**TheFundsChain: micro-architecture**

**Versatility**. DLT Technology: Hyperledger fabric. This note provides some design guidelines for chaincode micro-design. Our intent is to devise a versatile chaincode support for multiple scenarios. Experience showed that, although the same base concepts are iterated over and over again, each POC / customer demands require great efforts to put be put forward into a chaincode.

**Chaincodes, workflows, models and APIs**

Chaincodes are thought as workflow processing units backed by a state database. A single chaincode may participate in several workflows. Workflow transitions are the “transactions” of the DLT.

If we keep under the hood the immutable / secured / consensus-based state-transitions mechanisms, we just have a “controller” component in charge of one or several workflows. Workflow transitions are defined by an API, which in turn defines the underlying state data model and the input / output message models. In H.L terminology, such transitions are called “invoke functions”.

Besides transactions, users (or other chaincodes) may interact with a chaincode by posting queries, which follow a different process.

Our design strives to rapidly compose and deploy the APIs implementing workflows.

This shall make our platform highly versatile, adaptable to different business scenarios or even entirely new businesses.

Common Shift utils

* API does not directly use JSON: JSON marhsalling/unmarshalling is wrapped as a common utility (in order to factor out JSON marshalling quirks and peculiarities in GO, esp. numeric types).
* Introspection, e.g. retrieving the list of methods published by an API handler.
* Introspection methods mapping rules, relying on API conventions, e.g. InvokeFundSubscribe() is published as FUND\_SUBSCRIBE function available to HTTP clients. Same for QueryGetFundsHoldings(): GET\_FUNDS\_HOLDINGS.
* Sugar: rule: camel case replaced by ‘\_’ ; upper cased

Other utilities:

* Wrapper: a stamping function to generate signed / verifiable messages (e.g. used when synchronizing chaincodes)
  + E.g. wraps hyperledger/util/sign.go
* Wrapper: secure time (perhaps this one could be managed by zookeeper instead)

Shim Shift utils

* Provides a wrapper to shim (the internal API for chaincodes) for use from within APIs. The idea is to be able to mock this wrapper to be able to run unit tests on API without the burden of shim / chaincode deployment. It is very important to be able to validate our API before running into the complexities of an HL environment
* Wrapper implements: get/set state ; call local chaincode ; call remote chaincode
* Introspection about current peer: aka “who am I” method? (e.g. tentative wraps policy Evaluate())
* Wrapper: custom event subscribe / notify framework / secure time-based event subscribe / notify
  + E.g. provide a more concise interface than <https://www.ibm.com/developerworks/cloud/library/cl-ibm-blockchain-chaincode-development-using-golang/index.html>

API package

An API package is composed of:

* A state-model JSON schema definition
* For each INVOKE and QUERY function, JSON schema definition of input and output messages
* Exported GO types mapping all JSON definitions in this API
* An interface with all INVOKE and QUERY methods supported by the API
* An init method for the API, which shall be invoked at chaincode deployment time
* A handler type acting as an object implementing all API methods
* API structures are instantiated as either state, delegated to a local chaincode (same peer) or delegated to a remote.
* In the case of delegated state management, only partial information is generally available (e.g. input/output messages)

GO structures are generated from JSON using some existing GO tool. The quality of these tools is far from perfect and subject to bugs or changes. Try to take the best fit and possibly improve it. We are going to produce many JSON schemas, so it is important to reduce boiler plate typing in GO.

<https://adrianhesketh.com/2016/07/19/json-schema-to-go-struct-generator-roundup/>

<https://github.com/a-h/generate>

I tested them all. None of the tools mentioned here is fully functioning… A good tool would be a merger of the 4 tools tested.

SWAGGER: API should be documented using swagger. Swagger is cool to generate a nice documentation UI from API methods and messages.

API composition

API may be built upon several, more basic APIs, following a composition pattern (most favored pattern in GO). The newly built API may expose part or all of the underlying APIs.

Each JSON state model defined by an API is stored under its own key (e.g. couchdb document).

In the case of a compounded API (i.e. resulting from composition), one single document is created.

Example: a chaincode declaring a FundLegalAPI and a FundOperationsAPI creates 2 documents.

The FundOperationsAPI is itself composed of more basics API, such a FundOrdersAPI and some others. FundsOperationsAPI defines its composition scheme in JSON by using other schemas. One single JSON document is used.

API options

An API handler may be instantiated by a chaincode with a set of options. Options settings are defined by each API.

Example: the LegalAPI may be either specialized into FrenchLegalAPI and LuxLegalAPI or both may be merged with an option.

Example: a chaincode may choose to publish an API partially. Methods may be optionally disabled from the API. E.g. on a French fund, the “Transfer” method of the FundOrderAPI could be disabled. Options standardization will emerge later on.

Other example: an API may require tampering with the “one JSON schema / one document” default rule, for instance to generate as many documents as suits the API. Example: a FundHoldingsAPI may choose to split holdings in several documents and be able to distribute them across several shards (advanced couchdb usage). It should be possible for a chaincode to switch to such special API capabilities thanks to options.

