:years: 2015-2022

Architecture

This document captures the major architectural decisions in platform. The purpose of the document is to provide a guide to the overall structure of the platform; where it fits in the overall context of an MIS solution and its internals so that contributors can more effectively understand how changes that they are considering can be made, and the consequences of those changes.

The target audience for this report is both system integrators (who will use the document to gain an understanding of the structure of the platform and its design rationale) and platform contributors who will use the document to reason about future changes and who will update the document as the system evolves.

History

The Idea

Fineract was an idea born out of a wish to create and deploy technology that allows the microfinance industry to scale. The goal is to:

- Produce a gold standard management information system suitable for microfinance operations
- Acts as the basis of a platform for microfinance
- Open source, owned and driven by member organisations in the community
- Enabling potential for eco-system of providers located near to MFIs

Timeline

- 2006: Project intiated by Grameen Foundation
- Late 2011: Grameen Foundation handed over full responsibility to open source community.
- 2012: Mifos X platform started. Previous members of project come together under the name of Community for Open Source Microfinance (COSM / OpenMF)
- 2013: COSM / OpenMF officially rebranded to Mifos Initiative and receive US 501c3 status.
- 2016: Fineract 1.x began incubation at Apache

Resources

- Project URL is https://github.com/apache/fineract
- Issue tracker is https://issues.apache.org/jira/projects/FINERACT/summary
- Download from http://fineract.apache.org/

System Overview

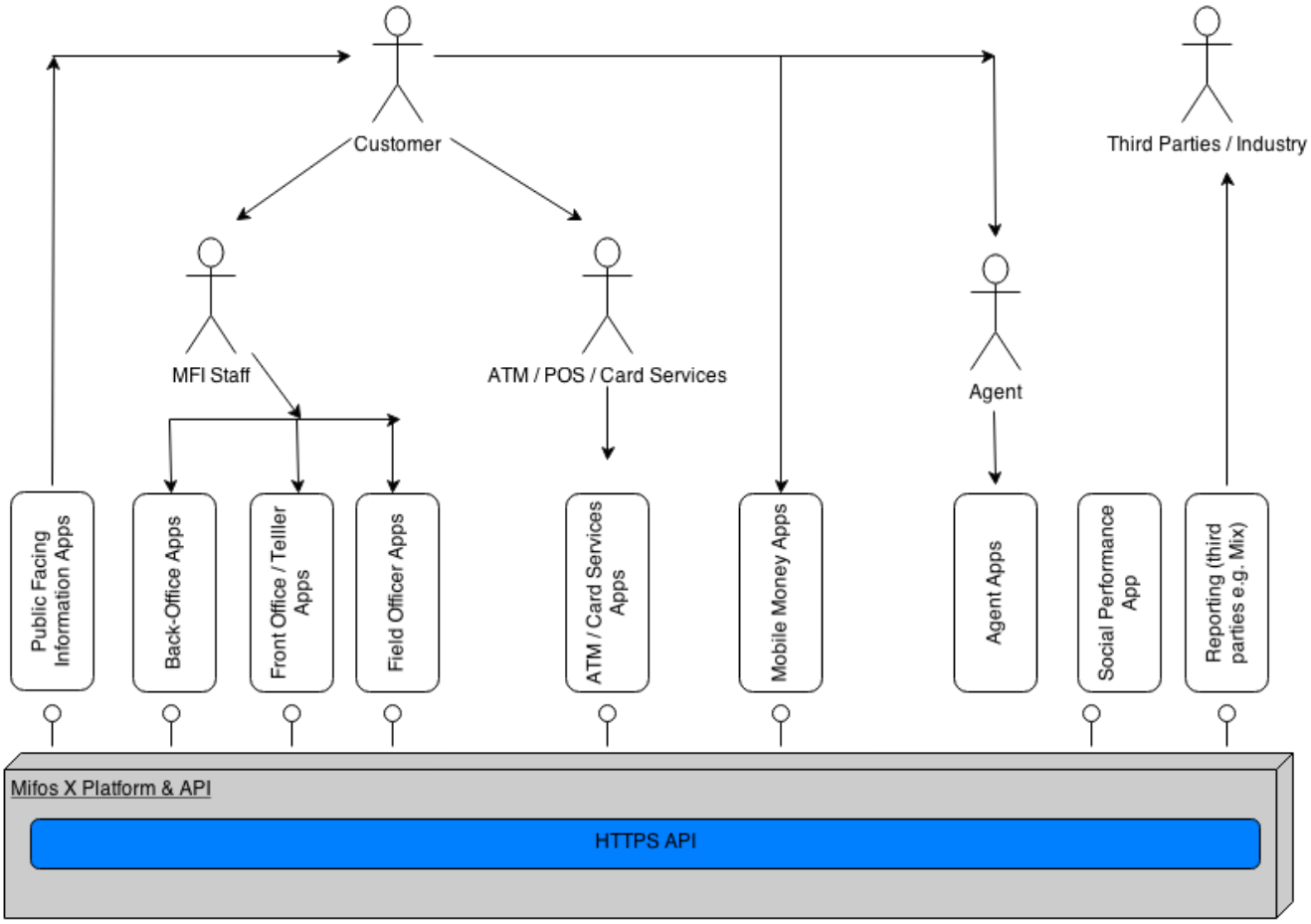


Figure 1. Platform System Overview

Financial institutions deliver their services to customers through a variety of means today.

- Customers can call direct into branches (teller model)
- Customers can organise into groups (or centers) and agree to meetup at a location and time with FI staff (traditional microfinance).
- An FI might have a public facing information portal that customers can use for variety of reasons including account management (online banking).
- An FI might be integrated into a ATM/POS/Card services network that the customer can use.
- An FI might be integrated with a mobile money operator and support mobile money services for customer (present/future microfinance).
- An FI might use third party agents to sell on products/services from other banks/FIs.

As illustrated in the above diagram, the various stakeholders leverage business apps to perform specific customer or FI related actions. The functionality contained in these business apps can be bundled up and packaged in any way. In the diagram, several of the apps may be combined into one app or any one of the blocks representing an app could be further broken up as needed.

The platform is the core engine of the MIS. It hides alot of the complexity that exists in the business and technical domains needed for an MIS in FIs behind a relatively simple API. It is this API that

frees up app developers to innovate and produce apps that can be as general or as bespoke as FIs need them to be.

Functional Overview

As ALL capabilities of the platform are exposed through an API, The API docs are the best place to view a detailed breakdown of what the platform does. See online API Documentation.

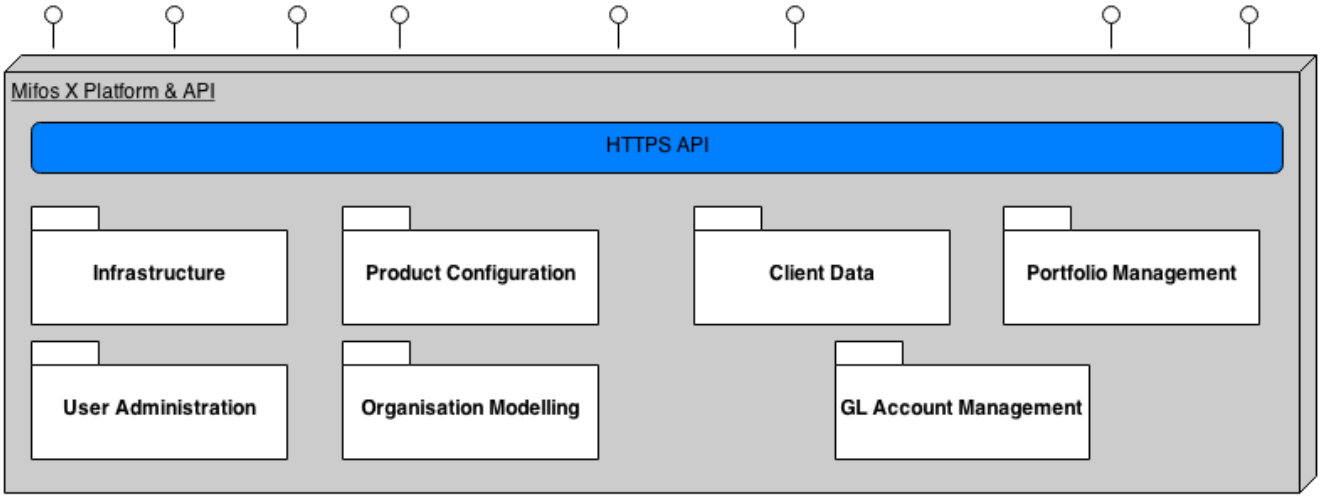


Figure 2. Platform Functional Overview

At a higher level though we see the capabilities fall into the following categories:

- Infrastructure
  - Codes
  - Extensible Data Tables
  - Reporting
- User Administration
  - Users
  - Roles
  - Permissions
- Organisation Modelling
  - Offices
  - Staff
  - Currency
- Product Configuration
  - Charges
  - Loan Products
  - Deposit Products
- Client Data

    - Know Your Client (KYC)
- Portfolio Management
    - Loan Accounts
    - Deposit Accounts
    - Client/Groups
- GL Account Management
    - Chart of Accounts
    - General Ledger

Principles

RESTful API

The platform exposes all its functionality via a **practically-RESTful API**, that communicates using JSON.

We use the term **practically-RESTful** in order to make it clear we are not trying to be fully REST compliant but still maintain important RESTful attributes like:

- Stateless: platform maintains no conversational or session-based state. The result of this is ability to scale horizontally with ease.
- Resource-oriented: API is focussed around set of resources using HTTP vocabulary and conventions e.g GET, PUT, POST, DELETE, HTTP status codes. This results in a simple and consistent API for clients.

See online API Documentation for more detail.

Multi-tenanted

The Fineract platform has been developed with support for multi-tenancy at the core of its design. This means that it is just as easy to use the platform for Software-as-a-Service (SaaS) type offerings as it is for local installations.

The platform uses an approach that isolates an FIs data per database/schema (See Separate Databases and Shared Database, Separate Schemas).

Extensible

Whilst each tenant will have a set of core tables, the platform tables can be extended in different ways for each tenant through the use of Data tables functionality.

Command Query Seperation

We seperate **commands** (that change data) from **queries** (that read data).

Why? There are numerous reasons for choosing this approach which at present is not an attempt at full blown CQRS. The main advantages at present are:

- State changing commands are persisted providing an audit of all state changes.
- Used to support a general approach to **maker-checker**.
- State changing commands use the Object-Oriented paradign (and hence ORM) whilst querys can stay in the data paradigm.

Maker-Checker

Also known as **four-eyes principal**. Enables apps to support a maker-checker style workflow process. Commands that pass validation will be persisted. Maker-checker can be enabled/disabled at fine-grained level for any state changing API. Fine grained access control

A fine grained permission is associated with each API. Administrators have fine grained control over what roles or users have access to.

