**Corda API**

* Client
* Core
* Finance
* Testing

**State APIS**

1. **Contract State Interface**

Using a state without any identifier. This state cannot be updated but can be consumed

Example : Art work – This state can’t be updated. However, it can be changed to different state of choice like SOLD, STILL FOR SALE, SHIPPED etc.

Notice there is no unique identifier with respect to this state

1. **Linear State Interface**

Got a unique Identifier along with state update.

**Example:**  Insurance Policy 100 has an exclusive identifier with respect to the customer like Customer ID+ Ins number. This could be a unique identifier

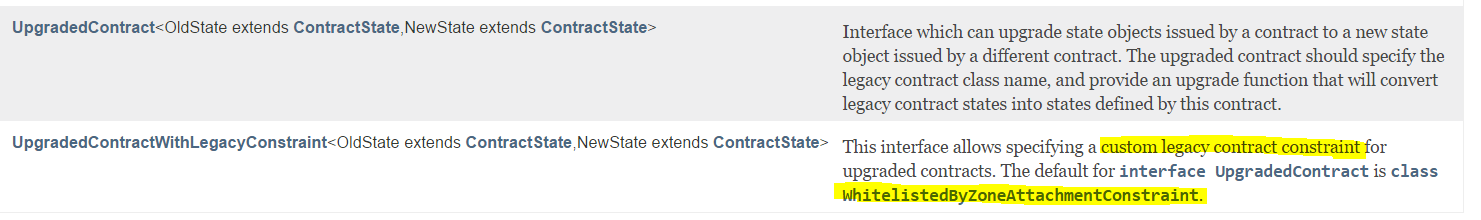
1. **QueryableState**

Support for querying the node database