Options should be specified with a JSON structure.

One should be able to distinguish between build-time options and deploy-time options.

API interactions

API may interact with each other. Depending on chaincode peer settings, an API call may be: internal, local to an another chaincode on the same peer, remote.

Ideally, we should be able to tune this thanks to options, but let’s not be too ambitious for a start. We’ll assume for a while that the API supports certain peer setups only and hard code the implementation of other API calls.

**Chaincode template**

A chaincode is implemented by simply composing API’s, and routing the methods obtained by introspection.

A chaincode based on our template declares dependencies to API packages

Reference implementation: chaintool – but we would prefer to work by introspecting methods rather than declaring yaml files…

<https://github.com/hyperledger/fabric-chaintool/blob/master/examples/helloworld/src/chaincode/main.go>

Attention: API initialization may specify an order of initialization.

To sum up:

* Chaincode main component boils down to 100 lines of code or so. It is no more the heart of our code (2 methods to switch input function toward API methods)
* We may separate full chaincode testing from API functional testing (use mocks for shim)

The chaincode main() performs a ROUTING function to API methods.

A chaincode contains one or several OBJECTS (e.g. order, registry, contract…).

Each object participates in zero or many state-machines, each FSM driven by a separate state variable.

The chaincode loads API methods. Invoke methods emit events to the FSM.

JSON schema tooling

We design JSON schemas with XMLSpy.

We need quite a bit of unitary schema definitions to overcome known JSON limitations. The main issue lies with numbers. We choose not to use floats at all and enforce all numbers to be integers. Decimals would be wrapped as a structure { int , precision }. Future use of floats is not precluded but shall require some care.

Warning: great care must be taken with type safety in GO as it I easy to panic a GO program with an overflow / some other type incompatibility issue. Therefore, coding style will use and abuse of type assertions provided by GO. This is a key security issue.

Similarly, JSON unmarshalling WILL be preceded by a JSON schema validator (again there are many available GO implementations of a JSON schema validator, each with its own flaws and limitations).

*Schema annotations*

JSON schema 4 does not (and will not…) support comments or annotations to drive code generation (other strategies – e.g. Java Jackson – use JSON schema generation from annotated code structures… but we lose the point of data modelling with a convenient tool such as XML spy…). Annotations would be welcome in order to support:

* Structure access policy
* GUI handling (e.g. something along the lines of JSON Hyper-schema)
* Serialization directives (such as storing in a separate Couchdb document, etc..)
* GO type generation directives (e.g. generating GO tags, marshalling instructions…)

No hope in sight here. Only solution is to tweak a JSON-to-GO type generator to interpret special optional attributes from the schema.

Example: annotation: { type: ‘object’ … } would bear a special meaning for generators (GO or Angular).

*Other GO boiler plate*

We like the idea behind the golangannotations package to generate some boiler plate REST API mapping. It is probably too much focused on its intended use (generating HTTP REST services) but the idea is here and may be reused to use annotations in order to keep API code concise and business-minded.

<https://github.com/MarcGrol/golangAnnotations>

I am thinking of something that would replace typical ACL boiler-plate with methods (unless a powerful ACL configuration mechanism is put in place). This really looks like an Apache config option:

// @deny(all)

// @allow(custodian, issuer)

(h \*APIHandler) myAPIMethod(…) (\*message) {

…

}

golangAnnotation provides out of the box an annotation for code generation on JSON structure (marshall/unmarshall template). Along the same lines, it is straightforward to extend the package.

Warning: annotations are not macros! They define generation targets on structs, functions and interfaces. Macros ae not really useful anyhow.

Easier to customize go annotations I think than writing our own code generator for go: generate…

Code generation is useful for:

* JSON => go struct declarations
* JSON go struct => get/Set methods with marshall/unmarshall, checks & etc.. (e.g. @JSONStruct)
* API methods ACL checks
* Other boiler plate ?

**GUI and tools**

We like the ideas behind Angular 4 framework. Together with something like Faces for Angular, that could be an accelerator to our model-driven idea of GUIs.

Especially, we believe that the use of a framework on top of AngularJS could be a good bet for quickly deploying cool GUIs.

To be tested:

* UI Frameworks:
  + PrimeNG: OK
  + Json-schema-to-typescript: OK
  + Angular4json-form
  + Angular UI Tree, Grid, Bootstrap and Angular Animations (cool for demos and tutorials)
    - Animated routes: OK
    - Testing: ng-bootstrap
* App de v frameworks?
  + Swagger has a codegen for nodeJS (generates skeleton for API)
  + Also has one for Golang but I would be much more cautious her…
* User management & auth : OKTA