Package Structure

The intention is for platform code to be packaged in a vertical slice way (as opposed to layers). Source code starts from https://github.com/apache/fineract/tree/develop/fineract-provider/src/main/java/org/apache/fineract

- accounting
- useradministration
- infrastructure
- portfolio
    - charge
    - client
    - fund
    - loanaccount
- accounting

Within each vertical slice is some common packaging structure:

- api - XXXApiResource.java - REST api implementation files
- handler - XXXCommandHandler.java - specific handlers invoked
- service - contains read + write services for functional area
- domain - OO concepts for the functional area

- data - Data concepts for the area
- serialization - ability to convert from/to API JSON for functional area

Design Overview

The implementation of the platform code to process commands through handlers whilst supporting maker-checker and authorisation checks is a little bit convoluted at present and is an area pin-pointed for clean up to make it easier to on board new platform developers. In the mean time below content is used to explain its workings at present.

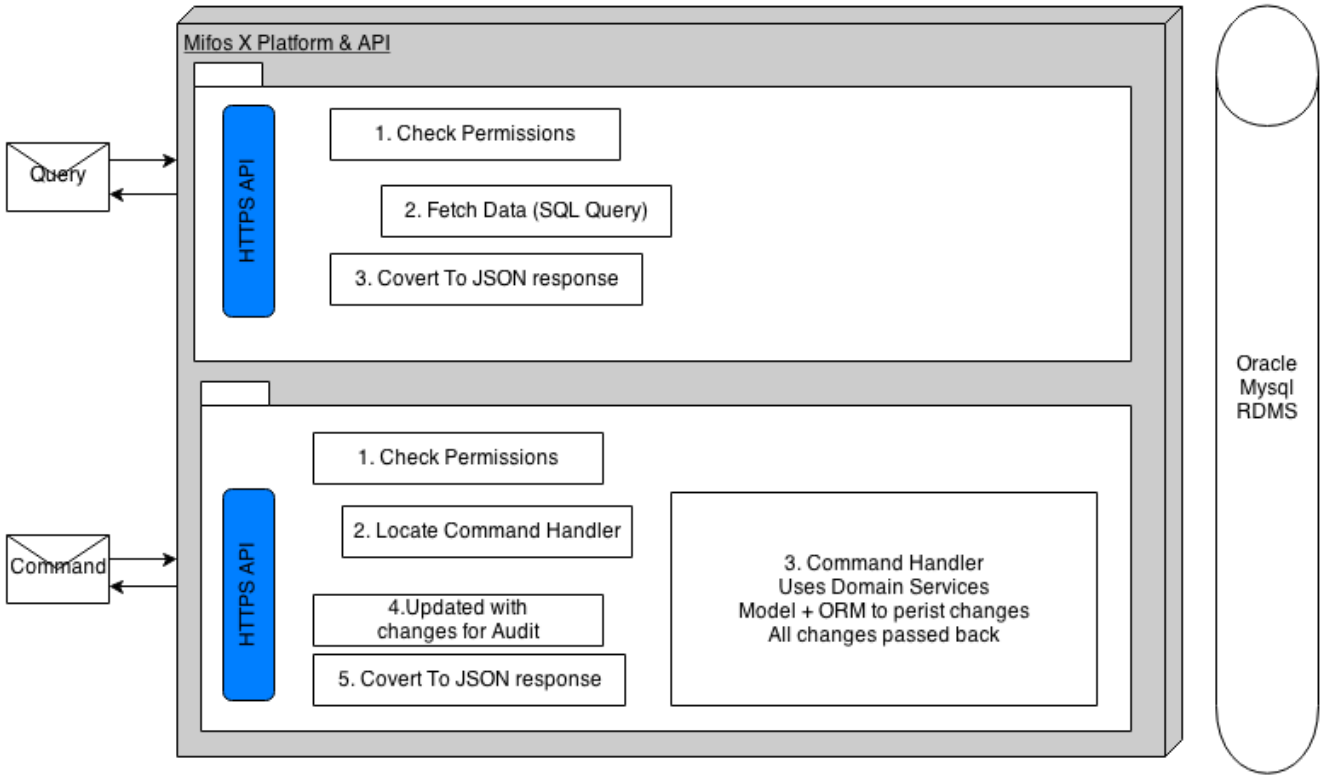


Figure 3. CQRS

Taking into account example shown above for the **users** resource.

- Query: GET /users
- HTTPS API: retrieveAll method on **org.apache.fineract.useradministration.api.UsersApiResource** invoked
- UsersApiResource.retrieveAll: Check user has permission to access this resources data.
- UsersApiResource.retrieveAll: Use 'read service' to fetch all users data ('read services' execute simple SQL queries against Database using JDBC)
- UsersApiResource.retrieveAll: Data returned to coverted into JSON response
- Command: POST /users (Note: data passed in request body)
- HTTPS API: create method on org.apache.fineract.useradministration.api.UsersApiResource invoked

UsersApiResource.create

```
    @POST
    @Operation(summary = "Create a User", description = "Adds new application user.\n"
+ "\n"
            + "Note: Password information is not required (or processed). Password
details at present are auto-generated and then sent to the email account given (which
is why it can take a few seconds to complete).\n"
            + "\n" + "Mandatory Fields: \n" + "username, firstname, lastname, email,
officeId, roles, sendPasswordToEmail\n" + "\n"
            + "Optional Fields: \n" +
"staffId,passwordNeverExpires,isSelfServiceUser,clients")
    @RequestBody(required = true, content = @Content(schema = @Schema(implementation =
UsersApiResourceSwagger.PostUsersRequest.class)))
    @ApiResponses({
            @ApiResponse(responseCode = "200", description = "OK", content = @Content
(schema = @Schema(implementation = UsersApiResourceSwagger.PostUsersResponse.class)))
})
    @Consumes({ MediaType.APPLICATION_JSON })
    @Produces({ MediaType.APPLICATION_JSON })
    public String create(@Parameter(hidden = true) final String apiRequestBodyAsJson)
{

        final CommandWrapper commandRequest = new CommandWrapperBuilder() //
                .createUser() //
                .withJson(apiRequestBodyAsJson) //
                .build();

        final CommandProcessingResult result = this.
commandsSourceWritePlatformService.logCommandSource(commandRequest);

        return this.toApiJsonSerializer.serialize(result);
```

Create a CommandWrapper object that represents this create user command and JSON request body. Pass off responsiblity for processing to PortfolioCommandSourceWritePlatformService.logCommandSource

```
        final JsonElement parsedCommand = this.fromApiJsonHelper.parse(json);
        JsonCommand command = JsonCommand.from(json, parsedCommand, this
.fromApiJsonHelper, wrapper.getEntityName(), wrapper.getEntityId(),
                wrapper.getSubentityId(), wrapper.getGroupId(), wrapper.getClientId(),
wrapper.getLoanId(), wrapper.getSavingsId(),
                wrapper.getTransactionId(), wrapper.getHref(), wrapper.getProductId(),
wrapper.getCreditBureauId(),
                wrapper.getOrganisationCreditBureauId(), wrapper.getJobName());

        return this.processAndLogCommandService.executeCommand(wrapper, command,
isApprovedByChecker);
    }

    @Override
```

```
    public CommandProcessingResult approveEntry(final Long makerCheckerId) {

        final CommandSource commandSourceInput = validateMakerCheckerTransaction
(makerCheckerId);
        validateIsUpdateAllowed();

        final CommandWrapper wrapper = CommandWrapper.fromExistingCommand
(makerCheckerId, commandSourceInput.getActionName(),
                commandSourceInput.getEntityName(), commandSourceInput.resourceId(),
commandSourceInput.subResourceId(),
                commandSourceInput.getResourceGetUrl(), commandSourceInput
.getProductId(), commandSourceInput.getOfficeId(),
                commandSourceInput.getGroupId(), commandSourceInput.getClientId(),
commandSourceInput.getLoanId(),
                commandSourceInput.getSavingsId(), commandSourceInput.
getTransactionId(), commandSourceInput.getCreditBureauId(),
                commandSourceInput.getOrganisationCreditBureauId(),
commandSourceInput.getIdempotencyKey());
        final JsonElement parsedCommand = this.fromApiJsonHelper.parse
(commandSourceInput.getCommandJson());
        final JsonCommand command = JsonCommand.fromExistingCommand(makerCheckerId,
commandSourceInput.getCommandJson(), parsedCommand,
                this.fromApiJsonHelper, commandSourceInput.getEntityName(),
commandSourceInput.resourceId(),
                commandSourceInput.subResourceId(), commandSourceInput.getGroupId(),
commandSourceInput.getClientId(),
                commandSourceInput.getLoanId(), commandSourceInput.getSavingsId(),
commandSourceInput.getTransactionId(),
                commandSourceInput.getResourceGetUrl(), commandSourceInput
.getProductId(), commandSourceInput.getCreditBureauId(),
                commandSourceInput.getOrganisationCreditBureauId(),
commandSourceInput.getJobName());

        return this.processAndLogCommandService.executeCommand(wrapper, command, true
);
    }

    @Transactional
    @Override
    public Long deleteEntry(final Long makerCheckerId) {

        validateMakerCheckerTransaction(makerCheckerId);
        validateIsUpdateAllowed();

        this.commandSourceRepository.deleteById(makerCheckerId);

        return makerCheckerId;
    }

    private CommandSource validateMakerCheckerTransaction(final Long makerCheckerId) {
```

```
        final CommandSource commandSourceInput = this.commandSourceRepository.
findById(makerCheckerId)
                .orElseThrow(() -> new CommandNotFoundException(makerCheckerId));
        if (!commandSourceInput.isMarkedAsAwaitingApproval()) {
            throw new CommandNotAwaitingApprovalException(makerCheckerId);
        }

        this.context.authenticatedUser().validateHasCheckerPermissionTo
(commandSourceInput.getPermissionCode());

        return commandSourceInput;
    }

    private void validateIsUpdateAllowed() {
        this.schedulerJobRunnerReadService.isUpdatesAllowed();
    }
```

