    - Take down payee DFSP after quote
    - Take down center DFSP before and after prepare
    - Take down payer ledger adapter after prepare
    - Take down client when a transfer is unknown (is retry initiated by the DFSP when the client is restarted?)

- Verify Idempotence - cause retries and verify only 1 transaction on ledgers for both DFSPs and the center.

Settlement

To support deferred net settlement, the central ledger can easily list:

- Fulfilled transfers
    - with fees broken out for separate accounts.
- Balances by DFSP
- Cancelled and rejected transfers
- Unknown expired transfers

The firt two are tested as part of fees testing. The latter two should be tested during state testing.

Refunds

Refunds are not currently implemented.

In this system all transfers are final, so a a refund is a second transfer in the opposite direction for the original amount including both principle and fees. It contains data to link it to the original transfer it is negating for auditing purposes.

DFSPs typically do not charge fees for the refund.

The refund is marked as such so that the central ledger can report on it appropriately.

Refunds may be charged a central fee if that is charged to every other transfer, which the DFSP can choose to pass on

to the customer or not.

Scenario Tests

Setup

See Jmeter setup here

Test Configuration

This is a functional end-to-end test and is driven by several components - environment configuration - test case data

Environment Configuration:

The main configuration is external to the test in a csv file called **scenarioConfigData.csv**. This file should be in the same directory as the jmx file for the test. It contains the: *the name of the two DFSP servers - these can be made the same server* the USSD port number - it assumes that both servers use the same port number * the expected total fee - it assumes the fee is the same in both directions These values can be adapted to your local environment by changing the file.

Test Case Data

There are two versions of the test. In one version, **Scenario Test (new users).jmx**, two accounts are created dynamically and used for a transfer and an an invoice. The user created will create users with a name in the format: TestUser_DDDDDDD where the D's are digits. Those same digits are used for the user's national ID and PIN. The user's first name is always John. That version of the test writes two files: *ScenarioUser1.csv* ScenarioUser2.csv

These files contain the user digits, and the user number generated by the system.

In the second version, **Scenario Test (existing users from file).jmx**, the user csv files are used to send money and invoices between the existing users with creating new users or accounts.

Test Functionality

The tests perform these scenarios: *Create new users on each DFSP* Check the initial balance (the amount is stored, but not validated) *Send Money from User1 to User2* Check the final balances and verify they are correct including the fee

Send Invoice from User1 to User2 User2 approves the invoice, sending money back to User1 * Balances are checked

Scenarios Not tested: account management, sub-accounts

Logging and Output

- scenarioTest.log - produced by the test. Shows success/failure by step and performance info
- jmeter.log - produced by jmeter. Shows test failure info.

Tests can be modified to send results in mail.

Integration Tests

Contains an initial attempt at providing performance and functional tests for Mojaloop assets.

Setup

You will need Apache JMeter 3.X to load these files.

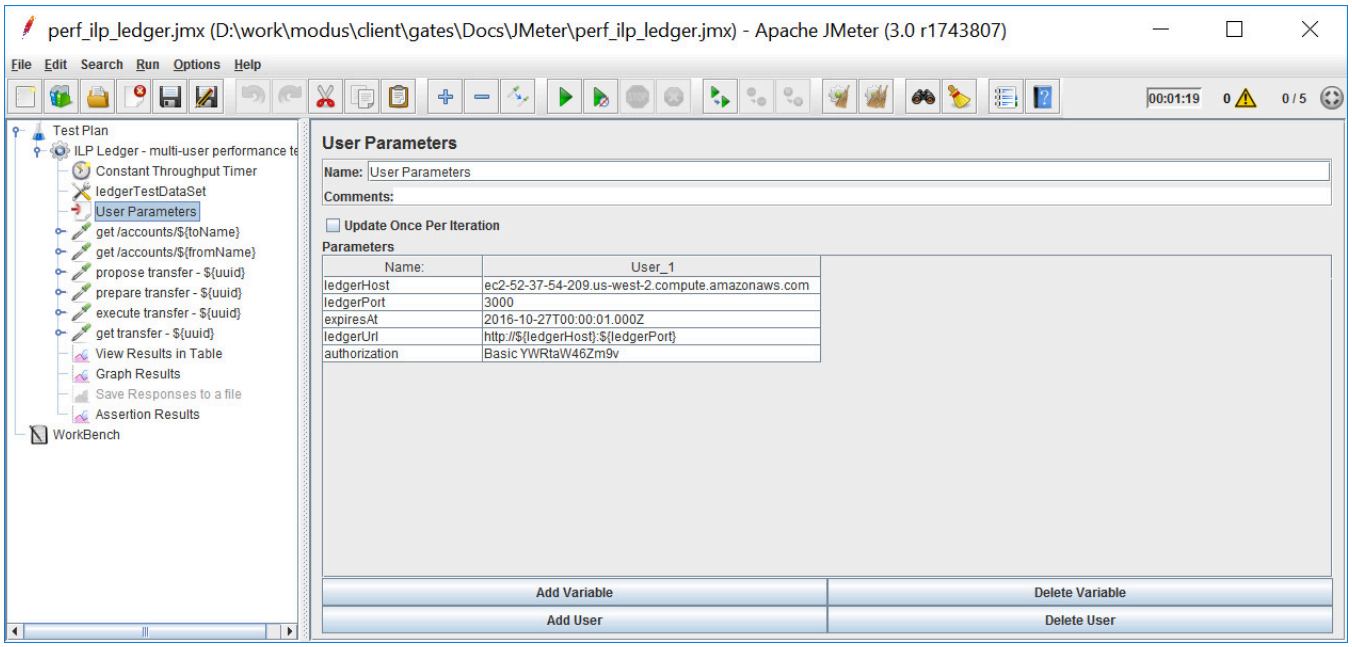
1. Download the appropriate archive for your operating system and expand it.
2. Download jmeter-plugin-manager and place it in the `lib/ext` folder under the path you extracted jmeter
3. Navigate to the `/bin` folder under the path you extracted jmeter
4. Start JMeter by executing `jmeter.bat` or `jmeter.sh` depending on your os
5. Navigate to Options Menu -> Plugin Manager -> Available Plugins (middle tab)
6. Select `JMeter Plugins JSON` and then click `Apply Changes and Restart JMeter`

Test Configuration

The test is driven by several components - environment configuration - test case data - desired throughput - number of concurrent clients

Environment Configuration:

User variables required to configure the test are located in the `User Paramerters` element as shown below

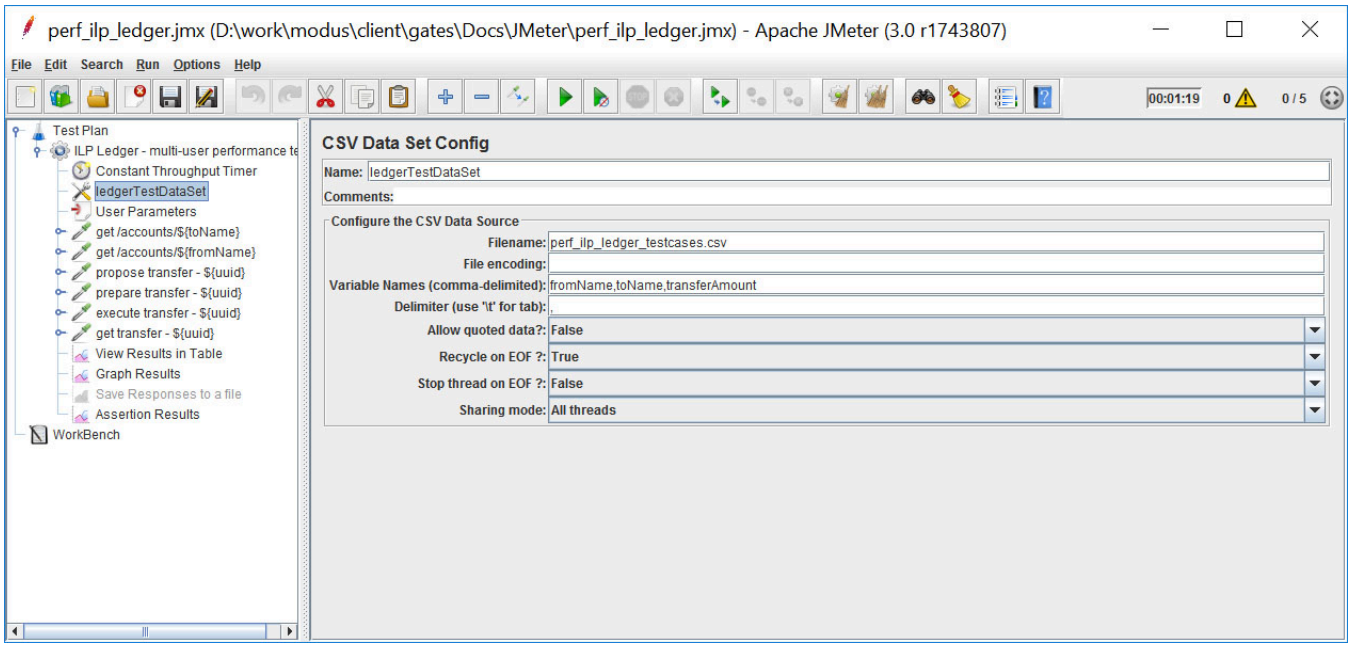


In this section you should configure the following attributes:

attribute	description
ledgerHost	the host name the test should connect to
ledgerPort	the port
expiresAt	what data for expresAt should be included. Some date in the future.
ledgerUrl	by default its being built by host and port but you can change it here. these goes inside some of the request messages
authorization	the value to be included in the Authorization header element. This value is correct for admin / foo. This is a temporary hack.

Test Case Configuration:

This test loads test cases from a csv file. You will need to either copy the sample file located here into your `jmeter bin` directory or edit this element and include the path to the file. The file contains the following data.



fromAccountName, toAccountName, transferAmount

example:

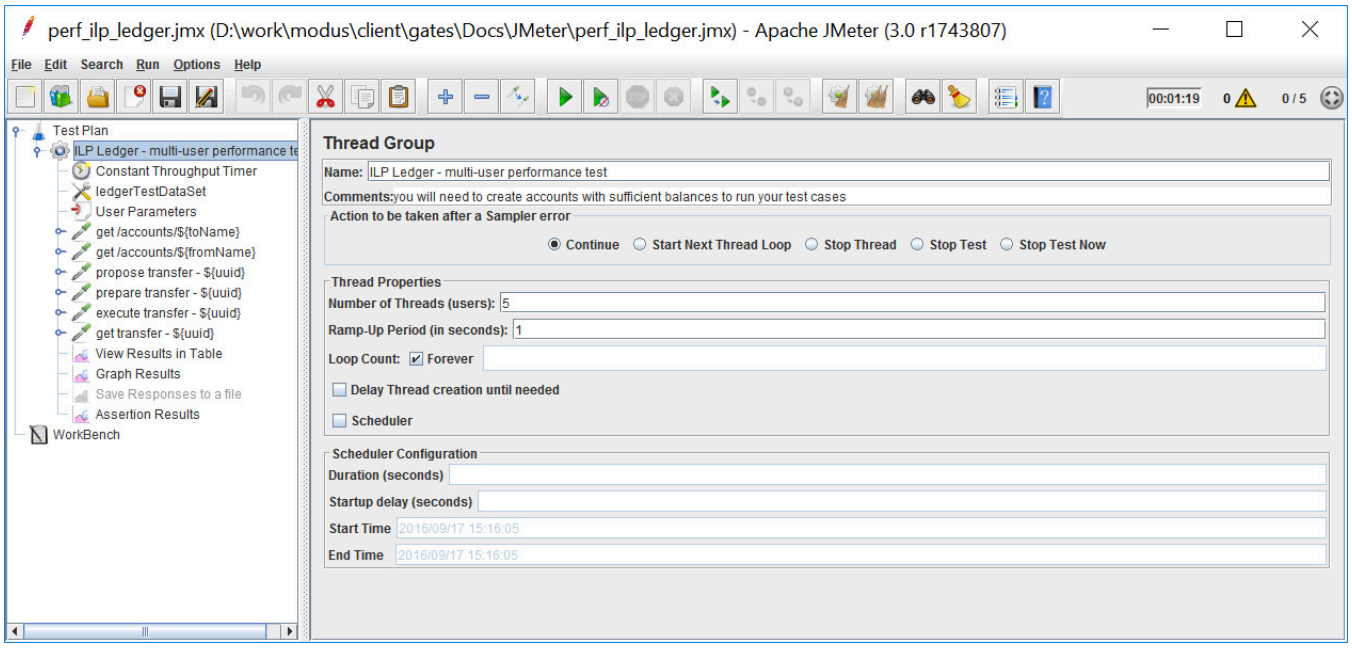
```
alice,bob,50
bob,alice,100
```

Each line in the file will cause be run by a client thread. When all of the lines have been executed the test will loop back to the beginning. This behavior can be changed.

Note that since this is a test that is intended to be run multiple times it does not create the accounts first. You will need to create accounts in your ledger with sufficient balances to sustain the test. When there are insufficnet funds in a ledger you will start seeing errors pop up in the results table.

Number of Concurrent Clients:

A `thread group` configures how many client threads should be used, how long they should take to ramp up to full load, and how many iterations of the test they should execute.

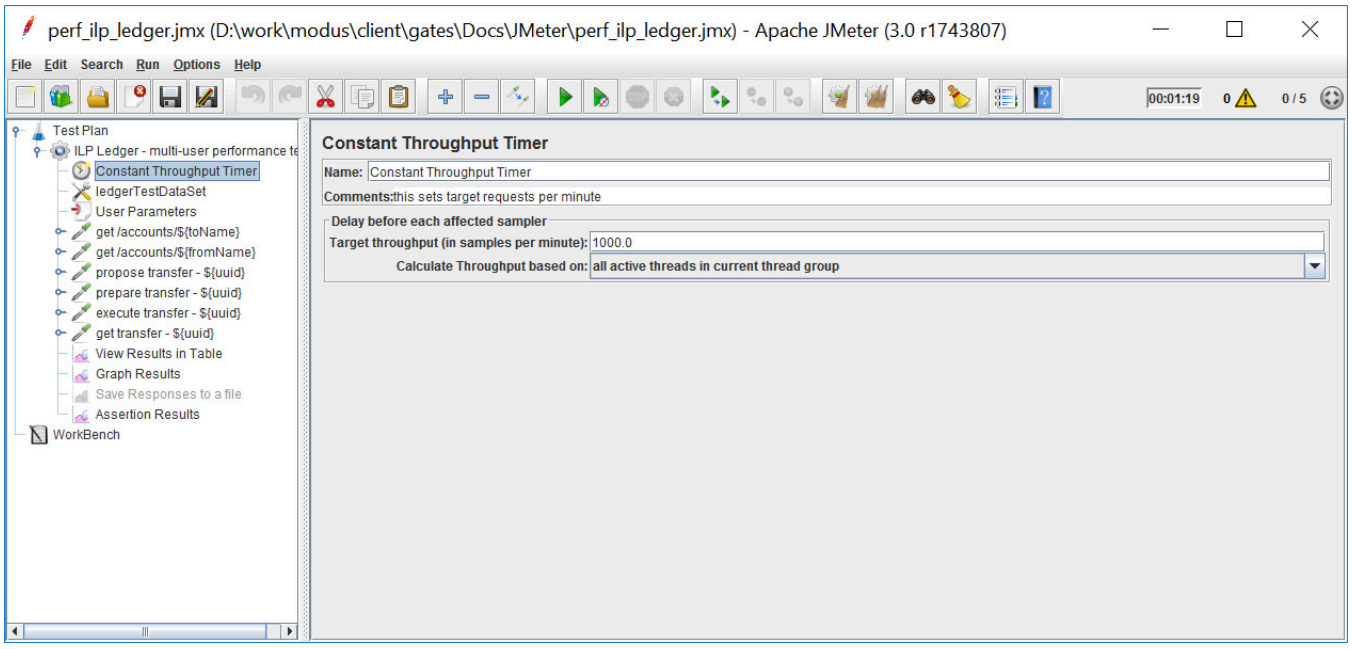


In this example we have 5 client threads ramping up over a 1 second window. They will loop forever. If the test receives an error it will continue to run.

As the number of client threads are increased throughput will increase to a point. There will be a point where throughput will continue to increase but per call latency will also start to climb. This is generally around the point where you cross over the number of threads allocated to the receiving thread pool on the service you are calling against. For example the default number of threads allocated to a http listener in mule is 16. As we increase client threads up to 16 throughput will steadily increase. Past 16 latency starts to climb because each client thread has to wait for some number of server side threads to complete before its request can be servied. This can be good to a point because we are optimizing out part of the round trip latency between the computer making the request and the server servicing it. The time for the request to get there becomes free from the perspective of overall throughput.

Throughput Configuration

The `constant throughput timer` constrains the amount of load generated by the client threads on the target system. It will constrain client connections so that they do not exceed the specified number of requests per minute. Note that these are total requests generated. The test case contains several requests per test iteration.

