Finance Data Model

Overview

Most of the finance schemas are well out of date and need to be updated. Recent clarifications on requirements have also introduced new complexities and challenges that need to be overcome. Given the scope of these changes and how interrelated the various schemas and APIs will be, creating a wiki page to capture this information seemed to make sense.

Schemas

Schemas exist for most of the financial records, but will need to be updated as they're out of date. The Acquisitions Interface Fields maybe be helpful to some degree, but isn't a direct 1-to-1 mapping with the schemas due to intricacies of the data model, e.g. internal/storage use are needed which aren't captured in that spreadsheet.

Acquisition Units

The complex relationships between the various record types makes it difficult to use an inheritance pattern like we've used in other apps. For this reason, most finance records will have an acquisitionUnits field. How these are applied is slightly different as well. The fiscal year detail view for instance displays ledgers, groups and funds (budgets). In order to keep things reasonably simple, if the user has the folio permission to view the fiscal year, and belongs to one of the acquisition units assigned to the fiscal year record, the ledger, group and fund (budget) data can also be viewed. The same concept applies across the board - e.g. ledgers detail view shows groups and funds (budgets). We will need to get creative with our APIs and database views to make this work.

Transactions

- Immutable
- Payments, credits, allocations, transfers, encumbrance (see below)
- Each of these will have their own API to allow for business logic rules, but may share a table in the storage layer.
- Since there isn't a directly link to a budget, Querying by toFundId, fromFundId, and fiscalYearId is needed to satisfy:
    - GET transactions by Budget
- Single transactions table, shared by multiple APIs

Schemas

transaction

Encumbrances

Property	Type	Default	Required	Notes
id	string		N	UUID - System generated if not specified
amount	number		Y	Ideally we could filter by a range of values; also be able to search by exact amount of transaction. NOTE: For encumbrances: This is initialAmountEncumbered - (amountAwaitingPayment + amountExpended)
description	string		N	
source	string		Y	The readable identifier of the record that resulted in the creation of this transaction
transactionType	string		Y	This describes the type of transaction
fromFundId	string		N	UUID of the fund money is moving from
toFundId	string		N	UUID of the fund money is moving to
fiscalYearId	string		Y	UUID of the fiscal year that the transaction is taking place in
sourceInvoiceId	string		N	UUID of the Invoice that triggered the creation of this transaction
sourceInvoiceLineId	string		N	UUID of the InvoiceLine that triggered the creation of this transaction. Needed to support an idempotent payments/credit API
sourceFiscalYearId	string		N	UUID of the fiscal year that triggered the creation of this transaction (Used during fiscal year rollover)
tags	tags		N	
currency	string		Y	currency code for this transaction - from the system currency.
paymentEncumbranceId	string		N	UUID of the encumbrance associated with this payment/credit taking place.
encumbrance	encumbrance		N	Encumbrance sub-object - holds encumbrance-specific information not applicable to other transaction types
awaitingPayment	awaiting_payment		N	Awaiting payment sub-object - holds awaiting payment-specific information
metadata	metadata		N	System generated metadata (createdBy/updatedBy/createdOn/etc.)

encumbrance

Property	Type	Default	Required	Notes
initialAmountEncumbered	number		Y	Shouldn't change once create
amountAwaitingPayment	number		N	
amountExpended	number		N	
status	string		Y	enum: Released, Unreleased
orderType	string		N	Taken from the purchase order. enum: One-time, Ongoing
subscription	boolean		N	Taken from the purchase order. Has implications for fiscal year rollover
reEncumber	boolean		N	Taken from the purchase order. Has implications for fiscal year rollover
sourcePurchaseOrderId	string		N	UUID of the purchase order associated with this encumbrance
sourcePoLineId	string		N	UUID of the poLine associated with this encumbrance

awaiting_payment

Property	Type	Default	Required	Notes
encumbranceId	string		N	UUID of the encumbrance to updated
amountAwaitingPayment	number		Y	The amount of money moving from encumbered awaitingPayment
releaseEncumbrance	boolean	false	Y	Whether or not remaining encumbered money should be released

invoice_transaction_summary

Property	Type	Default	Required	Notes
id	UUID		Y	UUID of the invoice these payments are associated with (Unique)
numAwaitngPayments	int		Y	Total number of awaiting payments created for this invoice upon invoice approval.
~~numEncumbrances~~	~~int~~		~~Y~~	~~Total number of encumbrance updates (encumbered awaitingPayment) for this invoice~~
numPaymentsCredits	int		Y	Total number of payments/credits expected for this invoice

temporary_invoice_transactions

Property	Type	Default	Required	Notes
id	UUID		N	UUID of the payment (Unique) - Generated by the system
transaction	transaction (jsonb)		Y	The payment or credit

NOTE: Since these are for internal consumption only, we might consider just using regular old tables for both of these instead of creating full blown schemas. (Update) - now that we're exposing invoice-transaction-summary APIs in the BL layer this may not make sense for that table anymore. I think we're going to have to create a schema

Potential Optimization: It might be worth pulling some information (e.g. fundId, amount, etc.) out into distinct columns of the temp table to make it easier /faster to apply the payments and update totals once they're all received...

order_transaction_summary

Property	Type	Default	Required	Notes
id	UUID		Y	UUID of the order these payments are associated with (Unique)
~~purchaseOrderId~~	~~UUID~~		~~Y~~	~~UUID of the order these payments are associated with (Unique)~~
numTransactions	int		Y	Total number of transactions (encumbrances) expected for this order

temporary_order_transactions

Property	Type	Default	Required	Notes
id	UUID		N	UUID of the transaction (Unique) - Generated by the system
purchaseOrderId	UUID		Y	UUID of the order these payments are associated with
transaction	transaction (jsonb)		Y	The transaction (encumbrance)

APIs

Business Logic Module

Method	Path	Request	Response	Description	Notes
POST	/finance/allocations	transaction	transaction	Create an allocation	calls POST /finance-storage/transactions
POST	/finance/credits	transaction	transaction	Create a credit	calls POST /finance-storage/transactions
POST	/finance/payments	transaction	transaction	Create a payment	calls POST /finance-storage/transactions
POST	/finance/transfers	transaction	transaction	Create a transfer	calls POST /finance-storage/transactions
POST	/finance /encumbrances	transaction	transaction	Create an encumbrance	calls POST /finance-storage/transactions

POST	/finance/pending-payments	transaction	transaction	Create an awaiting payment	calls POST /finance-storage/transactions
GET	/finance /transactions/<id>	NA	transaction	Get a transaction by Id	
GET	/finance /transactions	CQL Query	transaction_collection	Search/List transactions	
~~PUT~~	~~/finance /encumbrance /<id>~~	~~transaction~~	~~transaction~~	~~Update an encumbrance~~	~~Only allowed if transactionType == encumbrance. Needed for updating encumbrances (e.g. encumbered->awaiting payment)~~
POST	/finance/invoice-transaction-summaries	invoice_transaction_summary	invoice_transaction_summary	Create an invoice transaction summary	tells finance how many transactions (encumbrances/ payments & credits) to expect for a particular invoice. Create if a record doesn't already exist for the invoice, otherwise return existing record.
POST	/finance/order-transaction-summaries	order_transaction_summary	order_transaction_summary	Create an order transaction summary	tells finance how many transactions (encumbrances) to expect for a particular order.
~~POST~~	~~/finance/awaiting-payment~~	~~awaiting_payment~~	~~201~~	~~Move money from encumbered->awaitingPayment~~	~~Updates the specified encumbrance~~
POST	/finance/release-encumbrance /<encumbranceId>	NA	201	Release any remaining money encumbered back to the budget's available pool	Updates the specified encumbrance
~~GET~~	~~/finance/invoice-transaction-summaries /<invoiceId>~~	~~NA~~	~~invoice_transaction_summary~~	~~Get a invoice payment summary~~	
~~DELETE~~	~~/finance/invoice-transaction-summaries /<invoiceId>~~	~~NA~~	~~204~~	~~Delete an invoice-transaction-summary~~	

NOTE: While the various POST endpoints all talk to the same storage API, they live as separate endpoints to allow for business logic to differ between transaction types.

NOTE: Intentionally omitted DELETE transaction endpoint here since transactions are immutable. The storage module provides a DELETE endpoint for purposes of cleaning up API test artifacts, etc.

NOTE: May need to add a DELETE /finance/encumbrance/<id> to support the ability to cleanup encumbrances for orders that are removed/purged.

> **Update**: Probably not needed since this cleanup would be done by one of the business logic modules (mod-orders). I don't think we'll need to expose this functionality directly to the user/client.

