



<https://docs.corda.net/key-concepts.html>

Key concepts

This section describes the key concepts and features of the Corda platform. It is intended for readers who are new to Corda, and want to understand its architecture. It does not contain any code, and is suitable for non-developers.

This section should be read in order:

- [The network](#)
- [The ledger](#)
- [Identity](#)
- [States](#)
- [Contracts](#)
- [Transactions](#)
- [Flows](#)
- [Consensus](#)
- [Notaries](#)
- [Time-windows](#)
- [Oracles](#)
- [Nodes](#)
- [Tradeoffs](#)

The detailed thinking and rationale behind these concepts are presented in two white papers:

- [Corda: An Introduction](#)
- [Corda: A Distributed Ledger](#) (A.K.A. the Technical White Paper)

Explanations of the key concepts are also available as [videos](#).

The network

Summary

- *A Corda network is made up of nodes running Corda and CorDapps*
- *The network is permissioned, with access controlled by a doorman*
- *Communication between nodes is point-to-point, instead of relying on global broadcasts*

Network structure

A Corda network is an authenticated peer-to-peer network of nodes, where each node is a JVM run-time environment hosting Corda services and executing applications known as *CorDapps*.

All communication between nodes is direct, with TLS-encrypted messages sent over AMQP/1.0. This means that data is shared only on a need-to-know basis; in Corda, there are **no global broadcasts**.

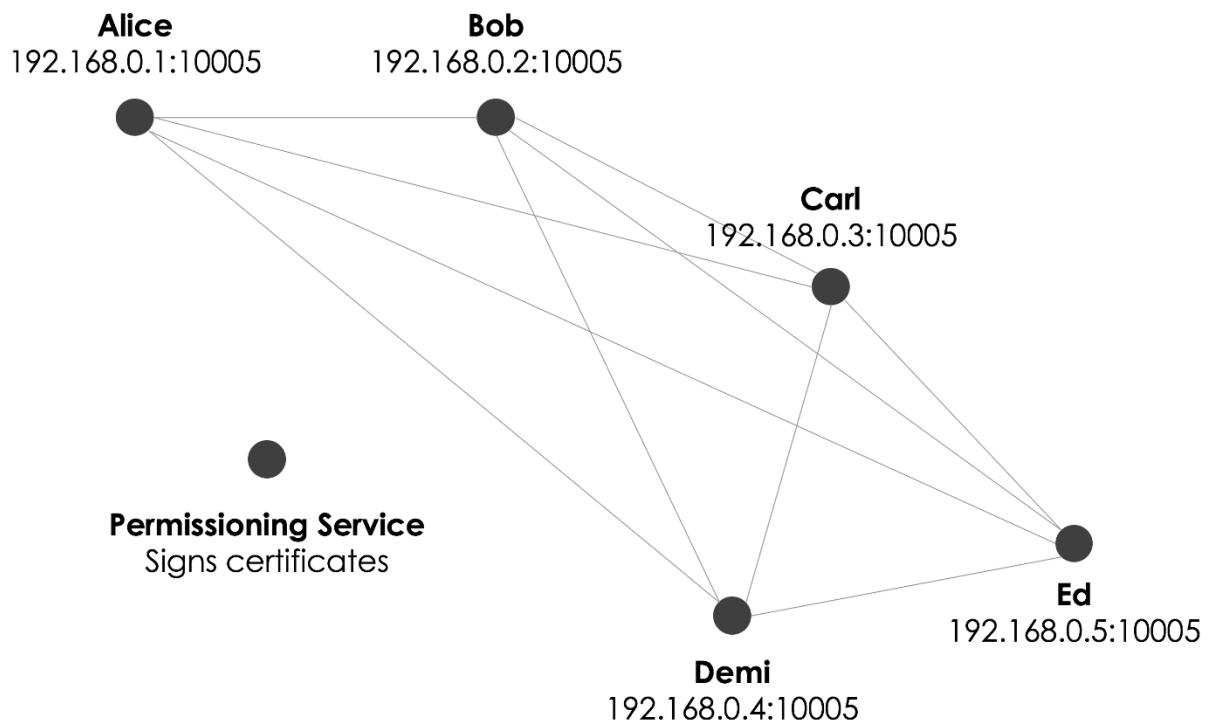
Each network has a **network map service** that publishes the IP addresses through which every node on the network can be reached, along with the identity certificates of those nodes and the services they provide.

The doorman

Corda networks are semi-private. Each network has a doorman service that enforces rules regarding the information that nodes must provide and the know-your-customer processes that they must complete before being admitted to the network.

To join the network, a node must contact the doorman and provide the required information. If the doorman is satisfied, the node will receive a root-authority-signed TLS certificate from the network's permissioning service. This certificate certifies the node's identity when communicating with other participants on the network.

We can visualize a network as follows:



Network services

Nodes can provide several types of services:

- One or more pluggable **notary services**. Notaries guarantee the uniqueness, and possibility the validity, of ledger updates. Each notary service may be run on a single node, or across a cluster of nodes.
- Zero or more **oracle services**. An oracle is a well-known service that signs transactions if they state a fact and that fact is considered to be true.

[Next](#) [Previous](#)

The ledger

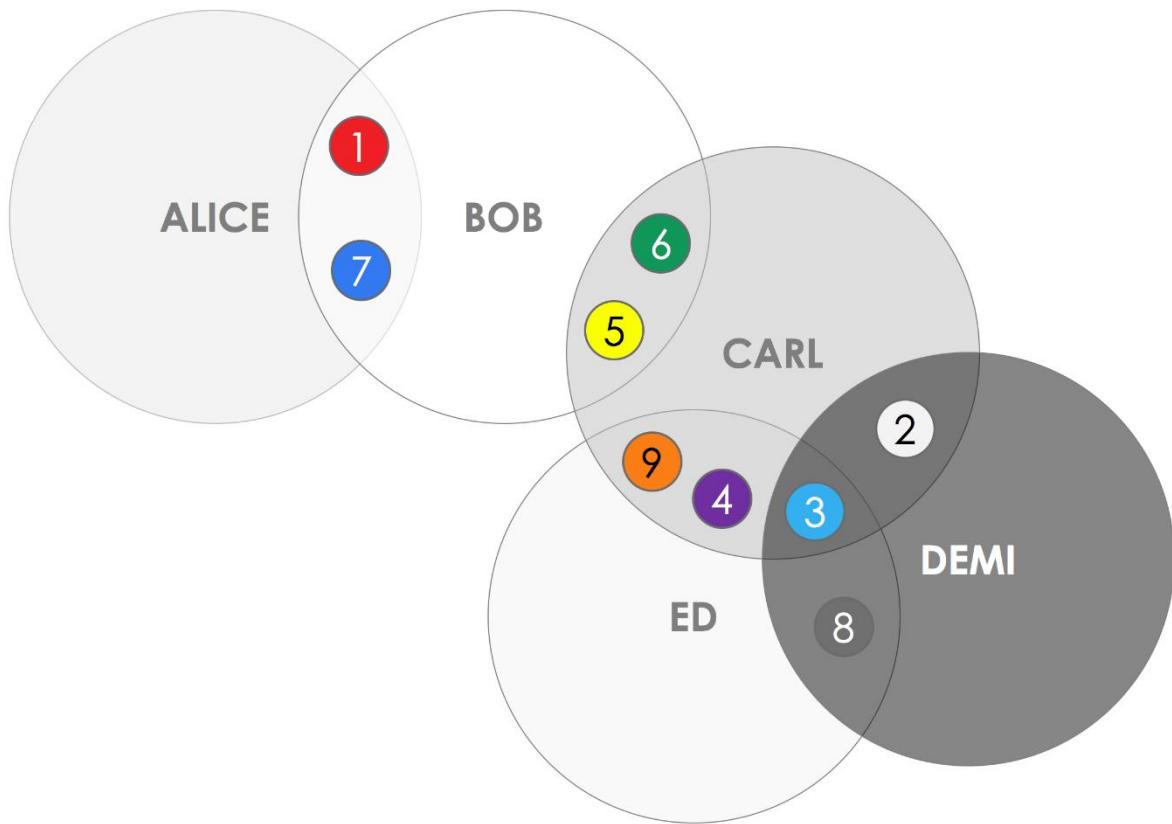
Summary

- *The ledger is subjective from each peer's perspective*
- *Two peers are always guaranteed to see the exact same version of any on-ledger facts they share*