**Example :** This is responsible for querying the state from its current schema using custom attributes

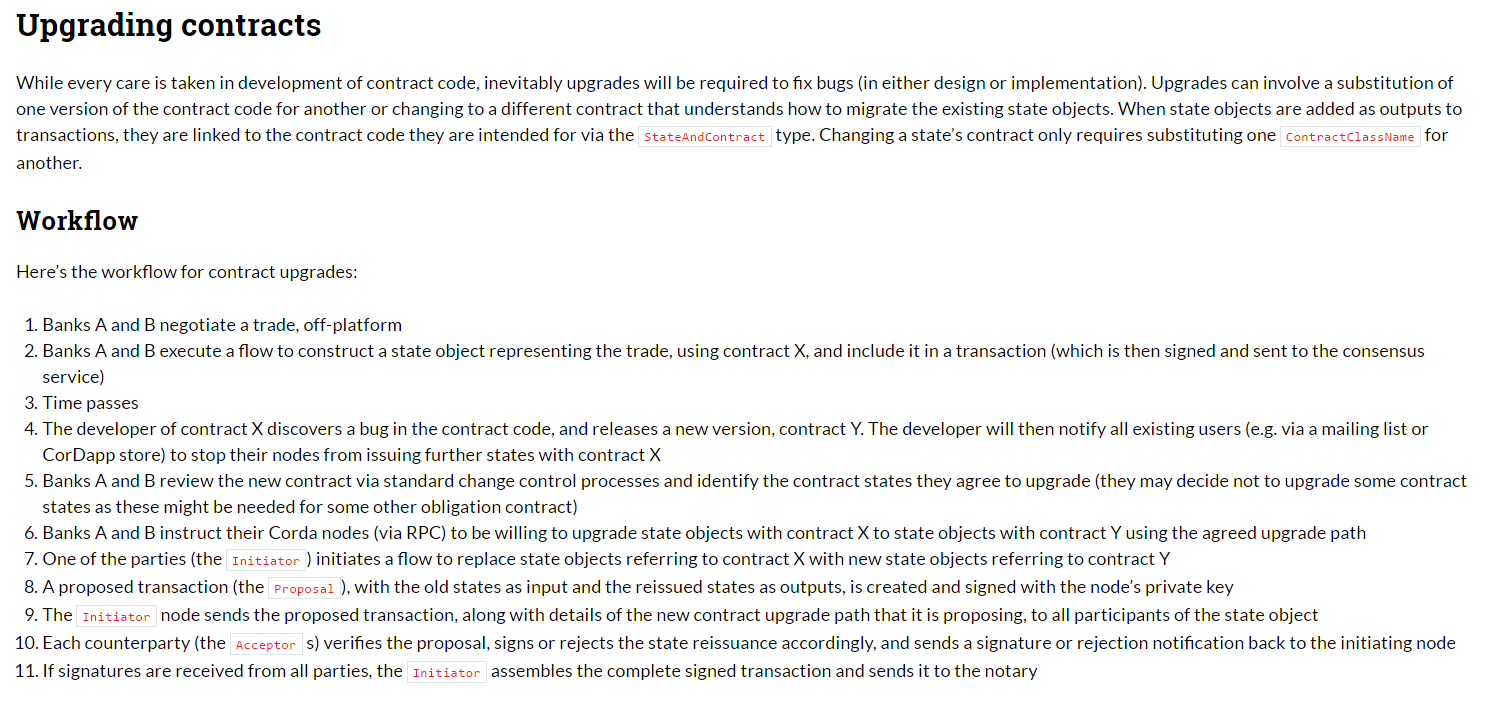
1. **UpgradedContract**

If suppose there is a bug identified after the contract is deployed. It can be upgraded if both the parties agree to it.

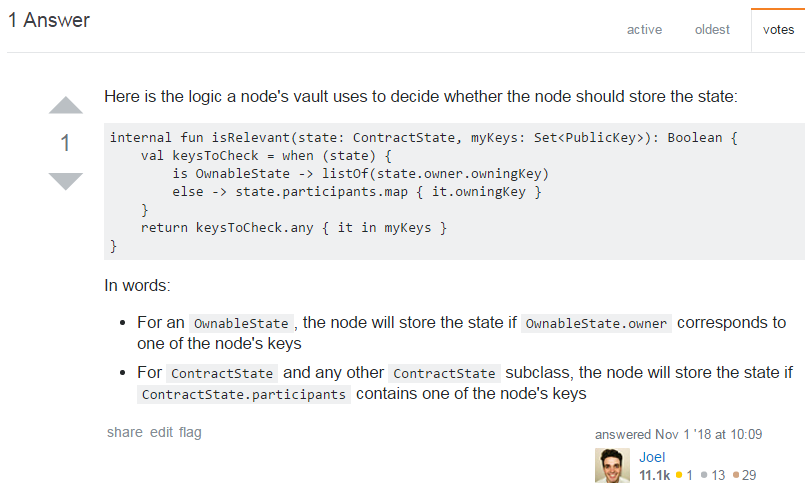


* 2nd one is when there is a custom whitelist zone

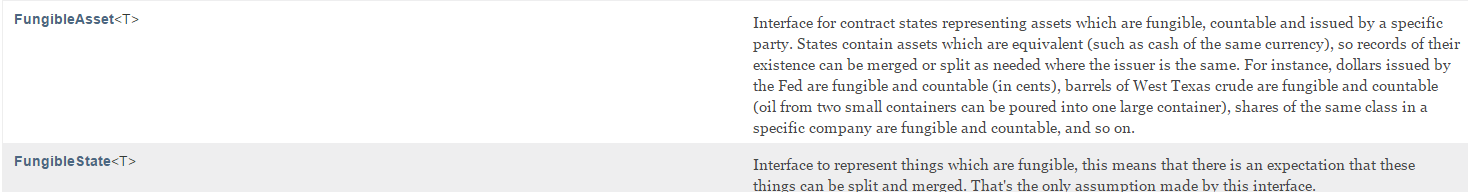
<https://docs.corda.net/contract-upgrade.html#workflow>

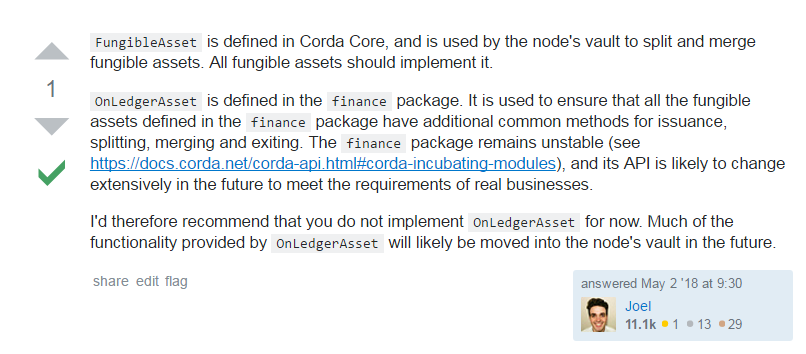


1. **OwnableState**



1. **Fungible State & Fungible Asset**





1. **Vault updates ( States Changes)**

* Ownable state 🡪 Owner who currently owns the state updates the vault
* Contract State 🡪 All the parties involved in the states updates the vault