Storage Module

Method	Path	Request	Response	Description	Notes
POST	/finance-storage/transactions	transaction	transaction	Create a transaction	Update transactions, budget(s), ledger(s) all within a DB transaction.
GET	/finance-storage/transactions /<id>	NA	transaction	Get a transaction by Id	
GET	/finance-storage/transactions	CQL Query	transaction_collection	Search/List transactions	
PUT	/finance-storage/transactions /<id>	transaction	transaction	Update an encumbrance	Only allowed if transactionType == encumbrance Update transactions, budget(s), ledger(s) all within a DB transaction.
DELETE	/finance-storage/transactions /<id>	NA	204	Delete a transaction	for internal/emergency admin use only.
POST	/finance-storage/invoice-transaction-summaries	invoice_transaction_summary	invoice_transaction_summary	Create an invoice transaction summary	tells finance how many transactions (awaitng_payments/ payments & credits) to expect for a particular invoice
GET	/finance-storage/invoice-transaction-summaries /<invoiceId>	NA	invoice_transaction_summary	Get a invoice transaction summary	
DELETE	/finance-storage/invoice-transaction-summaries /<invoiceId>	NA	204	Delete an invoice-transaction-summary	

POST	/finance-storage/order-transaction-summaries	order_transaction_summary	order_transaction_summary	Create an order transaction-summary	tells finance how many transactions (encumbrances) to expect for a particular order. Create if a record doesn't already exist for the order, otherwise return existing record.
GET	/finance-storage/order-transaction-summaries /<purchaseOrderId>	NA	order_transaction_summary	Get a order transaction summary	
DELETE	/finance-storage/order-transaction-summaries /<purchaseOrderId>	NA	204	Delete an order-transaction-summary	

Encumbrance as a Transaction

- needs to be viewed as a transaction, but can't be immutable
- ✗ Option A: make encumbrance an superset of transaction, store in separate table, and use views/joins to satisfy search/filter needs.
    - Some of the field names are unintuitive - "amount" for instance is vague and potentially confusing given that we have other fields: "initalAmountEncumbered", "amountAwaitingPayment", and "amountExpended"
- ✓ Option B: make encumbrance a special-case of transaction. This involves adding a bunch of optional fields to the transaction schema which would only be used for the encumbrance type of transaction.
    - Putting these extra fields in an "encumbrance" sub-object might make sense - would allow for some schema validation to be used instead of needing to do it all in code.
    - This would allow us to store all transaction types in a single table and avoid having to deal with views/joins and confusing field names.
    - ~~Need to implement a PUT endpoint for transactions, to support this approach, but can restrict its use (in code) to only transactions of type "encumbrance". I don't like putting even this much business logic in the storage layer, but it's fairly limited and allows for a cleaner design.~~
        - See below the API table about possibly not needing this endpoint.

Updating Allocated/Unavailable/Available and Other Running Totals

- ✗Option A: Have the business logic module perform calculations as money is moved
    - Harder to guarantee consistency/accuracy - is eventual consistency OK for finances?
    - Keeps business logic out of the storage module, consistent with approach taken in other acquisition apps
- ✗Option B: Perform summary/total calculations in the storage module and update all tables within a transaction. Either they all succeed, or the entire operation fails
    - Much better control over consistency/accuracy
    - Bleeds business logic into the storage module
    - Other business logic remains in mod-finance - e.g. acquisitionUnits, what can be modified vs what can't, etc.
- ✗Option C: Calculate these numbers on-the-fly in the business logic module
    - Difficult/inefficient to do if these numbers are needed to be shown in search results (they are)
    - Likely means we'll need a more complicated data model with more views and schemas
    - Since we're calculating these numbers on the fly, consistency is less of a problem... the numbers only live in one place
- ✓Option D: Hybrid of B and C
    - Calculate most of these totals in the storage module and update all tables in a transaction
    - For some special cases, we need to perform calculations on the fly
        - e.g. GET groups for a ledger/FY - Only sum the amounts from the funds in those groups which are part of the ledger for this FY...

Preventing Partially Paid Invoices

When an invoice is paid, numerous transactions (payments, credits) will be generated. Either all or none of these need to be successfully processed; we can't have partially paid invoices. This presents a challenge of scale. There could feasibly be hundreds of payments for an invoice. Several options were weight and the most attractive approach involves collecting or aggregating payments/credits in the storage layer until all are received, then processing them all at once in a database transaction. The other options are listed in the appendix.

This approach keeps the payment "simple" and consistent with the other transaction APIs. The storage layer performs some aggregation in the database of payments/credits based on the sourceInvoiceId specified in each payment. The flow looks something like this:

Introduce new tables:

- invoice_transaction_summary - Contains an invoiceId and the total number of transactions (payments/credits) expected when "paying" this invoice.
- temp_invoice_payments - Temporary storage of payments until all payments related to the invoice are received.

When an invoice is paid

- mod-invoice calls POST /finance/invoice-transaction-summary
    - mod-finance calls mod-finance-storage to create the entry if one doesn't already exist
- mod-invoice generates payments and calls POST /finance/payments
    - mod-finance checks calls POST /finance-storage/payments

When an payment is received,

- Persist the payment record in a temp_invoice_payments table
- Count the number of records in the temp_invoice_payments table. If this is the last payment, (the number of entries in the temp table == numTransactions)

        - In a database transaction:
            - Apply the payments/credits from the temp_invoice_payments table and perform the summary # updates (e.g. update budget, ledger, encumbrances, etc. - unavailable/available/encumbered/expended/etc.)
                - If everything works, cleanup the temp_invoice_payments table and return 200 OK
                - If something fails, no actual records are updated. The payment API call can fail and can be retried (POST payment /credit are idempotent)

When an "approved" invoice is removed:

- mod-invoice makes a call to mod-finance to ensure that the invoice_transaction_summary has been cleaned up. These records are small so even if on is orphaned (which shouldn't happen often if ever) it's not a huge deal.
- DELETE /finance-storage/invoice-transaction-summary/<invoiceId>

If a payment request fails

- mod-finance can safely retry it.
- After N failed tries an error can be returned to mod-invoice and/or the client who can later try to pay the invoice again once the (network?) problem is resolved.

"All or Nothing" Operations

There are several situations in acquisitions where a bunch of transactions need to happen all at once. Often the situation is such that either all of these transactions need to succeed, or none of them should. We first crossed this in the context of preventing partially paid invoices (see above). Fortunately, these cases are similar enough to one another that the design work and lessons learned during that work can be borrowed/reused. There scenarios can be split into two categories:

Transactions applying to an invoice

- When approving an invoice, money needs to move from encumbered to awaitingPayment in both the encumbrances and the corresponding budgets. It's also possible that encumbrances need to be released at this point, e.g. we encumbered $100 but the invoice is for $90, so we "release" $10 back to the budget's "available" pool.
    - **For tracking purposes we create awaiting payment for every fund distribution**
        - **amount= fundDistribution amount**
        - **awaitingPayment.encumbranceId = fundDistribution.encumbrance**
        - **NOTE: This means that the crossed-out statement below "The number of transactions should be the same for both of these events" is again true. In order to avoid complicated data migraiton scripts, we're adding a new field to invoice-transaction-summary for indicating the number of new awaiting payment created upon invoice approval. See the schema table for details.**
        - **NOTE: This also simplifies the calculations that need to occur when applying payments/credits as there should always be an associated awaiting payment.**
    - **In order to accomodate this we need to add awaitngPayment sub-object to transaction schema.**
- When paying an invoice, payments and credits are made. See the earlier section "Preventing Partially Paid Invoices" for details.

Here I think we can re-use the "invoice_transaction_summary" and temporary transaction table almost as-is for both of these cases. We might want to rename "invoice_transaction_summary" to a more generic "invoice_transaction_summary". What this means is that a mod-invoice would create the invoice summary record when the invoice is approved instead of when the invoice is paid. ~~The number of transactions should be the same for both of these events~~ The number of encumbrances might differ from the number of payments/credits, hence separate fields for the two counts. It's important to note that the invoice shouldn't be changing after it's approved. This is what allows us to calculate the number of payments/credits upon invoice approval. When determining if all of the payments/credits have been received, we'll need to also look at the transaction type to essentially ignore the "encumbrance" transactions in the temporary table. The temporary table can be cleaned up, but there's not guarantee on the timing of that, so we can't assume that only payments/credits will be present at the time of an invoice being paid.

Transactions applying to an Order

- When opening an order, encumbrances are created. We need to ensure that all encumbrances for an order are created before opening an order.
- When closing an order, encumbrances need to be released. That is if we encumbered money that we didn't end up spending, it needs to be "released" or moved back into the budget's "available" pool.

We looked at handling this in a similar fashion to creation of records in inventory; i.e. when opening the order we try to create all of the encumbrances, and if any of them fail, we return an appropriate error response and leave the order in "Pending" state. However, it was determined that was not a viable approach since it opens the door to having encumbrances in the system for "pending" orders.

The idea for handling this is similar to what we do above for invoices...

mod-finance-storage

- Create an order_transaction_summaries table and API to sit in front of it
- Create a temporary table for holding encumbrances (transactions)
- Add a link to the order (POL?) in the encumbrance sub-object of transaction. This is akin to sourceInvoiceId/sourceInvoiceLineId and is needed to accumulate all of the encumbrances associated with an order before applying changes.
- Once all encumbrances for an order are received, apply the changes in a DB transaction

These new tables will be used for both opening an order (create encumbrances) as well as closing an order (releasing encumbrances). The encumbrance status field will help differentiate the transactions for each of these events in the temporary table.

