:years: 2015-2022

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.

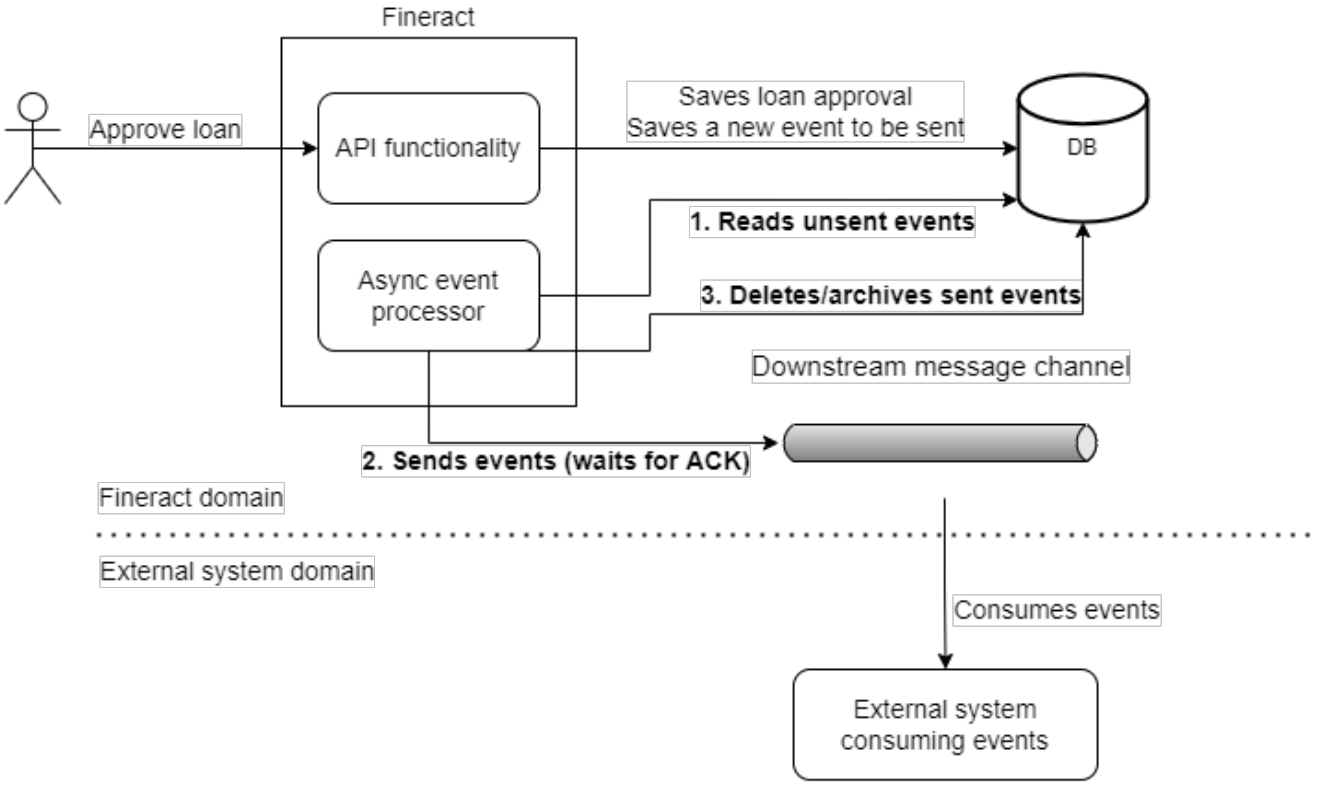
For example during the COB process, there might be events raised in separate business steps which needs to be sent out but they only need to be sent out at the end of the COB execution process instead of one-by-one.

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.