Check user has permission for this action. if ok, a) parse the json request body, b) create a JsonCommand object to wrap the command details, c) use CommandProcessingService to handle command

```
    @Override
    @Retry(name = "executeCommand", fallbackMethod = "fallbackExecuteCommand")
    public CommandProcessingResult executeCommand(final CommandWrapper wrapper, final
JsonCommand command,
            final boolean isApprovedByChecker) {
        // Do not store the idempotency key because of the exception handling
        setIdempotencyKeyStoreFlag(false);

        final boolean rollbackTransaction = configurationDomainService
.isMakerCheckerEnabledForTask(wrapper.taskPermissionName());
        String idempotencyKey = idempotencyKeyResolver.resolve(wrapper);
        exceptionWhenTheRequestAlreadyProcessed(wrapper, idempotencyKey);

        // Store idempotency key to the request attribute

        CommandSource savedCommandSource = commandSourceService.saveInitial(wrapper,
command, context.authenticatedUser(wrapper),
                idempotencyKey);
        storeCommandToIdempotentFilter(savedCommandSource);
        setIdempotencyKeyStoreFlag(true);

        final CommandProcessingResult result;
        try {
            result = findCommandHandler(wrapper).processCommand(command);
        } catch (Throwable t) { // NOSONAR
            commandSourceService.saveFailed(commandSourceService.findCommandSource
(wrapper, idempotencyKey));
            publishHookErrorEvent(wrapper, command, t);
            throw t;
        }
```

```
        CommandSource initialCommandSource = commandSourceService.findCommandSource
(wrapper, idempotencyKey);
        initialCommandSource.setResult(toApiJsonSerializer.serializeResult(result));
        initialCommandSource.updateResourceId(result.getResourceId());
        initialCommandSource.updateForAudit(result);

        boolean rollBack = (rollbackTransaction || result.isRollbackTransaction()) &&
!isApprovedByChecker;
        if (result.hasChanges() && !rollBack) {
            initialCommandSource.setCommandJson(toApiJsonSerializer.serializeResult
(result.getChanges()));
        }

        initialCommandSource.setStatus(CommandProcessingResultType.PROCESSED.getValue
());
        commandSourceService.saveResult(initialCommandSource);

        if ((rollbackTransaction || result.isRollbackTransaction()) &&
!isApprovedByChecker) {
            /*
             * JournalEntry will generate a new transactionId every time. Updating the
transactionId with old
             * transactionId, because as there are no entries are created with new
transactionId, will throw an error
             * when checker approves the transaction
             */
            initialCommandSource.updateTransaction(command.getTransactionId());
            /*
             * Update CommandSource json data with JsonCommand json data, line 77 and
81 may update the json data
             */
            initialCommandSource.setCommandJson(command.json());
            throw new RollbackTransactionAsCommandIsNotApprovedByCheckerException
(initialCommandSource);
        }
        result.setRollbackTransaction(null);

        publishHookEvent(wrapper.entityName(), wrapper.actionName(), command, result);

        return result;
    }

    private void storeCommandToIdempotentFilter(CommandSource savedCommandSource) {
        if (savedCommandSource.getId() == null) {
            throw new IllegalStateException("Command source not saved");
        }
        saveCommandToRequest(savedCommandSource);
    }
```

if a RollbackTransactionAsCommandIsNotApprovedByCheckerException occurs at this point. The original transaction will of been aborted and we only log an entry for the command in the audit table setting its status as 'Pending'.

- Check that if maker-checker configuration enabled for this action. If yes and this is not a 'checker' approving the command - rollback at the end. We rollback at the end in order to test if the command will pass 'domain validation' which requires commit to database for full check.
- findCommandHandler - Find the correct Hanlder to process this command.
- Process command using handler (In transactional scope).
- CommandSource object created/updated with all details for logging to 'm_portfolio_command_source' table.
- In update scenario, we check to see if there where really any changes/updates. If so only JSON for changes is stored in audit log.

Persistence

TBD

Database support

Fineract supports multiple databases:

- MySQL compatible databases (e.g. MariaDB)
- PostgreSQL

The platform differentiates between these database types in certain cases when there's a need to use some database specific tooling. To do so, the platform examines the JDBC driver used for running the platform and tries to determine which database is being used.

The currently supported JDBC driver and corresponding mappings can be found below.

JDBC driver class name	Resolved database type
`org.mariadb.jdbc.Driver`	MySQL
`com.mysql.jdbc.Driver`	MySQL
`org.postgresql.Driver`	PostgreSQL

The actual code can be found in the `DatabaseTypeResolver` class.

Data-access layer

The data-access layer of Fineract is implemented by using JPA (Java Persistence API) with the EclipseLink provider. Despite the fact that JPA is used quite extensively in the system, there are cases where the performance is a key element for an operation therefore you can easily find native SQLs as well.

The data-access layer of Fineract is compatible with different databases. Since a lot of the native queries are using specific database functions, a wrapper class - `DatabaseSpecificSQLGenerator` - has been introduced to handle these database specifics. Whenever there's a need to rely on new database level functions, make sure to extend this class and implement the specific functions provided by the database.

Fineract has been developed for 10+ years by the community and unfortunately there are places where entity relationships are configured with `EAGER` fetching strategy. This must not confuse anybody. The long-term goal is to use the `LAZY` fetching strategy for every single relationship. If you're about to introduce a new one, make sure to use `LAZY` as a fetching strategy, otherwise your PR will be rejected.

Database schema migration

As for every system, the database structure will and need to evolve over time. Fineract is no different. Originally for Fineract, Flyway was used until Fineract 1.6.x.

After 1.6.x, PostgreSQL support was added to the platform hence there was a need to make the data-access layer and the schema migration as database independent as possible. Becuase of that, from Fineract 1.7.0, Flyway is not used anymore but Liquibase is.

Some of the changesets in the Liquibase changelogs have database specifics into it but they only run for the relevant databases. This is controller by Liquibase contexts.

The currently available Liquibase contexts are:

- `mysql` - only set when the database is a MySQL compatible database (e.g. MariaDB)
- `postgresql` - only set when the database is a PostgreSQL database
- configured Spring active profiles
- `tenant_store_db` - only set when the database migration runs the Tenant Store upgrade
- `tenant_db` - only set when the database migration runs the Tenant upgrade
- `initial_switch` - this is a technical context and should **NOT** be used

The switch from Flyway (1.6.x) to Liquibase (1.7.x) was planned to be as smooth as possible so there's no need for manual work hence the behavior is described as following:

- If the database is empty, Liquibase will create the database schema from scratch
- If the database contains the latest Fineract 1.6.x database structure which was previously migrated with Flyway. Liquibase will seamlessly upgrade it to the latest version. Note: the Flyway related 2 database tables are left as they are and are not deleted.
- If the database contains an earlier version of the database structure than Fineract 1.6.x. Liquibase will **NOT** do anything and **will fail the application during startup**. The proper approach in this case is to first upgrade your application version to the latest Fineract 1.6.x so that the latest Flyway changes are executed and then upgrade to the newer Fineract version where Liquibase will seamlessly take over the database upgrades.

Troubleshooting

1. During upgrade from Fineract 1.5.0 to 1.6.0, Liquibase fails

After dropping the flyway migrations table (schema_version), Liquibase runs it's own migrations which fails (in recreating tables which already exisit) because we are aiming to re-use DB with existing data from Fineract 1.5.0.

Solution: The latest release version (1.6.0) doesn't have Liquibase at all, it still runs Flyway migrations. Only the develop branch (later to be 1.7.0) got sqitched to Liquibase. Do not pull the develop before upgrading your instance.

Make sure first you upgrade your instance (aka database schema with Fineract 1.6.0). Then upgrade with the current develop branch. Check if some migration scripts did not run which led to some operations failing due to slight differences in schema. Try with running the missing migrations manually.

Note: develop is considered unstable until released.

1. Upgrading database from MySQL 5.7 as advised to Maria DB 10.6, fails. If we use data from version 18.03.01 it fails to migrate the data. If we use databases running on 1.5.0 release it completes the startup but the system login fails.

Solution: A database upgrade is separate thing to take care of.

1. We are getting `ScehmaUpgradeNeededException: Make sure to upgrade to Fineract 1.6 first and then to a newer version` error while upgrading to `tag 1.6`.

1.6 version shouldn't include Liquibase. It will only be released after 1.6. Make sure Liquibase is dropping `schema_version` table, as there is no Flyway it is not required. Drop Flyway and use Liquibase for both migrations and database independence. In case, if you still get errors, you can use git SHA `746c589a6e809b33d68c0596930fcaa7338d5270` and Flyway migration will be done to the latest.

```
TENANT_LATEST_FLYWAY_VERSION = 392;
TENANT_LATEST_FLYWAY_SCRIPT_NAME =
"V392__interest_recovery_conf_for_rescedule.sql";
TENANT_LATEST_FLYWAY_SCRIPT_CHECKSUM = 1102395052;
```

Idempotency

Idempotency is the way to make sure your specific action is only executed once. For example, if you have a button that is supposed to send a repayment, you don't want to repayment twice if the user clicks the button twice. Idempotency is a way to make sure that the action is only executed once.

There are two ways to use idempotency:

- HTTP Request with idempotency key header

- Batch request with batch item header

How it works

The `idempotency key` with `action name` and `entity name` is unique, and identify a specific command in the system. If no idempotency key is assigned to the request, the system will generate one for you.

1. User send a request
2. The system checks there are already executed commands with the same `idempotency key` and `action name` and `entity name`
3. The action based on the result of the check
    - If the request is completed the system return with the already generated result
    - If not completed, return HTTP 409 response
    - If the request is not completed, we process the requests and store the results in the database

Idempotency in HTTP requests

To achieve idempotency in HTTP requests, you can use the HTTP header from `fineract.idempotency-key-header-name` configuration variables (default `Idemptency-Key`). This header is a unique identifier for the request. If you send the same request twice, the second request will be ignored and the response from the first request will be returned.

Idempotency in Batch requests

In batch requests, you can set the idempotency key for every batch item, in the batch item `header` fields. The header key is from `fineract.idempotency-key-header-name` configuration variables (default `Idemptency-Key`).

Result of the request

- When the request is already executed and completed, the system will return a `x-served-from-cache` header with the value `true` in the response and return the original request body.
- When the request is already executed but still not completed, the system will return to HTTP 409 error code
- When the request is not executed, the system runs it normally and stores the result in the date

Validation

Programmatic

Use the DataValidatorBuilder, e.g. like so:

```
new DataValidatorBuilder().resource("fileUpload")
    .reset().parameter("Content-Length").value(contentLength).notBlank()
.integerGreaterThanNumber(0)
    .reset().parameter("FormDataContentDisposition").value(fileDetails).notNull()
    .throwValidationErrors();
```

Such code is often encapsulated in `*Validator` classes (if more than a few lines, and/or reused from several places; avoid copy/paste), like so:

```
public class YourThingValidator {

    public void validate(YourThing thing) {
        new DataValidatorBuilder().resource("yourThing")
        ...
        .throwValidationErrors();
    }
}
```

Declarative

[FINERACT-1229](https://issues.apache.org/jira/browse/FINERACT-1229) is an open issue about adopting Bean Validation for *declarative* instead of *programmatic* (as above) validation. Contributions welcome!

Batch execution and jobs

Just like any financial system, Fineract also has batch jobs to achieve some processing on the data that's stored in the system.

The batch jobs in Fineract are implemented using Spring Batch. In addition to the Spring Batch ecosystem, the automatic scheduling is done by the Quartz Scheduler but it's also possible to trigger batch jobs via regular APIs.

Glossary

Job	A Job is an object that encapsulates an entire batch process.
Step	A Step is an object that encapsulates an independent phase of a Job.
Chunk oriented processing	Chunk oriented processing refers to reading the data one at a time and creating 'chunks' that are written out within a transaction boundary.