Business Logic API changes

The current design includes a generic PUT /finance/encumbrances/<id> endpoint that would be sufficient for this approach. However, this generic PUT endpoint doesn't really allow the caller to specify exactly what they're trying to do, making it difficult to properly validate the input. By incorporating specific endpoints for the various actions, we clarify the context. By having these endpoints take input that isn't a full blown transaction, we can save ourselves a lot of validation.

- POST /finance/awaiting-payment - takes a new schema w/ encumbranceId, amount. Retrieves the specified encumbrance from storage, applies the change and sends the updated record back to storage.
- POST /finance/release-encumbrance/<id> - no request payload. Retrieves the specified encumbrance, applies the changes and sends the updated record back to storage.

NOTE: These endpoints are slightly different from the rest of the finance API in that they're command endpoints, and aren't very RESTful in their design. That said, there is some precedence for this in several other places in acquisitions, e.g. mod-orders receiving/check-in endpoints

Calculations

Allocations

Update Budget identified by the transactions fiscal year (fiscalYearId) and source fund (fromFundId)

- If "fromFundId" is present,
    - allocated decreases by transaction amount
    - available decreases by transaction amount
    - recalculate overEncumbered
    - recalculate overExpended

Update Budget identified by the transaction's fiscal year (fiscalYearId) and the destination fund (toFundId)

- allocated increases by the transaction amount
- available increases by the transaction amount
- recalculate overEncumbered
- recalculate overExpended

Update LedgerFY identified by the transaction's fiscal year (fiscalYearId) and the source fund (fromFundId)

- If "fromFundId" is present
    - allocated decreases by transaction amount
    - available decreases by transaction amount

Update LedgerFY identified by the transaction's fiscal year (fiscalYearId) and the source fund (toFundId)

- allocated increases by the transaction amount
- available increases by the transaction amount

Transfers

Update Budget identified by the transactions fiscal year (fiscalYearId) and source fund (fromFundId)

- If "fromFundId" is present,
    - available decreases by the transaction amount
    - ~~unavailable increases by the transaction amount~~
    - recalculate overEncumbered
    - recalculate overExpended

Update Budget identified by the transactions fiscal year (fiscalYearId) and source fund (toFundId)

- available increases by the transaction amount
- ~~unavailable decreases by the transaction amount~~
- recalculate overEncumbered
- recalculate overExpended

Update LedgerFY identified by the transactions fiscal year (fiscalYearId) and source fund (fromFundId)

- If "fromFundId" is present,
    - available decreases by the transaction amount
    - ~~unavailable increases by the transaction amount~~

Update LedgerFY identified by the transactions fiscal year (fiscalYearId) and source fund (toFundId)

- available increases by the transaction amount
- ~~unavailable decreases by the transaction amount~~

Payments

Update the encumbrance identified by the transactions (paymentEncumbranceId)

- ~~awaitingPayment decreases by transaction amount (min 0)~~
- expended increases by transaction amount
- amount (effectiveEncumbrance) is recalculated

Delete related awaiting payment transaction

Update Budget identified by the transaction's fiscal year (fiscalYearId) and the source fund (fromFundId)

- ~~If there is an encumbrance associated with the payment~~
    - ~~awaitingPayment decreases by transaction amount (min 0)~~
- ~~Else~~
    - ~~available decreases by transaction amount (min 0)~~
    - ~~unavailable increases by the transaction amount~~
- Always
    - awaitingPayment decreases by transaction amount
    - expenditures increases by transaction amount
    - recalculate overExpended

~~Update LedgerFY identified by the transactions fiscal year (fiscalYearId) and source fund (fromFundId)~~

- ~~If there is no encumbrance is associated with the payment~~
    - ~~available decreases by the transaction amount (min 0)~~
    - ~~unavailable increases by the transaction amount~~

Credits

Update the encumbrance identified by the transactions (paymentEncumbranceId)

- expended decreases by transaction amount
- amount (effectiveEncumbrance) is recalculated
- ~~if(status == Released), the encumbered amount is immediately released (see update encumbrance section below)~~
    - ~~Update the budget identified by the transaction(encumbrance) fiscal year (fiscalYearId) and the source fund (fromFundId)~~
        - ~~encumbered decreases by the amount being released~~
        - ~~available increases by the amount being released~~
        - ~~unavailable decreases by the amount being released (min 0)~~

Delete related awaiting payment transaction

Update Budget identified by the transaction's fiscal year (fiscalYearId) and the destination fund (toFundId)

- ~~If there is a released encumbrance associated with the credit~~
    - ~~available increases by transaction amount~~
    - ~~unavailable decreases by transaction amount (min 0)~~
- ~~Else~~
    - ~~encumbered increases by the transaction amount~~
- Always
    - awaitingPayment increases by transaction amount
    - expenditures decreases by transaction amount
    - recalculate overExpended

~~Update LedgerFY identified by the transactions fiscal year (fiscalYearId) and destination fund (toFundId)~~

- ~~If there is a released encumbrance is associated with the credit~~
    - ~~available increases by the transaction amount~~
    - ~~unavailable decreases by the transaction amount (min 0)~~

Encumbrances

Upon Creation

- Update Budget identified by the transaction fiscal year (fiscalYearId) and the source fund (fromFundId)
    - encumbered increases by the transaction amount
    - available decreases by the transaction amount
    - unavailable increases by the transaction amount
    - overEncumbered is recalculated
- Update LedgerFY identified by the transaction fiscal year (fiscalYearId) and the source fund (fromFundId)
    - available decreases by the transaction amount
    - unavailable increases by the transaction amount

Upon Update

- ALWAYS DO THIS PART:
    - Update Budget identified by the transaction fiscal year (fiscalYearId) and the source fund (fromFundId)
        - TBD

  IN ADDITION TO ABOVE, DO THIS PART IF ENCUMBRANCE IS RELEASED:

    - If encumbrance.status = Released
        - Update the encumbrance
            - transaction.amount becomes 0 (save the original value for updating the budget)
        - Update the budget identified by the transaction fiscal year (fiscalYearId) and the source fund (fromFundId)
            - encumbered decreases by the amount being released
            - available increases by the amount being released
            - unavailable decreases by the amount being released (min 0)
        - **Update LedgerFY identified by the transactions fiscal year (fiscalYearId) and source fund (fromFundId)**
            - available increases by the transaction amount (min 0)
            - unavailable decreases by the transaction amount

Pending payments

Upon Creation

- if awaitngPayment.encumbranceId specified
    - ALWAYS DO THIS PART:
        - Update Budget identified by the transaction fiscal year (fiscalYearId) and the source fund (fromFundId)
            - encumbered decreases by the transaction.amount
            - awaitingPayment increases by the same amount
        - update related encumbrance:
            - transaction(encumbrance).amount is updated to (transaction(encumbrance.)amount - transaction(pending payment).amount)
        - if pendingPayment.amout > encumbrance.amount
            - Update the budget identified by the transaction(encumbrance) fiscal year (fiscalYearId) and the source fund (fromFundId)
                - available decreases by the pendingPayment.amout - encumbrance.amount
                - unavailable increases by the pendingPayment.amout - encumbrance.amount (min 0)
                - recalculate overExpended?
    - IN ADDITION TO ABOVE, DO THIS PART IF ENCUMBRANCE IS RELEASED:
        - If encumbrance.status = Released
            - Update the related encumbrance
                - transaction.amount becomes 0 (save the original value for updating the budget)
                - Update the budget identified by the transaction(encumbrance) fiscal year (fiscalYearId) and the source fund (fromFundId)
                    - encumbered decreases by the amount being released
                    - available increases by the amount being released
                    - unavailable decreases by the amount being released (min 0)
                - **Update LedgerFY identified by the transactions fiscal year (fiscalYearId) and source fund (fromFundId)**
                    - available increases by the transaction amount (min 0)
                    - unavailable decreases by the transaction amount
- if awaitngPayment.encumbranceId is empty
    - Update budget identified by the transaction fiscal year (fiscalYearId) and the source fund (fromFundId)
        - available decreases by transaction amount (min 0)
        - unavailable increases by the transaction amount
        - awaitingPayment increases by the transaction.amount
    - Update LedgerFY identified by the transactions fiscal year (fiscalYearId) and destination fund (toFundId)

        - available increases by the transaction amount
        - unavailable decreases by the transaction amount (min 0)

Example payment:

Budget:

allocated: 100

unavailable: 0

available: 100

encumbrance:

amount: 50

initialEncumbered: 50

Budget:

allocated: 100

unavailable: 50

available: 50

encumbered: 50

Pending payment:

amount: 51

encumbrance:

amount: 0

awaitingPayment: 51

initialEncumbered: 50

Budget:

allocated: 100

unavailable: 51

available: 49

encumbered: 0

awaitngPayment: 51

Payment:

amount: 51

encumbrance:

amount: 0

awaitingPayment: 0

initialEncumbered: 50

expended: 51

Budget:

allocated: 100

unavailable: 51

available: 49

encumbered: 0

awaitngPayment: 0

expended: 51

overExpended: rcalculate

Example credit:

Budget:

allocated: 100

unavailable: 0

available: 100

encumbrance:

amount: 50

initialEncumbered: 50

Budget:

allocated: 100

unavailable: 50

available: 50

encumbered: 50

Pending payment:

amount: -10

encumbrance:

amount: 50 + 10 = 60

awaitingPayment: 0

initialEncumbered: 50

Budget:

allocated: 100

unavailable: 40

available: 60

encumbered: 60

awaitngPayment: 0

Credit:

amount: 10

encumbrance:

amount: 50

awaitingPayment: 0

initialEncumbered: 50

expended: -10

Budget:

allocated: 100

unavailable: 40

available: 60

encumbered: 50

awaitngPayment: 0

expended: -10

overExpended: rcalculate

release:

encumbrance:

amount: 0

awaitingPayment: 0

initialEncumbered: 50

expended: -10

status: released

Budget:

allocated: 100

unavailable: max(40 -50, 0) = 0

available: 110

encumbered: 0

awaitngPayment: 0

expended: -10

overExpended: rcalculate

Without encumbrance:

Budget:

allocated: 100

unavailable: 0

available: 100

Pending payment:

amount: 50

Budget:

allocated: 100

unavailable: 50

available: 50

encumbered: 0

awaitngPayment: 50

expended: 0

Payment:

amount: 50

Budget:

allocated: 100

unavailable: 50

available: 50

encumbered: 0

awaitngPayment: 0

expended: 50

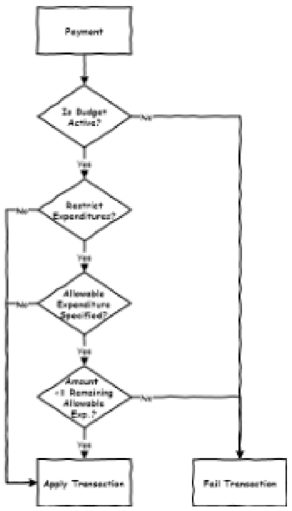
overExpended: rcalculate

Transaction Restrictions

It's often necessary to check if a transaction should be allowed to happen based on the current budget values. This section explains these restrictions and how they're applied.

Payments

There are several factors that come into play here. The flowchart below shows how we determine if a payment can be accepted or should be rejected.



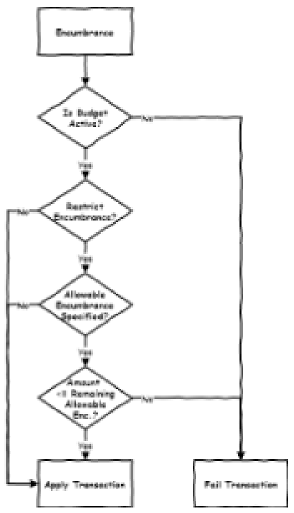
The "Remaining Allowable Exp." in the flowchart is calculated as follows:

[remaining amount we can expend] = (allocated * allowableExpenditure) - (allocated - (unavailable + available)) - (awaitingPayment + expended)

The "restrictExpenditures" flag actually lives in the ledger, so we'll need to find traverse: budget fund ledger.restrictExpenditures

Encumbrance

Creating encumbrances requires a similar set of checks as described above for payments.



The "Remaining Allowable Enc." in the flowchar is calculated as follows:

[remaining amount we can encumber] = (allocated * allowableEncumbered) - (encumbered + awaitingPayment + expended)

The "restrictEncumbrances" flag actually lives in the ledger, so we'll need to find traverse: budget fund ledger.restrictEncumbrances

Tags

Depending on the type of transaction, tags are in some cases inherited from another record.

- **Payments / Credits**
  - Inherit the tags that were associated with the invoice line
- **Encumbrances**
  - Inherit the tags that were associated with the order line if one is associated with the encumbrance
  - **DB: Yes**. Otherwise, Inherit the tags that were associated with the invoice line.~~Ann-Marie Breaux or Dennis Bridges to clarify this~~
- **Allocations / Transfers**
  - Don't inherit tags from anywhere - the user should be allowed to assign tags at the point of creating the allocation or transfer

FundTypes

- Simple controlled vocabulary

Schemas

fund_type

Property	Type	Default	Required	Notes
id	string		No	UUID - System generated if not specified
name	string		Yes	
count	string		No	The number of funds w/ this fund-type Not persisted (read only) - populated by mod-finance when listing fund-types

APIs

Business Logic Module

Method	Path	Request	Response	Description	Interface	Notes
POST	/finance /fund-types	fund_type	fund_type	Create a fund-type	finance. fund-types	

GET	/finance/fund-types/<id>	NA	fund_type	GET fund-type by Id	finance.fund-types	
GET	/finance/fund-types	CQL Query	collection<fund_type>	Search/List fund-types	finance.fund-types	Queries GET /finance-storage/fund-types, and augments the results with count /usage info. Will need to make additional calls to GET /finance-storage/funds?query=fundType=XYZ to get this information. These queries can happen in parallel.
PUT	/finance/fund-types/<id>	fund_type	fund_type	Update a fund-type	finance.fund-types	
DELETE	/finance/fund-types/<id>	NA	204	Delete a fund-type	finance.fund-types	

Storage Module

Method	Path	Request	Response	Description	Interface	Notes
POST	/finance-storage/fund-types	fund_type	fund_type	Create a fund-type	finance-storage.fund-types	
GET	/finance-storage/fund-types/<id>	NA	fund_type	GET fund-type by Id	finance-storage.fund-types	
GET	/finance-storage/fund-types	CQL Query	collection<fund_type>	Search/List fund-types	finance-storage.fund-types	
PUT	/finance-storage/fund-types/<id>	fund_type	fund_type	Update a fund-type	finance-storage.fund-types	
DELETE	/finance-storage/fund-types/<id>	NA	204	Delete a fund-type	finance-storage.fund-types	

Funds

- Record with info that doesn't change from FY to FY (name, code, acq. units, etc.)
- Foreign Keys;
    - funds.ledgerId = ledgers.id

Schemas

fund

Property	Type	Default	Required	Notes
id	string		No	UUID - System generated if not specified
name	string		Yes	e.g. African History
code	string		Yes	Unique. e.g. AFRICAHIST
fundStatus	string		Yes	enum - Active, Inactive, Frozen
fundTypeId	string		No	A descripter that allows the users to categorize funds and drive functionality in workflows like rollover etc. controlled vocab.
externalAccountNo	string		No	Corresponding account in the financial system. Will be recorded in payment generated as well.
description	string		No	
allocatedFromIds	array<string>		No	UUIDs of funds money can be allocated from
allocatedToIds	array<string>		No	UUIDs of funds money can be allocated to
tags	tags		No	
ledgerId	string		Yes	UUID of the ledger this fund belongs to
acqUnitIds	array<string>		No	Array of UUIDs corresponding to the acquisition units assigned to this fund.
metadata	metadata		No	System generated metadata (createdBy/updatedBy/createdOn/etc.)

compositeFund

This schema is essentially a fund w/ one additional field - groupIds. The business logic module's fund APIs will use this schema, and will manage the fund and group/fund relationships independently via calls to the storage APIs. This simplifies things on the UI end.

Property	Type	Default	Required	Notes
id	string		No	UUID - System generated if not specified
name	string		Yes	e.g. African History
code	string		Yes	Unique. e.g. AFRICAHIST
fundStatus	string		Yes	enum - Active, Inactive, Frozen
fundTypeId	string		No	A descripter that allows the users to categorize funds and drive functionality in workflows like rollover etc. controlled vocab.
externalAccountNo	string		No	Corresponding account in the financial system. Will be recorded in payment generated as well.
description	string		No	
allocatedFromIds	array<string>		No	UUIDs of funds money can be allocated from
allocatedToIds	array<string>		No	UUIDs of funds money can be allocated to
tags	tags		No	
ledgerId	string		Yes	UUID of the ledger this fund belongs to
acqUnitIds	array<string>		No	Array of UUIDs corresponding to the acquisition units assigned to this fund.
groupIds	array<string>		No	Array of UUIDs corresponding to the groups associated with this fund.
metadata	metadata		No	System generated metadata (createdBy/updatedBy/createdOn/etc.)

APIs

Business Logic Module

Method	Path	Request	Response	Description	Interface	Notes
POST	/finance/funds	composite_fund	composite_fund	Create a fund	finance.funds	Involves creating the fund as well as potientially adding groupFundFY records for the current fiscal year
GET	/finance/funds/<id>	NA	composite_fund	GET fund by Id	finance.funds	Needs to make multiple calls to gather not only the fund, but also the ids of the groups associated with this fund for the "current" fiscal year
GET	/finance/funds	CQL Query	collection<fund>	Search/List funds	finance.funds	proxy back to storage - NOTE that this returns a collection of fund not composite_fund
PUT	/finance/funds/<id>	composite_fund	201	Update a fund	finance.funds	May involve updating the fund and/or adding/removing groupFundFY records for the current fiscal year
DELETE	/finance/funds/<id>	NA	204	Delete a fund	finance.funds	Remove the fund and related groupFundFy records - may eventually become a soft delete for one or both record types.

NOTE: see the section below about determining the appropriate ("current") fiscal year.