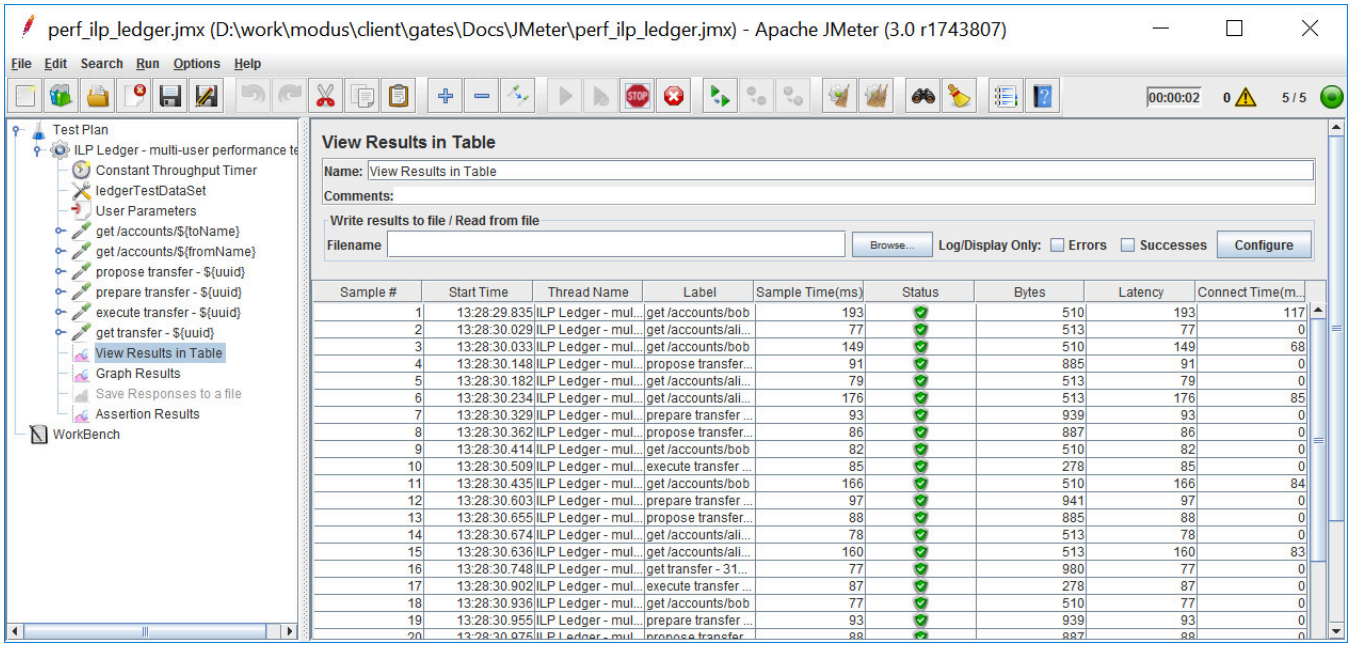


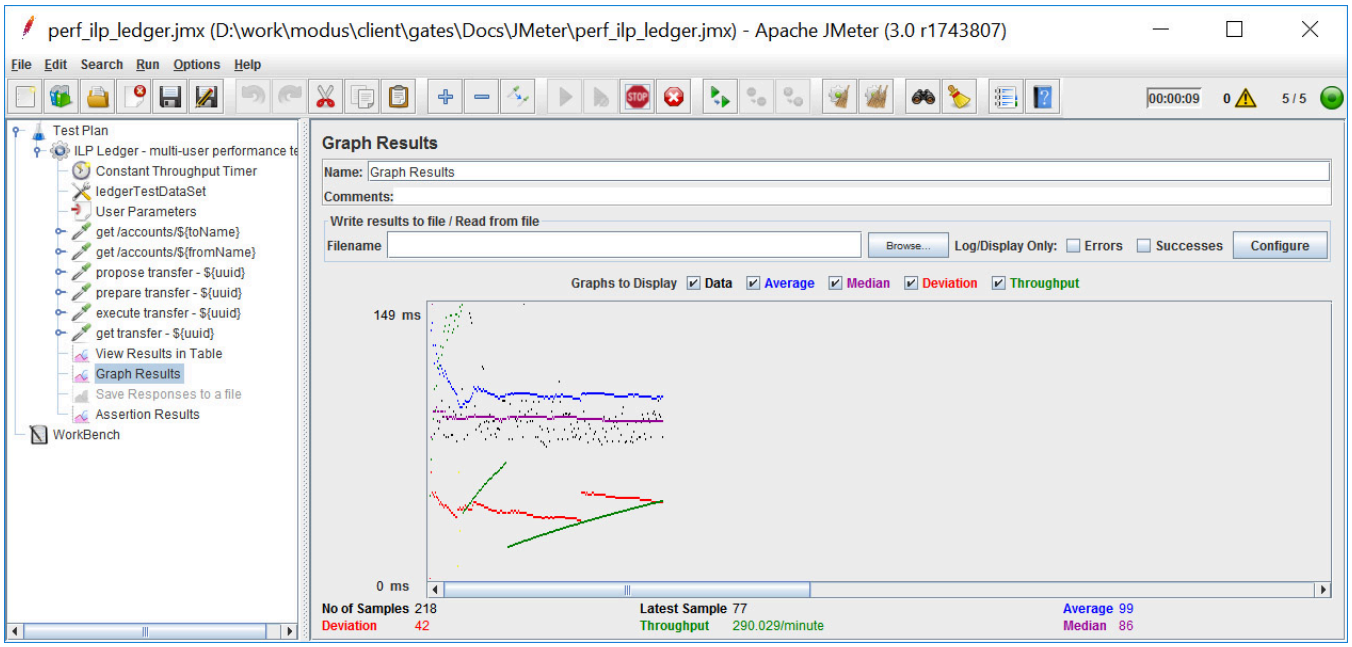
Executing the Test

To start the test select `Run -> Start` or click the  button in the center of the toolbar. To stop the test select `Run -> stop` or select the  button. To clear resuts from previous test runs select the  icon.

Viewing Results

Results are being collected into a table and a graph. Other listeners can be configured.





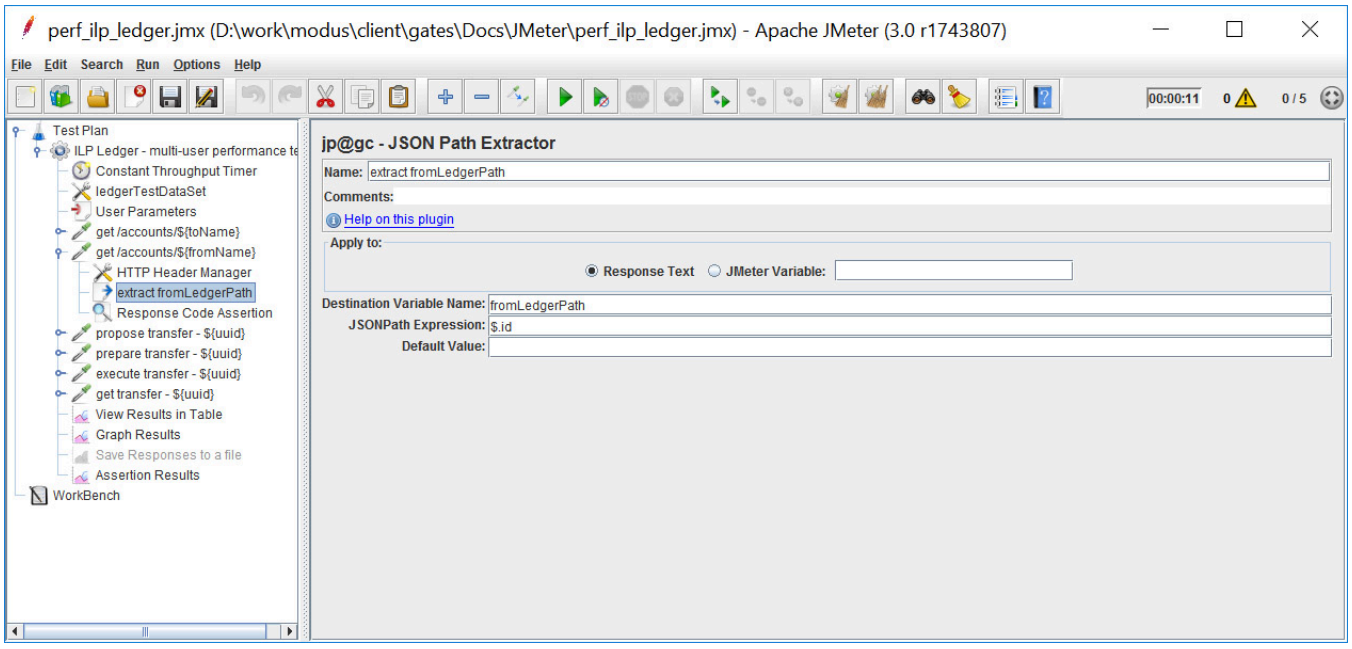
Debugging

If you are getting nothing but errors you can right click on the "Save Responses to a file" element near the bottom and choose enable. It will save the response from each step into its own file. You can also click on the `Test Plan` element at the top and select `Functional Test Mode`. This will cause JMeter to store the results of each call into a file. When debugging I recomment going into the `Thread Group` configuration and setting client threads to 1, unchecking forever and selecting 1 for the number of test iterations to execute. For this test the thread group element is the second one from the top and is called `ILP Ledger - multi-user performance test`

Extending / Reusing

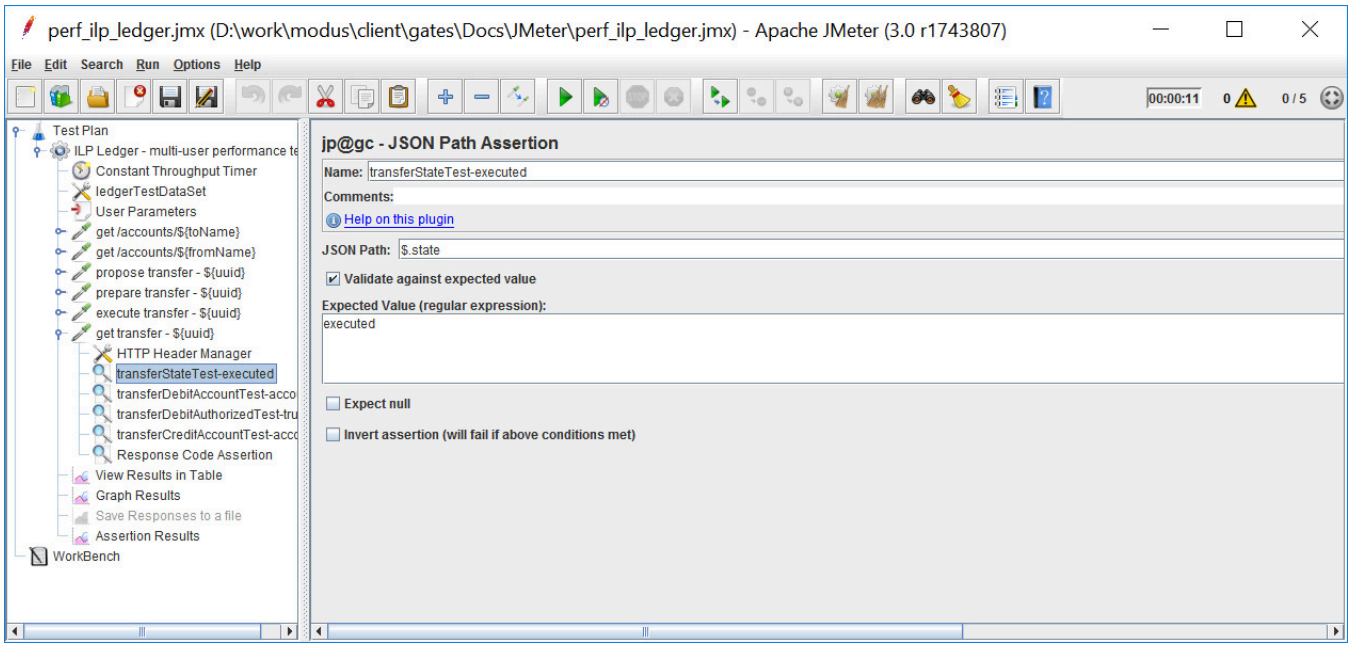
These basic tests can be used as the basis for testing any of our other rest based api's. There are many other connectors available other than rest as well.

JSON Path Extractor:



This element shows extracting data from the response of a call to lookup an account. The `id` field contains the full path of the ledger account and is used in the subsequent call to construct the transfer request. The data is coming from response text, from the element named `id` in the root of the response object, and is being placed in a variable called fromLedgerPath. This will be referenced later is `${fromLedgerPath}`

JSON Path Assertion:



This element shows reading in the value `state` from the root of the response payload and asserting that it equals `executed`. The response code assertion further down checks to see if the status was 200.

Resilience Modeling and Analysis

Wikipedia: "Failure mode and effects analysis (FMEA) . . . was one of the first systematic techniques for failure analysis. "

When FMEA is applied to software services it is called **Resilience Modeling and Analysis** (RMA), see white paper.

It was developed by reliability engineers in the late 1950s to study problems that might arise from malfunctions of military systems.

FMEA can be applied directly to software services to identify and rank possible service failures. Once identified, there are standard mitigation and testing patterns can be applied to resolve each failure. This process is used to prevent major failures and reduce downtime.

What RMA is

Resilience modeling assumes that there will be failures in a system. It doesn't focus on increasing reliability, which is measured below as mean time to failure (MTTF), instead if focuses on reducing time to detection and recovery (MTTD and MTTR). By reducing time to detection and time to recovery (the red area below), availability (the green area) is maximized.

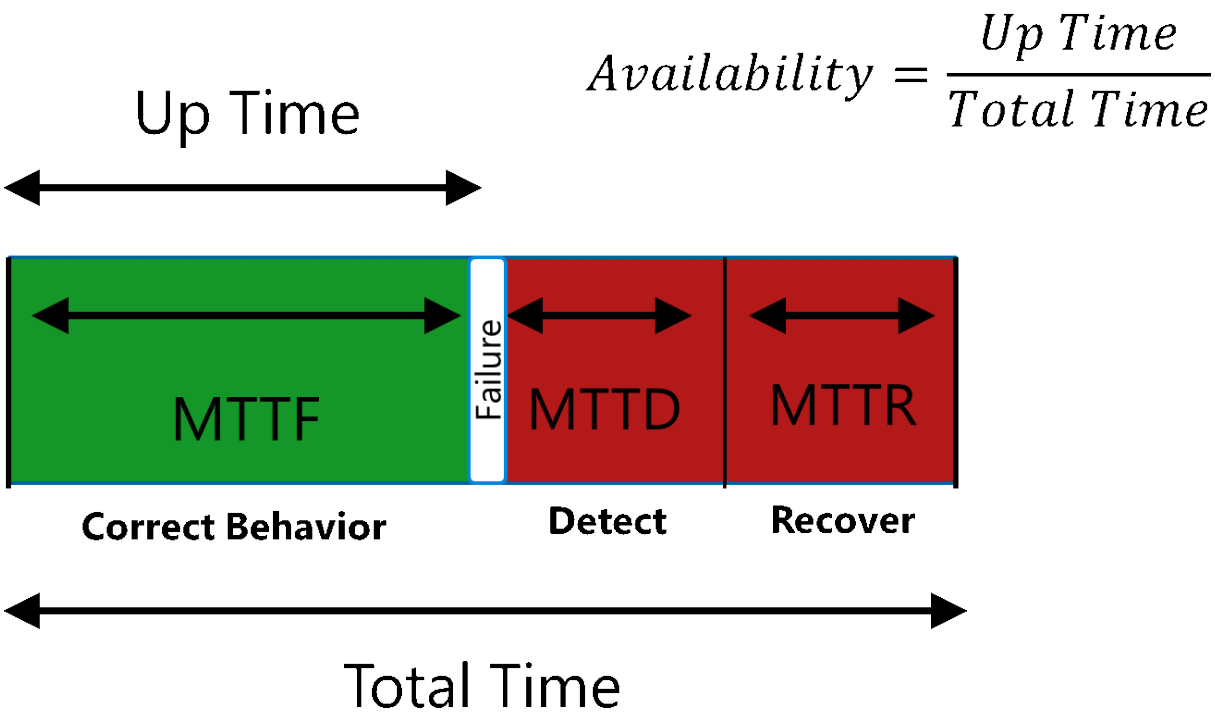


Figure - Availability from MTTF, MTTD, and MTTR

For RMA, developers look at the architecture data flows and ask: "what could go wrong here, how bad will that be, and how often will that happen?" RMA provides a prioritized list of potential faults. We extend RMA to add one or more ways to detect, mitigate, and test the handling for each of those faults.

RMA is a very similar process to threat modeling except that instead of looking for threats, we look for faults and instead of using a threat acronym like STRIDE we use DIAL:

- **D** - Discovery: name resolution, configuration
- **I** - Incorrectness: corruption, version mismatch, sequence errors, duplicates
- **A** - failure of Authorization or Authentication
- **L** - Latency; slow or no response, flooding, deadlocks, metering, timeouts

Standard Microservice Resilience Patterns

Because Mojaloop follows a microservices architecture there are a group of standard potential failures that all such services have. Because every microservice has the same issues, these issues can be grouped together by failure mode along with standard methods of detection, mitigation, and testing.

When we apply DIAL to microservices, we find several standard patterns for failure.

Failure Pattern #1 - Low Resources

A low resource condition is common to all software. It has the advantage that you can often detect and correct the problem before the failure occurs.

Example failures:

- Low memory
- Low disk space
- Excessive CPU
- Peak network traffic

Detection

Use a system monitoring tool (Ex: AWS, Nagios, App Analytics, Sensu, New Relic, SCOM, etc.)

Two stages:

1. Yellow: Raise event when resource is getting low and before it's a problem
2. Red: Alert when the resource is critically low or gone

For each microservice we create a table. Here's an example:

Resource	Green	Yellow	Red
CPU	<80%	>80%, 1-minute average	> 95%, 10-minute average
Disk	<80% full	80 to 95% full	> 95%
Memory	<80% available memory utilized	80 to 95%, 3-minute average	> 95%, 5-minute average
Network	<80% network, capacity 5-minute average	80 to 95%, 5-minute average	> 95%

In Mojaloop, we make use of the ELK stack and Metricbeats for gathering system data. This makes the data available to any number of alerting systems.

Mitigation

Graceful degradation: system continues to function, but some functionality may temporarily stop. As an example, in our case, new money transfer prepare requests might be slow or rejected while the services processes existing fulfillment work.

Fault Injections

There are many standard tools to fill disk space, allocate large amounts of memory, hog CPU cycles, and throttle the network.

Failure Pattern #2 - Service is down

Example Failures:

- Microservice down
- Mule down
- DB/SQL down

Detection

In our case, each microservice implements a health endpoint which returns an http 200 if the service is up. Microservice may implement a JSON return to indicate that the service is degraded (yellow/warning status). The health service works by doing a simple internal check of the service.

We use the free Mule runtime which can be extended on-site for a license fee for monitoring and dashboards to cover this or another monitoring service may be used.

Service failures also are detected by a calling service when that service receives a connection failure (ex: 404). This works for dependent services that don't implement a health service. These failures are logged and picked up by the logging engine (ELK stack) where they can be integrated with a monitoring service.