Partitioning	Partitioning refers to the high-level idea of dividing your data into so called partitions and distributing the individual partitions among Workers. The splitting of data and pushing work to Workers is done by a Manager.
Remote partitioning	Remote partitioning is a specialized partitioning concept. It refers to the idea of distributing the partitions among multiple JVMs mainly by using a messaging middleware.
Manager node	The Manager node is one of the objects taking a huge part when using partitioning. The Manager node is responsible for dividing the dataset into partitions and keeping track of all the divided partitions' Worker execution. When all Workers nodes are done with their partitions, the Manager will mark the corresponding Job as completed.
Worker node	A Worker node is the other important party in the context of partitioning. The Worker node is the one executing the work needed for a single partition.

Batch jobs in Fineract

Types of jobs

The jobs in Fineract can be divided into 2 categories:

- Normal batch jobs
- Partitionable batch jobs

Most of the jobs are normal batch jobs with limited scalability because Fineract is still passing through the evolution on making most of them capable to process a high-volume of data.

List of jobs

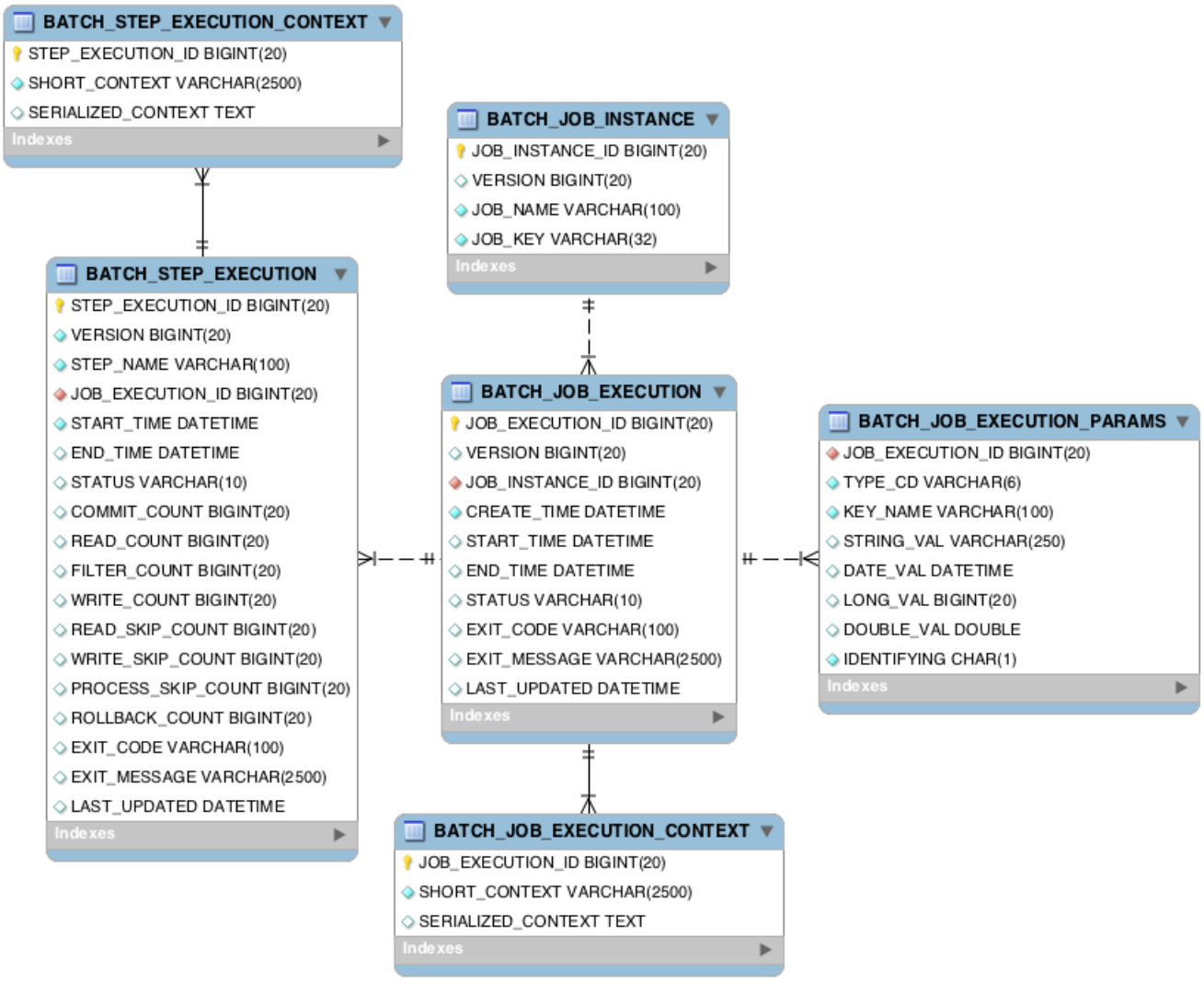
Job name	Active by default	Partitionable	Description
LOAN_CLOSE_OF_BUSINESS	No	Yes	TBD

Batch job execution

State management

State management for the batch jobs is done by the Spring Batch provided state management. The

data model consists of the following database structure:

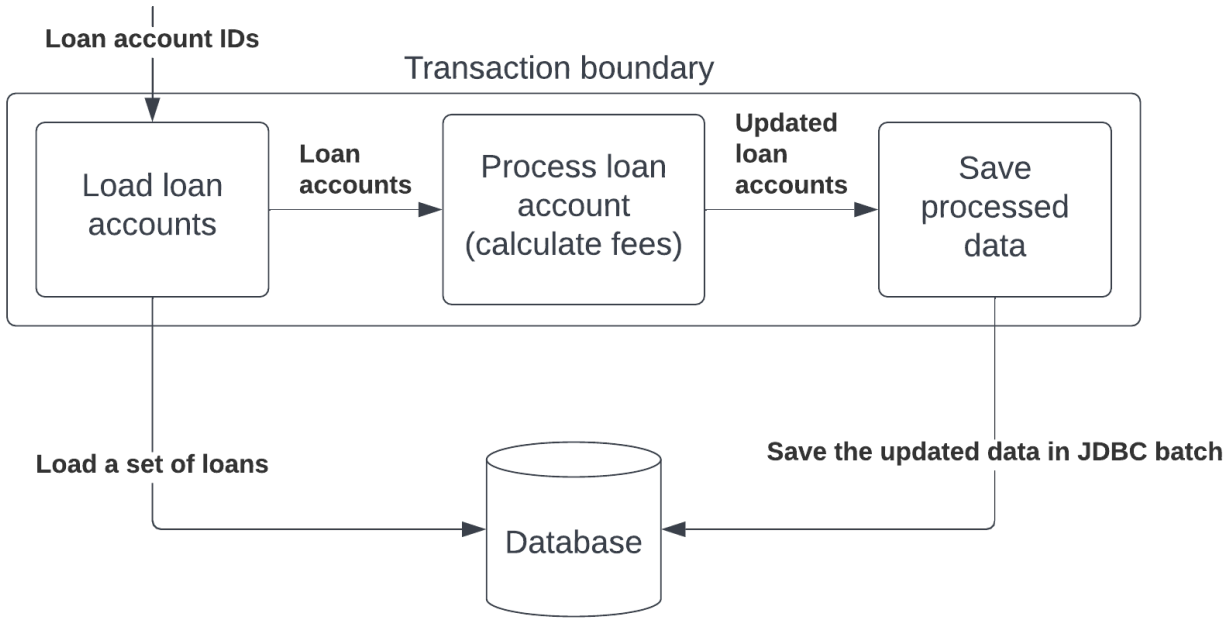


The corresponding database migration scripts are shipped with the Spring Batch core module under the `org.springframework.batch.core` package. They are only available as native scripts and are named as `schema-.sql` where is the short name of the database platform. For MySQL it's called `schema-mysql.sql` and for PostgreSQL it's called `schema-postgresql.sql`. When Fineract is started, the database dependent schema SQL script will be picked up according to the datasource configurations.

Chunk oriented processing

Chunking data has not been easier. Spring Batch does a really good job at providing this capability.

In order to save resources when starting/committing/rollbacking transactions for every single processed item, chunking shall be used. That way, it's possible to mark the transaction boundaries for a single processed chunk instead of a single item processing. The image below describes the flow with a very simplistic example.



In addition to not opening a lot of transactions, the processing could also benefit from JDBC batching. The last step - writing the result into the database - collects all the processed items and then writes it to the database; both for MySQL and PostgreSQL (the databases supported by Fineract) are capable of grouping multiple DML (INSERT/UPDATE/DELETE) statements and sending them in one round-trip, optimizing the data being sent over the network and granting the possibility to the underlying database engine to enhance the processing.

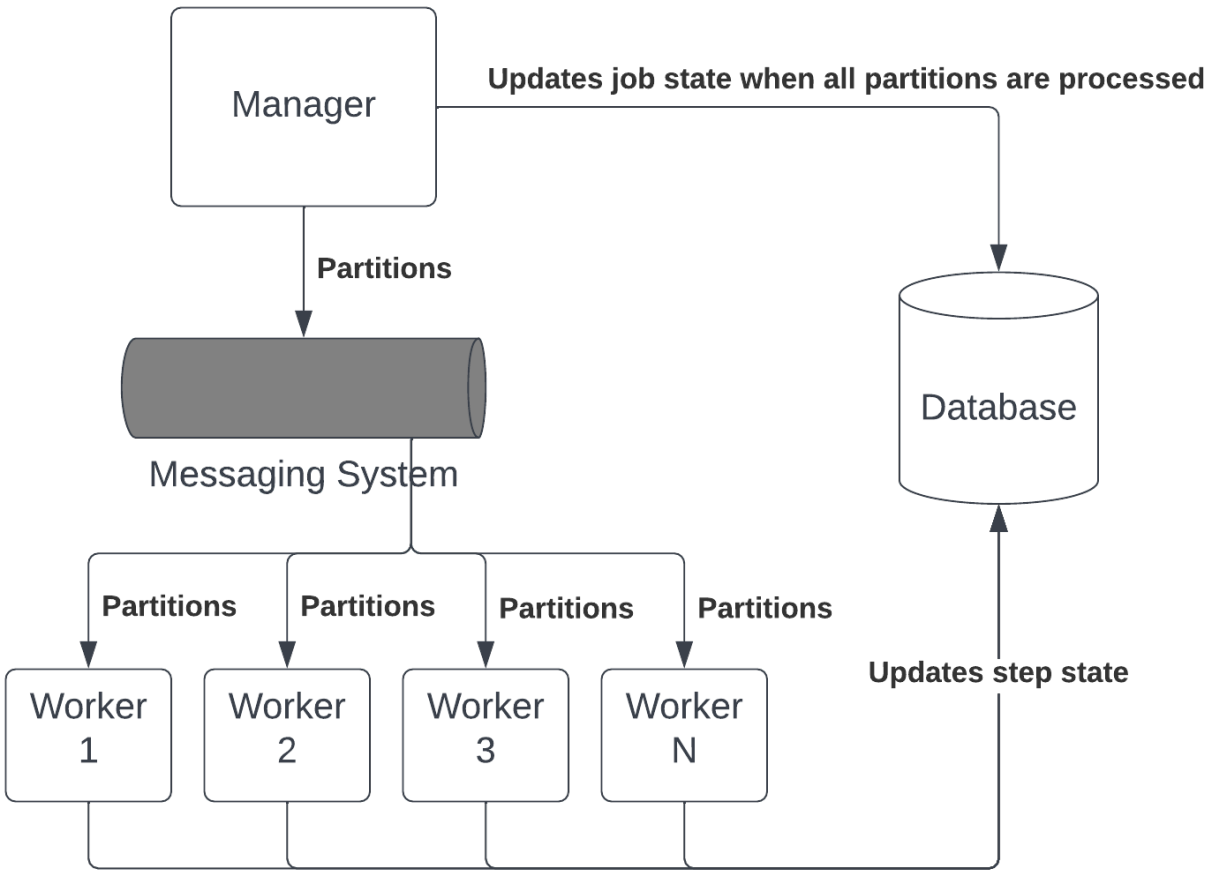
Remote partitioning

Spring Batch provides a really nice way to do remote partitioning. The 2 type of objects in this setup is a manager node - who splits and distributes the work - and a number of worker nodes - who picks up the work.