Storage Module

Method	Path	Request	Response	Description	Interface	Notes
POST	/finance-storage/funds	fund	fund	Create a fund	finance-storage.funds	
GET	/finance-storage/funds/<id>	NA	fund	GET fund by Id	finance-storage.funds	
GET	/finance-storage/funds	CQL Query	collection<fund>	Search/List funds	finance-storage.funds	
PUT	/finance-storage/funds/<id>	fund	201	Update a fund	finance-storage.funds	
DELETE	/finance-storage/funds/<id>	NA	204	Delete a fund	finance-storage.funds	

Determining the "current" fiscal year

A few places in the fund API require the module to figure out the appropriate fiscal year to use when associating groups/funds. The following process should probably be implemented in a way that can be easily reused.

- Given provided fund, get the associated ledger using fund.ledgerId
    - Query fiscal years:
        - series = ledger.fiscalYearOne.series
        - currentDate >= startPeriod
        - currentDate < endPeriod
- If no fiscal year is returned for this period, look for next period
    - Query fiscal years:
        - series = ledger.fiscalYearOne.series
        - currentDate + 1 year >= startPeriod
        - currentDate + 1 year < endPeriod
- If still nothing, return an error

Groups

- Record with info that doesn't change from FY to FY
- A Group/Fund/FY table will be needed to track fund/group relationships on a per FY basis.
- Foreign keys:
    - group_fund_fiscal_years.budgetId = budgets.id (alias: budget)
    - group_fund_fiscal_years.fundId = funds.id (alias: fund)
    - group_fund_fiscal_years.fiscalYearId = fiscal_years.id (alias: fiscalYear)
    - group_fund_fiscal_years.groupId = groups.id (alias: group)
    - group_fund_fiscal_years.fundId funds.ledgerId ledgers.id (alias: ledger)

Schemas

group

Property	Type	Default	Required	Notes
id	string		N	UUID - System generated if not specified
name	string		Y	Unique
description	string		N	
code	string		Y	Unique
status	string		Y	enum: Active, Inactive, Frozen
acqUnitIds	array<string>		N	Array of UUIDs corresponding to the acquisition units assigned to this group.
metadata	metadata		N	System generated metadata (createdBy/updatedBy/createdOn/etc.

groupFundFY

Property	Type	Default	Required	Notes
id	string		N	UUID - System generated if not specified
groupId	string		Y	UUID of group
fiscalYearId	string		N	UUID of fiscal year Not required in schema, but will always be present in storage - added by BL module
fundId	string		Y	UUID of fund
budgetId	string		N	UUID of the budget - seems redundant but needed for cross-index querying
~~allocated~~	~~number~~		~~N~~	~~not persisted~~
~~unavailable~~	~~number~~		~~N~~	~~not persisted~~
~~available~~	~~number~~		~~N~~	~~not persisted~~

Note: Removing the summary fields as they're moved to the groupFySummary schema

groupFySummary

Property	Type	Default	Required	Notes
groupId	string		Y	UUID of group
fiscalYearId	string		Y	UUID of fiscal year
allocated	number		Y	not persisted

unavailable	number		Y	not persisted
available	number		Y	not persisted

Note: No id field as this is only used between the UI and business logic module

APIs

Business Logic Module

Method	Path	Request	Response	Description	Interface	Notes
POST	/finance/groups	group	group	Create a group	finance. groups	
GET	/finance/groups /<id>	NA	group	GET groupby Id	finance. groups	
GET	/finance/groups	CQL Query	collection<group>	Search/List groups	finance. groups	
PUT	/finance/groups /<id>	group	group	Update a group	finance. groups	
DELETE	/finance/groups /<id>	NA	204	Delete a group	finance. groups	
~~GET~~	~~/finance/groups /<id>/budgets~~	~~CQL Query,~~ ~~Fiscal Year Query (req)~~	~~collection<budget>~~	~~List budgets in this group~~	~~finance. groups~~	~~Queries view API: Group + Budget + Fund /Group/FY (by groupId, fiscalYearId, group. acqUnits)~~ • ~~returns array of budget~~ ~~Enforces acquisition units based on group's assignments, ignores budget/fund acq. units.~~ ~~The fiscalYear query arg is the UUID of a fiscal year record~~
POST	/finance/group-fund-fiscal-years	group_fund_fy	204	Create a groupFundFY	finance.group-fund-fiscal-years	May not be needed in light of the new /finance /fund API w/ composite_fund
GET	/finance/group-fund-fiscal-years	CQL Query	collection<group_fund_fy>	Search/List groupFundFY	finance.group-fund-fiscal-years	May not be needed in light of the new /finance /fund API w/ composite_fund
DELETE	/finance/group-fund-fiscal-years /<id>	NA	204	Delete a groupFundFY	finance.group-fund-fiscal-years	May not be needed in light of the new /finance /fund API w/ composite_fund
GET	/finance/group-fiscal-year-summaries	CQL Query	collection<group_fy_summary>	Get summary info for a group/fiscal year combination based on the provided query	finance-group	Queries both of these APIs with the provided CQL • GET /finance-storage/budgets • GET /finance-storage/group-fund-fiscal-years The first provides budget data including allocated/available/unavailable The second provides fund/group associations. Iterate of the results and calculate summary data on a per-group basis. Return this as a group_fy_summary collection. **Does not support paging**

NOTE: removing GET /finance/groups/<id>/budgets. This functionality will be provided by GET /finance/budgets?query=GroupFundFY.groupId=XYZ and fiscalYearId=ABC and group.

NOTE: adding GET /finance/group-fiscal-year-summaries. This replaces GET /finance/ledgers/<id>/groups and GET /finance/fiscal-years/<id>/groups.

Storage Module

Method	Path	Request	Response	Description	Interface	Notes
POST	/finance-storage/groups	group	group	Create a group	finance-storage.groups	
GET	/finance-storage/groups/<id>	NA	group	GET group by Id	finance-storage.groups	

GET	/finance-storage/groups	CQL Query	collection<group>	Search/List groups	finance-storage.groups	
PUT	/finance-storage/groups/<id>	group	group	Update a group	finance-storage.groups	
DELETE	/finance-storage/groups/<id>	NA	204	Delete a group	finance-storage.groups	
POST	/finance-storage/group-fund-fiscal-years	group_fund_fy	204	Create a groupFundFY	finance-storage.group-fund-fiscal-years	
GET	/finance-storage/group-fund-fiscal-years/<id>	NA	group_fund_fy	GET groupFundFY by Id	finance-storage.group-fund-fiscal-years	may not be needed, but can't hurt
GET	/finance-storage/group-fund-fiscal-years	CQL Query	collection<group_fund_fy>	Search /List groupFundFY	finance-storage.group-fund-fiscal-years	
PUT	/finance-storage/group-fund-fiscal-years/<id>	group_fund_fy	group_fund_fy	Update a groupFundFY	finance-storage.group-fund-fiscal-years	may not be needed, but can't hurt
DELETE	/finance-storage/group-fund-fiscal-years/<id>	NA	204	Delete a groupFundFY	finance-storage.group-fund-fiscal-years	

Ledgers

- Record with info that doesn't change from FY to FY
- Ledger/FY table will be needed to track per FY information (e.g. summary #s)
- Foreign Keys:
    - ledger_fiscal_years.ledgerId = ledgers.id
    - ledger.fiscalYearOneId = fiscalYear.id

Schemas

ledger

Property	Type	Default	Required	Notes
id	string		N	UUID - system generated if not specified
name	string		Y	Unique
description	string		N	
code	string		Y	Unique
ledgerStatus	string		Y	enum: Active, Inactive, Frozen
restrictEncumbrances	boolean	true	Y	whether or not to bypass allowableEncumbrance restrictions against budgets associated with this ledger. Typically used for some period of time after rollover, before allocations are made to the new budgets.
restrictExpenditures	boolean	true	Y	whether or not to bypass allowableExpenditures restrictions against budgets associated with this ledger. Typically used for some period of time after rollover, before allocations are made to the new budgets.
fiscalYearOneId	string		Y	UUID of the first fiscal year to associate with this ledger.
allocated	number		N	not persisted
unavailable	number		N	not persisted
available	number		N	not persisted
currency	string		N	currency code
acqUnitIds	array<string>		N	Array of UUIDs corresponding to the acquisition units assigned to this ledger.
metadata	metadata		N	System generated metadata (createdBy/updatedBy/createdOn/etc.)

ledgerFY

Property	Type	Default	Required	Updated On Transaction	Notes
id	string		N	N	UUID - system generated if not specified
ledgerId	string		Y	N	UUID of ledger
fiscalYearId	string		Y	N	UUID of fiscal year
allocated	number		N	Y	

unavailable	number		N	Y	
available	number		N	Y	
currency	string		Y	N	currency code for this fiscal year