**Using node for gateways**

Npm Protocol, jsftp, pbf, gulp-scp,

**Appendix: typical wrap-ups to be offered by the API**

*Local chaincode call*

*f := "invoke"*

*invokeArgs := util.ToChaincodeArgs(f, "a", "b", "10")*

*response := stub.InvokeChaincode(chainCodeToCall, invokeArgs, channelID)*

*if response.Status != shim.OK {*

*errStr := fmt.Sprintf("Failed to invoke chaincode. Got error: %s",*

*string(response.Payload))*

*fmt.Printf(errStr)*

*return shim.Error(errStr)*

*}*

*// Query chaincode\_example02*

*f := "query"*

*queryArgs := util.ToChaincodeArgs(f, "a")*

*// if chaincode being invoked is on the same channel,*

*// then channel defaults to the current channel and args[2] can be "".*

*// If the chaincode being called is on a different channel,*

*// then you must specify the channel name in args[2]*

*response := stub.InvokeChaincode(chaincodeName, queryArgs, channelName)*

*if response.Status != shim.OK {*

*errStr := fmt.Sprintf("Failed to query chaincode. Got error: %s",*

*response.Payload)*

*fmt.Printf(errStr)*

*return shim.Error(errStr)*

*}*

*Aval, err = strconv.Atoi(string(response.Payload))*

*if err != nil {*

*errStr := fmt.Sprintf(*

*"Error retrieving state from ledger for queried chaincode: %s", err.Error())*

*fmt.Printf(errStr)*

*return shim.Error(errStr)*

*}*

*Remote chaincode call*

<https://github.com/hyperledger/fabric-sdk-go/blob/master/test/integration/end_to_end_test.go>

**Finite state machines**

API should implement finite state machines for rapid deployment / unit testing and maintenance.

There are number of "simple" (simplistic packages) available.

J'étudie <https://github.com/looplab/fsm>. Il reste simple mais assez complet. J'aime:

* Le mécanisme commun avec la version Javascript utilisable a priori avec nodejs
* La précision des callbacks sur événements (transitions / post&pre events / post&pre state)
  + before\_<EVENT>, leave\_<STATE>, after\_<EVENT>…
  + La factorisation avec before\_event, enter\_state…
  + Les méthodes is(), can() et cannot()
  + Error handling / explicit Cancel with custom Errors....
  + Asynchronous transitions
  + Almost zero dependancies
  + Compacity: 3 files…
  + Up to date: projet vivant
  + Importé par 92 autres packages (donc pas mal côté Hyperledger). Ça sent bon.

Côté nodejs, même profusion de packages FSM. Par exemple javascript-state-machine semble convenable. Machina est trop complexe, node-state trop simpliste…

*Gographviz*: rêvons un peu… J'aime bien le potentiel pour documenter de goviz (avec graphviz).

On doit bien pouvoir trouver un moyen de transformer une définition de FSM en Golang en un joli graphe type SDL… (et joindre ça à la doc).

Idéalement, un chaincode = un ou plusieurs workflows = autant de diagrammes d'état

J'aime aussi le potentiel de l'outil callgraph (pour s'y retrouver en interne…)

**Unit testing**

Go test est ok pour la base.

Unit testing: ici ce qui m'intéresse, c'est la génération d'un squelette de cas de tests.

Par exemple, à partir d'une FSM, on peut générer un cas pour chaque combinaison (event, state) et déjà prévoir un résultat.

On pourrait utiliser une annotation @fsmStruct(method=simul\_state\_1, … pour générer le squelette de test. Pas facile mais essentiel pour assurer une bonne couverture de test des FSM. Pas simple de faire un bon générateur go generate from scratch..

Ce qu'il faudrait, c'est annoter la structure qui décrit le FSM et générer un squelette de test pour chaque état et chaque transition. Pour une transition donnée, il y a en général plusieurs cas de tests (données initiales JSON injectées pour le test).

Franchement, tout le débat XUnit vs BDD test ne m'intéresse pas vraiment…

Je regarde ce que font les gars du Fabric pour m'en inspirer sur les choses importantes:

* Mocks
* Mise en place de config
* …
* Ils utilisent testify/asset : OK
* Ils ont mis en place une chaîne de tests d'intégration de CI en python (fabric/test)
* Les mocks sont faits à la main. Peut être est-ce mieux ainsi – c'est à voir.

**Documentation**

Go doc generation: go doc fourni le service minimum… Mais bon c'est simple au moins.

<https://godoc.org/github.com/fluhus/godoc-tricks>

On peut essayer slate (style Github)

Pour la vision API, swagger semble le top (utilisé par fabric-ca).

Ici, Fabric ne donne pas (à mon sens) le bon exemple. Tout est dans des markdown et rien dans le code.

**Demo building**

I found something very interesting. However, it's not free software. Our project probably does not represent a good use case…

Possible Use case: publish entry points in every chaincode / nodejs SDK – the noflow stuff should be able to produce a dynamic visualization of all data flow… <https://plans.flowhub.io/>

Based on: <https://noflojs.org/> (free). Nice demo: <http://app.flowhub.io/#project/7135158/7135158_noflo>

OpenMCT is the main building block for visualization (NASA open source, provided as a nodeJS package). OK to display a mashup of counters. How about message flows? They say it's possible with the so called "live mode". Couldn't see a working demo.

Another idea could be to capture all message logs and build an animated visualization (fabric nodes and message flows) from these logs.