In remote partitioning, the worker instances are receiving the work via a messaging system as soon as the manager splits up the work into smaller pieces.

Remote partitioning could be done 2 ways in terms of keeping the job state up-to-date. The main difference between the two is how the manager is notified about partition completions.

One way is that they share the same database. When the worker does something to a partition - for example picks it up for processing - it updates the state of that partition in the database. In the meantime, the manager regularly polls the database until all partitions are processed. This is visualized in the below diagram.



An alternative approach to this - when the database is not intended to be shared between manager and workers - is to use a messaging system (could be the same as for distributing the work) and the workers could send back a message to the manager instance, therefore notifying it about failure/completion. Then the manager can simply keep the database state up-to-date.

Even though the alternative solution decouples the workers even better, we thought it's not necessary to add the complexity of handling reply message channel to the manager.

Also, please note that the partitioned job execution is multitenant meaning that the workers will receive which tenant it should do the processing for.

Supported message channels

For remote partitioning, the following message channels are supported by Fineract:

- Any JMS compatible message channels (ActiveMQ, Amazon MQ, etc)

Fault-tolerance scenarios

There are multiple fault tolerance use-cases that this solution must and will support:

1. If the manager fails during partitioning
2. If the manager completes the partitioning and the partition messages are sent to the broker but while the manager is waiting for the workers to finish, the manager fails

3. If the manager runs properly and during a partition processing a worker instance fails

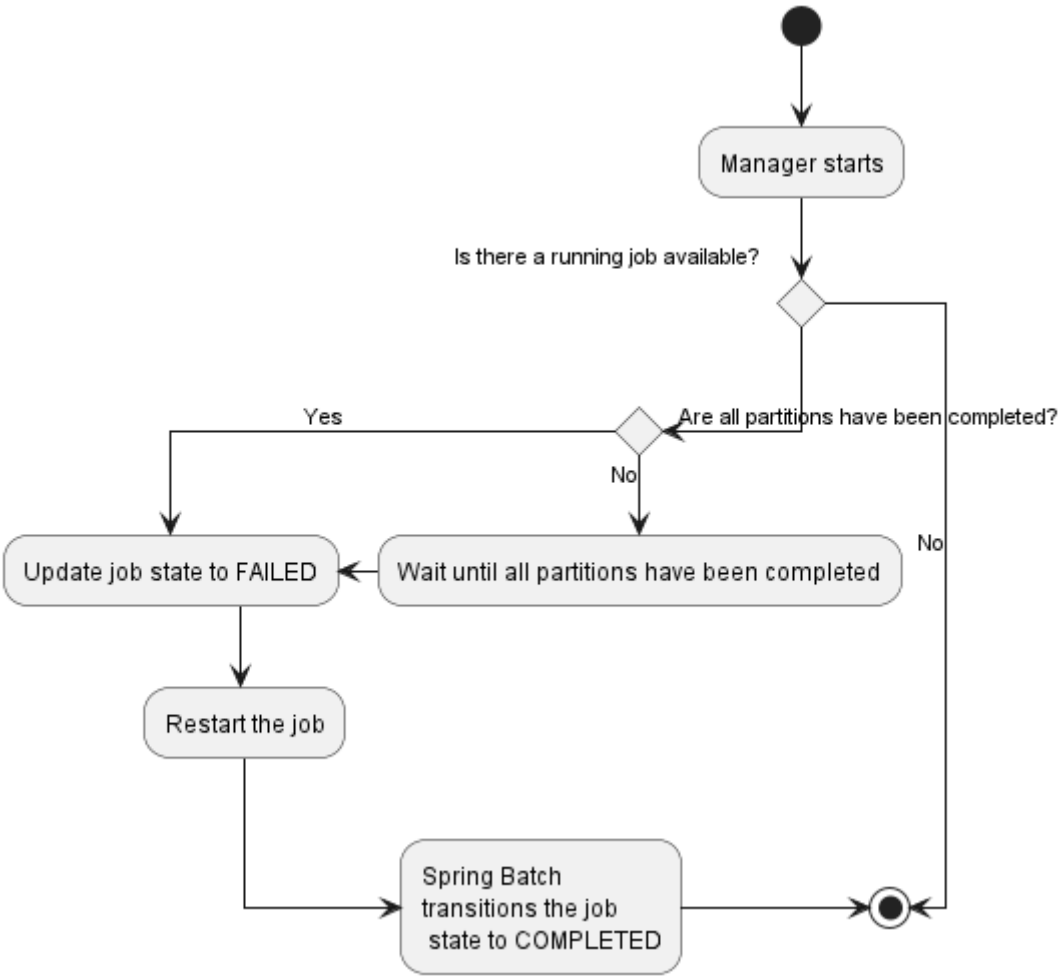
In case of scenario 1), the simple solution is to re-trigger the job via API or via the Quartz scheduler.

In case of scenario 2), there's no out-of-the-box solution by Spring Batch. Although there's a custom mechanism in place that'll resume the job upon restarting the manager. There are 2 cases in the context of this scenario:

- If all the partitions have been successfully processed by workers
- If not all the partitions have been processed by the workers

In the first case, we'll simply mark the stuck job as `FAILED` along with it's partitioning step and instruct Spring Batch to restart the job. The behavior in this case will be that Spring Batch will spawn a new job execution but will notice that the partitions have all been completed so it's not going to execute them once more.

In the latter case, the same will happen as for the first one but before marking the job execution as `FAILED`, we'll wait until all partitions have been completed.



In case of scenario 3), another worker instance will take over the partition since it hasn't been finished.

Configurable batch jobs

There's another type of distinction on the batch jobs. Some of them are configurable in terms of their behavior.

The currently supported configurable batch jobs are the following:

- LOAN_CLOSE_OF_BUSINESS

The behavior of these batch jobs are configurable. There's a new terminology we're introducing called **business steps**.

Business steps

Business steps are a smaller unit of work than regular Spring Batch Steps and the two are not meant to be mixed up because there's a large difference between them.

A Spring Batch Step's main purpose is to decompose a bigger work into smaller ones and making sure that these smaller Steps are properly handled within a single database transaction.

In case of a business step, it's a smaller unit of work. Business steps live **within** a Spring Batch Step. Fundamentally, they are simple classes that are implementing an interface with a single method that contains the business logic.

Here's a very simple example:

```
public class MyCustomBusinessStep implements BusinessStep<Loan> {
    @Override
    public Loan process(Loan loan) {
        // do something
    }
}
```

```
public class LoanCOBItemProcessor implements ItemProcessor<Loan, Loan> {
    @Override
    public Loan process(Loan loan) {
        List<BusinessStep<Loan>> bSteps = getBusinessSteps();
        Loan result = loan;
        for (BusinessStep<Loan> bStep : bSteps) {
            result = bStep.process(result);
        }
        return result;
    }
}
```

Business step configuration

The business steps are configurable for certain jobs. The reason for that is because we want to

allow the possibility for Fineract users to configure their very own business logic for generic jobs, like the Loan Close Of Business job where we want to do a formal "closing" of the loans at the end of the day.

All countries are different with a different set of regulations. However in terms of behavior, there's no all size fits all for loan closing.

For example in the United States of America, you might need the following logic for a day closing:

1. Close fully repaid loan accounts
2. Apply penalties
3. Invoke IRS API for regulatory purposes

While in Germany it should be:

1. Close fully repaid loan accounts
2. Apply penalties
3. Do some fraud detection on the account using an external service
4. Invoke local tax authority API for regulatory purposes

These are just examples, but you get the idea.

The business steps are configurable through APIs:

Retrieving the configuration for a job:

```
GET /fineract-provider/api/v1/jobs/{jobName}/steps?tenantIdentifier={tenantId}
HTTP 200

{
  "jobName": "LOAN_CLOSE_OF_BUSINESS",
  "businessSteps": [
    {
      "stepName": "APPLY_PENALTY_FOR_OVERDUE_LOANS",
      "order": 1
    },
    {
      "stepName": "LOAN_TAGGING",
      "order": 2
    }
  ]
}
```

Updating the business step configuration for a job:

```
PUT /fineract-provider/api/v1/jobs/{jobName}/steps?tenantIdentifier={tenantId}
```

```
{
  "businessSteps": [
    {
      "stepName": "LOAN_TAGGING",
      "order": 1
    },
    {
      "stepName": "APPLY_PENALTY_FOR_OVERDUE_LOANS",
      "order": 2
    }
  ]
}
```

The business step configuration for jobs are tracked within the database in the `m_batch_business_steps` table.

Loan account locking

Keeping a consistent state of loan accounts become quite important when we start talking about doing a business day closing each day for loans.

There are 2 concepts for loan account locking:

1. Soft-locking loan accounts
2. Hard-locking loan accounts

Soft-locking simply means that when the Loan COB has been kicked off but workers not yet processing the chunk of loan accounts (i.e. the partition is waiting in the queue to be picked up) and during this time a real-time write request (e.g. a repayment/disbursement) comes in through the API, we simply do an "inlined" version of the Loan COB for that loan account. From a practical standpoint this will mean that before doing the actual repayment/disbursement on the loan account on the API, we execute the Loan COB for that loan account, kind of like prioritizing it.

Hard-locking means that when a worker picks up the loan account in the chunk, real-time write requests on those loan accounts will be simply rejected with an `HTTP 409`.

The locking is strictly tied to the Loan COB job's execution but there could be other processes in the future which might want to introduce new type of locks for loans.

The loan account locking is solved by maintaining a database table which stores the locked accounts, it's called `m_loan_account_locks`.