Overview

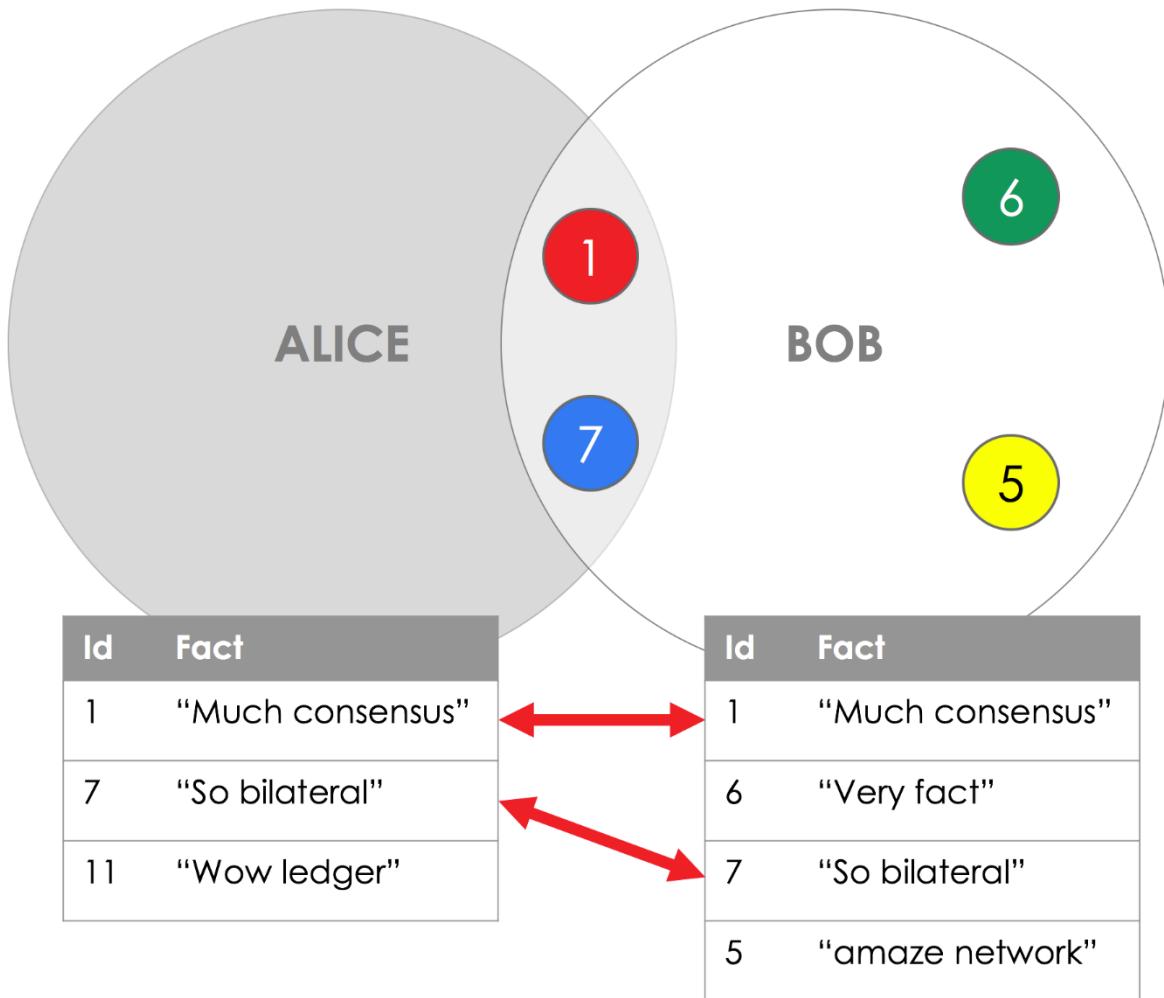
In Corda, there is **no single central store of data**. Instead, each node maintains a separate database of known facts. As a result, each peer only sees a subset of facts on the ledger, and no peer is aware of the ledger in its entirety.

For example, imagine a network with five nodes, where each coloured circle represents a shared fact:



We can see that although Carl, Demi and Ed are aware of shared fact 3, **Alice and Bob are not**.

Equally importantly, Corda guarantees that whenever one of these facts is shared by multiple nodes on the network, it evolves in lockstep in the database of every node that is aware of it:



For example, Alice and Bob will both see the **exact same version** of shared facts 1 and 7.

[Next](#) [Previous](#)

Identity

Summary

- *Identities in Corda can represent legal identities or service identities*
- *Identities are attested to by X.509 certificate signed by the Doorman or a well known identity*
- *Well known identities are published in the network map*
- *Confidential identities are only shared on a need to know basis*

Identities in Corda can represent:

- Legal identity of an organisation
- Service identity of a network service

Legal identities are used for parties in a transaction, such as the owner of a cash state. Service identities are used for those providing transaction-related services, such as notary, or oracle. Service identities are distinct to legal identities so that distributed services can exist on nodes owned by different organisations. Such distributed service identities are based on [CompositeKeys](#), which describe the valid sets of signers for a signature from the service. See [API: Core types](#) for more technical detail on composite keys.

Identities are either well known or confidential, depending on whether their X.509 certificate (and corresponding certificate path to a trusted root certificate) is published:

- Well known identities are the generally identifiable public key of a legal entity or service, which makes them ill-suited to transactions where confidentiality of participants is required. This certificate is published in the network map service for anyone to access.
- Confidential identities are only published to those who are involved in transactions with the identity. The public key may be exposed to third parties (for example to the notary service), but distribution of the name and X.509 certificate is limited.

Although there are several elements to the Corda transaction privacy model, including ensuring that transactions are only shared with those who need to see them, and planned use of Intel SGX, it is important to provide defense in depth against privacy breaches. Confidential identities are used to ensure that even if a third party gets access to an unencrypted transaction, they cannot identify the participants without additional information.

Certificates

Nodes must be able to verify the identity of the owner of a public key, which is achieved using X.509 certificates. When first run a node generates a key pair and submits a certificate signing request to the network Doorman service (see [Network permissioning](#)). The Doorman service applies appropriate identity checks then issues a certificate to the node, which is used as the node certificate authority (CA). From this initial CA certificate the node automatically creates and

signs two further certificates, a TLS certificate and a signing certificate for the node's well known identity. Finally the node builds a node info record containing its address and well known identity, and registers it with the network map service.

