Modular Data Model for Iroha v1.x

Intro

Data Model

A data model (further referred to as **DM**) is a business model abstraction. It may provide interfaces to execute some commands and query some state. A DM implementation is a module that can be attached to an Iroha node. In that case, the set of commands delivered to a DM module is strictly determined by the ledger, which enables to build extensible blockchain applications.

Iroha 1

Recently we had an experience of integrating HL Burrow into Iroha 1. It can also be seen as a DM module, but it did not achieve enough level of abstraction that could be used for any other DM module. In this document we want to design a more universal concept of integrating any DMs into Iroha with a uniform and simple process.

Client API

We exploit the transparent extensibility of protobuf messages: a message can be parsed correctly with a subset of its fields. So, Iroha parses a subset related to the execution flow till the data model, and DM parses the rest. Protobuf wire protocol does not contain any names, so we only have to ensure that field numbers correspond to right data type and semantics.
This approach keeps Iroha agnostic of the semantics and contents of specific payloads, and allows for seamless extensibility of both Iroha commands and queries and the DM module payload.

DM module identifier:

Includes a unique name and a version string.

```
message DataModelId {
      string name = 1;
      string version = 2;
}
```

```
}
```

Command

New command type: **CallModel**

```
message CallModel {
     message Payload {}
     Payload payload = 1;
     DataModelId dm_id = 2;
}
```

Query

New query type: **QueryModel**

```
message QueryModel {
     message Payload {}
     Payload payload = 1;
     DataModelId dm_id = 2;
}
message QueryModelResponse {
     message Payload {}
     Payload payload = 1;
}
```

And a new **endpoint**:

```
rpc QueryModel (QueryModel) returns (stream QueryModelResponse);
```

For now we consider only streaming query API as unary queries are logically a particular case of streaming and can be implemented later if needed for whatever reason.

Example

TicTacToe, a classical example

```
// -- common --
message Coordinates {
     int32 x = 1;
     int32 y = 2;
}
```

```
// -- commands --
message TicTacToeMakeMove {
	Coordinates coordinates = 2;
}
message TicTacToeGiveUp {}

message TicTacToeCommand {
	// extends CallModel
	message Payload {
		oneof command {
			TicTacToeMakeMove = 1;
			TicTacToeGiveUp = 2;
		}
	}
	Payload payload = 1;
	DataModelId dm_id = 2;
}

// -- queries --
message TicTacToeWatchUpdates {}
message TicTacToeGetBoard {}

message TicTacToeQuery {
	// extends QueryModel
	message Payload {
		oneof query {
			TicTacToeWatchUpdates = 1;
			TicTacToeGetBoard = 2;
		}
	}
	Payload payload = 1;
	DataModelId dm_id = 1;
}

// -- query responses --
message TicTacToeMoveRecord {
	String user = 1;
	Coordinates coordinates = 2;
}
message TicTacToeBoard {
	repeated TicTacToeMoveRecord = 1;
}
```

```
message TicTacToeQueryResponse {
      // extends QueryModelResponse
      message Payload {
            oneof response {
                  TicTacToeMoveRecord = 1;
                  TicTacToeBoard = 2;
            }
      }
      Payload payload = 1;
}
```

Note the “`extends`” comments in some messages. They are parsed by Iroha node by their extended parent type, and passed to the right backend, which then parses it by the extender type.

DM module API

Result is a structure that provides semantics of a variant of a void value or a string error. An example can be seen in Iroha Burrow integration: https://github.com/hyperledger/iroha/blob/847bc0b7cddf5709df509ffa156d89f4e0998856/irohad/ametsuchi/impl/common_c_types.h#L28.

Command

`Result execute(CallModel const&)`	apply the command
`void revert_block()`	undo the effect of all `execute` commands since last `commit_block`. // db transaction
`void revert_transaction()`	undo the effect of all `execute` commands since last `start_transaction` or `commit_block`, whichever is most recent. // db savepoint
`void commit_block()`	A savepoint for `revert_block()`, `revert_transaction()` and queries
`void start_transaction()`	A savepoint for `revert_transaction()`
`void reset_state()`	Completely wipe out the state. Needed when overwriting the ledger.

Query

`Result query(QueryModel const&, QueryModelResponseWriter&)` - perform the query and write the results to the writer channel. Query is performed on the state as on last `commit_block()`. This can be extended in the future to support queries for specific height as the module knows about the blocks.

Module configuration

For the earlier stages of development of this feature, we can configure available modular backends in configuration file. This means that runtime changes are not available.
A record of all configured modules seems required in the ledger, like a command in the settings table. This will allow us to keep the ledger when modules are updated. When we face the need to update a module version, we can also add a command to disable specific executor on the ledger.
So the minimal basic workflow looks like this:

- Iroha is started
- It reads several modular DM initialization params from the config
- For each DM it:
    - Initializes the DM
    - Registers the DM in command and query executors
- Then it checks that the set of configured DMs matches exactly the set of enabled DMs in the genesis block.

Pipeline

Command

The new command acts just like any other command till the command executor. Once the command executor receives the new command, it parses the DM identifier of the command and tries to find a matching executor. If found, it passes the reference to the original protobuf message into the specific executor and gets a status back (success or failure). If the executor was not found, an error is returned immediately.
Add a general permission for CallModule command.

Query

Seems straightforward. A new separate endpoint just passes the query to a DM module.
Add a general permission for QueryModule command.

Iroha internals

DM module registry

Manages the modules of DMs.
It is used by

- Command executor for executing CallModel commands
- Transaction executor for starting & reverting txs
- Consensus module for starting & reverting blocks
- Irohad::resetWsv() for clearing state

`registerModule(...)`	Add a DM module with specific name & version
`Result execute(CallModel const&)`	Call execute of specific DM if it is registered, or return Error otherwise
`void revert_block()`	Call `revert_block()` for each registered module
`void revert_transaction()`	Call `revert_transaction()` for each registered module
`void commit_block()`	Call `commit_block()` for each registered module
`void start_transaction()`	Call `start_transaction()` for each registered module
`void reset_state()`	Call `reset_state()` for each registered module