When a loan account is present in the table above, it simply means there's a lock applied to it and whether it's a soft or hard lock can be determined by the `lock_owner` column.

And when a loan account is locked, loan related write API calls will be either rejected or will trigger an inline Loan COB execution. There could be a corner case here when the Loan COB fails to process some loan accounts (due to a bug, inconsistency, etc) and the loan accounts stay locked. This is an intended behavior to mark loans which are not supposed to be used until they are "fixed".

Since the fixing might involve making changes to the loan account via API (for example doing a repayment to fix the loan account's inconsistent state), we need to allow those API calls. Hence, the lock table includes a `bypass_enabled` column which disables the lock checks on the loan write APIs.

Technology

- Java: http://www.oracle.com/technetwork/java/javase/downloads/index.html
- JAX-RS using Jersey
- JSON using Google GSON
- Spring I/O Platform: http://spring.io/platform
  - Spring Framework
  - Spring Boot
  - Spring Security
  - Spring Data (JPA) backed by EclipseLink
- MySQL: http://www.oracle.com/us/products/mysql/overview/index.html
- PostgreSQL

TBD

Modules

We are currently working towards a fully modular codebase and will publish more here when we are ready.

Even if we are not quite there yet with full modularity you can already create your own custom modules to extend Fineract. Please see chapter [_custom_modules].

Introducing Business Date into Fineract - Community version

Business date as a concept does not exist as of now in Fineract. It would be business critical to add such a functionality to support various banking functionalities like "Closing of Business day", "Having Closing of Business day relevant jobs", "Supporting logical date management".

Glossary

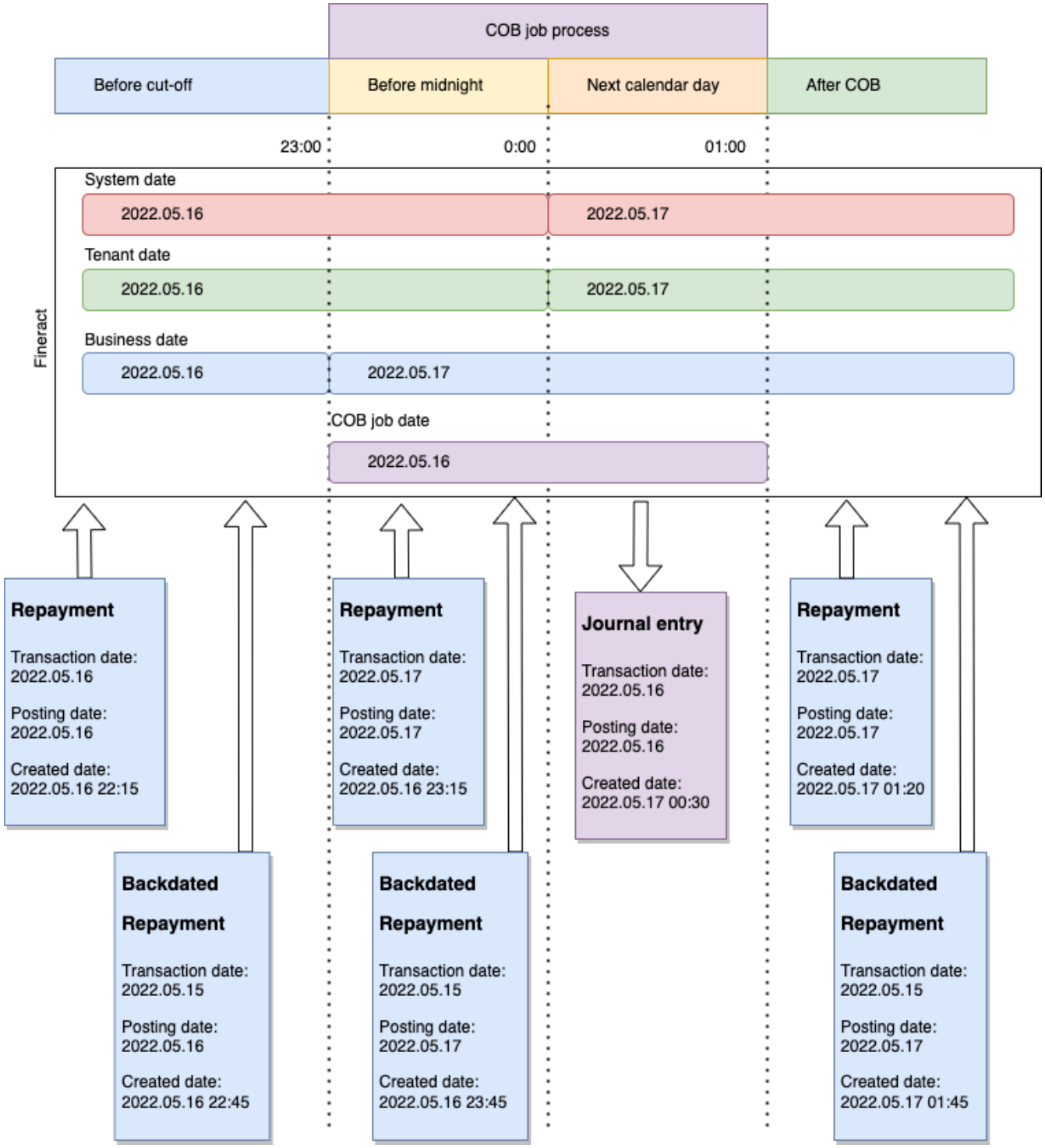
*COB	Close of Business; concept of closing a business day
*Business day	Timeframe that logically group together actions on a particular business date

*Business date	Logical date; its value is not tied to the physical calendar. Represents a business day
*Cob date	Logical date; Represents the business date for actions during COB job execution
*Created date	When the transaction was created (audit purposes). Date + time
*Last modified date	When the transaction was last modified (audit purposes). Date + time
*Submitted on date / Posting date	When the transaction was posted. Tenant date or business date (depends on whether the logical date concept was introduced or not)
*Transaction date / Value date	The date on which the transaction occurred or to be accounted for

Current behaviour

- Fineract support 3 types of dates:
  - System date
    - Physical/System date of the running environment
  - Tenant date
    - Timezoned version of the above system date
  - User-provided date
    - Based on the provided date (as string) and the provided date format
- There is no support of logical date concept
  - Independent from the system / tenant date
- Jobs are scheduled against system date (CRON), but aligned with the tenant timezone.
- During the job execution all the data and transactions are using the actual tenant date
  - It could happen some transactions are written for 17th of May and other for 18th of May, if the job was executed around midnight
- There is no support of COB
  - No backdated transactions by jobs
  - There is no support to logically group together transactions and store them with the same transaction date which is independent of the physical calendar of the tenant
- All the transactions and business logic are tied to a physical calendar

Business date



Design

By introducing the business day concept we are not tied anymore to the physical calendar of the system or the tenant. We got the ability to define our own business day boundaries which might end 15 minutes before midnight and any incoming transactions after the cutoff will be accounted for the following business day.

It is a logical date which makes it possible to separate the business day from the physical calendar:

- Close a business day before midnight
- Close a business day at midnight
- Close a business day after midnight

Closing a Business Day could be a longer process (see COB jobs) meanwhile some processes shall still be able to create transactions for that business day (COB jobs), but others are meant to create the transactions for the next (incoming transactions): Business date concept is there to sort that out.

Business date concept is essential when:

- Having COB jobs:
  - When the COB was triggered:
    - All the jobs which processing the data must still accounted for actual business day
    - All the incoming transactions must be accounted to the next business day
- Business day is ending before / after midnight (tenant date / system date)
- Testing purposes:
  - Since the transactions and job execution is not tied anymore to a physical calendar, we can easily test a whole loan lifecycle by altering the business date
- Handling disruption of service: For any unseen reason the system goes down or there are any disruption in the workflow, the “missed days” can easily be processed one by one as nothing happened
  - There is a disruption at 2022-06-02
  - The issue is fixed by 2022-06-05
  - The COB flow can be executed for 2022-06-03 and when it is finished for 2022-06-04 and after when the time arrives for 2022-06-05

This logical date is manageable via:

- Job
- API

To maintain such separation from physical calendar we need to introduce the following new dates:

- Business date
- COB date
  - Can be calculated based on the actual business date
    - Depend on COB date strategy (see below)

Business date

The - logical - date of the actual business day, eg: 2022-05-06

- It does not support time parts
- It can be managed manually (via API call) or automatically (via scheduled job)
- All business actions during the business day shall use this date:
  - Posting / submitted on date of transactions
  - Submitted on date of actions

    - (Regular) jobs
- It will be used in every situation where the transaction date / value date is not provided by the user or the user provided date shall be validated.
    - Opening date
    - Closing date
    - Disbursal date
    - Transaction/Value date
    - Posting/Submitted date
    - Reversal date
- Will not be use for audit purposes:
    - Created on date
    - Updated on date

COB date

The - logical - date of the business day for job execution, eg: 2022-05-05

- It can be calculated based on the the business date
    - COB date = business date - 1 day
    - Automatically modified alongside with the business date change
- It does not support time parts
- It is automatically managed by business date change
    - Configurable
- It is used only via COB job execution
    - When we create / modify any business data during the COB job execution, the COB date is to be used:
        - Posting date of transactions
        - Submitted on date of actions
        - Transaction / value date of any actions

Some basic example

Apply for a loan

#1

Tenant date: 2022-05-23 14:22:12

Business date: 2022-05-22

Submitted on date: 2022-05-23

Outcome: **FAIL**

Message: **The date on which a loan is submitted cannot be in the future.**

Reason: Even the tenant date is 2022-05-23, but the business date was 2022-05-22 which means anything further that date must be considered as a future date.

#2

Tenant date: 2022-05-23 14:22:12

Business date: 2022-05-22

Submitted on date: 2022-05-22

Outcome: **SUCCESS**

Loan application details:

- Submitted on date: 2022-05-22

Repayment for a loan

#1

Tenant date: 2022-05-25 11:22:12

Business date: 2022-05-24

Transaction date: 2022-05-25

Outcome: **FAIL**

Message: **The transaction date cannot be in the future.**

Reason: Even the physical date is 2022-05-25, but the business date was 2022-05-24 which means anything further that date must be considered as a future date.