Mitigations

If service down is detected, the first mitigation is to restart the service. We use ansible playbooks for service startup.

Restart the service in a known good configuration. We use Ansible playbooks to deploy and configure the services. These can be run to redeploy and restart all the stateless services or restart the stateful ones. Restarting the service should generate a configuration change event.

Additional failover processes may be deployed in production.

The monitoring service should alert an operator when the service has been down for a threshold amount of time.

Fault Injections

Stop the service

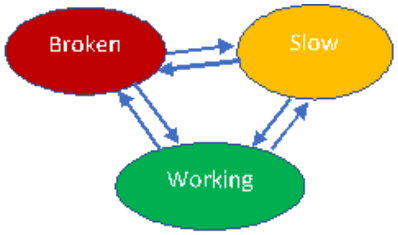
Failure Pattern #3 - Health Modeling

What is Health Modeling?

Health modeling is included here as a part of resilience, but it has a larger role in helping operations maintain the service. Health modeling answers. "what state is the service in, and what action can I take to correct it". In services, a health model defines a pattern that typically looks like "check the service state, attempt to fix it automatically if it's broken, alert the operator if we can't"

The first part of heath modeling is defining the actionable states of the system.

A very simple health model might have three states: broken, slow, and working. The most general form of this model is a finite state model or petri net showing the three states and all possible transitions between them:



where the transitions are typically events that come from log events or health checks. Ex: The transition from Working to Broken might be "health check doesn't return 200".

In a simple model like this one, where the severity of the problems can be stack ranked, it's easy to model the system as a chain of responsibility pattern:

```
If (health state doesn't return 200) then broken

if (health state returns 200 with "slow" in JSON) then slow

else working
```

This kind of pattern is very easy to code and test. You can have any number of if/then statements in this kind of model, and multiple consecutive statements can lead to the same state. State checks are ordered from worst outcome to best. If any statement is true, the chain stops.

The second part of the health model is recovery process. This is also an ordered set of operations that can be shown as one or more chain of responsibility patterns - either for the entire system or for a group of states within it. Example:

```
if (broken for more than 5 minutes) alert operator

If (broken for more than 2 minutes) then raise event and run Ansible
playbook to redeploy service

If (broken) then raise event and run Ansible playbook to restart
service
```

```
if (slow and # of services > N) then alert operator

if (slow) then raise event and run playbook to add additional
microservice

if (working and more than 1 service and a service is idle) then run
playbook to scale down services
```

Describing the actions like this makes it easy to automate the responses and understand what should happen when problems occur.

A General Microservice Health Model

An advantage of microservices is that every microservice has the same kinds of possible states and transitions. A general health model can apply to most of the operations in any microservice. Once we have that model we only need to worry about special cases specific to our service.

Our simple health model has only four states (in order):

- **Stopped** - the service is stopped or unresponsive
- **Misconfigured** - a catch-all state for something has gone wrong that the code can't automatically fix. We don't support auto-rollback of a new deployment, but If you want to you can add "Mis-deployed" state above this one to cover the case where a new deployment has been done N minutes ago yet is still in the misconfigured state.
- **Slow** - performance is below an acceptable threshold
- **Working**

The main difference between the simple health model example above and our model is the addition of the misconfigured state. Below are details for handling each state.

Stopped Service

Example Failures:

- Microservice down
- Mule down
- DB/SQL down

Detection

In our case, each microservice implements a health endpoint which returns an http 200 if the service is up. Microservice may implement a JSON return to indicate that the service is degraded (yellow/warning status). The health service works by doing a simple internal check of the service. We use the free Mule runtime which can be extended on-site for a license fee for monitoring and dashboards to cover this or another monitoring service may be used.

Service failures also are detected by a calling service when it receives a connection failure (ex: 404). This works for dependent services that don't implement a health service. These failures are logged and picked up by the logging engine (ELK stack) where they can be integrated with a monitoring service.

Mitigations

```
    If the service has been down for N + M minutes alert an operator
    Else, restart the service using an ansible playbook
```

Additional failover processes may be deployed in production. Restarting the service generates a configuration change event.

Fault Injection

Stop the service to test that the service will be restarted.

Configuration error examples

Many possible failures lead to the misconfigured state. In all cases, the configuration error detection can come from a logged message since the service is running and logging, it's just not communicating. That message should have a log type to indicate that there's a config error. The type of checks will depend on what communication methods the microservices supports. Here's a list:

Http

- Auth: major security failure. Unable to call upstream service and/or all clients can't get data
- Misconfigured URL
- Misconfigured network
- API version mismatch

Web sockets

- Multiple clients on same socket
- Port not open or configured (ex: in Docker)
- Socket not configured (DFSP initiates)
- Major version mismatch
- client access auth failure - client service logs config error
- error on notify - receiver logs error (we may consider retries here before failing)

SQL

- Misconfigured Connection string
- Misconfigured network

General - infrastructure version incorrect (ex: OS, Docker, JScript)

Besides all the config failures, it can be helpful if the service has a "configuration good" log event that gets fired after startup or a configuration change. This allows the model to know when a state has returned to "working". Since we deal with configuration problems manually, this is not required, but in an automated setup it would be needed.

Detection

- Run a scheduled test to ping the health service
- Listen for actionable log messages marked with the configuration type

Mitigation

Use the mitigations above for restarting the service, but add this check in the middle:

- If the service has been down for N minutes. Use an Ansible playbook to redeploy and configure the service. A stateful service can be also reinstalled, but leaving the existing data volume untouched.

Fault Injections

- Change Http or socket config

- Change client or server auth
- Force update of component to incorrect version or configuration

Slow Service

If performance is below an acceptable threshold the health state will return that. The model here follows the example above

Detection

- if (health state returns 200 with "slow" in JSON) then slow

Mitigation

```
-   if (slow) then raise event and run playbook to add additional
    microservice
-   if (working and more than 1 service and a service is idle) then
run
    playbook to scale down
```

Fault Injection

- Use a tool to load the processor

Mojaloop Specific Health Modeling

Mojaloop has two additional potential faults that need to be addressed that could cause a participating DFSP to lose money. These are *overloaded ledgers* and *dropped messages*.

Overloaded Ledger

Example: Payer sends \$100. Payee DFSP agrees and starts fulfilment. The payer ledger is overload and doesn't resolve the transfer in time, however, it's been fulfilled by the payee DFSP and the center. Payer DFSP losses \$100 during settlement.

A similar problem happens if the central ledger doesn't resolve the transfer. Then the payee DFSP can be out the \$100 during settlement.

Detection

Track remaining time on all transfers. If a transfer is not reported before the timeout window (either fulfilled, rejected, or cancelled), then it's delinquent and needs to be checked on.

Mitigation

1) Thread priority: Ledgers handle fulfillments before preparing new transfers. 2) Query on timeout: If payer DFSP hasn't explicitly heard a "fullfil", "reject", or "cancel" message at the end of a transfer timeout, it queries the center to get the current status of the transfer. The payee DSFP can check the status at anytime, such as before a settlement window or on restart of it's services after a failure.

Extending this pattern: This solution treats the central ledger as the source of truth. It can be extended to multiple hubs where the transfer goes through many hops before getting to the final destination. This works following an eventual consistency model the same as above but with multiple central hubs. Each central hub acts exactly the same way and uses the same transfer ID for the transfer. At the end of a timeout, if any participant doesn't know the status, they check and retry the next participant up the chain. Whenever and participant changes a ledger they send status both up and back down the chain. On the downside, they should get a corresponding change notification. If that doesn't happen they retry.

Fault Injection

Take down the payee ledger adapter

Messages dropped

As with the ledger overload problem, if the ledger notifications are missed or dropped the payer or payee ledger can lose money.

Detection 1. Payer DFSP detects when no message is received within the timeout 2. Payee DFSP doesn't recieve an ACKnowledge and fulfillment notification from the central ledger.

Mitigation 1) Query on timeout: The payer DFSP can resolve this with the same mechanism as an overloaded ledger. 2) Wait for Notify Fulfillment message: The payee DFSP would lose money if it fullfils a transfer, but doesn't deliver the notification of it to the central ledger. To mitigate this, the payee DFSP doesn't notify the payee or hand any money out until it recieves an fulfillment notification from the central ledger. The payee DFSP can choose to check on a transfer status with the center, but it is not recommended to do this for every transfer. 3) Retries (w/idempotent writes): the ILP-Connector may retry fulfillment messages automatically (opt-in) if there's no response. For this to work properly, the ledgers must implement idempotent writes on the transfer ID.

Fault Injection

Drop/block the fulfillment notifications

Setting up the Development Environment

Install development tools

Install Visual Studio Code editor

Download and install VS Code from Download Visual Studio Code.

Install Node.js platform

Download and install lastest stable version of Node.js from Node.js downloads. Check the version by typing `node -v` in the console. It should be at least 4.5.0.

Update npm package manager

Node comes with npm installed. Update to the lastest version with the command `npm install npm -g`. Check the version of npm with the command `npm -v`. It should be higher than 3.10.

Install git

Install git as appropriate place for your operating system.

Clone the project

Generate and add SSH key to github. If you are not sure how to do that, you can follow the guides here Generate an

SSH key Navigate to the directory where the project should be cloned and type in the console for example:

```
git clone git@github.com:LevelOneProject/dfsp-directory.git
```

Configuration file for the database

Create configuration file named .ut_dfsp_directory_devrc in the home directory (C:/Users/[username]) that contains the individual access settings for the database. The content of the file should be the following:

```
[db.db]
database=dfsp-directory-<name>-<surname>
user=<name>.<surname>
password=<password>

[db.create]
user=<admin user>
password=<admin password>
```

The common settings can be found in the dev.json file in the server directory of the project.

Run npm install

In the project diectory (dfsp-directory) run `run npm install`.

Launch Configurations

Debugging in VS Code requires launch configuration file - `launch.json`. To create it click on the Configure gear icon on the Debug view top bar, choose debug environment and VS Code will generate a launch.json file under workspace's .vscode directory. Generated for Node.js debugging launch.json should look like the following:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "directory",
```

```
            "type": "node",
            "request": "launch",
            "program": "${workspaceRoot}/index.js",
            "stopOnEntry": false,
            "args": [],
            "cwd": "${workspaceRoot}",
            "preLaunchTask": null,
            "runtimeExecutable": null,
            "runtimeArgs": [
                "--nolazy"
            ],
            "env": {
                "NODE_ENV": "development"
            },
            "externalConsole": false,
            "sourceMaps": false,
            "outDir": null
        }
    ]
}
```

- `name`: name of configuration; appears in the launch configuration drop down menu
- `type`: type of configuration; possible values: "node", "mono"
- `program`: wrkspace relative or absolute path to the program
- `stopOnEntry`: automatically stop program after launch
- `args`: command line arguments passed to the program
- `cwd`: workspace relative or absolute path to the working directory of the program being debugged. Default is the current workspace
- `runtimeExecutable`: workspace relative or absolute path to the runtime executable to be used. Default is the runtime executable on the PATH
- `runtimeArgs`: optional arguments passed to the runtime executable
- `env`: environment variables passed to the program
- `sourceMaps`: use JavaScript source maps (if they exist)
- `outDir`: if JavaScript source maps are enabled, the generated code is expected in this directory

Required Extensions for VS Code

In VS Code press `ctrl` + `shift` + `p` and type in the new open input field Install Extensions, or press `ctrl` + `shift`

+ x to go directly to the extensions.

beautify

Find `beautify` in the Extensions marketplace, install and enable it. This extension enables running js-beautify in VS Code. The generated .jsbeautifyrc loads code styling. It should has the following settings:

```
{
  "end_with_newline": true,
  "wrap_line_length": 160,
  "e4x": true,
  "jslint_happy": true,
  "indent_size": 2
}
```

Formatting code can be done with `Shift + Alt + F`.

CircleCI

Find `CircleCl` in the Extensions marketplace and install it. To enable it go to CircleCI and create an API token. Add it as `circleci.apiKey` in the Workspace Settings in VS Code (File -> Preferences -> Workspace Settings):

```
{
    "circleci.apiKey": [API token]
}
```

ESLint

This extension contributes the following variables to the Default settings of VS Code:

```
"eslint.enable": true,
"eslint.options": {}
```

- `eslint.enable`: enabled by default
- `eslint.options`: options to configure how eslint is started. They can be specified as valid for all projects in the User Settings (File -> Preferences -> User Settings) or only for a project in the Workspace Settings (File -> Preferences -> Workspace Settings) in which case the User Settings will be overwritten.

Each project includes the module ut-tools as developmen dependency. You need to point eslint config file to the eslint

settings used.

Example

