Working with transactions in modules based on RMB

Current approach to working with transactions in RMB

To enable the work with a database in FOLIO project there is a custom solution implemented on top of the VERT.X Postgres Client. The main feature of working with RMB and VERT.X is the usage of the asynchronous approach. Sequential execution of operations requires handling the completion of each operation and occurring errors. Each subsequent operation can be executed only if the previous one is succeeded. In order to maintain data consistency there is a need to execute the operations in transaction and be able to rollback the changes in case an error occurred. At the moment, this possibility is implemented as follows:

1. *A database connection object is created and the SQL command “BEGIN” is executed
2. The connection object is passed as a parameter to the Postgres client's methods and, accordingly, all commands are executed within a single connection
3. All errors are handled and Futures are succeeded
4. If an error occurs, rollback must be explicitly called
5. At the end of the transaction, the endTransaction() method must be explicitly called
6. *After the transaction is ended, the SQL command "COMMIT" is executed

The First and the last operations RMB PostgresClient does automatically

Example method with two operation in scope of one transaction

```
public Future<Void> example() {
  Future future = Future.future();
  PostgresClient client = PostgresClient.getInstance(vertx, tenantId);
  // start tx
  client.startTx(tx -> {
    // first operation
    client.get(tx, "upload_definition", UploadDefinition.class, new Criterion(), true, false, getHandler -> {
      if (getHandler.succeeded()) {
        // second operation
        client.save(tx, "upload_definition", UUID.randomUUID().toString(), getHandler.result(), saveHandler -> {
          if (saveHandler.succeeded()) {
            client.endTx(tx, endHandler -> {
              if (endHandler.succeeded()) {
                future.succeeded();
              } else {
                client.rollbackTx(tx, rollbackHandler -> {
                  future.fail(getHandler.cause());
                });
              }
            });
          } else {
            client.rollbackTx(tx, rollbackHandler -> {
              future.fail(getHandler.cause());
            });
          }
        });
      } else {
        client.rollbackTx(tx, rollbackHandler -> {
          future.fail(getHandler.cause());
        });
      }
    });
  });
  return future;
}
```

Locking tables in a database

When developing a slightly more complex business logic, the difficulty arises in the fact that certain operations may take some time and, accordingly, at this moment there is a possibility that it will be necessary to process another such request. Without locking a record in the database, there is a high probability of “lost changes” when the second request overwrites the changes made by the first one. Since VERTX.X is asynchronous, any locks and synchronous code executions are unacceptable, and the Persistence Context is absent. The most obvious solution is to use locks on the record in the database using the “SELECT FOR UPDATE” statement. Accordingly, to perform a safe update of the record in the database, you should:

1. Create Transaction Object

2. Select data from database for update (using “SELECT FOR UPDATE”)
3. Do some kind of business logic operation
4. Update entity in database
5. Complete transaction

Such scenario was needed at mod-data-import for file upload functionality.

Steps for file upload example:

- Load UploadDefinition from database
- Check for FileDefinition
- Save file to the local storage
- Update UploadDefinition status
- Save changed UploadDefinition entity into the database

Upload Definition mutator interface for callback usage

```
/**
 * Functional interface for change UploadDefinition in blocking update statement
 */
@FunctionalInterface
public interface UploadDefinitionMutator {
  /**
   * @param definition - Loaded from DB UploadDefinition
   * @return - changed Upload Definition ready for save into database
   */
  Future<UploadDefinition> mutate(UploadDefinition definition);
}
```