#2

Tenant date: 2022-05-25 11:22:12

Business date: 2022-05-24

Transaction date: 2022-05-23

Outcome: **SUCCESS**

Loan transaction details:

- Submitted on date: 2022-05-24
- Transaction date: 2022-05-23

- Created on date: 2022-05-25 11:22:12

Changes in Fineract

We shall modify at all the relevant places where the tenant date was used:

- With very limited exceptions all places where the tenant date is used we need to modify to use the business date.
- Replace system date with tenant date or business date (exceptions may apply)
- Add missing Value dates and Posting dates to entities
- Having a generic naming conventions for JPA fields and DB fields
- Renaming the fields accordingly
- Evaluate value date (transaction date) and posting date (submitted on date), created on date usages
- Jobs to be checked and modified accordingly
- Native queries to be checked and modified accordingly
- Reports to be checked and modified accordingly
- Every table where update is supported the AbstractAuditableCustom should be implemented
- Amend Transactions and Journal entries date handling to fit for business date concept
- For audit fields we shall introduce timezoned datetimes and store them in database accordingly
  - Storing DATETIME fields without Timezone is potential problem due to the daylight savings
  - Also some external libs (like Quartz) are using system timezone and Fineract will using Tenant timezone for audit fields. To be able to distinct them in DB we shall use DATETIME with TIMESTAMP column types and use timezoned java time objects in the application

Reliable event framework

Fineract is capable of generating and raising events for external consumers in a reliable way. This section is going to describe all the details on that front with examples.

Framework capabilities

ACID (transactional) guarantee

The event framework must support ACID guarantees on the business operation level.

Let’s see a simple use-case:

1. A client applies to a loan on the UI
2. The loan is created on the server
3. A loan creation event is raised

What happens if step 3 fails? Shall it fail the original loan creation process?

What happens if step 2 fails but step 3 still gets executed? We're raising an event for a loan that hasn't been created in reality.

Therefore, raising an event is tied to the original business transaction to ensure the data that's getting written into the database along with the respective events are saved in an all-or-nothing fashion.

Messaging integration

The system is able to send the raised events to downstream message channels. The current implementation supports the following message channels:

- ActiveMQ

Ordering guarantee

The events that are raised will be sent to the downstream message channels in the same order as they were raised.

Delivery guarantee

The framework supports the at-least-once delivery guarantee for the raised events.

Reliability and fault-tolerance

In terms of reliability and fault-tolerance, the event framework is able to handle the cases when the downstream message channel is not able to accept events. As soon as the message channel is back to operational, the events will be sent again.

Selective event producing

Whether or not an event must be sent to downstream message channels for a particular Fineract instance is configurable through the UI and API.

Standardized format

All the events sent to downstream message channels are conforming a standardized format using Avro schemas.

Extendability and customizations

The event framework is capable of being easily extended with new events for additional business operations or customizing existing events.

Ability to send events in bulk

The event framework makes it possible to sort of queue events until they are ready to be sent and

send them as a single message instead of sending each event as a separate, individual one.

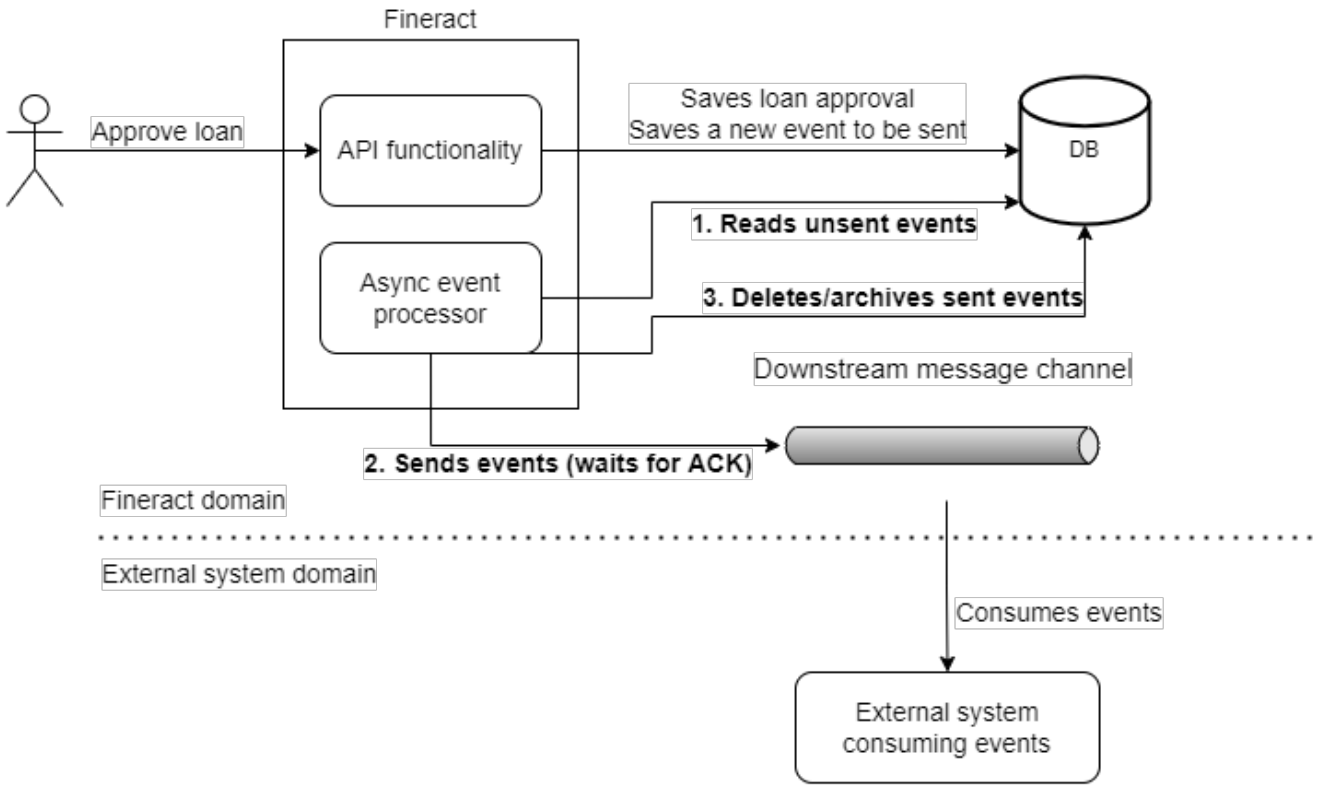
For example during the COB process, there might be events raised in separate business steps which needs to be sent out but they only need to be sent out at the end of the COB execution process instead of one-by-one.

Architecture

Intro

On a high-level, the concept looks the following. An event gets raised in a business operation. The event data gets saved to the database - to ensure ACID guarantees. An asynchronous process takes the saved events from the database and puts them onto a message channel.

The flow can be seen in the following diagram:



Foundational business events

The whole framework is built upon an existing infrastructure in Fineract; the Business Events.

As a quick recap, Business Events are Fineract events that can be raised at any place in a business operation using the `BusinessEventNotifierService`. Callbacks can be registered when a certain type of Business Event is raised and other business operations can be done. For example when a Loan gets disbursed, there's an interested party doing the Loan Arrears Aging recalculation using the Business Event communication.

The nice thing about the Business Events is that they are tied to the original transaction which means if any of the processing on the subscriber's side fail, the entire original transaction will be rolled back. This was one of the requirements for the Reliable event framework.

Event database integration

The database plays a crucial part in the framework since to ensure transactionality, - without doing proper transaction synchronization between different message channels and the database - the framework is going to save all the raised events into the same relational database that Fineract is using.

Database structure

The database structure looks the following

Name	Type	Description	Example
`id`	number	Auto incremented ID. Not null.	`1`
`type`	text	The event type as a string. Not null.	`LoanApprovedBusinessEvent`
`schema`	text	The fully qualified name of the schema that was used for the data serialization, as a string. Not null.	`org.apache.fineract.avro.loan.v1.LoanAccountDataV1`
`data`	BLOB (MySQL/MariaDB), BYTEA (PostgreSQL)	The event payload as Avro binary. Not null.	
`created_at`	timestamp	UTC timestamp when the event was raised. Not null.	`2022-09-06 14:20:10.148627 +00:00`
`status`	text	Enum text representing the status of the external event. Not null, indexed.	`TO_BE_SENT`, `SENT`
`sent_at`	timestamp	UTC timestamp when the event was sent.	`2022-09-06 14:30:10.148627 +00:00`

idempotency_key	text	Randomly generated UUID upon inserting a row into the table for idempotency purposes. Not null.	68aed085-8235-4722-b27d-b38674c19445
business_date	date	The business date to when the event was generated. Not null, indexed.	2022-09-05

The above database table contains the unsent events which later on will be sent by an asynchronous event processor.

Upon successfully sending an event, the corresponding statuses will be updated.

Avro schemas

For serializing events, Fineract is using Apache Avro. There are 2 reasons for that:

- More compact storage since Avro is a binary format
- The Avro schemas are published with Fineract as a separate JAR so event consumers can directly map the events into POJOs

There are 3 different levels of Avro schemas used in Fineract for the Reliable event framework which are described below.

Standard event schema

The standard event schema is for the regular events. These schemas are used when saving a raised event into the database and using the Avro schema to serialize the event data into a binary format.

For example the OfficeDataV1 Avro schema looks the following:

▼ OfficeDataV1.avsc

```
{
    "name": "OfficeDataV1",
    "namespace": "org.apache.fineract.avro.office.v1",
    "type": "record",
    "fields": [
        {
            "default": null,
            "name": "id",
            "type": [
                "null",
                "long"
            ]
```

```
        },
        {
            "default": null,
            "name": "name",
            "type": [
                "null",
                "string"
            ]
        },
        {
            "default": null,
            "name": "nameDecorated",
            "type": [
                "null",
                "string"
            ]
        },
        {
            "default": null,
            "name": "externalId",
            "type": [
                "null",
                "string"
            ]
        },
        {
            "default": null,
            "name": "openingDate",
            "type": [
                "null",
                "string"
            ]
        },
        {
            "default": null,
            "name": "hierarchy",
            "type": [
                "null",
                "string"
            ]
        },
        {
            "default": null,
            "name": "parentId",
            "type": [
                "null",
                "long"
            ]
        },
        {
            "default": null,
```

```
            "name": "parentName",
            "type": [
                "null",
                "string"
            ]
        },
        {
            "default": null,
            "name": "allowedParents",
            "type": [
                "null",
                {
                    "type": "array",
                    "items": "org.apache.fineract.avro.office.v1.OfficeDataV1"
                }
            ]
        }
    ]
}
```

Event message schema

The event message schema is just a wrapper around the standard event schema with extra metadata for the event consumers.

Since Avro is strongly typed, the event content needs to be first serialized into a byte sequence and that needs to be wrapped around.

This implies that for putting a single event message onto a message queue for external consumption, data needs to be serialized 2 times; this is the 2-level serialization.