APIs

Business Logic Module

Method	Path	Request	Response	Description	Interface	Notes
POST	/finance/ledgers	ledger	ledger	Create a ledger	finance. ledgers	
GET	/finance/ledgers/<id>	Fiscal Year Query (opt)	ledger	GET ledger by Id	finance. ledgers	If FY specified:Queries GET /finance-storage /ledger-fiscal-year • Return Ledger w/ summary #s • Enforces acquisition units based on ledger's assignments, ignores budget's acq. units. else • return ledger w/o summary #s The fiscalYear query arg is the UUID of a fiscal-year record
GET	/finance/ledgers	CQL Query Fiscal Year Query (opt)	collection<ledger>	Search/List ledgers	finance. ledgers	If FY specified: • Queries GET /finance-storage/ledger-fiscal-year • Return Ledgers w/ summary #s • Enforces acquisition units based on ledger's assignments, ignores budget's acq. units. else • return ledger w/o summary #s The fiscalYear query arg is the UUID of a fiscal-year record
PUT	/finance/ledgers/<id>	ledger	ledger	Update a ledger	finance. ledgers	
DELETE	/finance/ledgers/<id>	NA	204	Delete a ledger	finance. ledgers	
~~GET~~	~~/finance/ledgers/<id> /budgets~~	~~CQL Query,~~ ~~Fiscal Year Query (req)~~	~~collection<budget>~~	~~Get budgets for a ledger~~	~~finance. ledgers~~	~~Queries GET /finance-storage/group-fund-fiscal-years (by fiscalYear, ledger, ledger.acqUnits)~~ ~~Enforces acquisition units based on ledger's assignments, ignores budget's acq. units.~~ ~~The fiscalYear query arg is the UUID of a fiscal-year record~~
~~GET~~	~~/finance/ledgers/<id> /groups~~	~~CQL Query,~~ ~~Fiscal Year Query (req)~~	~~collection<group_fy_summary>~~	~~Get groups for a ledger~~	~~finance. ledgers~~	~~Queries GET /finance-storage/budgets (by fiscalYearId, ledgerId, ledger.acqUnits)~~ ~~Queries GET /finance-storage/group-fund-fiscal-years (by ledgerId, fiscalYearId, ledger.acqUnits)~~ • ~~pages all results or gets them all in one go~~ • ~~sums #s for each group~~ • ~~returns array of group_fund_fy~~ ~~Enforces acquisition units based on ledger's assignments, ignores group's acq. units.~~ ~~Does not support paging~~ ~~The fiscalYear query arg is the UUID of a fiscal-year record~~
GET	/finance/ledgers/<id> /current-fiscal-year	NA	fiscal_year	Get the current fiscal year for a given ledger record with <id>	finance. ledgers	

NOTE: Removing GET /finance/ledgers/<id>/budgets. This functionality will be provided by GET /finance/budgets?query=fiscalYearId=ABC and ledger.id=DEF and ledger.acqUnits=\"XYZ\"

NOTE: Removing GET finance/ledger/<id>/groups. This functionality will be provided by GET /finance/group-fiscal-year-summaries?query=fiscalYearId=ABC and ledger.Id=DEF and ledger.acqUnits=\"XYZ\"

Storage Module

Method	Path	Request	Response	Description	Interface	Notes
POST	/finance-storage/ledgers	ledger	ledger	Create a ledger	finance-storage.ledgers	
GET	/finance-storage/ledgers/<id>	NA	ledger	GET ledgerby Id	finance-storage.ledgers	
GET	/finance-storage/ledgers	CQL Query	collection<ledger>	Search/List ledgers	finance-storage.ledgers	
PUT	/finance-storage/ledgers/<id>	ledger	201	Update a ledger	finance-storage.ledgers	
DELETE	/finance-storage/ledgers/<id>	NA	204	Delete a ledger	finance-storage.ledgers	
GET	/finance-storage/ledger-fiscal-years	CQL Query	collection<ledgerFY>	Search/List LedgerFYs	finance-storage.ledgers	

Updating the LedgerFY table

As you can see above, there is not API for directly manipulating the LedgerFY records. These records are automatically updated by the storage module when related changes are made.

- When a Ledger is created,
  - create a LedgerFY record for the FY identified by ledger.fiscalYearOneId
  - create a fiscalYear for the next period in the fiscalYear series (TODO: more details)
  - create a LedgerFY record for the FY just created
- ~~When a FiscalYear is created, create a new LedgerFY record for each existing Ledger.(only if fiscalYear endDate > now)~~ Handle this during rollover...

The approach we're taking will result in LedgerFY records being creating which may never be used - this is especially true in cases where there are simultaneous or overlapping fiscal years, e.g. Texas Fiscal Year (TFY19) and Qatar Fisal Year (QFY19) might both have similar or overlapping periods, but apply to two separate campuses, each with their own finances. A given ledger would only really apply to one or the other, but LedgerFY records would be created for both. We need to do this to support rollover. The LedgerFY records themselves are quite small, so it shouldn't be too much of a problem.

For now we don't have to worry about managing LedgerFY records upon update or delete of a ledger or FiscalYear. The plan is to implement a soft delete on both so reference integrity should be maintained without intervention.

The LegerFY records should be created in the storage module in database transactions (which also include the creation of the Ledger/FiscalYear record which caused the LedgerFY creation), possibly via trigger functions.

NOTE: The idea of adding a "fiscal year" series which relates fiscal years together e.g. "Texas Fiscal Year" (TFY19, TFY20, TFY21), and "Qatar Fiscal Year" (QFY19, QFY20, QFY21) has been discussed and would help with both rollover and would also reduce the number of unnecessary/unused LedgerFY records being created. This is still in the idea phase and has not yet been fleshed out. If we choose to go this route, adding it later shouldn't be too disruptive.

Budgets

- Essentially a Fund/FY object
- Foreign Keys:
  - Budget.fiscalYearId = FiscalYear.id
  - Budget.fundId = Fund.id
  - Budget.ledgerId = Ledger.id
- View:
  - Joins budgets + group_fund_fiscal_years + groups
  - Needed to query budgets by group criteria, e.g. group.acqUnitIds
  - queried by GET /finance-storage/group-budgets
  - Hopefully this is only a temporary solution that can be removed once RMB supports bi-directional cross-index sub-queries, e.g. *budget group_fund_fiscal_years group*

Schemas

budget

Property	Type	Default	Required	Updated On Transaction	Notes	Requirements tickets
id	string		N	N	UUID - system generated if not specified	
name	string		Y	N	= <fund.code>-<fiscal_year.code>, eg. AFRICAHIST-FY19. Unique	

budgetStatus	string		Y	N	enum: Active, Frozen, Planned, Closed	
allowableEncumbrance	number		N	N	percentage	
allowableExpenditure	number		N	N	percentage	
allocated	number	0	Y	Y	Running total of all the allocation transactions against the budget.	
awaitingPayment	number		N	Y (encumbrance only)	Running total of all the invoiced amounts (i.e. waiting to be paid)	
available	number	0	N	Y	= (allocation + <transfers>) - awaiting_payment - encumbered - expenditures NOTE: Doesn't include allowableEncumbered amount	
encumbered	number		N	Y (encumbrance only)	Running total of money set aside for purchases	
expenditures	number	0	N	Y (encumbrance only)	Running total of all the payments recorded by the fund	
unavailable	number		N	Y	= awaiting_payment + encumbered + expenditures NOTE: Doesn't include overEncumbered amount	
overEncumbrance	number		N	Y (encumbrance only)	Amount the budget is over encumbered. i.e. overEncumbered = MAX(0, encumbered - (MAX(0, (allocated - awaitingPayment - expended))	
overExpended	number		N	Y (encumbrance only)	Amount the budget is over expended. i.e. overExpended = max(0, expended +awaitingPayment - allocated)	
netTransfers	number	0	N	Y	This would actually be calculated by summing all the "Transfers" on this budget.	**UIF-215** - Getting issue details... STATUS
fundId	string		Y	N	UUID of fund	
fiscalYearId	string		Y	N	UUID of fiscal year	
ledgerId	string		Y	N	UUID of ledger	
acquisitionUnits	array<string>		N	N	Array of UUIDs corresponding to the acquisition units assigned to this budget.	
tags	tags		N	N	inherited from fund	
metadata	metadata		N	Y	System generated metadata (createdBy/updatedBy/createdOn/etc.)	