From the signing certificate the organisation can create both well known and confidential identities. Use-cases for well known identities include clusters of nodes representing a single identity for redundancy purposes, or creating identities for organisational units.

It is up to organisations to decide which identities they wish to publish in the network map service, making them well known, and which they wish to keep as confidential identities for privacy reasons (typically to avoid exposing business sensitive details of transactions). In some cases nodes may also use private network map services in addition to the main network map service, for operational reasons. Identities registered with such network maps must be considered well known, and it is never appropriate to store confidential identities in a central directory without controls applied at the record level to ensure only those who require access to an identity can retrieve its

States

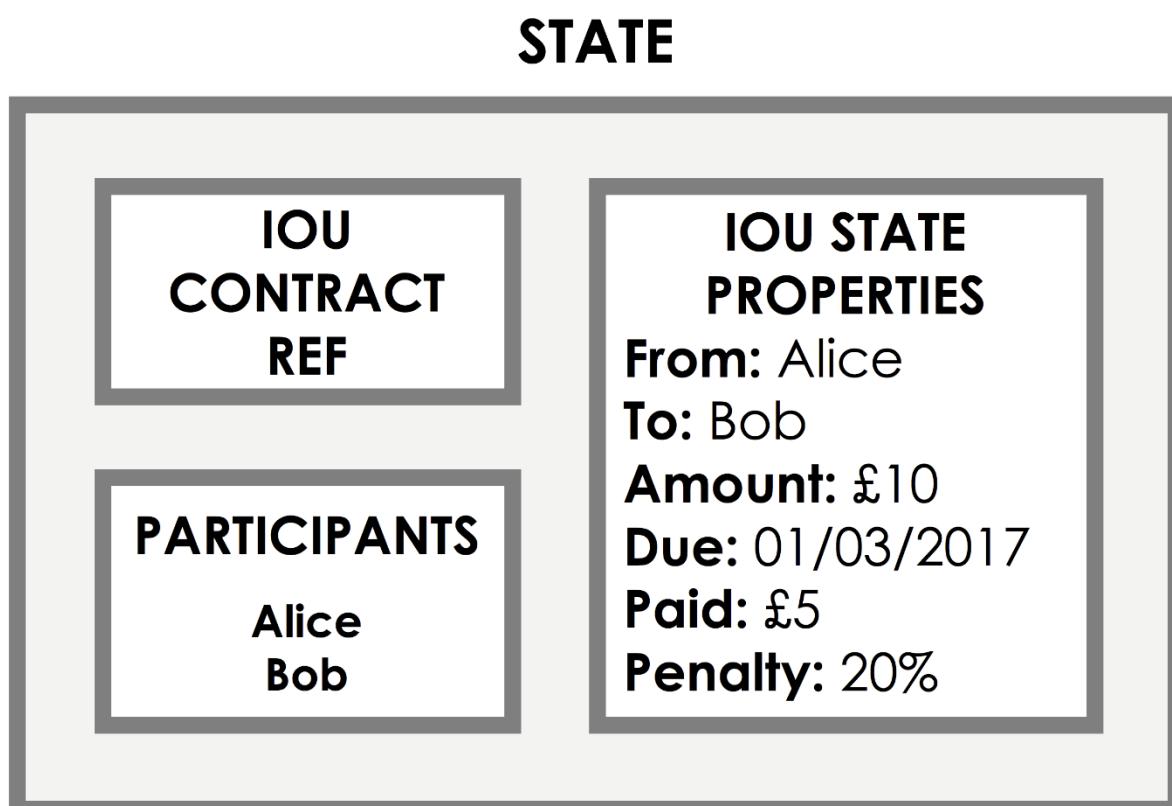
Summary

- States represent on-ledger facts
- States are evolved by marking the current state as historic and creating an updated state
- Each node has a vault where it stores any relevant states to itself

Overview

A *state* is an immutable object representing a fact known by one or more Corda nodes at a specific point in time. States can contain arbitrary data, allowing them to represent facts of any kind (e.g. stocks, bonds, loans, KYC data, identity information...).

For example, the following state represents an IOU - an agreement that Alice owes Bob an amount X:



Specifically, this state represents an IOU of £10 from Alice to Bob.

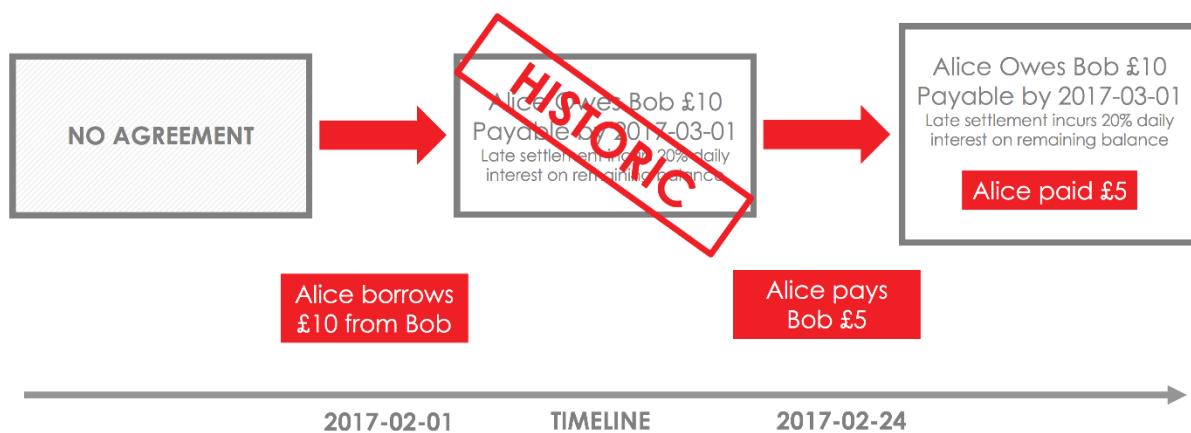
As well as any information about the fact itself, the state also contains a reference to the *contract* that governs the evolution of the state over time. We discuss contracts in [Contracts](#).

State sequences

As states are immutable, they cannot be modified directly to reflect a change in the state of the world.

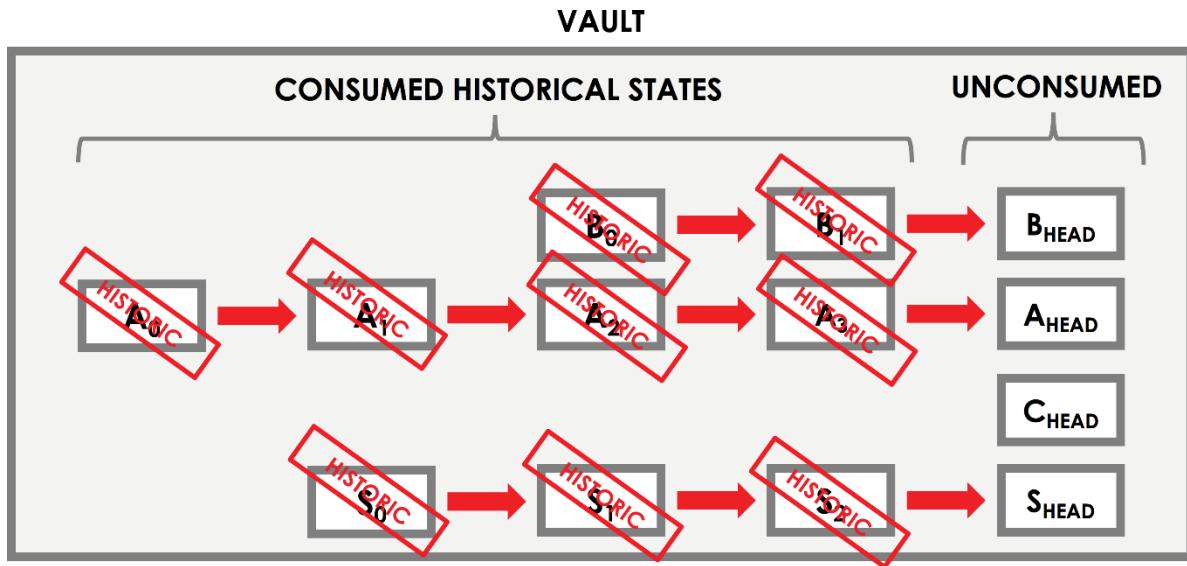
Instead, the lifecycle of a shared fact over time is represented by a **state sequence**. When a state needs to be updated, we create a new version of the state representing the new state of the world, and mark the existing state as historic.