Method for update entity with record locking

```
public Future<UploadDefinition> updateBlocking(String uploadDefinitionId, UploadDefinitionMutator mutator) {
  Future<UploadDefinition> future = Future.future();
  String rollbackMessage = "Rollback transaction. Error during upload definition update. uploadDefinitionId" +
uploadDefinitionId;
  pgClient.startTx(tx -> {
    try {
      StringBuilder selectUploadDefinitionQuery = new StringBuilder("SELECT jsonb FROM ")
        .append(schema)
        .append(".")
        .append(UPLOAD_DEFINITION_TABLE)
        .append(" WHERE _id ='")
        .append(uploadDefinitionId).append("' LIMIT 1 FOR UPDATE;");
      pgClient.execute(tx, selectUploadDefinitionQuery.toString(), selectResult -> {
        if (selectResult.failed() || selectResult.result().getUpdated() != 1) {
          pgClient.rollbackTx(tx, r -> {
            logger.error(rollbackMessage, selectResult.cause());
            future.fail(new NotFoundException(rollbackMessage));
          });
        } else {
          Criteria idCrit = new Criteria();
          idCrit.addField(UPLOAD_DEFINITION_ID_FIELD);
          idCrit.setOperation("=");
          idCrit.setValue(uploadDefinitionId);
          pgClient.get(tx, UPLOAD_DEFINITION_TABLE, UploadDefinition.class, new Criterion(idCrit), false, true,
uploadDefResult -> {
            if (uploadDefResult.failed()
              || uploadDefResult.result() == null
              || uploadDefResult.result().getResultInfo() == null
              || uploadDefResult.result().getResultInfo().getTotalRecords() < 1) {
              pgClient.rollbackTx(tx, r -> {
                logger.error(rollbackMessage);
                future.fail(new NotFoundException(rollbackMessage));
              });
            } else {
              try {
                UploadDefinition definition = uploadDefResult.result().getResults().get(0);
                mutator.mutate(definition)
                  .setHandler(onMutate -> {
                    if (onMutate.succeeded()) {
                      try {
                        CQLWrapper filter = new CQLWrapper(new CQL2PgJSON(UPLOAD_DEFINITION_TABLE + ".jsonb"),
"id==" + definition.getId());
                        pgClient.update(tx, UPLOAD_DEFINITION_TABLE, onMutate.result(), filter, true,
updateHandler -> {
                          if (updateHandler.succeeded() && updateHandler.result().getUpdated() == 1) {
                            pgClient.endTx(tx, endTx -> {
                              if (endTx.succeeded()) {
                                future.complete(definition);
                              } else {
                                logger.error(rollbackMessage);
                                future.fail("Error during updating UploadDefinition with id: " +
uploadDefinitionId);
                              }
                            });
                          } else {
                            pgClient.rollbackTx(tx, r -> {
                              logger.error(rollbackMessage, updateHandler.cause());
                              future.fail(updateHandler.cause());
                            });
                          }
                        });
                      } catch (Exception e) {
                        pgClient.rollbackTx(tx, r -> {
                          logger.error(rollbackMessage, e);
                          future.fail(e);
                        });
                      }
                    } else {
                      pgClient.rollbackTx(tx, r -> {
                        logger.error(rollbackMessage, onMutate.cause());
                        future.fail(onMutate.cause());
                      });
                    }
                  });
              } catch (Exception e) {
                pgClient.rollbackTx(tx, r -> {
                  logger.error(rollbackMessage, e);
                  future.fail(e);
                });
              }
            }
          });
        }
      });
    } catch (Exception e) {
      pgClient.rollbackTx(tx, r -> {
        logger.error(rollbackMessage, e);
        future.fail(e);
      });
    }
  });
  return future;
}
```

This UploadDefinitionDaoImpl.updateBlocking using `Future.compose` and `Future.setHandler`: https://github.com/julianladisch/mod-data-import/blob/future-compose/src/main/java/org/folio/dao/UploadDefinitionDaoImpl.java#L50-L112

Method with file upload logic

```
@Override
public Future<UploadDefinition> uploadFile(String fileId, String uploadDefinitionId, InputStream data,
OkapiConnectionParams params) {
  return uploadDefinitionService.updateBlocking(uploadDefinitionId, uploadDefinition -> {
    Future<UploadDefinition> future = Future.future();
    Optional<FileDefinition> optionalFileDefinition = uploadDefinition.getFileDefinitions().stream().filter
(fileFilter -> fileFilter.getId().equals(fileId))
      .findFirst();
    if (optionalFileDefinition.isPresent()) {
      FileDefinition fileDefinition = optionalFileDefinition.get();
      FileStorageServiceBuilder
        .build(vertx, tenantId, params)
        .map(service -> service.saveFile(data, fileDefinition, params)
          .setHandler(onFileSave -> {
            if (onFileSave.succeeded()) {
              uploadDefinition.setFileDefinitions(replaceFile(uploadDefinition.getFileDefinitions(), onFileSave.
result()));
              uploadDefinition.setStatus(uploadDefinition.getFileDefinitions().stream().allMatch
(FileDefinition::getLoaded)
                ? UploadDefinition.Status.LOADED
                : UploadDefinition.Status.IN_PROGRESS);
              future.complete(uploadDefinition);
            } else {
              future.fail("Error during file save");
            }
          }));
    } else {
      future.fail("FileDefinition not found. FileDefinition ID: " + fileId);
    }
    return future;
  });
}
```

Problems of the current approach

Using the asynchronous approach when working with a database, has a number of limitations and difficulties in when it comes to writing sequential or transactional business logic. Such as following:

- The inability to simultaneously run multiple operations within a single connection

```
public void test1() {
  PostgresClient client = PostgresClient.getInstance(vertx, "diku");
  client.startTx(tx -> {
    for (int i = 0; i < 5; i++) {
                // ConnectionStillRunningQueryException: [2] - There is a query still being run here -
race -> false
      client.save(tx, UPLOAD_DEFINITION_TABLE, UUID.randomUUID().toString(), new UploadDefinition().
withId(UUID.randomUUID().toString()), reply -> {
        if (reply.succeeded()) {
          System.out.println(reply.result());
        } else {
          System.out.println(reply.cause().getLocalizedMessage());
        }
      });
    }
  });
}
```

  - This is not a problem of RMB but a limitation of PostgreSQL: "After successfully calling `PQsendQuery`, call `PQgetResult` one or more times to obtain the results. `PQsendQuery` cannot be called again (on the same connection) until `PQgetResult` has returned a null pointer, indicating that the command is done." (from 34.4. Asynchronous Command Processing). "One thread restriction is that no two threads attempt to manipulate the same `PGconn` object at the same time. In particular, you cannot issue concurrent commands from different threads through the same connection object. (If you need to run concurrent commands, use multiple connections.)" (from 34.19. Behavior in Threaded Programs).
- A transaction is used within one connection and, accordingly, all subsequent actions must be performed in handlers and all errors must be manually processed. "Callback hell" as a result.

        - Using `Future.compose` and `Future.setHandler` avoids nested handlers, catches all unchecked exceptions and moves the error handling into a single place (`setHandler`). No callback hell.
        - Example: UploadDefinitionDaoImpl.updateBlocking in https://github.com/julianladisch/mod-data-import/blob/future-compose/src/main/java/org/folio/dao/UploadDefinitionDaoImpl.java#L50-L112
- There is no support for some kind of persistence context and all locks must be done manually in the database by separate requests
- There is no possibility to easily manage the transaction isolation level at the application level
- Since all objects are stored in JSON format in the database, there is no way to be sure that the data is stored correctly. At the moment there is an opportunity to save any JSON in any table
    - RMB validates the data when is arrives at the REST API.
- Because of the storage of objects in JSON format, it is not possible to build durable relationships between entities
    - It is possible using foreign keys, the Post Tenant API has `foreignKeys` to automatically create them.
- It is necessary either to store all the links in one big JSON object in one table or put them into the other tables and create primary keys that do not guarantee contact with another entity. Need to make several requests to load data by primary keys and insert them into the parent object
    - Use `LEFT JOIN` to get all data in a single query. Examples: users_groups_view created by a manual `CREATE VIEW`, instance_holding_item_view using Post Tenant API view support.
    - Use `jsonb_set` to merge the data into the parent object (Example: https://www.db-fiddle.com/f/umimX7fG7HE1zEFQAtzsDS/0)
- RMB don't have all needed methods for working with transactions and queries.
    - These are coded when needed. Please file an issue against RMB for each single method you actually need.
- RMB don't have method get() that using custom SQL script with transaction support
    - ✓ ~~RMB-304~~ - PostgresClient.get(txConn, sql, params, replyHandler) for SELECT in a transaction CLOSED

Possible solutions

- Refactoring of the RMB's PostgresClient and add a couple of new methods for updating with blocking, transactions and loading records with the blocking of the record itself in the scope of the transaction
    - Create a single approach to query data from the database. For now get, save, update, delete methods at PostgresClient have different query mechanisms: CQLWrapper, Criterion, UUID. Need to create a wrapper for the querying functionality or encapsulate this logic.
    - Add a possibility to run custom SQL with “SELECT” statements.
        - Use one of the `PostgresClient.select` methods.
    - Add methods for an update with row blocking in the table.
        - There are several non-transactional update methods and two transactional update methods. Please file an issue against RMB for each missing transactional update method that you need.
        - **RMB-388** - PostgresClient.getById with transaction, with "SELECT … FOR UPDATE" OPEN
    - Add batchUpdate method
        - There are 9 update methods. Please file an issue against RMB for each single method that you need with batch capability.
        - **RMB-374** - Add PostgresClient.updateBatch methods OPEN
    - Change saveBatch method. For now, it doesn’t work with ids in the table.
        - Is this **RMB-204** - add PostgresClient.saveBatchById OPEN what you need? If not please file an issue against RMB for each single method that you need.
    - Create a transaction helper for PostgresClient similar to rxjava 2 `SQLClientHelper` (PostgresClient doesn't support rxjava2, but creating a similar helper is straightforward).
- Investigate the possibility of using other database tools. Conduct a study on the compatibility of the vert.x framework and existing solutions for the database, perhaps even synchronous solutions.