APIs

Business Logic Module

Method	Path	Request	Response	Description	Interface	Notes
POST	/finance/budgets	budget	budget	Create a budget	finance.budgets	
GET	/finance/budgets/<id>	NA	budget	GET budget by Id	finance.budgets	
GET	/finance/budgets	CQL Query	collection<budget>	Search /List budgets	finance.budgets	Special handling here... inspect query - if contains criteria in groups, e.g. "group.*" use the "group-budgets API, otherwise use the budgets API. While not ideal this saves us from having to create and later deprecate/remove (a breaking change) a separate API for this. Instead we can make a non-breaking change (just remove the logic - the client is unaware anything changed)
PUT	/finance/budgets/<id>	budget	budget	Update a budget	finance.budgets	
DELETE	/finance/budgets/<id>	NA	204	Delete a budget	finance.budgets	

Storage Module

Method	Path	Request	Response	Description	Interface	Notes
POST	/finance-storage /budgets	budget	budget	Create a budget	finance-storage. budgets	
GET	/finance-storage /budgets/<id>	NA	budget	GET budget by Id	finance-storage. budgets	
GET	/finance-storage /budgets	CQL Query	collection<budget>	Search/List budgets	finance-storage. budgets	need to be able to query by fields from budget/ledger /fiscalYear/groupFundFy
PUT	/finance-storage /budgets/<id>	budget	budget	Update a budget	finance-storage. budgets	
DELETE	/finance-storage /budgets/<id>	NA	204	Delete a budget	finance-storage. budgets	
GET	/finance-storage /group-budgets	CQL Query	collection<budget>	Search/List budgets by group criteria	finance-storage. budgets	queries a view joining budgets, group_fund_fiscal_years, groups

Fiscal Years

Schemas

fiscal_year

Property	Type	Default	Required	Notes
id	string		N	UUID - system generated if not specified
name	string		Y	e.g. Texas Fiscal Year
code	string		Y	Unique e.g FY19
description	string		N	
series	string		N	The alphabetical part of the code - set by mod-finance - used to group/relate fiscal years together. E.g. codes=FY19, FY20, FY21 all have series=FY
periodStart	date		Y	
periodEnd	date		Y	
currency	string		N	Currency code used (recorded when the FY ends, otherwise the system currency can be assumed)
acqUnitIds	array<string>		N	Array of UUIDs corresponding to the acquisition units assigned to this fiscal year.
metadata	metadata		N	System generated metadata (createdBy/updatedBy/createdOn/etc.)

APIs

Business Logic Module

Method	Path	Request	Response	Description	Interface	Notes
POST	/finance/fiscal-years	fiscal_year	fiscal_year	Create a fiscalYear	finance. fiscal-years	May result in creation of other records
GET	/finance/fiscal-years /<id>	NA	fiscal_year	Get fiscalYear by Id	finance. fiscal-years	
GET	/finance/fiscal-years	CQL Query	collection<fiscal_year>	Search/List fiscalYears	finance. fiscal-years	
PUT	/finance/fiscal-years /<id>	fiscal_year	fiscal_year	Update a fiscalYear	finance. fiscal-years	
DELETE	/finance/fiscal-years /<id>	NA	204	Delete a fiscalYear	finance. fiscal-years	
~~GET~~	~~/finance/fiscal-years /<id>/ledgers~~	~~CQL Query~~	~~collection<ledgerFY>~~	~~Get ledgers for a fiscalYear~~	~~finance. fiscal-years~~	~~Queries GET /finance-storage/ledger-fiscal-year (on fiscalYear.id and fiscalYear.acqUnits)~~ ~~Enforces acquisition units based on fiscalYear's assignments, ignores ledger's acq. units.~~

~~GET~~	~~/finance/fiscal-years/<id>/groups~~	~~CQL Query~~	~~collection<group_fy_summary>~~	~~Get groups for a fiscalYear~~	~~finance.fiscal-years~~	~~Queries GET /finance-storage/budgets (on fiscalYear.id and fiscalYear.acqUnits)~~ ~~Queries GET /finance-storage/group-fund-fiscal-years (on fiscalYear.id and fiscalYear.acqUnits)~~ ~~Iterates over results and calculates summary #s for each group~~ ~~Enforces acquisition units based on fiscalYear's assignments, ignores group's acq. units.~~ ~~Does not support paging~~
~~GET~~	~~/finance/fiscal-years/<id>/budgets~~	~~CQL Query~~	~~collection<budget>~~	~~Get budgets for a fiscalYear~~	~~finance.fiscal-years~~	~~Queries GET /finance-storage/ledger-budgets (on budget.fiscalYearId and fiscalYear.acqUnits)~~ ~~Enforces acquisition units based on fiscalYear's assignments, ignores budget's acq. units.~~

NOTE: GET /finance/fiscal-years/<id>/ledgers has been removed. This functionality is covered by GET /finance/ledgers?query=ledgerFY.fiscalYearId=ABC

NOTE: GET /finance/fiscal-years/<id>/budgets has been removed. This functionality is covered by GET /finance/budgets?query=fiscalYearId=ABC

NOTE: GET finance/ledger/<id>/groups has been removed. This functionality will be provided by GET /finance/group-fiscal-year-summaries?query=fiscalYearId=ABC and fiscalYear.acqUnits=\"XYZ\"

Storage Module

Method	Path	Request	Response	Description	Interface	Notes
POST	/finance-storage/fiscal-years	fiscal_year	fiscal_year	Create a fiscalYear	finance-storage.fiscal-years	
GET	/finance-storage/fiscal-years/<id>	NA	fiscal_year	GET fiscalYear by Id	finance-storage.fiscal-years	
GET	/finance-storage/fiscal-years	CQL Query	collection<fiscal_year>	Search/List fiscalYears	finance-storage.fiscal-years	
PUT	/finance-storage/fiscal-years/<id>	fiscal_year	fiscal_year	Update a fiscalYear	finance-storage.fiscal-years	
DELETE	/finance-storage/fiscal-years/<id>	NA	204	Delete a fiscalYear	finance-storage.fiscal-years	

Expense Classes

Schemas

expense_class

Property	Type	Default	Required	Unique	Notes	Requirements tickets
id	string		N	Y	UUID - system generated if not specified	UIF-211 - Getting issue details... STATUS
name	string		Y	Y		
code	string		N	Y		
externalAccountNumberExt	string		N	N		

APIs

Business Logic Module

Method	Path	Request	Response	Description	Interface	Notes
POST	/finance/expense-classes	expense-class	expense_class	Create a expense_class	finance.expense_class	UIF-211 - Getting issue details... STATUS
GET	/finance/expense-classes/<id>	NA	expense_class	GET expense_class by Id	finance.expense_class	
GET	/finance/expense-classes	CQL Query	collection<expense_class>	Search/List expense_class	finance.expense_class	

PUT	/finance/expense-classes/<id>	expense_class	expense_class	Update a expense_class	finance. expense_class	
DELETE	/finance/expense-classes/<id>	NA	204	Delete a expense_class	finance. expense_class	

Storage Module

Method	Path	Request	Response	Description	Interface	Notes
POST	/finance-storage/expense-classes	expense_class	expense_class	Create a expense_class	finance-storage. expense_classes	UIF-211 - Getting issue details... STATUS
GET	/finance-storage/expense-classes/<id>	NA	expense_class	GET expense_class by Id	finance-storage. expense_classes	
GET	/finance-storage/expense-classes	CQL Query	collection<expense_class>	Search/List expense_class	finance-storage. expense_classes	
PUT	/finance-storage/expense-classes/<id>	expense_class	expense_class	Update a expense_class	finance-storage. expense_classes	
DELETE	/finance-storage/expense-classes/<id>	NA	204	Delete a expense_class	finance-storage. expense_classes	

Database Views

Several views are needed to accommodate some of the inter-record queries. These have been mentioned throughout the document. This section pulls that info into a single place and adds additional details (e.g. related API definitions)

N.B. This was written prior to RMB-395 - Getting issue details... STATUS . These views are no longer needed in light of this new cross-index subquery functionality. I'm leaving this section as-is in case this approach does not prove adequate for our needs.

LedgerFY

- join: ledgerFY + ledger + fiscalYear
  - ledger.id == ledgerFY.ledgerId
  - fiscalYear.id == ledgerFY.fiscalYearId
- jsonb: ledger_fy
- NOTE: we need fiscalYear here because when calling this from `GET /finance/fiscal-years/<id>/ledgers` we need to query against the fiscalYear's acquisition units.

API

Method	Path	Request	Response	Description	Interface	Notes
GET	/finance-storage/ledger-fiscal-years	CQL Query	collection<ledger_fy>	Search/list ledger/FY	finance-storage. ledgers	Needed for Fiscal Year details view, ledger accordion

GroupFundFY

- joins: Fund + Budget + Fund/Group/FY + Ledger + FIscalYear
  - fund.id == FundGroupFY.fundId.
  - budget.fiscalYearId == FundGroupFY.fiscalYearId AND budget.fundId = FundGroupFY.fundId
  - ledger.id == fund.ledgerId
  - fiscalYear.id == budget.fiscalYearId
- jsonb: group_fund_fy
- NOTE: we need ledger here because when calling this from `GET /finance/ledgers/<id>/groups` we need to query against the ledger's acquisition units.
- NOTE: we need fiscalYear here because when calling this from `GET /finance/fiscal-years/<id>/groups` we need to query against the FY's acquisition units.

API

Method	Path	Request	Response	Description	Interface	Notes
GET	/finance-storage/group-fund-fiscal-years	CQL Query	collection<group_fund_fy>	Query group/fund/FY view	finance-storage.groups	Needed for Ledger and Fiscal Year details views, groups accordion

LedgerBudget

- joins: Fund + Budget + Ledger + FiscalYear
    - fund.id == budget.fundId.
    - ledger.id == fund.ledgerId
    - fiscalYear.id == budget.fiscalYearId
- jsonb: budget
- NOTE: we need ledger here because when calling this from `GET /finance/ledgers/<id>/budgets` we need to query against the ledger's acquisition units.
- NOTE: we need fiscalYear here because when calling this from `GET /finance/fiscal-years/<id>/budgets` we need to query against the fiscalYear's acquisition units.