This sequence of state replacements gives us a full view of the evolution of the shared fact over time. We can picture this situation as follows:



The vault

Each node on the network maintains a *vault* - a database where it tracks all the current and historic states that it is aware of, and which it considers to be relevant to itself:



We can think of the ledger from each node's point of view as the set of all the current (i.e. non-historic) states that it is aware of.

Contracts

Summary

- A valid transaction must be accepted by the contract of each of its input and output states
- Contracts are written in a JVM programming language (e.g. Java or Kotlin)
- Contract execution is deterministic and its acceptance of a transaction is based on the transaction's contents alone

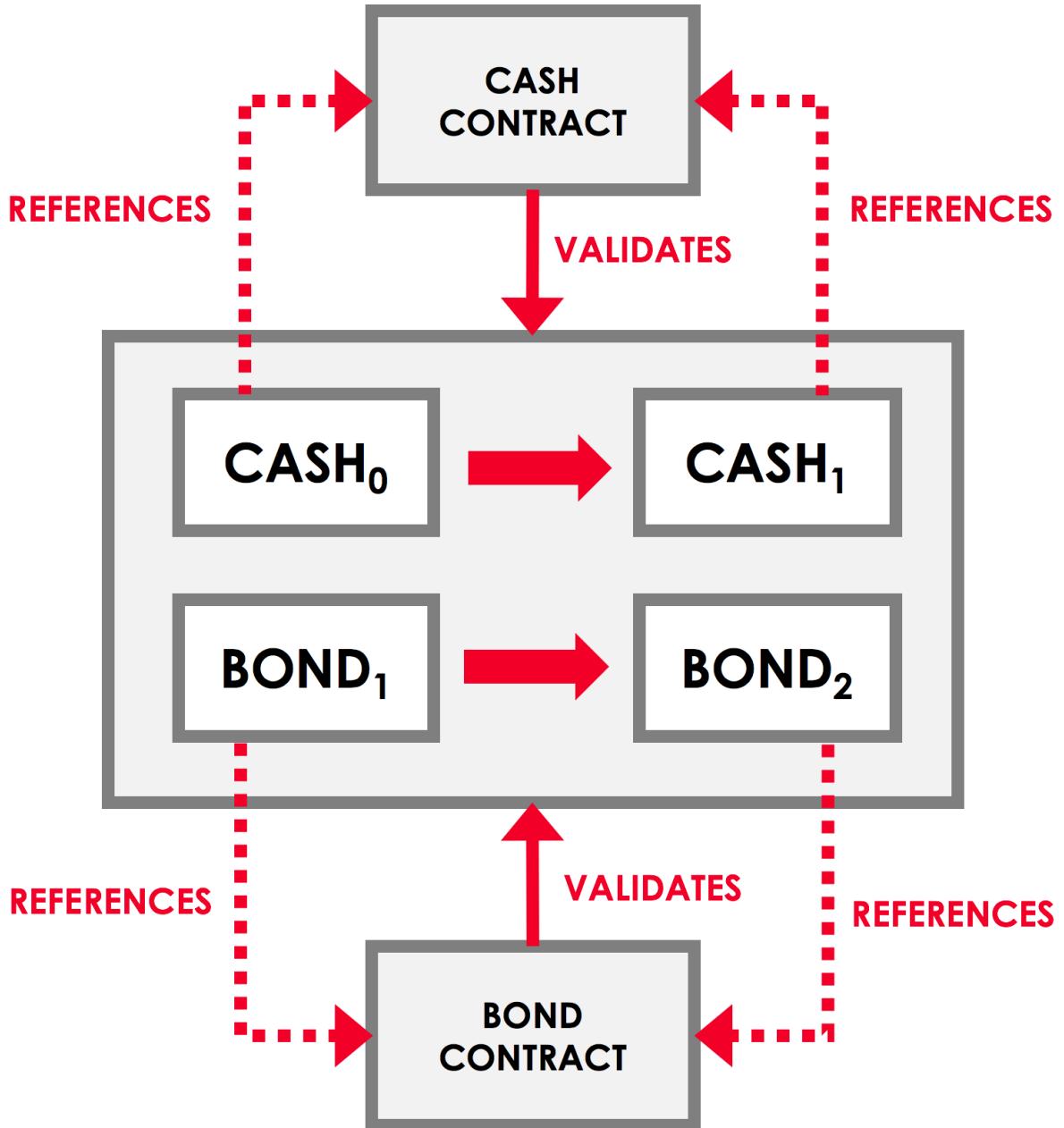
Transaction verification

Recall that a transaction is only valid if it is digitally signed by all required signers. However, even if a transaction gathers all the required signatures, it is only valid if it is also **contractually valid**.

Contract validity is defined as follows:

- Each state points to a *contract*
- A *contract* takes a transaction as input, and states whether the transaction is considered valid based on the contract's rules
- A transaction is only valid if the contract of **every input state** and **every output state** considers it to be valid

We can picture this situation as follows:



The contract code can be written in any JVM language, and has access to the full capabilities of the language, including:

- Checking the number of inputs, outputs, commands, timestamps, and/or attachments
- Checking the contents of any of these components
- Looping constructs, variable assignment, function calls, helper methods, etc.
- Grouping similar states to validate them as a group (e.g. imposing a rule on the combined value of all the cash states)

A transaction that is not contractually valid is not a valid proposal to update the ledger, and thus can never be committed to the ledger. In this way, contracts

impose rules on the evolution of states over time that are independent of the willingness of the required signers to sign a given transaction.

The contract sandbox

Transaction verification must be *deterministic* - a contract should either **always accept** or **always reject** a given transaction. For example, transaction validity cannot depend on the time at which validation is conducted, or the amount of information the peer running the contract holds. This is a necessary condition to ensure that all peers on the network reach consensus regarding the validity of a given ledger update.

To achieve this, contracts evaluate transactions in a deterministic sandbox. The sandbox has a whitelist that prevents the contract from importing libraries that could be a source of non-determinism. This includes libraries that provide the current time, random number generators, libraries that provide filesystem access or networking libraries, for example. Ultimately, the only information available to the contract when verifying the transaction is the information included in the transaction itself.

Contract limitations

Since a contract has no access to information from the outside world, it can only check the transaction for internal validity. It cannot check, for example, that the transaction is in accordance with what was originally agreed with the counterparties.