1. Serializing the event
2. Serializing the already serialized event into an event message using the message wrapper

The message schema looks the following:

MessageV1.avsc

```
{
    "name": "MessageV1",
    "namespace": "org.apache.fineract.avro",
    "type": "record",
    "fields": [
        {
            "name": "id",
            "doc": "The ID of the message to be sent",
            "type": "int"
        },
        {
            "name": "source",
```

```
            "doc": "A unique identifier of the source service",
            "type": "string"
        },
        {
            "name": "type",
            "doc": "The type of event the payload refers to. For example
LoanApprovedBusinessEvent",
            "type": "string"
        },
        {
            "name": "category",
            "doc": "The category of event the payload refers to. For example LOAN",
            "type": "string"
        },
        {
            "name": "createdAt",
            "doc": "The UTC time of when the event has been raised; in
ISO_LOCAL_DATE_TIME format. For example 2011-12-03T10:15:30",
            "type": "string"
        },
        {
            "name": "businessDate",
            "doc": "The business date when the event has been raised; in
ISO_LOCAL_DATE format. For example 2011-12-03",
            "type": "string"
        },
        {
            "name": "tenantId",
            "doc": "The tenantId that the event has been sent from. For example
default",
            "type": "string"
        },
        {
            "name": "idempotencyKey",
            "doc": "The idempotency key for this particular event for consumer de-
duplication",
            "type": "string"
        },
        {
            "name": "dataschema",
            "doc": "The fully qualified name of the schema of the event payload. For
example org.apache.fineract.avro.loan.v1.LoanAccountDataV1",
            "type": "string"
        },
        {
            "name": "data",
            "doc": "The payload data serialized into Avro bytes",
            "type": "bytes"
        }
    ]
```

```
}
```

Bulk event schema

The bulk event schema is used when multiple events are supposed to be sent together. This schema is used also when serializing the data for the database storing but the idea is quite simple. Have an array of other event schemas embedded into it.

Since Avro is strongly typed, the array within the bulk event schema is an array of `MessageV1` schemas. That way the consumers can decide which events they want to deserialize and which don't.

This elevates the regular 2-level serialization/deserialization concept up to a 3-level one:

1. Serializing the standard events
2. Serializing the standard events into a bulk event
3. Serializing the bulk event into an event message

Versioning

Avro is quite strict with changes to an existing schema and there are a number of compatibility modes available.

Fineract keeps it simple though. Version numbers - in the package names and in the schema names - are increased with each published modification; meaning that if the OfficeDataV1 schema needs a new field and the `OfficeDataV1` schema has been published officially with Fineract, a new `OfficeDataV2` has to be created with the new field instead of modifying the existing schema.

This pattern ensures that a certain event is always deserialized with the appropriate schema definition, otherwise the deserialization could fail.

Code generation

The Avro schemas are described as JSON documents. That's hardly usable directly with Java hence Fineract generates Java POJOs from the Avro schemas. The good thing about these POJOs is the fact that they can be serialized/deserialized in themselves without any magic since they have a `toByteBuffer` and `fromByteBuffer` method.

From POJO to ByteBuffer:

```
LoanAccountDataV1 avroDto = ...
ByteBuffer buffer = avroDto.toByteBuffer();
```

From ByteBuffer to POJO:

```
ByteBuffer buffer = ...
LoanAccountDataV1 avroDto = LoanAccountDataV1.fromByteBuffer(buffer);
```

The ByteBuffer is a stateful container and needs to be handled carefully. Therefore Fineract has a built-in ByteBuffer to byte array converter; `ByteBufferConverter`.

Downstream event consumption

When consuming events on the other side of the message channel, it's critical to know which events the system is interested in. With the multi-level serialization, it's possible to deserialize only parts of the message and decide based on that whether it makes sense for a particular system to deserialize the event payload more.

Whether events are important can be decided based on:

- the `type` attribute in the message
- the `category` attribute in the message
- the `dataschema` attribute in the message

These are the main attributes in the message wrapper one can use to decide whether an event message is useful.

If the event needs to be deserialized, the next step is to find the corresponding schema definition. That's going to be sent in the `dataschema` attribute within the message wrapper. Since the attribute contains the fully-qualified name of the respective schema, it can be easily resolved to a Class object. Based on that class, the payload data can be easily deserialized using the `fromByteBuffer` method on every generated schema POJO.

Message ordering

One of the requirements for the framework is to provide ordering guarantees. All the events have to conform a happens-before relation.

For the downstream consumers, this can be verified by the `id` attribute within the messages. Since it's going to be a strictly-monotonic numeric sequence, it can be used for ordering purposes.

Event categorization

For easier consumption, the terminology event category is introduced. This is nothing else but the bounded context an event is related to.

For example the LoanApprovedBusinessEvent and the LoanWaiveInterestBusinessEvent are both related to the Loan bounded contexts.

The category in which an event resides in is included in the message under the `category` attribute.

The existing event categories can be found under the Event categories section.

Asynchronous event processor

The events stored in the database will be picked up and sent by a regularly executed job.

This job is a Fineract job, scheduled to run for every minute and will pick a number of events in order. Those events will be put onto the downstream message channel in the same order as they were raised.

Purging events

The events database table is going to grow continuously. That's why Fineract has a purging functionality in place that's gonna delete old and already sent events.

It's implemented as a Fineract job and is disabled by default. It's called TBD.

Usage

Using the event framework is quite simple. First, it has to be enabled through properties or environment variable.

The respective options are the following:

- the `fineract.events.external.enabled` property
- the `FINERACT_EXTERNAL_EVENTS_ENABLED` environment variable

These configurations accept a boolean value; `true` or `false`.

The key component to interact with is the `BusinessEventNotifierService#notifyPostBusinessEvent` method.

Raising events

Raising events is really easy. An instance of a BusinessEvent interface is needed, that's going to be the event. There are plenty of them available already in the Fineract codebase.

And that's pretty much it. Everything else is taken care of in terms of event data persisting and later on putting it onto a message channel.

An example of event raising:

```
@Override
public CommandProcessingResult createClient(final JsonCommand command) {
    ...
    businessEventNotifierService.notifyPostBusinessEvent(new
ClientCreateBusinessEvent(newClient));
    ...
    return ...;
}
```

The above code is copied from the `ClientWritePlatformServiceJpaRepositoryImpl` class.

Example event message content

Since the message is serialized into binary format, it's hard to represent in the documentation therefore here's a JSON representation of the data, just as an example.

```
{
    "id": 121,
    "source": "a65d759d-04f9-4ddf-ac52-34fa5d1f5a25",
    "type": "LoanApprovedBusinessEvent",
    "category": "Loan",
    "createdAt": "2022-09-05T10:15:30",
    "tenantId": "default",
    "idempotencyKey": "abda146d-68b5-48ca-b527-16d2b7c5daef",
    "dataschema": "org.apache.fineract.avro.loan.v1.LoanAccountDataV1",
    "data": "..."
}
```

The source attribute refers to an ID that's identifying the producer service. Fineract will regenerate this ID upon each application startup.

Raising bulk events

Raising bulk events is really easy as well. The 2 key methods are:

- `BusinessEventNotifierService#startExternalEventRecording`
- `BusinessEventNotifierService#stopExternalEventRecording`

First, you have to start recording your events. This recording will be applied for the current thread. And then you can raise as many events as you want with the regular `BusinessEventNotifierService#notifyPostBusinessEvent` method, but they won't get saved to the database immediately. They'll get "recorded" into an internal buffer.

When you stop recording using the method above, all the recorded events will be saved as a bulk event to the database; and serialized appropriately.

From then on, the bulk event works just like any of the event. It'll be picked up by the processor to send it to a message channel.

Event categories

TBD

Selective event producing

TBD

Customizations

The framework provides a number of customization options:

- Creating new events (that's already given by the Business Events)
- Creating new Avro schemas
- Customizing what data gets serialized for existing events

In the upcoming sections, that's what going to be discussed.

Creating new events

Creating new events is super easy. Just create an implementation of the `BusinessEvent` interface and that's it.

From then on, you can raise those events in the system, although you can't publish them to an external message channel. If you have the event framework enabled, it's going to fail with not finding the appropriate serializer for your business event.

There are existing serializers which might be able to handle your new event. For example the `LoanBusinessEventSerializer` is capable of handling all `LoanBusinessEvent` subclasses so there's no need to create a brand new serializer.

The interface looks the following:

`BusinessEvent.java`

```
public interface BusinessEvent<T> {

    T get();

    String getType();

    String getCategory();

    Long getAggregateRootId();
}
```

Quite simple. The `get` method should return the data you want to pass within the event instance. The `getType` method returns the name of the business event that's gonna be saved as the `type` into the database.

Creating a new business event only means that it can be used for raising an event. To make it compatible with the event framework and to be sent to a message channel, some extra work is needed which are described below.

Creating new Avro schemas and serializers

First let's talk about the event serializers because that's what's needed to make a new event compatible with the framework.

The serializer has a special interface, `BusinessEventSerializer`.

`BusinessEventSerializer.java`

```
public interface BusinessEventSerializer {

    <T> boolean canSerialize(BusinessEvent<T> event);

    <T> byte[] serialize(BusinessEvent<T> rawEvent) throws IOException;

    Class<? extends GenericContainer> getSupportedSchema();
}
```

An implementation of this interface shall be registered as a Spring bean, and it'll be picked up automatically by the framework.

You can look at the existing serializers for implementation ideas.

New Avro schemas can be easily created. Just create a new Avro schema file in the `fineract-avro-schemas` project under the respective bounded context folder, and it will be picked up automatically by the code generator.

BigDecimal support in Avro schemas

Apache Avro by default doesn't support complex types like a BigDecimal. It has to be implemented using a custom snippet like this:

```
{
    "logicalType": "decimal",
    "precision": 20,
    "scale": 8,
    "type": "bytes"
}
```

It's a 20 precision and 8 scale BigDecimal.

Obviously it's quite challenging to copy-paste this snippet to every single BigDecimal field, so there's a customization in place for Fineract. The type `bigdecimal` is supported natively, and you're free to use it like this:

```
{
    "default": null,
    "name": "principal",
```

```
    "type": [
        "null",
        "bigdecimal"
    ]
}
```

This `bigdecimal` type will be simple replaced with the BigDecimal snippet showed above during the compilation process.

Custom data serialization for existing events

In case there's a need some extra bit of information within the event message that the default serializers are not providing, you can override this behavior by registering a brand-new custom serializer (as shown above).

Since there's a priority order of serializers, the only thing the custom serializer need to do is to be annotated by the `@Order` annotation or to implement the `Ordered` interface.

An example custom serializer with priority looks the following:

```
@Component
@RequiredArgsConstructor
@Order(Ordered.HIGHEST_PRECEDENCE)
public class CustomLoanBusinessEventSerializer implements BusinessEventSerializer {
    ...

    @Override
    public <T> boolean canSerialize(BusinessEvent<T> event) {
        return ...;
    }

    @Override
    public <T> byte[] serialize(BusinessEvent<T> rawEvent) throws IOException {
        ...
        ByteBuffer buffer = avroDto.toByteBuffer();
        return byteBufferConverter.convert(buffer);
    }

    @Override
    public Class<? extends GenericContainer> getSupportedSchema() {
        return ...;
    }
}
```

All the default serializers are having `Ordered.LOWEST_PRECEDENCE`.

Appendix A: Properties and environment variables

Property name	Environment variable	Default value	Description
`fineract.events.external.enabled`	`FINERACT_EXTERNAL_EVENTS_ENABLED`	`false`	Whether the external event sending is enabled or disabled.