```
{
    "eslint.options": {
        "configFile": "/[path-to-project]/node_modules/ut-tools/
eslint/l1p.eslintrc"
    }
}
```

Guidance for writing code for Mojaloop

- Pragmatic REST Guidelines describes how we use restful interfaces
- We also have standards describing how the restful interfaces are documented
- Our Branching Strategy and versioning is fairly simple
- An example of how to Build and Publish Node.js using CircleCI to the private package repository.
- Guidance for setting up and building a new Java repository
- We have standards for Code-Quality-Metrics
- Code Style lists our preferred code style checking tools.

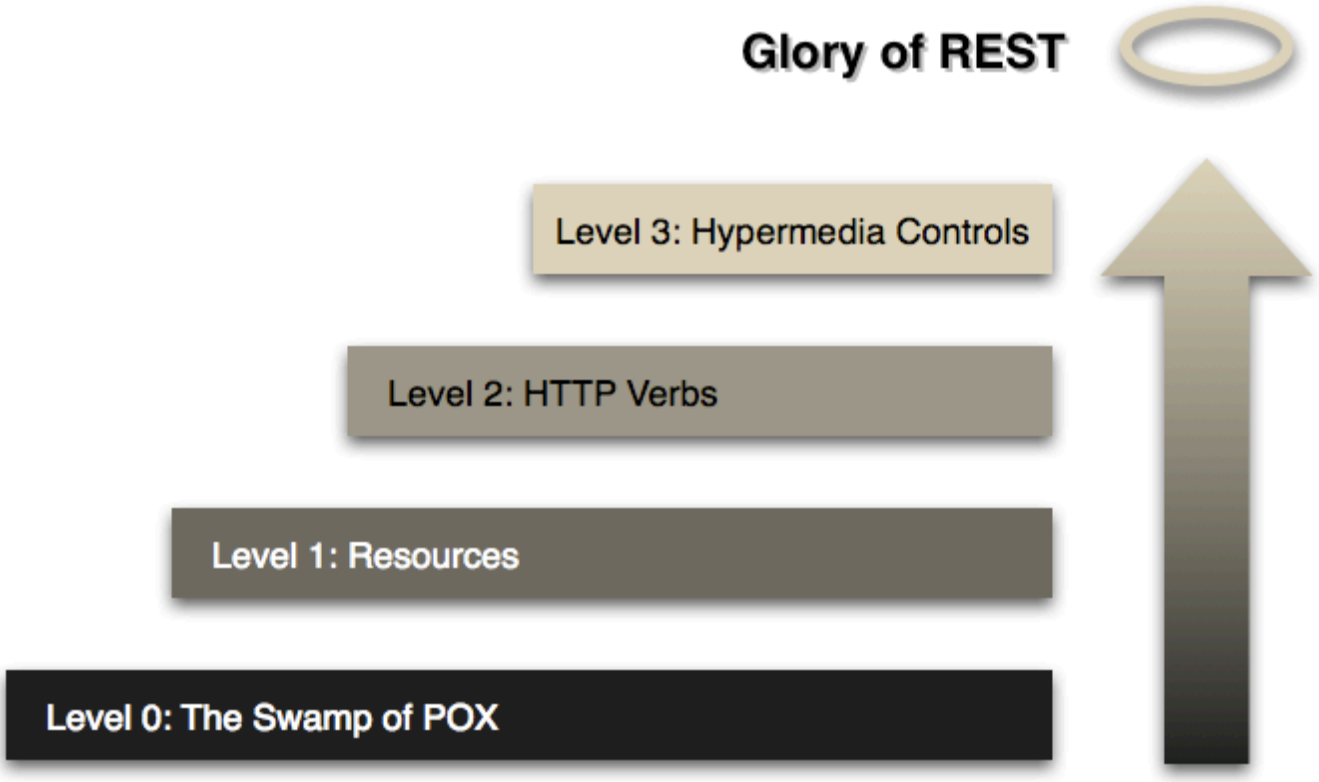
Pragmatic REST

For the Mojaloop Project

With the emergence of API strategy as a scaling tool for Internet service businesses, the focus on interconnect technology has shifted. Building on the principles that enabled the Web to form and scale, REST (Representational State Transfer) has become a design preference for Internet service APIs. But while the REST principles, proposed in Roy Fielding's dissertation that defined them, have academic value as a basis for research, a pure REST design is not at present practical for most applications. We are advocating a kind of Pragmatic REST-a design pattern that adopts the beneficial components of RESTful design without requiring strict adherence to academic purity.

The Richardson Maturity Model

Martin Fowler has referenced[1] a structured model of RESTful adoption developed by Leonard Richardson and explained at a QCon talk. Fowler refers to this as the Richardson Maturity Model of RESTful design.



Martin Fowler, referencing Rest in Practice,[2] summarizes the genesis of RESTful design:

> use Restful web services to handle many of the integration problems that enterprises face. At its heart . . . is the notion that the web is an existence proof of a massively scalable distributed system that works really well, and we can take ideas from that to build integrated systems more easily.

A pragmatic approach to RESTful design uses the best parts of Fielding's conceptual framework to allow developers and integrators to understand what they can do with the API as rapidly as possible and without writing extraneous code.

At its most fundamental, a RESTful design is resource-centric and uses HTTP verbs. At its most advanced, a design that follows pure academic REST utilizes the HATEOAS principle by implementing Hypermedia Controls. We are advocating a Level 2 RESTful design for Mojaloop.

Why not Hypermedia Controls?

Although HATEOAS is a fascinating principle-it advocates that a server should respond to each client action with a list of all possible actions that can lead the client to its next application state. And further, clients *must not* rely on out of band information (like a written API spec) for what actions can be performed on which resources or on the format of URIs.

It is this final proscription that fails the test of Pragmatic REST: While HATEOAS is an interesting theoretical approach

to limit coupling, it does not easily apply to Mojaloop (or any other contract API design). When we take into account our audience for the interconnect APIs, we find a group of commercial entities that will be operating under a set of highly specific scheme rules. Interactions between the participants, and between participant and central service hub, will be highly specified to assign acceptable commercial risk that can be priced at very low cost to end-users. This requires *ex-ante* predictability of the API which is anathema to the HATEOAS principle defined by Fielding.

Pragmatic RESTful Principles

URIs Define Resources

A well-designed URI pattern makes an API easy to consume, discover, and extend, just as a carefully designed API does in a traditional programming language. Pure REST disdains this principle in favor of HATEOAS. But pragmatic REST follows a normal pattern for URI definitions to improve human understanding, even if HATEOAS principles are employed for discovery.

URI paths that refer to a collection of objects should consist of a plural noun, e.g. /customers, to refer to a set of customers. When a collection can have only one instance, the singular noun should be used to avoid confusion. E.g. GET /transfers/:id/fulfillment is correct, since there is only one fulfillment object per identified transfer.

URI paths that refer to a single object should consist of a plural noun (representing the collection), followed by a predefined unique identifier. E.g., /customers/123456 to refer to the specific customer with number 123456. The identifier must be unique within the containing collection and persist for the life of the object within that collection. IDs must not be ordinal values-ordinal retrieval of objects from a collection is possible using query parameters on the collection URI.

URI paths may have a prefix to identify the environment, version, or other context of the resource. Nothing should follow the identifying path but collections and object references.

URI path and query segment identifiers should be chosen from the Roman character set, [0-9A-Za-z]. Use *camelCase* to define the elements of the URI path. Do not use snake_case.

For the avoidance of doubt, "_" (underscore) and "-" (hyphen) should not be used in URI path or query segment identifiers.

This probably seems a bit parochial. The purpose is to find a well-defined URI format that is consistent with wide-spread practice, easy to define, predictable, and that maps to native environments and conventions. It isn't going to satisfy everyone. Here is reasoning behind this constraint:

CapitalCase and camelCase are the defacto standard for NodeJS and JavaScript and are a common constraint in URI definition: URI path segments are often mapped to JS internal resources and so conforming to JS naming conventions makes sense.

Field names in JSON and SQL should also follow this convention since they are often automatically mapped into variable name space and can be referenced in URIs as path or query segment identifiers.

We should also avoid the use of "\$" unless it is required by a library (e.g. JQuery). IBM JCL has passed away; let it rest in peace. There are better scope control tools to separate name spaces than introducing non-roman symbols.

We should avoid "-" (hyphen) in path segment and query parameter names as it does not map into variable names, SQL, or JSON field name identifiers.

Underscore characters must be escaped in markdown source by prefixing each with a "\" character.

Snake_case has been reported to be slightly easier to read than camelCase in variable names, but it actually does not improve readability of URIs, as it visually interferes with path and query segment delimiters making it difficult to visually parse them. And when URIs are underlined in presentation, the underscores become illegible.

URI Parameters

Use a standard and predictable set of optional parameters in a consistent way.

A set of standard query parameters should be used for collections to enable caller control over how much of the collection they see. E.g. "count" to determine how many objects to return, "start" to determine where to start counting in the result set, and "q" as a generic free-form search query. We will define the standard set of parameters as we go and will apply them consistently.

Verbs

Singular objects should support GET for read, PUT for complete replacement (or creation when the primary key is specified by the client and is persistent, e.g. a payment card PAN), and DELETE for delete.

Collections should support GET to read back the whole or part of a collection, and POST to add a new object to the collection.

Singular objects may support POST as a way to change their state in specified ways. Posting a JSON document to a singular object URI may allow selected field values to be updated or trigger a state change or action without replacing the whole object.

GET must be implemented in a *nullipotent* manner-that is, GET never causes side effects and never modifies client-visible system state (other than logging events or updating instrumentation, e.g.).

PUT and DELETE must be implemented in an *idempotent* manner-that is, changes are applied consistently to the system data in a way that is dependent only on the state of the resource and inputs but on nothing else. The action has no additional effect if it is executed more than once with the same input parameters and does not depend on the order of other operations on a containing collection or other resources held in the collection. For example, removing a resource from a collection can be considered an idempotent operation on the collection. Using PUT to fully replace (or create) a

uniquely identified resource when the URI is fully known to the client is also idempotent. This implies that the system may reorder operations to improve efficiency, and the client does not need to know whether a resource exists before attempting to replace it.

POST and PATCH[3] are not idempotent operations. POST is used to create new resources where the resource identifier is assigned by the server or where a single identified internal resource is implied by the target URI (e.g. POST /transfers, but PUT /transfers/:id/fulfillment).

Data Format

We favor JSON[4] related data formats over XML. In some cases, data formats will be binary or XML, as defined by pre-existing standards, and these will be precisely specified. Binary formats should have a formal syntax to avoid ambiguous representational translations (e.g. character set translations, big- or little-endian representations of numeric values, etc).

Date and time values used in APIs should comply to the ISO 8601 standard, and further profiled by the w3c Note on Date and Time Formats.[5] This w3c note should lead to the reduction in complexity and error scope of communicating components that must exchange tangible dates and times. There will be cases where we use non-ISO format date or time as required by an external standard, e.g. ISO 7813 expiry dates.

Existing standard XML formats should have an XSD schema for the acceptable subset profile used within the project. For particularly complex data formats, we may use a common format profile translator to map between our project subset of the standard format and the wire format used by a standardized protocol (e.g.). This will limit coupling to complex formats in a more maintainable way.

When specifying the PATCH action for a resource, we will use a consistent patch document format (e.g. JSON Patch[6]).

Return Codes

Use HTTP return codes in a consistent way and according to their standard definitions. The standard codes are defined in RFC 2616.[7]

Machine Readable Error Format

The API should provide a machine readable error result in a well-defined JSON format. {TBD whether to use a response envelope and how to format errors, faults, and success envelopes. RESTful design relies on headers to carry protocol-defined errors, debug info can also be carried in headers. We should be clear on why we are using an envelope and how this supports normal production communication between client and server.

Versioning

API URIs should include a version identifier in the format v*M* as a leading path element (where *"M"* is the Major component of the multi-part version number). The API and its version identifier element must conform to the semantic versioning[8] 2.0 specification for API versioning.

A client must specify the Major version number in each request. It is not possible for a client to express a requirement for a specific minor version.

The full API version number is specified in the response header (TBD) for all successful and error responses.

While an API version contract will be influenced by Major, minor, *and* patch levels, only the Major version number is a production API binding element-that is, a production client cannot request a particular minor version or patch level and a production server will not accept a URI request that specifies these extra elements.

However, in pre-production environments, it is anticipated that some combination of minor, patch, pre-release, and metadata suffixes would be supported in client requests (as defined in *semver* [3]) and *may* be expressed in *pre-production* URIs to assist with development and integration scenarios.

We May Need to Give REST a Rest

As we design the interconnection APIs between components and between participating systems, we may find API requirements that don't precisely match the Pragmatic REST pattern defined here. We will evaluate these case-by-case and make the best choice to support the project goals.

Non-Functional Requirements

As we develop the APIs, we will make consistent choices about non-functional requirements to reinforce the project goals.

1: http://martinfowler.com/articles/richardsonMaturityModel.html, retrieved August 18, 2016.

2: https://www.amazon.com/gp/product/0596805829, retrieved August 18, 2016.

3: RFC 5789, *PATCH Method for HTTP*, https://tools.ietf.org/html/rfc5789, retrieved August 18, 2016.

4: *Introducing JSON*, http://json.org/, retrieved August 18, 2016.

5: http://www.w3.org/TR/1998/NOTE-datetime-19980827, retrieved August 22, 2016.

6: *JSON Patch*, http://jsonpatch.com/, retrieved August 18, 2016.

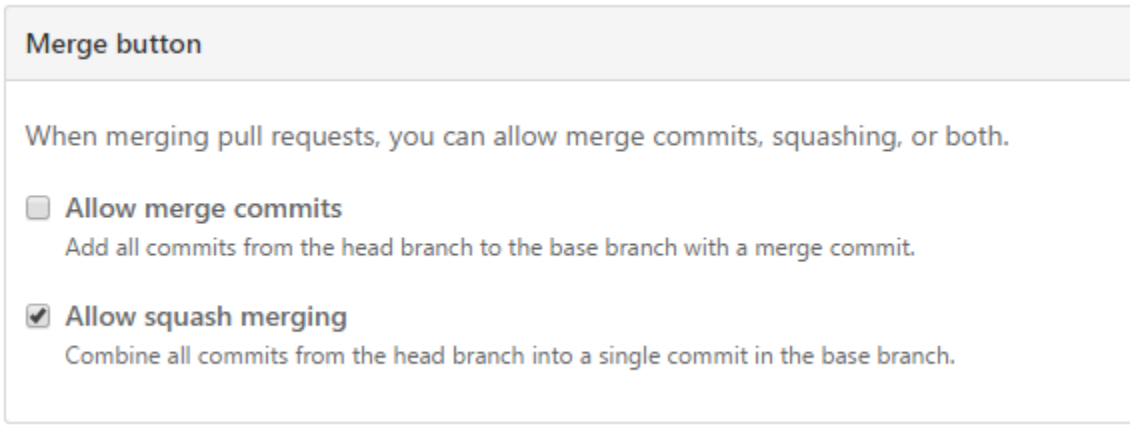
7: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

8: *Semantic Versioning 2.0.0*, http://semver.org/, retrieved August 18, 2016.

The master branch for each repo must represent a good state where all tests have been run and pass. You should update master at the end of user stories when the code is ready for checkin. The master branch is used to build components for test and integration with other components.

Setup the repo to squash commits instead of merging

We like code checked into master to have a simple clean story, so we want all repos to use the squash option: Under your repository settings uncheck allow merge:



This will force changes to master to be single named commit (see Squash your commits).

Versioning

Based on semantic versioning (Breaking.Feature.Fix), as long a component is not shipped, the major version will be less than 1 (example: 0.15.1). Most user stories will increment the middle number. Minor changes and bug fixes increment the last number. Example:

`git tag #` this will tell you the current tags including the version

`git tag v0.3.0` # this sets a version tag

```
git push --tags
```

Integrated tests

Make sure integrated build and testing is turned on for the repo at: https://circleci.com/add-projects. This will build and run tests after any push on any branch. The test override section of the circle.yml has the commands to run for the tests.

Quick and dirty

If you are making a small fix on a local branch dev that branches right off of master, you might follow a pattern like this (there are better as we'll see below):

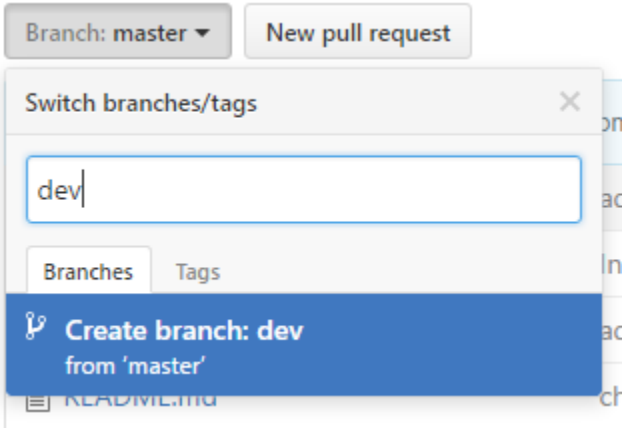
```
(master) git checkout -b dev # create the dev branch and move to it

[code change here]

git commit -m "a small change to the dev branch"

git rebase dev master # get the latest changes from master and move the master head up to dev

git tag v0.3.1 # update the version

git push && git push --tags # update the server and run the tests
```

This will pick of the latest changes anyone else made on master and add yours onto them. We use rebase instead of merge to keep a clear story of the changes. If you have a bunch of extraneous commits, consider using git rebase -i to do an interactive rebase and remove a bunch of the extra noise.You can choose to keep your dev branch around if you are going to reuse it or else delete it.

The procedure above doesn't really work for larger changes since you can't share the development branch with others, you can't do code reviews on it in Github, and the integration tests won't run on your dev branch in advance of the push - you have to do the tests manually. This makes it inferior for larger changes, and we won't use it at all once we go past v1 or go to open source. With that in mind, let's look at a better solution.

Cloned or Forked repos

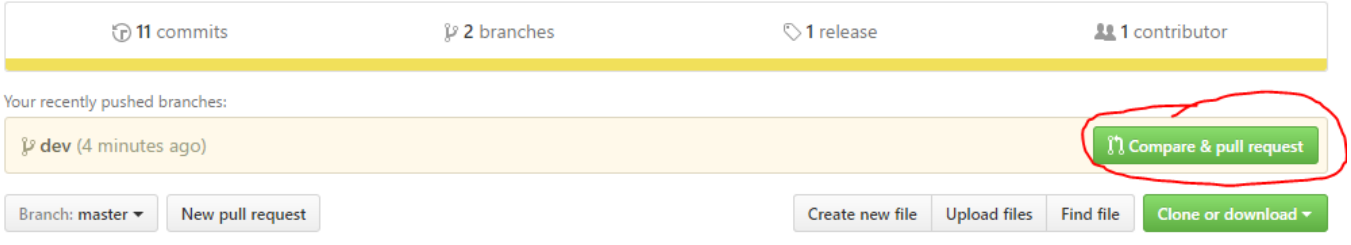
Instead of just a local branch like above, you create the branch on the server.



Then, locally:

```
git checkout dev # creates the local copy of the remote dev branch and switches to it

# make some code changes

(dev) git commit -m "user story #111 "

git tag v0.4.0 # update the version

git pull --rebase

git push && git push --tags # kicks off tests on dev branch, yea!
```

Now create a pull request from dev to master



After you resolve the comments, you can push the code.

Open Source

An open source contributor would follow almost the same steps. They'd fork instead of cloning. They'd do the same steps (without tags). The final step would be done by one of the repo owners.

Publishing to the Private Repo

See the DeployTest repo for example code

There are three files you need for build/deploy in CircleCI

package.json

```
"name": "@Mojaloop/YourRepoName",
"license": "SEE LICENSE IN LICENSE",
```

The package is identified with the Mojaloop organization name and License file.

CirceCI.yml

```
machine:
  node:
    version: 6.5.0

deployment:
  releases:
    branch: master
  commands:
    - npm publish
```

The CircleCI file defines the version of Node to use and when to publish to the repo (typically from a push to master branch).

.npmrc

```
registry=https://modusbox.jfrog.io/modusbox/api/npm/level1-npm-
release
_auth = ${NPM_TOKEN}
email = ${NPM_EMAIL}
always-auth = true
```

The .npmrc file for the repo defines where the private npm package repo is, the authorization token to write to it and the email to notify. To test that publishing works locally, you'll run *npm publish*. You'll want a local .npmrc file at cd ~ that defines the real values for your email and the auth token. Here are instructions to get the auth token.

CircleCI doesn't have your local environment variables so you have to add the email and auth token to it. Use the UI at https://circleci.com/gh/mojaloop/YourRepoNameHere/edit#env-vars to add the two variables.

Your account will need full read and write access to your private npm repo.

See more info at Adding a new project into CircleCI

Installing from the Private Repo

So, you have another repo that takes a dependency on the package you've published...in order to npm install from the private repo you need a few pieces in the package.json

```
"repository": {
   "type": "git",
   "url": "git@github.com:Mojaloop/thecurrentrepo.git"
 },
 "dependencies": {
    "@Mojaloop/thepackageyoupublished": "^0.1.0"
 },
 "publishConfig": {
    "registry": "https://modusbox.jfrog.io/modusbox/api/npm/
level1-npm-release"
 }
```

Functional quality metrics

Unit test metrics

High coverage and low dependencies show that the code is testable and therefore well isolated and easy to maintain. Low complexity also makes code readable and maintainable and helps enforce single responsibility. Real unit tests run very fast as they don't call external components.

Code Quality Metrics	New and Project Code
Unit test coverage	>= 80% block coverage
Unit test speed	<= 10 seconds
Dependencies/method	<= 10
Complexity/method	<= 7

Component

Functional testing typically covers pair combinations of the system states.

Integration

Functional tests have one test per message and error. Messages and errors that are handled the same way use the same test.

Contract

Limited to what the consuming teams need that isn't covered by existing unit, component, and integration tests. Often added to over time.

End to End

End to end tests cover acceptance tests from scenarios.

NodeJS

We follow the Standard style guidelines. Add `npm install standard` to you package.json.

Java

For Java we follow Checkstyle.

Story

A story should be granular to be completed within 1 sprint (2 week iteration)

See Story Template

bug

See Bug Template

duplicate

Link item to duplicate item and close

Epic

An epic is a work item that is bigger than a sprint, but less than a program increment for one team. See Scenarios and Epics

question

Specific label for general questions/discussions.

Debugging

Using Logging

Introduction

This document provides Mojaloop services logging guidelines in order to provide end-end traceability of interactions, aid in troubleshooting and publish metrics to the backend

Desired Goals

- End-to-end Traceability of a particular transaction
- Understand service behavior
- Debuggging
- Metrics for a particular transaction

General:

All logs statement must begin with **ISO8601 compliant timestamp**. The timestamp must have millisecond resolution. It should be follwed by a log level. Available logs levels are **ERROR, WARN, INFO, DEBUG** For example