Peers should therefore check the contents of a transaction before signing it, *even if the transaction is contractually valid*, to see whether they agree with the proposed ledger update. A peer is under no obligation to sign a transaction just because it is contractually valid. For example, they may be unwilling to take on a loan that is too large, or may disagree on the amount of cash offered for an asset.

Oracles

Sometimes, transaction validity will depend on some external piece of information, such as an exchange rate. In these cases, an oracle is required. See [Oracles](#) for further details.

Each contract also refers to a legal prose document that states the rules governing the evolution of the state over time in a way that is compatible with traditional legal systems. This document can be relied upon in the case of legal disputes.

Transactions

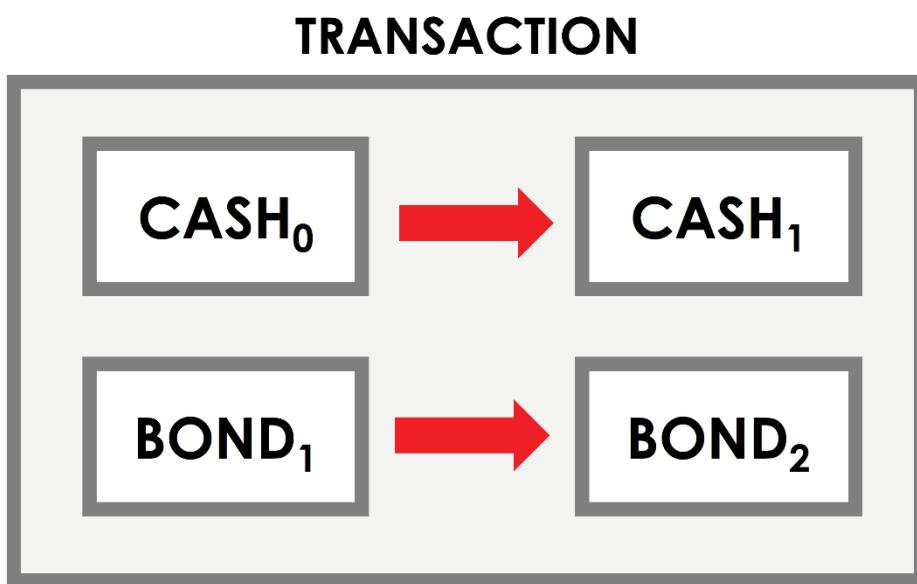
Summary

- *Transactions are proposals to update the ledger*
- *A transaction proposal will only be committed if:*
 - *It doesn't contain double-spends*
 - *It is contractually valid*
 - *It is signed by the required parties*

Overview

Corda uses a *UTXO* (unspent transaction output) model where every state on the ledger is immutable. The ledger evolves over time by applying *transactions*, which update the ledger by marking zero or more existing ledger states as historic (the *inputs*) and producing zero or more new ledger states (the *outputs*). Transactions represent a single link in the state sequences seen in [States](#).

Here is an example of an update transaction, with two inputs and two outputs:



A transaction can contain any number of inputs and outputs of any type:

- They can include many different state types (e.g. both cash and bonds)
- They can be issuances (have zero inputs) or exits (have zero outputs)

- They can merge or split fungible assets (e.g. combining a \$2 state and a \$5 state into a \$7 cash state)

Transactions are *atomic*: either all the transaction's proposed changes are accepted, or none are. There are two basic types of transactions:

- Notary-change transactions (used to change a state's notary - see [Notaries](#))
- General transactions (used for everything else)

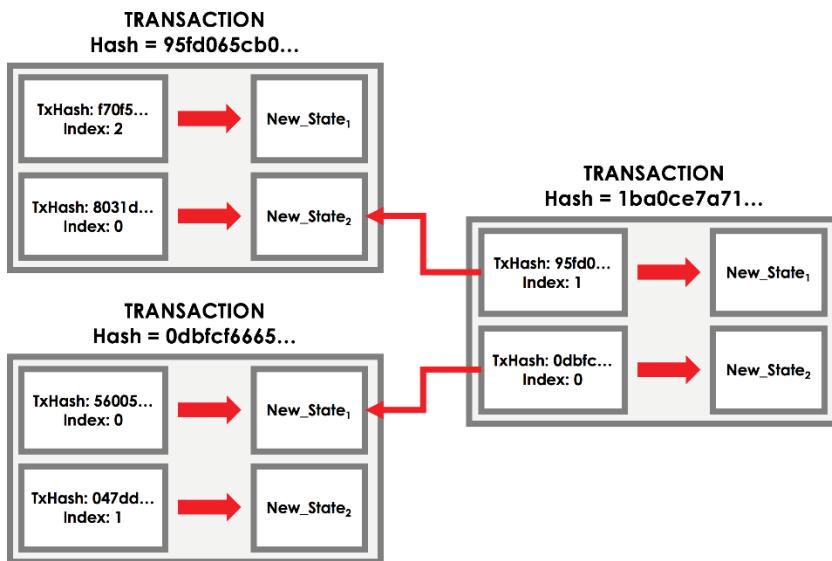
Transaction chains

When creating a new transaction, the output states that the transaction will propose do not exist yet, and must therefore be created by the proposer(s) of the transaction. However, the input states already exist as the outputs of previous transactions. We therefore include them in the proposed transaction by reference.

These input states references are a combination of:

- The hash of the transaction that created the input
- The input's index in the outputs of the previous transaction

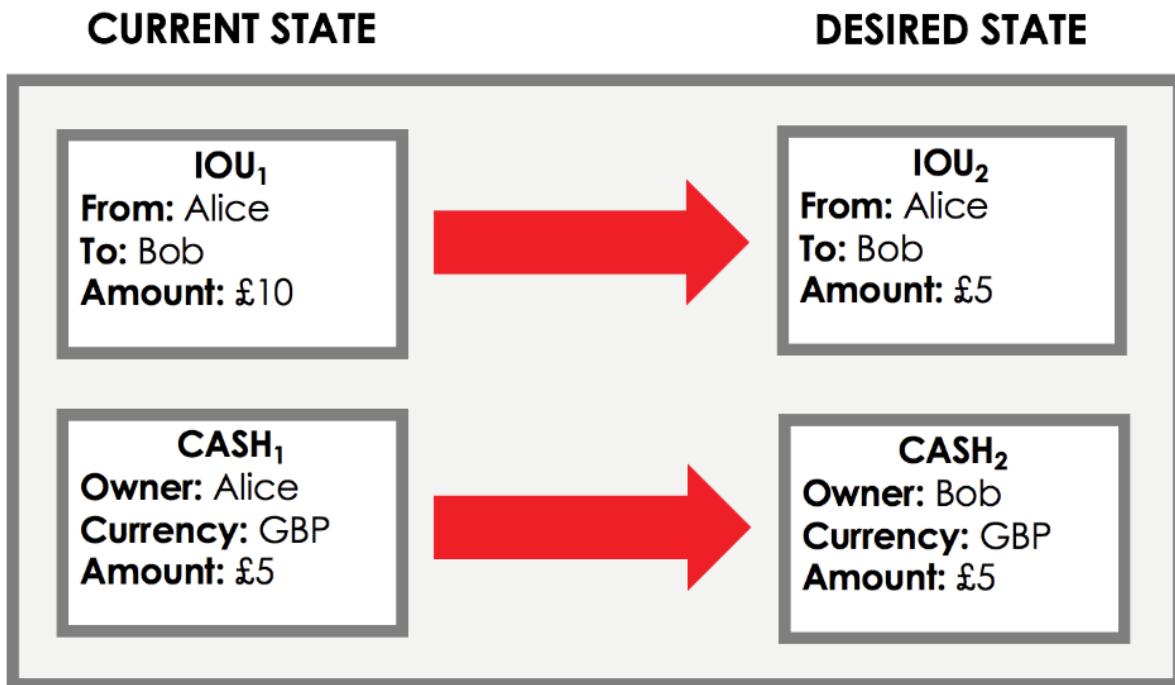
This situation can be illustrated as follows:



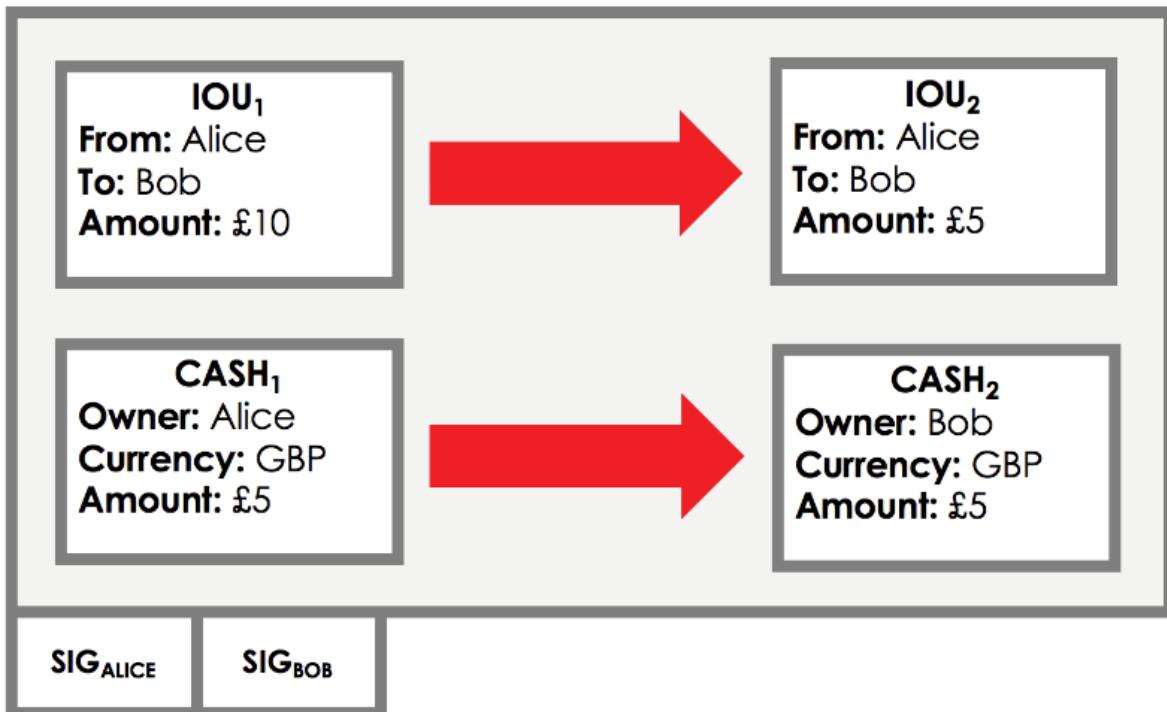
These input state references link together transactions over time, forming what is known as a *transaction chain*.

Committing transactions

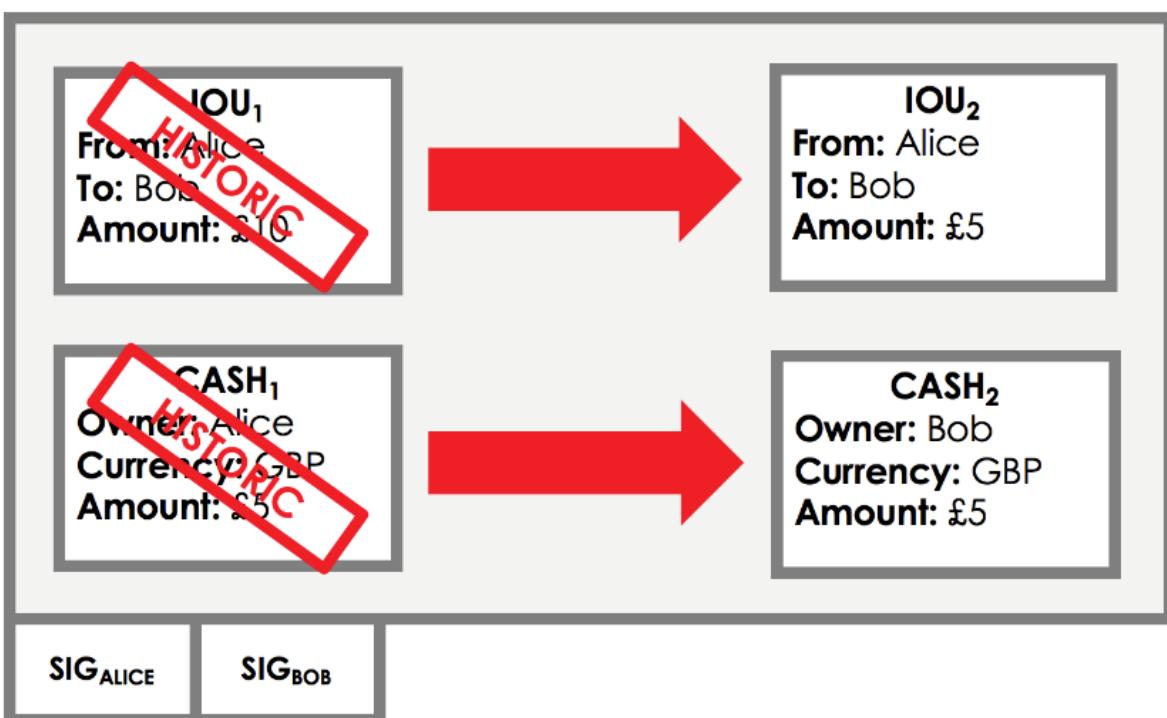
Initially, a transaction is just a **proposal** to update the ledger. It represents the future state of the ledger that is desired by the transaction builder(s):



To become reality, the transaction must receive signatures from all of the *required signers* (see **Commands**, below). Each required signer appends their signature to the transaction to indicate that they approve the proposal:



If all of the required signatures are gathered, the transaction becomes committed:



This means that:

- The transaction's inputs are marked as historic, and cannot be used in any future transactions
- The transaction's outputs become part of the current state of the ledger

Transaction validity

Each required signers should only sign the transaction if the following two conditions hold:

- **Transaction validity:** For both the proposed transaction, and every transaction in the chain of transactions that created the current proposed transaction's inputs:
 - The transaction is digitally signed by all the required parties
 - The transaction is *contractually valid* (see [Contracts](#))
- **Transaction uniqueness:** There exists no other committed transaction that has consumed any of the inputs to our proposed transaction (see [Consensus](#))

If the transaction gathers all the required signatures but these conditions do not hold, the transaction's outputs will not be valid, and will not be accepted as inputs to subsequent transactions.