Whenever states are irrelevant, they are not stored in vault. However transactions are recorded in transaction storage

**Contract State ( Primary Interface )**

* State should know list of parties involved with the state

Types of contract state ( Extended Further )

1. **Linear state** – got a linear id which can be tracked ( Ex: Batch ID)
2. **Ownable State** – Owner of the state ( another method represents new owner)
3. **Fungible State** – Split and Merge situation ( Cash 10 split into two 5’s)
   1. This can also define Fungible asset type – Ex: Cash
4. **Queryable State** – Query using custom attributes
5. **Schedulable State** – Pay at certain time
6. **Transaction State** –state wrapped up in Transaction Builder
7. **Reference State –** Input State reference

**VAULT**

* Transaction gets recorded in vault only when the corresponding Node/Party is part of the state . Otherwise it is not stored
* Owned state – default it is stored because the node is the owner of the state

Transaction State

Transaction budling or building for execution becomes a transaction state

Reference State

* Any state which are referred by the current state becomes reference state. Ex: USD/INR conversion rate state

**Notary Change**

**Case 1 : Reference State Input in a tx belongs to different notary**

A reference state is added to a transaction which is assigned to a different notary to the input and output states then all those inputs and outputs must be moved to the notary which the reference state uses

**Case 2 : Two or more Reference State Input in a tx belongs to multiple notary**

* unlikely that the party using the reference states has the authority to change the notary for the state
* it is likely that a transaction containing reference states with two different notaries cannot be committed to the ledger.

**State Pointers**

**<todo>**

**Persistence - API**

* Storing the contract state in RDMBS helps to run rich queries over the table
* Default database – H2 DB
* Every ContractState may implement the QueryableState interface if it wishes to be inserted into a custom table in the node’s database and made accessible using SQL
  + Mapped Schema is used to store the state which will be taken cared by SchemaService
* X509 Name is only stored as part of identity. If identity is anonymous then null is stored
* Java Persistence API – via Service Hub – Useful when you need to store off chain data