```
2017-04-28T17:16:20.561Z INFO ilp-routing:routing-tables debug
bumping route ledgerA: mojaloop.dfsp1.  ledgerB:   nextHop:
mojaloop.ist.dfsp2
```

1. End-to-end Traceability:

To provide end-to-end traceability of Level One payment related interactions, L1P components shall include **L1p-Trace-Id** in all log statements where available. The following snippet must be included in all of the log lines: `L1p-Trace-Id=<current_trace_id>` For a given unit of work, related interactions between services, the L1p-Trace-Id is required to be unique. It is recommended that UUID be used for uniqueness requirement. This would allow to quickly retrieve all logging for a given L1p-Trace-Id.

2. Rest Service Calls:

All Rest Service calls must include **L1p-Trace-Id** as a header HTTP Header. The value of this header must be set to *Payment ID* for all payment interactions. For all non-payment interaction the originating DFSP must generate and use an UUID for the value of the header. In the case where the **L1p-Trace-Id** is not present as a header, an error needs to be logged with context about the call with the missing header. The service must set the **L1p-Trace-Id** header with appropriate value whether the current interaction is during the course of processing a payment or not.

3. Web Socket Notifications:

[To Be Filled in]

4. Addtional Context in Logs

It is recommended to log other identifiers that can help retrieve logs statement across multiple layers in the Mojaloop stack. Some examples are Transfer Id, User ID (USSD id, email, login name), AppName, and AccountId etc. `Relevant-Id=<id_value>`

5. Metrics Logging

Logs can be used to publish metrics to metrics service. There are 2 types of metrics that are supported. The details about supported metrics are available here. The snippet below shows the systax for publishing metrics:

1. Counter `... L1P_METRIC_COUNTER:[counter-namespace.name] ...` where L1P_METRIC_COUNTER is a keyword followed by a colon and the desired metric name* is within [ ]. This would increment the counter identified by metric name by 1.
2. Timer `... L1P_METRIC_TIMER:[timer-namespace.name][50] ...` where L1P_METRIC_TIMER is a keyword followed by a colon, the desired metric name *is within [ ] and timed value in millisenconds with [ ].*

This would add the time value in milliseconds to timer identified by metric name.

* metric name is composed of 3 elements that separated by a period.
1. environment which captures where the application is runnning e.g. *dfsp1-test, dfsp2-qa* etc. 1. application instance id which should identify the process 1. metric name which could contain alpahnumeric characters and could have java package name style prefix

Using Kibana

How to use Kibana, its dashboards and query language for L1P tracing and debugging purposes.

Kibana is the ELK Stack (Elastic Stack) window into the Elasticsearch data. It allows you to monitory, query, visualize and create reports on Elasticsearch data. This document is a L1P Kibana User Guide that will show how to use Kibana for the most common L1P use cases. Kibana features used at L1P projects will be described and then specific use cases will be described in detail.

Kibana Dashboards and Visualizations

Kibana allows you to create visualizations based on the Elasticsearch data. Furthermore, Kibana allows you to create dashboards based on one or more visualizations.

Elasticsearch data is generated by the several components:

- Logstash
- Filebeat
- Metricbeat
- Heartbeat

For the data generated by the Beats family of shippers Kibana also contains out of the box Dashboards and Visualizations. These come prepackaged as part of the given Beat family shipper. After installing the given Beats family shipper its dashboards and visualizations can be imported.

Dashboards Import

Following are the scripts to install the Beats family shipper dashboards:

- /usr/share/filebeat/scripts/import_dashboards
- /usr/share/metricbeat/scripts/import_dashboards
- /usr/share/heartbeat/scripts/import_dashboards

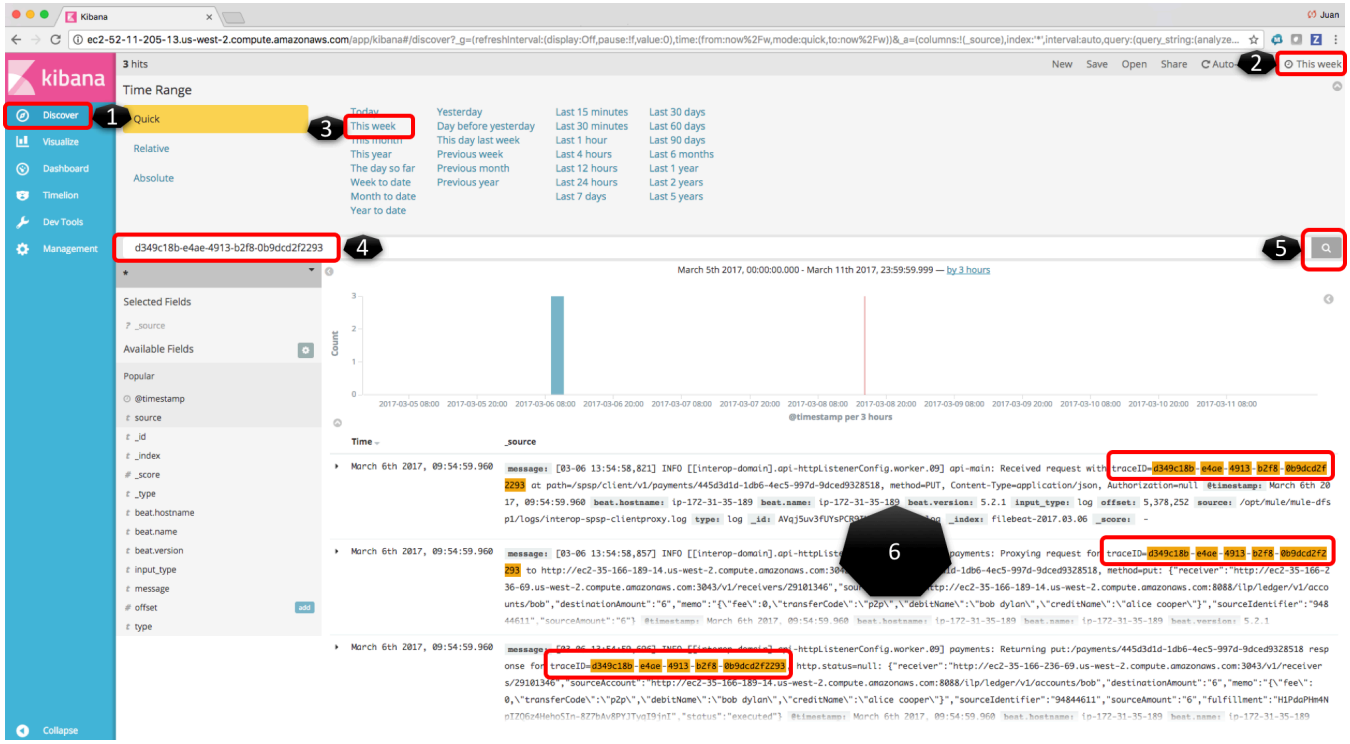
In order to access the visualization and dashboards that are imported by these scripts, go to Kibana and navigate to the Visualizations or Dashboards menu options at the left hand menu.

Access to Kibana via NGINX on your browser

http://EC2_INSTANCE_URL

L1P Kibana Use Cases

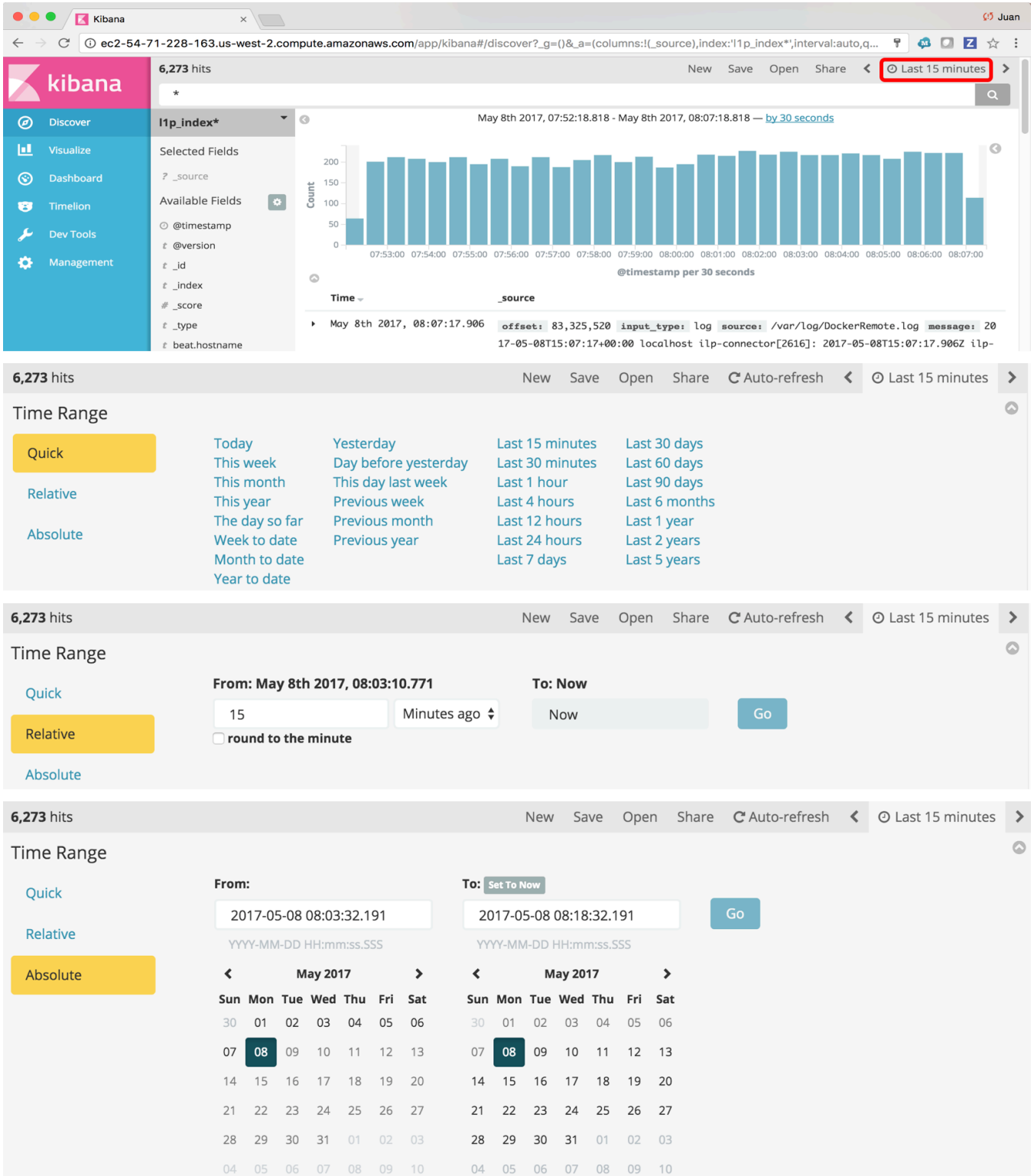
How to monitor logs?



Follow the following steps:

1. Navigate to "Discover" menu
2. Expand Time Range, by clicking the "Time picker" icon on top right corner
3. Set Time Range, pick between Quick, Relative and Absolute modes and set time range
4. Enter your search criteria (e.g. L1p-Trace-Id)
5. Perform search
6. Review logs returned

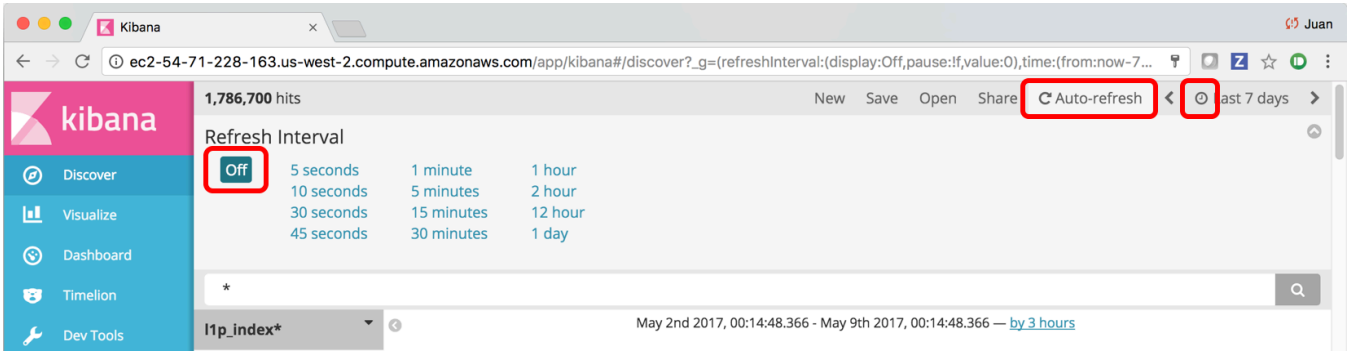
How to set time range?



By default, Kibana will show you logs for the last 15 min. To set the time range from the Discover Kibana page click on the area where with the "Time picker" icon on the top right corner. This will expand the "Time Range" control panel

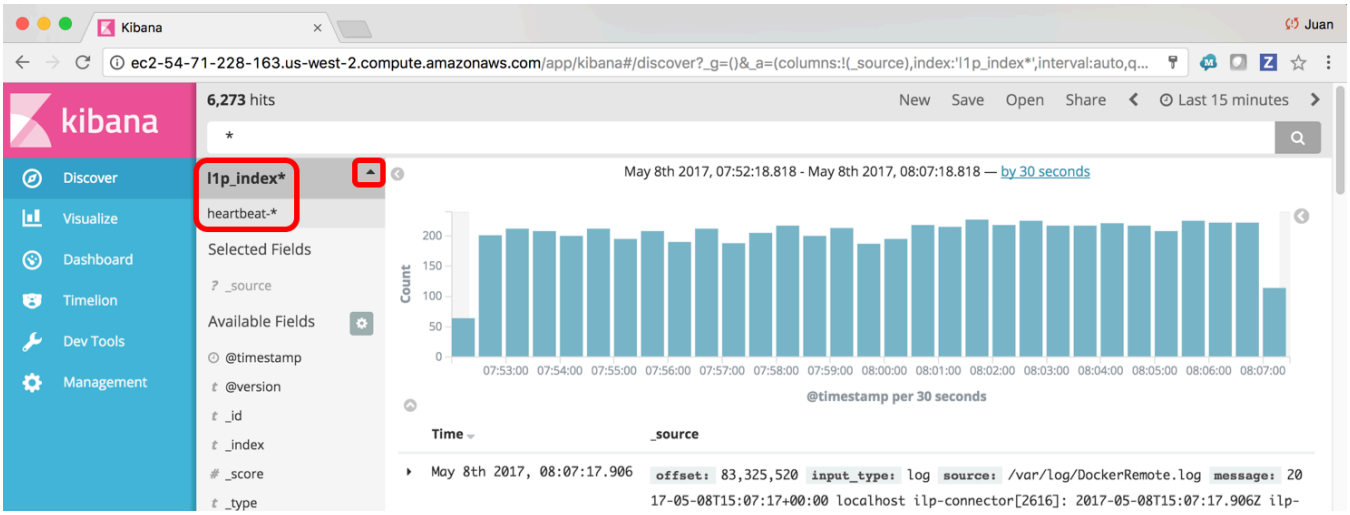
down containing three different modes to set the time range; "Quick", "Relative" and "Absolute" as shown in screenshots below.

How to enable auto refresh of search results?



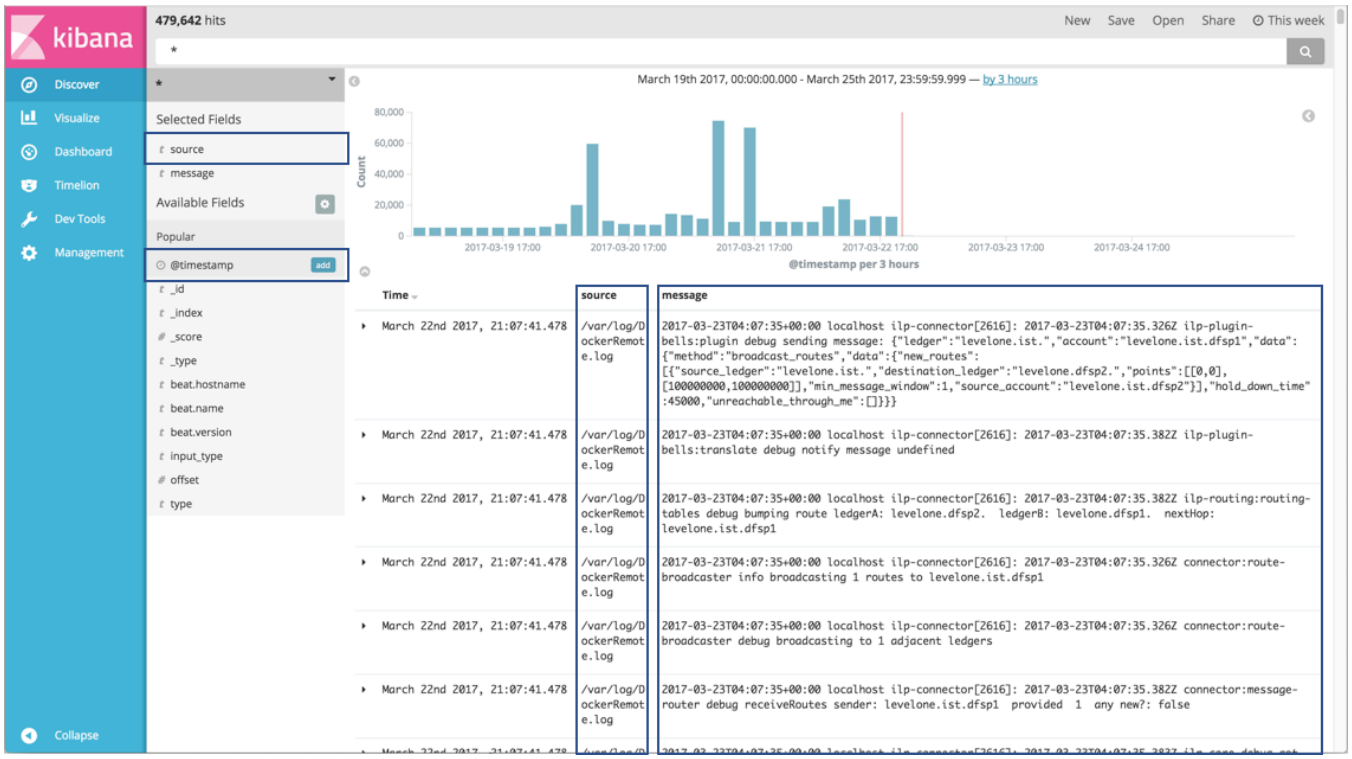
Search results can be set to auto refresh, so your search results and visualizations do not contain stale data. Optionally, you can manually refresh results by clicking "Refresh". The Auto-refresh can be enabled by clicking the "Time picker" icon, clicking the "Auto-refresh" link and then set it to on and specify the refresh rate.

How change to which indices you are searching?



When you submit a search request, the indices that match the currently-selected index pattern are searched. The current index pattern is shown below the toolbar. To change which indices you are searching, click the index pattern and select a different index pattern. NOTE: By default, only 1 index pattern is shown, you must click the arrow to expand the index section to show you the different indexes available.

How to add/remove fields from Kibana's Discover window to monitor logs?



Navigate to the Kibana Discover page and hover over the field you would like to add/remove from the search results table and click the add/remove button.

How to trace a L1P transaction based on its L1p-Trace-Id?

Navigate to the Kibana Discover page and enter a search criterial like:

L1p-Trace-Id=d349c18b-e4ea-4913-b2f8-0b9dcd2f2293

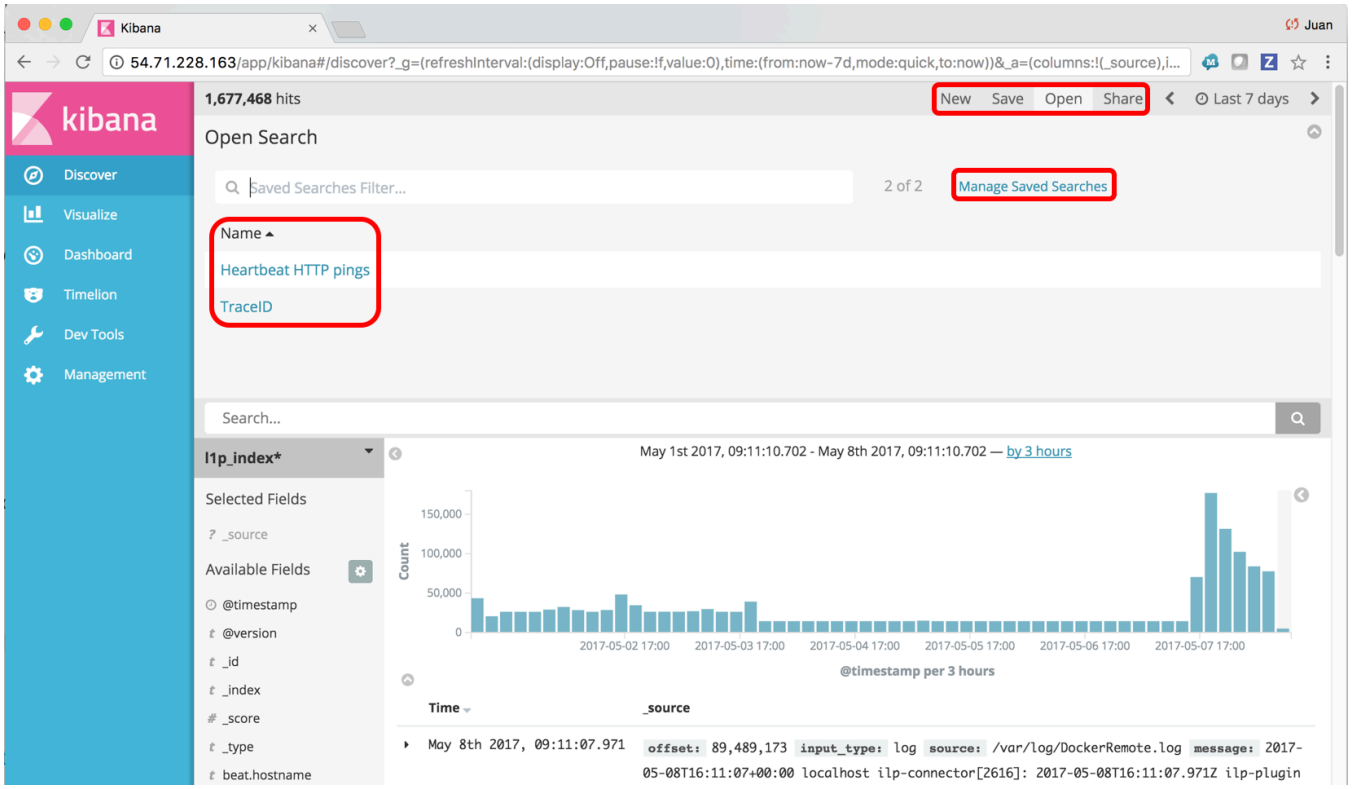
Examples of common L1P Kibana Queries

Sample Query	Description
l1p_trace_id:"47eb4790-0ccd-487a-9f05-f239ecf5d8a2"	Search for a specific L1P-Trace-Id
"89969d15-582a-4fee-9d2a-0dc732acc130"	Search for a specific L1P-Trace-Id
exists:l1p_trace_id	Search for log entries that contain a l1p_trace_id value
_index:"l1p_index_2017.04.18"	Search for a specific Index
l1p_trace_id:"1614cfa4-e792-4b01-9537-2f3eb8001b5e" **AND** _index:"l1p_index_2017.06.28"	Search for a specific L1P-Trace-Id **AND** inside a specific index

Quering behaviour and rules in Kiabana:

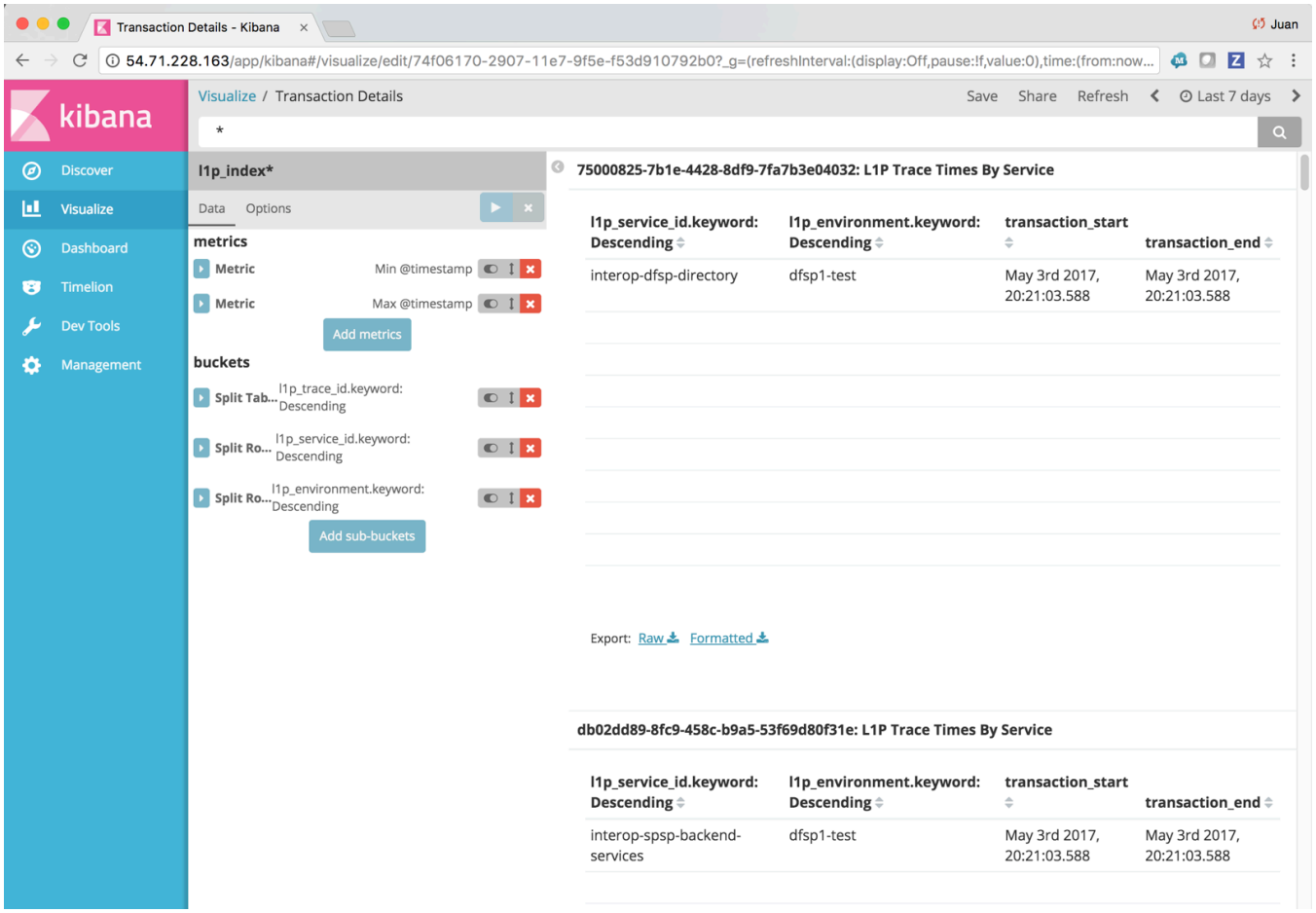
- query behaves as unstructured text search, with some special commands, and if you get the command syntax wrong it just does an unstructured text search
- by default it searches for entries containing any or your search terms
- hyphen is consider as a delimiter
- in order to search for a string literal use double quotes NOT single quotes (single quotes are ignored)
- to search for a single filed enter field name, then a colon and then the value within double quotes
- exists and missing are examples of commands that can be used, e.g exists:exception
- AND and OR are case-senditive, must use upper case
- can use parenthesis when searching for several things, e.g. exists:exception AND (l1p_trace_id:"1614cfa4-e792-4b01-9537-2f3eb8001b5e" OR _payment_id:"1614cfa4-e792-4b01-9537-2f3eb8001b5e"
- field names are also case-sensitive, examples of fields are l1p_trace_id and index

How to save a search, how to open a saved search and how to manage saved searches?



Kibana allows you to save a search criteria. From the Kibana Discover page just hit the "Save" link on the top right hand corner just before the "Time Picker" icon to save your search. To access your saved search, hit the "Open" link. To delete or edit your save search hit the "Open" link and then the "Manage Saved Searches" link.

How to monitor transaction duration between l1p_components?



A Kibana Visualization on top of the custom l1p_index was created for this purpose and can be accessed by navigating to Kibana Visualize page and selecting the "Transaction Details" link. NOTE: Once the trace id is contained at all component and services logs this table will correctly display the durations.

Reference to Kibana's Official Documentation

https://www.elastic.co/guide/en/kibana/current/introduction.html

Appendix

Terminology

These are the preferred terms and definitions for Mojaloop.

Details

Term	Alternative and Related Terms	Mojaloop Definition
Access Point	POS ("Point of Sale"), Customer Access Point, ATM, Branch	Places or capabilities that are used to initiate or receive a payment. Access points can include bank branch offices, ATMs, terminals at the POS, agent outlets, mobile phones, and computers.
Account Lookup System		Account Lookup System is an abstract entity used for retrieving information regarding in which FSP an account, wallet or identity is hosted. The Account Lookup System itself can be hosted in its own server, as part of a financial switch, or in FSPs.
Account Number	Account ID	A unique number representing an account. There can be multiple accounts for each end user.
Active User		A term used by many providers in describing how many of their account holders are frequent users of their service.
Addressing	Directories, Aliasing	The use of necessary information (account number, phone number, etc.) for a paying user to direct payment to a receiving user.
Agent	Agent till, Agent outlet	An entity authorized by the provider to handle various functions such as customer enrollment, cash-in and cash-out using an agent till.
Agent Outlet	Access point	A physical location that carries one or more agent tills, enabling it to perform

Term	Alternative and Related Terms	Mojaloop Definition
		enrollment, cash-in and cash-out transactions for customers on behalf of one or more providers. National law defines whether an agent outlet may remain exclusive to one provider. Agent outlets may have other businesses and support functions.
Agent Till	Registered agent	An agent till is a provider-issued registered "line", either a special SIM card or a POS machine, used to perform enrollment, cash-in and cash-out transactions for clients. National law dictates which financial service providers can issue agent tills.
Aggregator	Merchant Aggregator	A specialized form of a merchant services provider who typically handles payments transactions for a large number of small merchants. Scheme rules often specify what aggregators are allowed to do.
Anti Money Laundering	AML; also "Combating the Financing of Terrorism", or CFT	Initiatives to detect and stop the use of financial systems to disguise use of funds criminally obtained.
Application Program Interface	API	A software program that makes it possible for application programs to interact with each other and share data.
Arbitration		The use of an arbitrator, rather than courts, to resolve disputes.
Authentication	Verification, Validation	The process of ensuring that a person or a transaction is valid for the process (account opening, transaction initiation, etc.) being peformed.
Authorization		A process used during a "pull" payment (such as a card payment), when the payee requests (through their provider) confirmation from the payer's bank that the transaction is good.
Automated Clearing House		An electronic clearing system in which payment orders are exchanged among payment service providers, primarily via magnetic media or telecommunications networks, and then cleared amongst the participants. All operations are handled by a data processing center. An ACH typically clears credit transfers and debit transfers, and in some cases also cheques.
Bank	Savings Bank, Credit Union, Payments Bank	A charted financial system within a country that has the ability to accept deposits and make and receive payments into those accounts.
Bank Accounts and Transaction Services	Mobile Banking, Remote Banking, Digital	A transaction account held at a bank. This account may be accessible by a mobile phone, in which case it is sometimes referred to as "mobile banking".

Term	Alternative and Related Terms	Mojaloop Definition
	Banking	
Bank-Led Model	Bank-Centric Model	A reference to a system in which banks are the primary providers of digital financial services to end users. National law may require this.
Basic Phone		Minimum device required for DFS
Bilateral Net Settlement System		A settlement system in which participants' bilateral net settlement positions are settled between every bilateral combination of participants.
Bilateral Netting		An arrangement between two parties to net their bilateral obligations. The obligations covered by the arrangement may arise from financial contracts, transfers or both.
Bill Payment	C2B, Utility payments, school payments	Making a payment for a recurring service, either in person ("face to face") or remotely.
Biometric Authentication		The use of a physical characteristic of a person (fingerprint, IRIS, etc.) to authenticate that person.
Blacklist		A list or register of entities (registered users) that are being denied/blocked from a particular privilege, service, mobility, access or recognition. Entities on the list will NOT be accepted, approved and/or recognized. It is the practice of identifying entities that are denied, unrecognized, or ostracized. Where entities are registered users (or user accounts, if granularity allows) and services are informational (e.g. balance check), transactional (e.g. debit/ credit) payments services or lifecycle (e.g. registration, closure) services.
Blockchain	Digital currency, cryptocurrency, distributed ledger technology	The technology underlying bitcoin and other cryptocurrencies-a shared digital ledger, or a continually updated list of all transactions.
Borrowing		Borrowing money to finance a short term or long term need
Bulk Payer		An organization (or rarely, an individual), that needs to pay to many users at once.
Bulk Payments	G2C, B2C , G2P, social transfers	Making and receiving payments from a government to a consumer: benefits, cash transfers, salaries, pensions, etc.
Bulk Payments Services		A service that allows a government agency or an enterprise to make payments to a large number of payees - typically consumers but can be

Term	Alternative and Related Terms	Mojaloop Definition
		businesses as well.
Bulk upload service		A service enabling the import of multiple transactions per session, most often via a bulk data transfer file which is used to initiate payments. Example: salary payment file.
Bundling	Packaging, Tying	A business model in which a provider which groups a collection of services into one product which an end user agrees to buy or use.
Business		Entity such as a public limited or limited company or corporation that uses mobile money as a service, e.g. taking bill payments, making bill payments and disbursing salaries
Cash Management	Agent Liquidity Management	Management of cash balances at an agent.
Cash-In		Receiving eMoney credit in exchange for physical cash - typically done at an agent.
Cash-Out		Receiving physical cash in exchange for a debit to an eMoney account - typically done at an agent.
Chip Card	EMV Chip Card, Contactless Chip Card	A chip card contains a computer chip: it may be either contactless or contact (requires insertion into terminal). Global standards for chip cards are set by EMV.
CICO		Cash In Cash Out
Clearing		The process of transmitting, reconciling, and, in some cases, confirming transactions prior to settlement, potentially including the netting of transactions and the establishment of final positions for settlement. Sometimes this term is also used (imprecisely) to cover settlement. For the clearing of futures and options, this term also refers to the daily balancing of profits and losses and the daily calculation of collateral requirements.