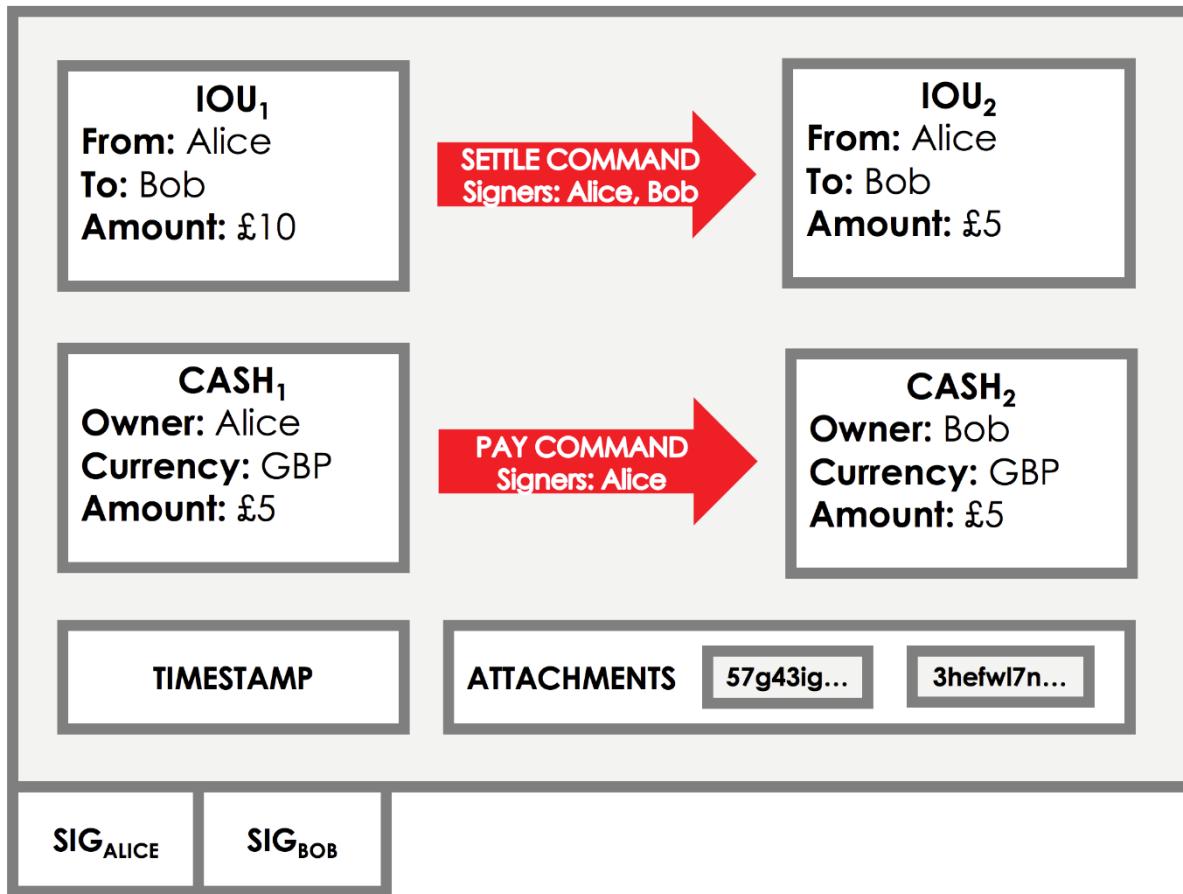
Other transaction components

As well as input states and output states, transactions may contain:

- Commands
- Attachments
- Timestamps

For example, a transaction where Alice pays off £5 of an IOU with Bob using a £5 cash payment, supported by two attachments and a timestamp, may look as

follows:



We explore the role played by the remaining transaction components below.

Suppose we have a transaction with a cash state and a bond state as inputs, and a cash state and a bond state as outputs. This transaction could represent two different scenarios:

- A bond purchase
- A coupon payment on a bond

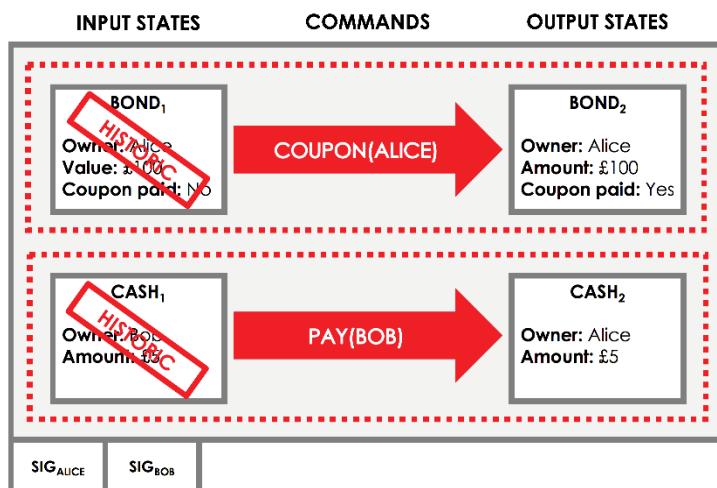
We can imagine that we'd want to impose different rules on what constitutes a valid transaction depending on whether this is a purchase or a coupon payment. For example, in the case of a purchase, we would require a change in the bond's current owner, whereas in the case of a coupon payment, we would require that the ownership of the bond does not change.

For this, we have *commands*. Including a command in a transaction allows us to indicate the transaction's intent, affecting how we check the validity of the transaction.

Each command is also associated with a list of one or more *signers*. By taking the union of all the public keys listed in the commands, we get the list of the transaction's required signers. In our example, we might imagine that:

- In a coupon payment on a bond, only the owner of the bond is required to sign
- In a cash payment, only the owner of the cash is required to sign

We can visualize this situation as follows:



Sometimes, we have a large piece of data that can be reused across many different transactions. Some examples:

- A calendar of public holidays
- Supporting legal documentation
- A table of currency codes

For this use case, we have *attachments*. Each transaction can refer to zero or more attachments by hash. These attachments are ZIP/JAR files containing arbitrary content. The information in these files can then be used when checking the transaction's validity.

Time-windows

In some cases, we want a transaction proposed to only be approved during a certain time-window. For example:

- An option can only be exercised after a certain date
- A bond may only be redeemed before its expiry date

In such cases, we can add a *time-window* to the transaction. Time-windows specify the time window during which the transaction can be committed. We discuss time-windows in the section on Time-windows.

Flows

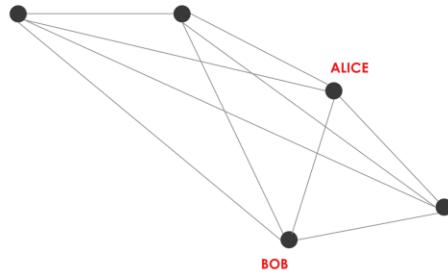
Summary

- *Flows automate the process of agreeing ledger updates*
- *Communication between nodes only occurs in the context of these flows, and is point-to-point*
- *Built-in flows are provided to automate common tasks*

Motivation

Corda networks use point-to-point messaging instead of a global broadcast. This means that coordinating a ledger update requires network participants to specify exactly what information needs to be sent, to which counterparties, and in what order.