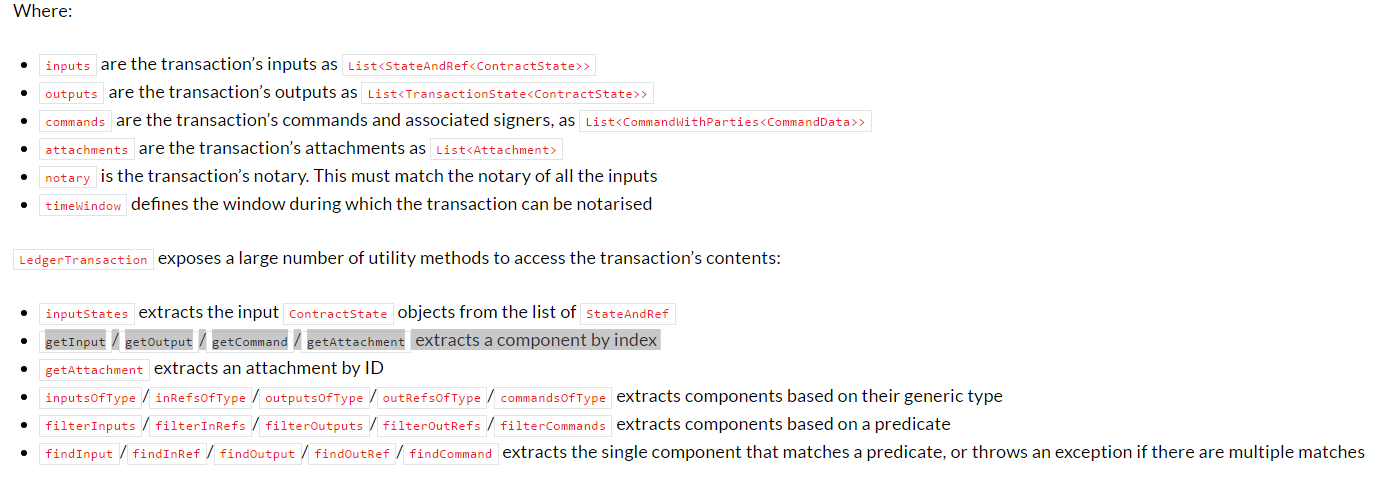
**Smart Contract – API**

VERIFY

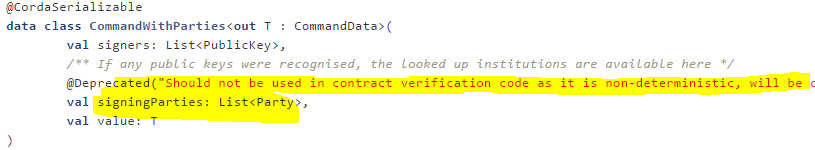
Tx proposal

(Ledger Tx )

Input & Output States



* Signatures verification (signing parties – in command) should not be used in contract verification code as it is non-deterministic, will be disabled for some future target platform version onwards and will take effect only for CorDapps targeting those versions.



**Contract Constraint – API**

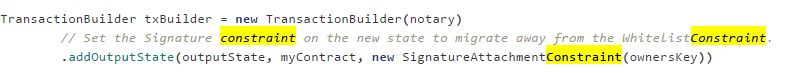
Helps in

1. Upgrade a Contract
2. Resist attacks

Following are the types of Constraints

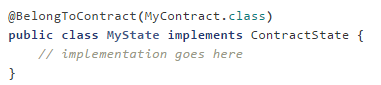
1. Hash Constraints HashAttachmentConstraint
2. Zone Whitelist - [WhitelistedByZoneAttachmentConstraint](https://docs.corda.net/api-contract-constraints.html#how-to-use-the-signatureattachmentconstraint-if-states-were-already-created-on-the-network-with-the-whitelistedbyzoneattachmentconstraint)
3. Signature Constraints - SignatureAttachmentConstraint
4. No Constraints @NoConstraintPropagation

**Constraints are mentioned in the flows. ( Transaction building process)**

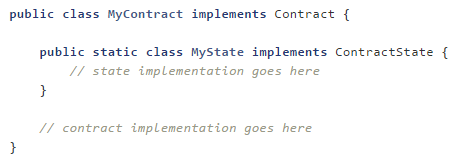


**When a contract is created, it is provided with a Fully defined Class Name**

* States are identified by fully qualified class name of a Contract implementation, and also a constraint is attached to it
* Contract State should always refer the respective Contract



(or)



APP

Version number provided while signing

* CORDAPP is installed with specific version . Ex : Version 1 , Respective State, Contract files, attachments are Zipped as JAR files

**Inefficient way**

The obvious way to write a CorDapp is to put all you states, contracts, flows and support code into a single Java module. This will work but it will effectively publish your entire app onto the ledger. That has two problems: (1) it is inefficient, and (2) it means changes to your flows or other parts of the app will be seen by the ledger as a “new app”, which may end up requiring essentially unnecessary upgrade procedures.

**Efficient way**

It’s better to split your app into multiple modules: one which contains just states, contracts and core data types. And another which contains the rest of the app.

**Transaction Builder - Process**

Transaction is built with the following parameters

1. Input States are referred / Possible Output States to be created post transaction
2. Attachment – Common reference doc, contract code, private encrypted sections
3. Command - Intent of the Transaction , also contains the signers list
4. Notary reference
5. Time Window ( Optional )

With all these steps Transaction is built and proposed to other Party

**Example :**

The below instructions are grouped inside FLOW

* Gather inputs 🡪 via the VaultService interface on the ServiceHub to locate the input states. In the below use case, inputs are needed from both the parties

State {

----

}

Flow {

TX Builder

Corresponds to a Contract

}

Party A Party B Notary

Gather Inputs (State ref)

from vault

Gather Inputs (State ref)

from vault ( if necessary)

Command which is part of transaction

provided the intent and the same can be

used by contract to verify it

Contract Code is executed

(Depending upon the rules,

Verify() method is executed)

Contract Code is executed

(Depending upon the rules,

Verify() method is executed)