Clearing House		A central location or central processing mechanism through which financial institutions agree to exchange payment instructions or other financial obligations (e.g. securities). The institutions settle for items exchanged at a designated time based on the rules and procedures of the clearinghouse. In some cases, the clearinghouse may assume significant counterparty, financial, or risk management responsibilities for the clearing system.
Closed-Loop		A payment system used by a single provider, or a very tightly constrained group of providers.
Combatting	CFT (Counter	Initiatives to detect and stop the use of financial systems to transfer funds to

Term	Alternative and Related Terms	Mojaloop Definition
Terrorist Financing	Financing of Terrorism)	terrorist organizations or people.
Commission		An incentive payment made, typically to an agent or other intermediary who acts on behalf of a DFS provider. Provides an incentive for agent.
Commit		Commit means that the electronic funds that were earlier reserved are now moved to the final state of the financial transaction. The financial transaction is completed. The electronic funds are no longer locked for usage.
Counterparty	Payee, payer, borrower, lender	The other side of a payment or credit transaction. A payee is the counterparty to a payer, and vice-versa.
Coupon		A token that entitles the holder to a discount or that may be exchanged for goods or services
Credit History	Credit bureaus, credit files	A set of records kept for an end user reflecting their use of credit, including borrowing and repayment.
Credit Risk Management		Tools to manage the risk that a borrower or counterparty will fail to meet its obligations in accordance with agreed terms.
Credit Scoring		A process which creates a numerical score reflecting credit worthiness.
Cross Border Trade Finance Services		Services which enable one business to sell or buy to businesses or individuals in other countries; may include management of payments transactions, data handling, and financing.
Cross-FX Transfer		Transfer involving multiple currencies including a foreign exchange calculation
Customer Database Management		The practices that providers do to manage customer data: this may be enabled by the payment platform the provider is using.
Data Protection	PCI-DSS	The practices that enterprises do to protect end user data. "PCI-DSS" is a card industry standard for this.
Deposit Guarantee System	Deposit Insurance	A fund that insures the deposits of account holders at a provider; often a government function used specifically for bank accounts.
DFSP On-boarding		On-boarding a DFSP is the process of adding a new DFSP to this financial network.
Digital Financial		The regulated entity providing digital financial services to users. Manages the wallet for the users. May manage other types of digital assets such as savings

Term	Alternative and Related Terms	Mojaloop Definition
Service Provider (DFSP)		accounts, loans etc. Depending on countries and regulations, a DFSP can be a bank, a telco, a Mobile Money Operator or some other private entity.
Digital Financial Services	Mobile Financial Services	Digital financial services include methods to electronically store and transfer funds; to make and receive payments; to borrow, save, insure and invest; and to manage a person's or enterprise's finances.
Digital Liquidity		A state in which a consumer willing to leave funds (eMoney or bank deposits) in electronic form, rather than performing a "cash-out".
Digital Payment	Mobile Payment, Electronic Funds Transfer	A broad term including any payment which is executed electronically. Includes payments which are initiated by mobile phone or computer. Card payments in some circumstances are considered to be digital payments. The term "mobile payment" is equally broad, and includes a wide variety of transaction types which in some way use a mobile phone.
Dispute Resolution		A process specified by a provider or by the rules of a payment scheme to resolve issues between end users and providers, or between an end user and its counter party.
Domestic Remittance	P2P; Remote Domestic Transfer of Value	Making and receiving payments to another person in the same country.
Electronic Invoicing, ERP, Digital Accounting, Supply Chain Solutions Services, Business Intelligence		Services that support merchant or business functions relating to DFS services.
eMoney	eFloat, Float, Mobile Money, Electronic Money, Prepaid Cards	A record of funds or value available to a consumer stored on a payment device such as chip, prepaid cards, mobile phones or on computer systems as a non-traditional account with a banking or non-banking entity.
eMoney Accounts and Transaction	Digital Wallet, Mobile Wallet, Mobile Money	A transaction account held at a non-bank. The value in such an account is refrred to as eMoney.

Term	Alternative and Related Terms	Mojaloop Definition
Services	Account	
eMoney Issuer	Issuer, Provider	A provider (bank or non-bank) who deposits eMoney into an account they establish for an end user. eMoney can be created when the provider receives cash ("cash-in") from the end user (typically at an agent location) or when the provider receives a digital payment from another provider.
Encryption	Decryption	The process of encoding a message so that it can be read only by the sender and the intended recipient.
End User	Consumer, Customer, Merchant, Biller	The customer of a digital financial services provider: the customer may be a consumer, a merchant, a government, or another form of enterprise.
Escrow	Funds Isolation, Funds Safeguarding, Custodian Account, Trust Account.	A means of holding funds for the benefit of another party. eMoney Issuers are usually required by law to hold the value of end users' eMoney accounts at a bank, typically in a Trust Account. This accomplishes the goals of funds isolation and funds safeguarding.
External Account		An account hosted outside the FSP, regularly accessible by an external provider interface API.
FATF		The Financial Action Task Force is an intergovernmental organization to combat money laundering and to act on terrorism financing.
Feature Phone		A mobile telephone without significant computational capabilities.
Fees		The payments assessed by a provider to their end user. This may either be a fixed fee, a percent-of-value fee, or a mixture. A Merchant Discount Fee is a fee charged by a Merchant Services Provider to a merchant for payments acceptance. Payments systems or schemes, as well as processors, also charge fees to their customer (typically the provider.)
Financial Inclusion		The sustainable provision of affordable digital financial services that bring the poor into the formal economy.
Financial Literacy		Consumers and businesses having essential financial skills, such as preparing a family budget or an understanding of concepts such as the time value of money, the use of a DFS product or service, or the ability to apply for such a service.
FinTech		A term that refers to the companies providing software, services, and products for digital financial services: often used in reference to newer technologies.

Term	Alternative and Related Terms	Mojaloop Definition
Float		This term can mean a variety of different things. In banking, float is created when one party's account is debited or credited at a different time than the counterparty to the transaction. eMoney, as an obligation of a non-bank provider, is sometimes referred to as float.
Fraud	Fraud Management, Fraud Detection, Fraud Prevention	Criminal use of digital financial services to take funds from another individual or business, or to damage that party in some other way.
Fraud Risk Management	Also know as fraud risk management service (FRMS)	Tools to manage providers' risks, and at times user's risks (e.g. for merchants or governments) in providing and/or using DFS services.
FX		Foreign Exchange.
Government Payments Acceptance Services		Services which enable governments to collect taxes and fees from individuals and businesses.
HCE		A communication technology that enables payment data to be safely stored without using the Secure Element in the phone.
Identity	National Identity, Financial Identity, Digital Identity	A credential of some sort that identifies an end user. National identities are issued by national governments. In some countries a financial identity is issued by financial service providers.
Immediate Funds Transfer	Real Time	A digital payment which is received by the payee almost immediately upon the payer having initiated the transaction.
Insurance Products		A variety of products which allow end user to insure assets or lives that they wish to protect.
Insuring Lives or assets		Paying to protect the value of a life or an asset.
Interchange	Swipe Fee, Merchand Discount Fee	A structure within some payments schemes which requires one provider to pay the other provider a fee on certain transactions. Typically used in card schemes to effect payment of a fee from a merchant to a consumer's card issuing bank.

Term	Alternative and Related Terms	Mojaloop Definition
International Remittance	P2P; Remote Cross-border Transfer of Value, Cross-Border Remittance	Making and receiving payments to another person in another country.
Interoperability	Interconnectivity	When payment systems are interoperable, they allow two or more proprietary platforms or even different products to interact seamlessly. The result is the ability to exchange payments transactions between and among providers. This can be done by providers participating in a scheme, or by a variety of bilateral or multilateral arrangements. Both technical and business rules issues need to be resolved for interoperability to work.
Interoperability settlement bank		Entity that facilitates the exchange of funds between the FSPs. The settlement bank is one of the main entities involved in any inter-FSP transactions.
Investment Products		A variety of products which allow end users to put funds into investments other than a savings account.
Irrevocable	Non-Repudiation	A transaction that cannot be "called back" by the payer; an irrevocable payment, once received by a payee, cannot be taken back by the payer.
Interoperability service for transfer	IST	Inter system trunk that allows for routing of payments.
Know Your Customer	KYC, Agent and Customer Due Diligence, Tiered KYC, Zero Tier	The process of identifying a new customer at the time of account opening, in compliance with law and regulation. The identification requirements may be lower for low value accounts ("Tiered KYC"). The term is also used in connection with regulatory requirements for a provider to understand, on an ongoing basis, who their customer is and how they are using their account.
L1P Bulk Payment Facilitator		An organization that processes L1P compliant payments and resulting reports on behalf of Bulk Payers.
Liability	Agent Liability, Issuer Liability, Acquirer Liability	A legal obligation of one party to another; required by either national law, payment scheme rules, or specific agreements by providers. Some scheme rules transfer liabilities for a transaction from one provider to another under certain conditions.
Liquidity	Agent liquidity	The availability of liquid assets to support an obligation. Banks and non-bank providers need liquidity to meet their obligations. Agents need liquidity to

Term	Alternative and Related Terms	Mojaloop Definition
		meet cash-out transactions by consumers and small merchants.
Loans	Microfinance, P2P Lending, Factoring, Cash Advances, Credit, Overdraft, Facility	Means by which end users can borrow money.
M2C		Merchant to Customer or Consumer.
mCommerce	eCommerce	Refers to buying or selling in a remote fashion: by phone or tablet (mCommerce) or by computer (eCommerce)
Merchant	Payments Acceptor	An enterprise which sells goods or services and receives payments for such goods or services.
Merchant Acquisition	Onboarding	The process of enabling a merchant for the receipt of electronic payments.
Merchant payment - POS	C2B, Proximity Payments	Making a payment for a good or service in person ("face to face"); includes kiosks and vending machines.
Merchant payment - Remote	C2b, eCommerce Payment, Mobile Payment	Making a payment for a good or service remotely; transacting by phone, computer, etc.
Merchant Payments Acceptance Services	Acquiring services	A service which enables a merchant or other payment acceptor to accept one or more types of electronic payments. The term "acquiring" is typically used in the card payments systems.
Merchant Service Provider	Acquirer	A provider (bank or non-bank) who supports merchants or other payments acceptors requirements to receive payments from customers. The term "acquirer" is used specifically in connection with acceptance of card payments transactions.
MFSP Platform		Mobile financial service providers
Mobile Network Operator		An enterprise which sells mobile phone services, including voice and data communication.
Money Transfer Operator		A specialized provider of DFS who handles domestic and/or international remittances.

Term	Alternative and Related Terms	Mojaloop Definition
Multilateral Net Settlement Position		The sum of the value of all the transfers a participant in a net settlement system has received during a certain period of time less the value of the transfers made by the participant to all other participants. If the sum is positive, the participant is in a multilateral net credit position; if the sum is negative, the participant is in a multilateral net debit position.
Multilateral Net Settlement System		A settlement system in which each settling participant settles (typically by means of a single payment or receipt) the multilateral net settlement position which results from the transfers made and received by it, for its own account and on behalf of its customers or non-settling participants for which it is acting.
Multilateral Netting		Netting on a multilateral basis is arithmetically achieved by summing each participant's bilateral net positions with the other participants to arrive at a multilateral net position. Such netting is conducted through a central counterparty (such as a clearing house) that is legally substituted as the buyer to every seller and the seller to every buyer. The multilateral net position represents the bilateral net position between each participant and the central counterparty.
NDFSP		national digital financial service providers
Near Field Communication	NFC	A communication technology used within payments to transmit payment data from an NFC equipped mobile phone to a capable terminal.
Netting		The offsetting of obligations between or among participants in the settlement arrangement, thereby reducing the number and value of payments or deliveries needed to settle a set of transactions.
Non Bank-Led Model	MNO-Led Model	A reference to a system in which non-banks are the providers of digital financial services to end users. Non-banks typically need to meet criteria established by national law and enforced by regulators.
Non-Bank	Payments Institution, Alternative Lender	An entity that is not a chartered bank, but which is providing financial services to end users. The requirements of non-banks to do this, and the limitations of what they can do, are specified by national law.
Nostro Account		From the Payer's perspective: Payer FSP funds/accounts held/hosted at Payee FSP
Notification		Notice to payer or payee regarding the status of a transfer.
Off-Us Payments	Off-net payments	Payments made in a multiple-participant system or scheme, where the payer's provider is a different entity as the payee's provider.

Term	Alternative and Related Terms	Mojaloop Definition
On-Us Payments	On-net payments	Payments made in a multiple-participant system or scheme, where the payer's provider is the same entity as the payee's provider.
Open-Loop		A payment system or scheme designed for multiple providers to participate in. Payment system rules or national law may restrict participation to certain classees of providers.
Operations Risk Management		Tools to manage providers' risks in operating a DFS system.
Organization		Non-business An entity such as a business, charity or government department that uses mobile money as a service, e.g. taking bill payments, making bill payments and disbursing salaries
Over The Counter Services	OTC, Mobile to Cash	Services provided by agents when one end party does not have an eMoney account: the (remote) payer may pay the eMoney to the agent's account, who then pays cash to the non- account holding payee.
Participant		A provider who is a member of a payment scheme, and subject to that scheme's rules.