API

NOTE: We decided that having a separate API duplicates functionality provided by the GET budgets by query endpoint. Unless we discover a reason why the subquery support in RMB won't for for this, there's no need for a separate API.

Method	Path	Request	Response	Description	Interface	Notes
~~GET~~	~~/finance-storage/ledger-budgets~~	~~CQL Query~~	~~collection<budget>~~	~~Query ledger /budget view~~	~~finance-storage.budgets~~	~~Needed for Ledger and Fiscal Year details views, funds accordion~~

End-to-End Flow

1. Setup - Create and link financial structures
    - Fiscal Year(s) are created
        - a ledgerFY with new FY is created for each ledger?? Only if not in the past?
        - a budget with new FY is created for each fund?? Only if not in the past?
        - a groupFundFY with new FY is created for each fund/group pair?? Only if not in the past?
    - Ledgers are created
        - a ledgerFY is created for each FY?? Only if not in the past?
    - Funds are created
    - Budgets are created
    - Groups are created
    - Funds are added to groups
        - a groupFundFY is created for each fund/group/FY tuple?? Only if FY not in the past?
2. Money is allocated to budgets
    - Allocations (transactions) are created
    - Budgets are updated
        - budget.allocated increases
        - budget.available/unavailable are re-calculated
    - Ledgers are updated
        - ledger.allocated/available/unavailable are re-calculated
3. An order is created and opened
    - Encumbrances are created for each of the POL's fund distributions
        - Need to check if the budget.allocated >= amount being encumbered
        - Need to check if the budget.allowableEncumbered >= (budget.encumbered + amount being encumbered)
    - Budgets are updated
        - budget.encumbered increases
        - budget.available/unavailable are re-calculated
    - Ledgers are updated
        - ledger.allocated/available/unavailable are re-calculated
4. An invoice is created and approved
    - Voucher is generated
    - Encumbrances are updated
        - transaction.encumbrance.amountAwaitingPayment increases
        - transaction.amount (effective encumbered amount is recalculated
    - Budgets are updated
        - budget.encumbered budget.awaitingPayment
        - budget.available/unavailable are re-calculated
    - Ledgers are updated
        - ledger.allocated/available/unavailable are re-calculated
5. Invoice is paid
    - Voucher is marked as paid
    - Payments (transactions) are created
    - Encumbrances are updated
        - transaction.encumbrance.amountAwaitingPayment transaction.encumbrance.amountExpended
        - transaction.amount (effective encumbered amount is recalculated
    - Budgets are updated
        - budget.awaitingPayment budget.expenditures
        - budget.available/unavailable are re-calculated
    - Ledgers are updated
        - ledger.allocated/available/unavailable are re-calculated
    - Orders/POLs are updated
        - purchaseOrder.paymentStatus = partially/fullyPaid
        - poLine.paymentStatus = partially/fullyPaid
    - For each order, If fully paid

        - repeat 4, 5 as needed
    - Else
        - Encumbrances are released
            - transaction.amount (effective encumbered amount is recalculated (becomes 0)
        - Budgets are updated
            - budget.encumbered decreases
            - budget.available/unavailable are re-calculated
        - Ledgers are updated
            - ledger.allocated/available/unavailable are re-calculated
        - Order is closed
6. Fiscal Year ends/starts
    a. TBD

JIRA

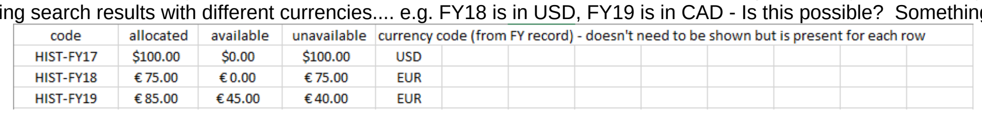
Convenient place to put links to features/stories in JIRA

- TBD

Mockups

https://drive.google.com/drive/folders/1Vc401TCsooCeUWNwBQForjYpADE7CMW8

Open Issues/Other Considerations

- The set of columns to show in search results has not been reviewed by the small group yet. That could potentially affect this if there are changes.
- How to express allocateTo/allocateFrom "all" funds? none? groups?
    - It was discussed that we might at some point add separate fields for specifying groups of funds that a fund can allocatedTo/From. At the moment this is the best idea we've come up with. One potential downside is that it might encourage proliferation of groups which will have a negative impact on performance in the current design due to needing to perform group calculations on the fly.
- How to capture changes in currency? Historically?
    - No need to track the currency code in each fund, but when the fiscalYear ends, we should capture the system currency in use at the time.
    - Displaying search results with different currencies.... e.g. FY18 is in USD, FY19 is in CAD - Is this possible? Something like:
        - 

code	allocated	available	unavailable	currency code (from FY record) - doesn't need to be shown but is present for each row
HIST-FY17	$100.00	$0.00	$100.00	USD
HIST-FY18	€ 75.00	€ 0.00	€ 75.00	EUR
HIST-FY19	€ 85.00	€ 45.00	€ 40.00	EUR

        - Discussed with the FE guys, and this shouldn't be a problem.

Appendix

Preventing Partially Paid Invoices

When an invoice is paid, numerous transactions (payments, credits) will be generated. Either all or none of these need to be successfully processed; we can't have partially paid invoices. This presents a challenge of scale. There could feasibly be hundreds of payments for an invoice.

✗ Option A: Allow multiple payments to be sent at once to the payment API.

The idea here is simple... update the POST payments API to accept a collection of payments/credits and process them all at once.

Pros/Cons

- The request payload could be quite large
- Processing all the payments in a single call might take a prohibitively long time

✓ Option B: Collect/Aggregate payments/credits in the storage layer until all are received, then process together

This approach keeps the payment "simple" and consistent with the other transaction APIs. The storage layer performs some aggregation in the database of payments/credits based on the sourceInvoiceId specified in each payment. The flow looks something like this:

Introduce a new type of record... let's just call it a "saga" for now - what's proposed here isn't strictly a saga, but it's similar in ways. Maybe a "paymentManifest" or something is a better name; anyway - its not important right now.

Saga:

- invoiceId (UUID of the invoice these payments/credits are related to)
- numTransactions (int - total number of payments/credits)A payment or credit is posted

Flow:

- Using the sourceInvoiceId, get the invoiceLines and sum up the total number of fundDistributions - this is numTransactions.
- Get the saga using payments sourceInvoiceId. If one does exist, create one.
- Update the saga record and persist the payment record in a temporary table - do all this in a database transaction
- If this is the last payment in the saga, (the number of entries in the temp table == numTransactions) - NOTE: we'll probably want to use a unique_index on the id field or something to ensure the same payment isn't being counted more than once.
    - Persist the payments/credits from the temporary table and perform the summary # updates (e.g. update budget, ledger, encumbrances, etc. - unavailable/available/encumbered/expended/etc.) - do this in a database transaction
        - If everything works, cleanup the temporary table and return 200 OK
        - If something fails, no actual records are updated. The payment API call can fail and can be retried (POST payment/credit are idempotent)

NOTE: we might be able to move some of this logic to the BL module. The trade-off there is that the storage module would contain less logic, but would have a more complicated API.

Pros/Cons:

- Might be able to do some incremental processing instead of having to do it all at once
- Multiple single payments can be sent - avoids the massive request payload
- Individual payments/credits can be repeated if needed (the POST payment API is idempotent)

Other Ideas:

- Option B w/ optional batching... it may be convenient to allow batching (send multiple payments/credits at once); would help cut down on number of API calls needed to pay an invoice.
    - One call per invoiceLine vs one call per fund distribution
- TBD

Required Queries (WIP)

View Budget Transactions

In order to support searching for transactions for a given budget: https://drive.google.com/open?id=1w-kNrURaAQ0cjiHl0NzADaLnackJN0mT

1. GET transactions by Budget, search/filter on:
    a. transactionType
    b. source
    c. fromFundId
    d. toFundId
    e. status
    f. tags
    g. date (range)
    h. fiscalYearId
    i. description
    j. amount

View Budget Details

In order to support Budget details view: https://drive.google.com/open?id=1ezTFSS6Rypv_wg8eVBoMbsLRoci-2UR-

1. GET Budget by Id - including allocated/avail/unavail summary
2. GET FY by Id

View Group Details

In order to support Group details view: https://drive.google.com/open?id=1qxEIiISiNX6wSTQUEbTwfcx-fwL8xwbv

1. GET Group by Id - including allocated/avail/unavail summary
2. GET Funds by Group/FY - including allocated/avail/unavail

View Fund Details

In order to support Fund details view: https://drive.google.com/open?id=1ksgVbdEc426JCMkUDuey_GlB-M9c3to_

1. GET Fund by Id
2. GET Budget by FY (current FY - Equals, previous FYs - LessThan, next FY - GreaterThan)

View Ledger Details

In order to support Ledger details view: https://drive.google.com/open?id=1MMQdlKU1-OFlpCDCClHQMvTi-xPTf2K

1. GET funds for ledger/FY - including allocated/avail/unavail

2. GET groups for ledger/FY - including allocated/avail/unavail
3. GET FYs for a ledger
4. GET funds for a group
5. GET ledger for FY - including allocated/avail/unavail summary