Partner Bank		Financial institution supporting the FSP and giving it access to the local banking ecosystem.
Payee	Receiver	The receipient of funds in a payment transaction.
Payee FSP		Payer's financial service providers.
Payer	Sender	The payer of funds in a payment transaction.
Payer FSP		Payee's financial service providers.
Paying for Purchases	C2B - Consumer to Business	Making payments from a consumer to a business: the business is the "payment acceptor" or merchant.
Payment System	Payment Network, Money Transfer System	Encompasses all payment-related activities, processes, mechanisms, infrastructure, institutions and users in a country or a broader region (eg a common economic area).
Payment System Operator	Mobile Money Operator, Payment Service Provider	The entity that operates a payment system or scheme.
Peer FSP		The counterparty Mobile Money Provider's Platform financial service

Term	Alternative and Related Terms	Mojaloop Definition
Mobile Money Platform		provider.
PEP		Politically Exposed Person. Someone who has been entrusted with a prominent public function. A PEP generally presents a higher risk for potential involvement in bribery and corruption by virtue of their position and the influence that they may hold (e.g. 'senior foreign political figure', 'senior political figure', foreign official', etc.).
Phone Number		Non identifying number associated with one or more end users as contact information for the end user. These numbers use the E.164 standard. Phone numbers are not required as a user number, though they can be used that way if a government or DFSP insists.
Platform	Payment Platform, Payment Platform Provider	A term used to describe the software or service used by a provider, a scheme, or a switch to manage end user accounts and to send and receive payment transactions.
Point of Sale Device	Terminal, Acceptance Device, POS, mPOS	Any device meant specifically for managing the receipt of electronic payments.
Posting	Clearing	The act of the provider of entering a debit or credit entry into the end user's account record.
Prefunding		The process of adding funds to Vostro/Nostro accounts.
Prepaid Cards		eMoney product for general purpose use where the record of funds is stored on the payment card (on magnetic stripe or the embedded integrated circuit chip) or a central computer system, and which can be drawn down through specific payment instructions to be issued from the bearer's payment card.
Processor	Gateway	An enterprise that manages, on an out-sourced basis, various functions for a digital financial services provider. These functions may include transaction management, customer database management, and risk management. Processors may also do functions on behalf of payments systems, schemes, or switches.
Promotion		FSP marketing initiative offering the user a transaction/service fee discount on goods or services. May be implemented through the use of a coupon.
Provider	Financial Service	The entity that provides a digital financial service to an end user (either a consumer, a business, or a government.) In a closed-loop payment system,

Term	Alternative and Related Terms	Mojaloop Definition
	Provider, Payment Service Provider, Digital Financial Services Provider	the Payment System Operator is also the provider. In an open-loop payment system, the providers are the banks or non-banks which participate in that system.
Pull Payments		A payment type which is initiated by the payee: typically a merchant or payment acceptor, whose provider "pulls" the funds out of the payer's account at the payer's provider.
Push Payments		A payment type which is initiated by the payer, who instructs their provider to debit their account and "push" the funds to the receiving payee at the payee's provider.
Quoting		The process a DFSP uses to ask for the fees and ILP packet from the destination
Reconcilation		Cross FSP Reconciliation is the process of ensuring that two sets of records, usually the balances of two accounts, are in agreement between FSPs. Reconciliation is used to ensure that the money leaving an account matches the actual money transferred. This is done by making sure the balances match at the end of a particular accounting period.
Recourse		Rights given to an end user by law, private operating rules, or specific agreements by providers, allowing end users the ability to do certain things (sometimes revoking a transaction) in certain circumstances.
Refund		A repayment of a sum of money.
Registration	Enrollment, Agent Registration	The process of opening a provider account. Separate processes are used for consumers, merchants agents, etc.
Regulator		A governmental organization given power through national law to set and enforce standards and practices. Central Banks, Finance and Treasury Departments, Telecommunications Regulators, and Consumer Protection Authorities are all regulators involved in digital financial services.
Reservation		Part of a 2-phase transfer operation in which the funds to be transferred are 'segregated' (i.e. made unusable) for a predetermined duration, commonly governed by a timeout period, to any other transfer attempts.
Reversal		The process of reversing a completed transfer.
Risk	Fraud	The practices that enterprises do to understand, detect, prevent, and manage

Term	Alternative and Related Terms	Mojaloop Definition
Management	Management	various types of risks. Risk management occurs at providers, at payments systems and schemes, at processors, and at many merchants or payments acceptors.
Risk-based Approach		A regulatory and/or business management approach that creates different levels of obligation based on the risk of the underlying transaction or customer.
Rollback		The process of reversing a completed transfer.
RTGS		Real time gross settlement
Rules		The private operating rules of a payments scheme, which bind the direct participants (either providers, in an open-loop system, or end users, in a closed-loop system).
Saving and Investing		Keeping funds for future needs and financial return
Savings Products		An account at either a bank or non-bank provider, which stores funds with the design of helping end users save money.
Scheme		A set of rules, practices and standards necessary for the functioning of payment services.
Secure Element		A secure chip on a phone that can be used to store payment data.
Security Level		Security specification of the system which defines effectiveness of risk protection.
Sending or Receiving Funds		Making and receiving payments to another person
Settlement		An act that discharges obligations in respect of funds or securities transfers between two or more parties.
Settlement System	Net Settlement, Gross Settlement, RTGS	A system used to facilitate the settlement of transfers of funds, assets or financial instruments. Net settlement system: a funds or securities transfer system which settles net settlement positions during one or more discrete periods, usually at pre-specified times in the course of the business day. Gross settlement system: a transfer system in which transfer orders are settled one by one.
Short Message Service		A service for sending short messages between mobile phones.

Term	Alternative and Related Terms	Mojaloop Definition
SIM Card	SIM ToolKit, Thin SIM	A smart card inside a cellular phone, carrying an identification number unique to the owner, storing personal data, and preventing operation if removed. A SIM Tool Kit is a standard of the GSM system which enables various value-added services. A "Thin SIM" is an additional SIM card put in a mobile phone.
Smart Phone		A device that combines a mobile phone with a computer.
Standards Body	EMV, ISO, ITU, ANSI, GSMA	An organization that creates standards used by providers, payments schemes, and payments systems.
Storing Funds	Account, Wallet	Keeping funds in secure electronic format. May be a bank account or an eMoney account.
Super Agent	Master agent	In some countries, agents are managed by Super Agents or Master Agents who are responsible for the actions of their agents to the provider.
Supplier Payment	B2B - Business to Business, B2G - Business to Government	Making a payment from one business to another for supplies, etc: may be in-person or remote, domestic or cross border. Includes cross-border trade.
SVA (Stored Value Account)		Accounts in which funds are kept in a secure, electronice format.
Switch		An entity which receives transactions from one provider and routes those transactions on to another provider. A switch may be owned or hired by a scheme, or be hired by individual providers. A switch will connect to a settlement system for inter-participant settlement.
Systemic Risk		In payments systems, the risk of collapse of an entire financial system or entire market, as opposed to risk associated with any one individual provider or end user.
Tax Payment	C2G, B2G	Making a payment from a consumer to a government, for taxes, fees, etc.
Tokenization		The use of subsitute a token ("dummy numbers") in lieu of "real" numbers, to protect against the theft and misuse of the "real" numbers. Requires a capability to map the token to the "real" number.
Trading	International Trade	The exchange of capital, goods, and services across international borders or territories
Transaction Accounts		Transaction account is broadly defined as an account held with a bank or other authorized and/or regulated service provider (including a non-bank) which can be used to make and receive payments. Transaction accounts can be further differentiated into deposit transaction accounts and eMoney

Term	Alternative and Related Terms	Mojaloop Definition
		accounts. Deposit transaction account is a deposit account held with banks and other authorised deposit-taking financial institutions that can be used for making and receiving payments. Such accounts are known in some countries as current accounts, chequing accounts or other similar terms.
Transaction Cost		The cost to a DFS provider of delivering a digital financial service. This coudl be for a bundle of services (e.g. a "wallet") or for individual transactions.
Transfer		A general term for sending money. Local transfer refers to sending money within a ledger or DFSP, interledger transfers go between ledgers or DFSPs.
Trusted Execution Environment		An development execution environment that has security capabilities and meets certain security-related requirements.
Ubiquity		The ability of a payer to reach any (or most) payees in their country, regardless of the provider affiliation of the receiving payee. Requires some type of interoperability.
Unbanked	Underbanked, Underserved	Unbanked people do not have a transaction account. Underbanked people may have a transaction account but do not actively use it. Underserved is a broad term referring to people who are the targets of financial inclusion initiatives. It is also sometimes used to refer to a person who has a transaction account but does not have additional DFS services.
User Creation		A process for creating an individual user in the system.
User Number	User ID	A number that identifies an end user. We assume it uses E.164 format. Depending on the country and DFSP, this can be a phone number, some form of DFSP provided ID or some form of national ID. The DFSPs associates the number to a phone number and a primary account number. Money is generally sent to the user number, and not directly to the account.
User On-boarding		A process for creating an individual user in the system and all additional related actions such as creation of user PIN, creation of user account, KYC data capture (photo, fingerprints), etc.
USSD		A communication technology that is used to send text between a mobile phone and an application program in the network.
Voucher		A token that entitles the holder to a discount or that may be exchanged for goods or services.
Wallet		Repository of funds for an account.
Whitelist		A list or register of entities (registered users) that are being provided a

Term	Alternative and Related Terms	Mojaloop Definition
		particular privilege, service, mobility, access or recognition, especially those that were initially blacklisted. Entities on the list will be accepted, approved and/or recognized. Whitelisting is the reverse of blacklisting, the practice of identifying entities that are denied, unrecognized, or ostracized. Where entities are registered users (or user accounts, if granularity allows) and services are informational (e.g. balance check), transactional (e.g. debit/ credit) payments services or lifecycle (e.g. registration, closure) services.
'x'-initiated		Used when referring to the side that initiated a transaction, e.g. agent-initiated cash-out vs. user-initiated cash-out"

Evolution of Mojaloop

Here we document the reasoning behind certain tools, technology and process choices for Mojaloop.

- **Open source** - the entire project may be made open source in accordance with the level one principles. All tools and processes must be open source friendly and support an Apache 2.0 license and no restrictive licenses.
- **Agile development** - The requirements need to be refined as the project is developed, therefore we picked agile development over waterfall or lean.
- **Scaled Agile Framework** - there are four initial development teams that are geographically separate. To keep the initial phase of the project on track, the scaled agile framework (SAFe) was picked. This means work is divided into program increments (PI) that are typically four 2 week sprints long. As with the sprints, the PI has demo-able objective goals defined in each PI meeting.
- **Threat Modeling, Resilience Modeling, and Health Modeling** - because this code needs to exchange money in an environment with very flaky infrastructure it must have good security, resilience, and easily report it's health state and automatically attempt to return to it. To achieve this we employ basic tried and true modeling practices.
- **Automated Testing** - for the most part, most testing will be automated to allow for easy regression. See the automated testing strategy.
- **Microservices** - Because the architecture needs to easily deploy, scale, and have components be easily replaced or upgraded it will be built as a set of microservices.
- **APIs** - In order to avoid confusion from too many changing microservices, we use strongly defined APIs. APIs will be defined using OpenAPI or RAML. Teams document their APIs with Swagger v2.0 or RAML v0.8 so they can automatically test, document, and share their work. Swagger is slightly preferred as there are free tools. Mule will make use of RAML 0.8. Swagger can be automatically converted to RAML v0.8, or manually to RAML v1.0 if additional readability is desired.
- **Services** - Microservices are grouped and deployed in a few services such as the DFSP, Central Directory, etc. Each of these will have simple defined interfaces, configuration scripts, tests, and documentation.
- **Database Storage** - although Microsoft SQL Server is widely used in Africa, we need a SQL backend that is open source friendly and can scale in a production environment. Thus, we chose PostgreSQL. The database is called through an adapter and the stored procedures are kept in simple ANSI SQL so that it can be replaced later with little trouble.
- **USSD** - Smart phones are only 25% of the target market and not currently supported by most money transfer service, so we need a protocol that will work on simple feature phones. Like M-Pesa, we are using USSD between the phone and the digital financial service provider (DFSP).

- **Operating System** - Again, Microsoft Windows is widely used in many target countries, but we need an operating system that is free of license fees and is open source compatible. We are using Linux. We don't have a dependency on the particular flavor, but are using the basic Amazon Linux. In the Docker containers, Alpine Linux is used.
- **Interledger** - The project needed a lightweight, open, and secure transport protocol for funds. Interledger.org provides all that. It also provides the ability to connect to other systems. We also considered block chain systems, but block chain systems send very large messages which will be harder to guarantee delivery of in third world infrastructure. Also, while blockchain systems provide good anonymity, that is not a project goal. To enable fraud detection, regulatory authorities need to be able to request records of transfers by account and person.
- **MuleSoft** - For the most part, the mule server is simple a host and pass through for Level One Client API calls. However, it will be necessary to deploy Mojaloop system into existing financial providers. MuleSoft provides an excellent adapter so that the APIs can be easily hooked up to existing systems while providing cross cutting concerns like logging, fault tolerance, and security. The core pieces used don't require license fees.
- **NodeJS** - NodeJS is designed to create simple microservices and it has a huge set of open source libraries available. Node performance is fine and while Node components don't scale vertically a great deal, we plan to scale horizontally, which it does fine. The original Interledger code was written in NodeJS as was the level one prototype this code is based on. Most teams used Node already, so this made sense as a language.
- **NodeJS "Standard"** - Within NodeJS code, we use Standard as a code style guide and to enforce code style.
- **Java** - Mule can't run NodeJS directly, so some adapters to mule and interop pieces are written in Java. This is a very small part of the overall code.
- **Checkstyle** - Within Java code, we use Checkstyle as a code style guide and style enforcement tool.
- **GitHub** - GitHub is the standard source control for open source projects so this decision was straightforward. We create a story every time for integration work. Create bugs for any issues. Ensure all stories are tracked throughout the pipeline to ensure reliable awx.
- **Slack** - Slack is used for internal team communication. This was largely picked because several team already used it and liked it as a lightweight approach compared to email.
- **ZenHub** - We needed a project management solution that was very light weight and cloud based to support distributed teams. It had to support epics, stories, and bugs and a basic project board. VS and Jira online offerings were both considered. For a small distributed development team an online service was better. For an open source project, we didn't want ongoing maintenance costs of a server. Direct and strong GitHub integration was important. It was very useful to track work for each microservice with that microservice. Jira and VS both have more overhead than necessary for a project this size and don't integrate as cleanly with GitHub as we'd want. ZenHub allowed us to start work immediately. A disadvantage is the lack of support for cumulative flow diagrams and support for tracking # of stories instead of points, so we do these manually with a spreadsheet updated daily and the results published to the "Project Management" Slack channel (Cumulutative flow is being added to Zenhub, but wasn't available for most of the project).

- **AWS** - We needed a simple hosting service for our Linux instances, and we aren't going to use many of the extra services like geo-redundancy, since we expect customers may wish to self-host. AWS is an industry standard and works well. We considered Azure, which also would have worked, but it's harder for the Gates Foundation to get an Azure subscription than AWS.
- **Docker** - the project needs to support both local and cloud execution. We have many small microservices that have very simple specific configurations and requirements. The easiest way to guarantee that the service works the same way in every environment from local development, to cloud, to hosted production is to put each microservice in a Docker container along with all the prerequisites it needs to run. The container becomes a secure, closed, pre-configured, runnable unit.
- **CircleCI** - to get started quickly we needed an online continuous build and testing system that can work with many small projects and a distributed team. Jenkins was considered, but it requires hosting a server and a lot of configuration. CircleCI allowed for a no host solution that could be started with no cost and very limited configuration. We thought we might start with CircleCI and move off later if we outgrew it, but that hasn't been needed.
- **Artifactory** - After the build we need private repository to put our NodeJS packages and Docker containers until they are formally published. Docker and AWS both do this, and any solution would work. We chose Artifactory from JFrog simply because one team already had an account with it and had it setup.
- **SonarQube** - We need an online dashboard of code quality (size, complexity, issues, and coverage) that can aggregate the code from all the repos. We looked at several online services (CodeCov, Coveralls, and Code Climate), but most couldn't do complexity or even number of lines of code. Code Climate has limited complexity (through ESLint), but costs 6.67/seat/month. SonarQube is free, though it required us to setup and maintain our own server. It gave the P1 features we wanted.
- **Markdown** - Documentation is a deliverable for this project, just like the code, and so we want to treat it like the code in terms of versioning, review, check in, and tracking changes. We also want the documentation to be easily viewable online without constantly opening a viewer. GitHub has a built-in format called Markdown which solves this well. The same files work for the Wiki and the documents. They can be reviewed with the check in using the same tools and viewed directly in GitHub. We considered Google Docs, Word and PDF, but these binary formats aren't easily diff-able. A disadvantage is that markdown only allows simple formatting - no complex tables or font changes - but this should be fine when our main purpose is clarity.
- **Draw.io** - We need to create pictures for our documents and architecture diagrams using an (ideally free) open source friendly tool, that is platform agnostic, supports vector and raster formats, allows WYSIWYG drawing, works with markdown, and is easy to use. We looked at many tools including: Visio, Mermaid, PlantUML, Sketchboard.io, LucidChart, Cacoo, Archi, and Google Drawings. Draw.io scored at the top for our needs. It's free, maintained, easy to use, produces our formats, integrates with DropBox and GitHub, and platform agnostic. In order to save our diagrams, we have to save two copies - one in SVG (scalable vector) format and the other in PNG (raster). We use the PNG format within the docs since it can be viewed directly in GitHub. The SVG is used as the master copy as it is editable.
- **Dactyl** - We need to be able to print the online documentation. While it's possible to print

markdown files directly one at a time, we'd like to put the files into set of final PDF documents, where one page might end up in more than one final manual. Dactyl is a maintained open source conversion tool that converts between markdown and PDF. We originally tried Pandoc, but it had bugs with converting tables. Dactyl fixes that and is much more flexible.

- **Ansible** - We need a way to set microservice configurations and monitor/verify that those configuration are correct. We need the tool to be very simple to setup and use. It must support many OS's and work for both the cloud and local environments as well as Docker and non-Docker setup. We looked at many tools including: Chef, Puppet, Cloud Formation, Docker Compose/Swarm, Kubernetes, Terraform, Salt, and Ansible. Chef and Puppet were eliminated because they have a very large learning curve and large setup requirements. AWS Cloud Formation and Docker were both limited to specific environments and we need broader support. Terraform was a good tool, but works differently with each environment, so a configuration for cloud can't be reused locally. Kubernetes is also good, but is designed to send commands across large scale environments, not configure a few specific microservices. Salt and Ansible can both do the job. Salt is more scalable and performant, as it puts an agent on each server to orchestrate config. Ansible is much simpler, having an agentless direct setup. We went with Ansible because of the simple setup and learning curve. We don't need the speed and scale Salt provides and accept a slightly lower performance. Ansible allows us to define the expected state of the microservice in a playbook. If the state is incorrect, Ansible can automatically alert and correct the state. In this way it is both a monitoring and configuration tool.

Frequently Asked Questions

What is Mojaloop? Mojaloop is open-source software for building interoperable digital payments platforms on a national scale. It makes it easy for different kinds of providers to link up their services and deploy low-cost financial services in new markets.

How does it work? Most digital financial providers run on their own networks, which prevents customers who use different services from transacting with each other. Mojaloop functions like a universal switchboard, routing payments securely between all customers, regardless of which network they're on. It consists of three primary layers, each with a specific function: an interoperability layer, which connects bank accounts, mobile money wallets, and merchants in an open loop; a directory service layer, which navigates the different methods that providers use to identify accounts on each side of a transaction; a transactions settlement layer, which makes payments instant and irrevocable; and components which protect against fraud.

Who is it for? There are many components to the code, and everyone either directly or indirectly working with digital financial transactions-fintech developers, bankers, entrepreneurs, startups-is invited to explore and use whatever parts are useful or appealing. The software as a whole is meant to be implemented on a national scale, and so it will be most applicable to mobile money providers, payments associations, central banks, and country regulators.

Developers at fintech and financial services companies can use the code in three ways: adapt the code to the financial services standards for a country, use the code to update their own products and services or create new ones, and improve the code by proposing updates and new versions of it for other users.

For example: *A central bank may commission the use of the software by their commercial partners to speed up the deployment of a national payment gateway.* A major payment processor can use the software to modernize their current offering, to achieve lower transaction costs without major R&D investments. *A fintech startup can use the code to understand practically how to comply with interoperable payment APIs.* A bank can use the code to modify their internal systems so that they easily interoperate with other payment providers.

Why does it exist? Providers trying to reach developing markets with innovative, low-cost digital financial services have to build everything on their own. This raises costs and segregates services from each other. Mojaloop can be used as a foundation to help build interoperable platforms, lowering costs for providers and allowing them to integrate their services with others in the market.

Who's behind it? Mojaloop was built in collaboration with a group of leading tech and fintech companies: Ripple, Dwolla, Software Group, ModusBox and Crosslake Technologies. Mojaloop was created by the Gates Foundation's Mojaloop, which is aimed at leveling the economic playing field by crowding in expertise and resources to build inclusive payment models to benefit the world's poor. It is free to the public as open-source software under the Apache 2.0 License.

What platforms does Mojaloop run on? The Mojaloop platform was developed for modern cloud-computing environments. Open-source methods and widely used platforms, like Node.js, serve as the foundation layer for

Mojaloop. The microservices are packaged in Docker and can be deployed to local hardware or to cloud computing environments like Amazon Web Services or Azure.

Is it really open-source? Yes. All core modules, documentation and white papers are available under a Apache 2.0 License. Mojaloop relies on commonly used open-source software, including node.js, MuleCE, Java and PostgreSQL. Mojaloop also uses the Interledger Protocol to choreograph secure money transfers. The licenses for all of these platforms and their imported dependencies allow for many viable uses of the software.

How can I contribute to Mojaloop? You can contribute by helping us create new functionality on our roadmap or by helping us improve the platform. For our roadmap, go to the Roadmap.md. We recommend starting with the onboarding guide and sample problem. This has been designed by the team to introduce the core ideas of the platform and software, the build methods, and our process for check-ins.

What is supported? Currently the Central ledger components are supported by the team. The DFSP components are outdated and thus the end-to-end environment and full setup is challenging to install.

Phase 3 Roadmap

Cmpletion of Phase 3 is scheduled for February 2019

Functional Epics

- Event Logging Framework: Support operational reporting and auditing of processing
- Error Handling Framework: Consistent reporting in line with specification and support operational auditing
- API Gateway: Provide role, policy-based access, security, abstraction, throttling & control, identity management
- Endpoints for P2P, Merchant: Provide endpoints to support P2P and Merchant payments
- Settlements: Complete settlements process to handle failures and reconciliation positions
- Central directory/Account lookup service: Provide native implementation for ALS to confirm the API specification to provide user lookup
- Fraud & Risk Management System: Provide support for a fraud and risk management system
- Forensic Logging: Support forensic logging to support auditing and reporting
- Reporting API: Provide an API for reporting

Operational Epics

- Testing Framework: Provide a framework for automated regression, functional and other testing to ensure quality
- Performance Improvements: Provide a framework for automated regression, functional and other testing to ensure quality
- ELK framework & logging: Provide framework or dashboards for Operational support, Debugging and Resolving issues
- DevOps: Provide flexibility, dynamism in deployments, improve monitoring and reliability

mechanisms
- Rules Engine: Provide a framework to enforce, implement Business, Scheme rules

Non-Functional Epics

- Deprecate Postgres: Avoid usage of multiple databases to improve supportability and maintenance and move to MySQL
- Security & Threat Modeling: Address security vulnerabilities, issues and provide a report on status of the System's security so that they can be addressed
- Documentation: Update documentation to support adoption by community, for labs, deployment by various partners
- API-Led Design: Refactor central services so that schema validation, paths can be addressed thoroughly (automatically) and decrease maintenance, development effort (for those services don't already follow this)
- API-led connectivity is a methodical way to connect data to applications through reusable and purposeful APIs.

Beyond Phase 3

Below is a list of larger initiatives and epics by area that will help to further develop the Mojaloop project. Some of these have been entered as epics or stories, but most are still in the "concept" phase.

Functional Epics

- Native Implementation P2P: Implementation of resources to support Payee initiated and other Use Cases from the Specification, along with supporting P2P Use case completely
- Native Implementation Payee: Implementation of resources to support Payee initiated

transactions and ones that involve OTPs
- Bulk Payments: Design & Implementation of resources to support Bulk Payments

Central Services

- Directory Interoperability
- Multi-currency and schemes
- Enforcing Currency configurations
- Fees: UI for configuring fees
- Increase performance
- Fraud Scores and Reasons
- Role management
- DSP Management
- Stopping/Pausing a DFSP
- boarding protocol

DFSP/Account Management

- Agent Network
- NFCC identity merchant
- Persistent merchant ID
- Onboarding protocol
- Change password
- Password requirements
- Hold/Restart account

Security

- Central certificate service
- Implement fee quote transfer service in the center
- Prevent user guessing from rogue DFSPs
- Preferred authorizations

Market Deployment

- Integration with major mobile money vendors in Africa (PDP initiative)

CI/CD & Testing

- Implement auto deployment to test environment
- Automatically run acceptance tests in test environment as part of build/deploy
- Automate bulk import tests
- Forensic log test
- Account management test