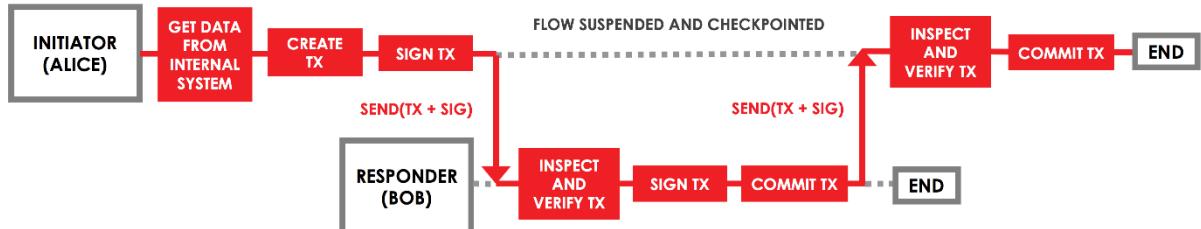
Here is a visualisation of the process of agreeing a simple ledger update between Alice and Bob:



The flow framework

Rather than having to specify these steps manually, Corda automates the process using *flows*. A flow is a sequence of steps that tells a node how to achieve a specific ledger update, such as issuing an asset or settling a trade.

Here is the sequence of flow steps involved in the simple ledger update above:



Running flows

Once a given business process has been encapsulated in a flow and installed on the node as part of a CorDapp, the node's owner can instruct the node to kick off this business process at any time using an RPC call. The flow abstracts all the networking, I/O and concurrency issues away from the node owner.

All activity on the node occurs in the context of these flows. Unlike contracts, flows do not execute in a sandbox, meaning that nodes can perform actions such as networking, I/O and use sources of randomness within the execution of a flow.

Inter-node communication

Nodes communicate by passing messages between flows. Each node has zero or more flow classes that are registered to respond to messages from a single other flow.

Suppose Alice is a node on the network and wishes to agree a ledger update with Bob, another network node. To communicate with Bob, Alice must:

- Start a flow that Bob is registered to respond to
- Send Bob a message within the context of that flow
- Bob will start its registered counterparty flow

Now that a connection is established, Alice and Bob can communicate to agree a ledger update by passing a series of messages back and forth, as prescribed by the flow steps.

Subflows

Flows can be composed by starting a flow as a subprocess in the context of another flow. The flow that is started as a subprocess is known as a *subflow*. The parent flow will wait until the subflow returns.

The flow library

Corda provides a library of flows to handle common tasks, meaning that developers do not have to redefine the logic behind common processes such as:

- Notarising and recording a transaction
- Gathering signatures from counterparty nodes
- Verifying a chain of transactions

Further information on the available built-in flows can be found in [Flow library](#).

Concurrency

The flow framework allows nodes to have many flows active at once. These flows may last days, across node restarts and even upgrades.

This is achieved by serializing flows to disk whenever they enter a blocking state (e.g. when they're waiting on I/O or a networking call). Instead of waiting for the flow to become unblocked, the node immediately starts work on any other scheduled flows, only returning to the original flow at a later date.

[Next](#) [Previous](#)

Consensus

Summary

- *To be committed, transactions must achieve both validity and uniqueness consensus*
- *Validity consensus requires contractual validity of the transaction and all its dependencies*

Uniqueness consensus prevents double-spend Two types of consensus

Determining whether a proposed transaction is a valid ledger update involves reaching two types of consensus:

- *Validity consensus* - this is checked by each required signer before they sign the transaction

- *Uniqueness consensus* - this is only checked by a notary service

Validity consensus

Validity consensus is the process of checking that the following conditions hold both for the proposed transaction, and for every transaction in the transaction chain that generated the inputs to the proposed transaction:

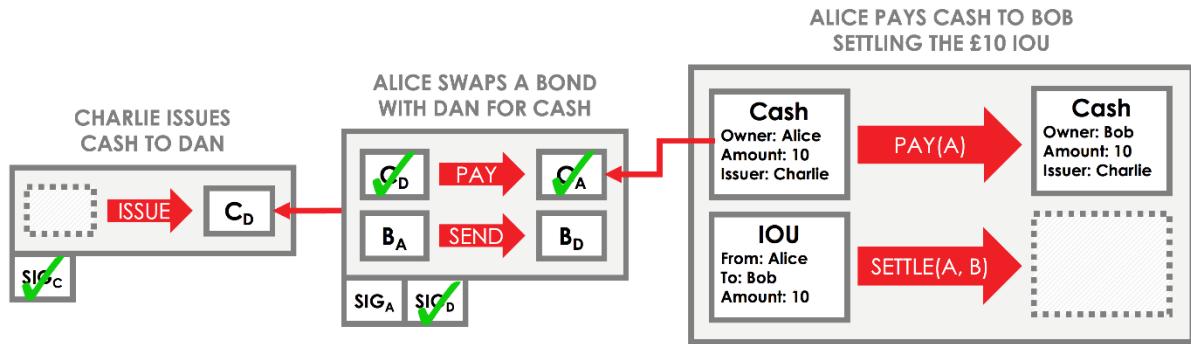
- The transaction is accepted by the contracts of every input and output state
- The transaction has all the required signatures

It is not enough to verify the proposed transaction itself. We must also verify every transaction in the chain of transactions that led up to the creation of the inputs to the proposed transaction.

This is known as *walking the chain*. Suppose, for example, that a party on the network proposes a transaction transferring us a treasury bond. We can only be sure that the bond transfer is valid if:

- The treasury bond was issued by the central bank in a valid issuance transaction
- Every subsequent transaction in which the bond changed hands was also valid

The only way to be sure of both conditions is to walk the transaction's chain. We can visualize this process as follows:



When verifying a proposed transaction, a given party may not have every transaction in the transaction chain that they need to verify. In this case, they can request the missing transactions from the transaction proposer(s).

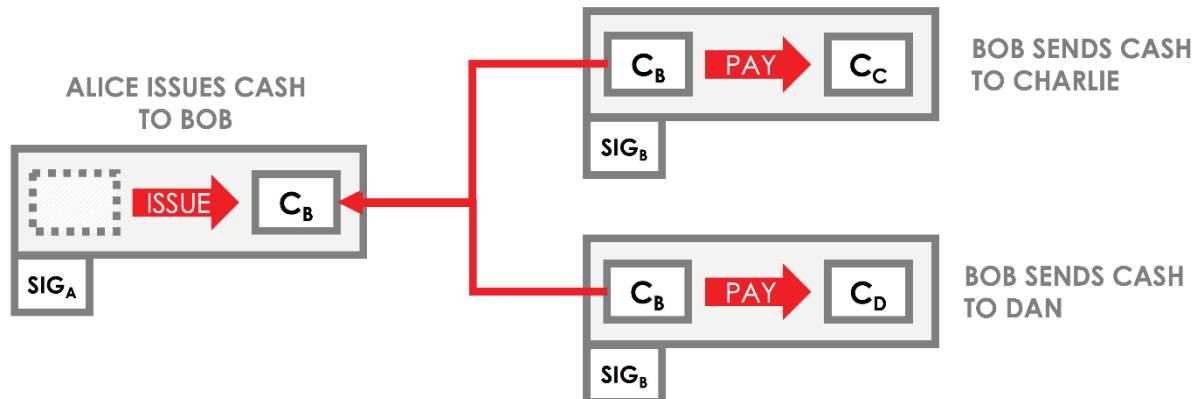
The transaction proposer(s) will always have the full transaction chain, since they would have requested it when verifying the transaction that created the proposed transaction's input states.

Uniqueness consensus

Imagine that Bob holds a valid central-bank-issued cash state of \$1,000,000. Bob can now create two transaction proposals:

- A transaction transferring the \$1,000,000 to Charlie in exchange for £800,000
- A transaction transferring the \$1,000,000 to Dan in exchange for €900,000

This is a problem because, although both transactions will achieve validity consensus, Bob has managed to “double-spend” his USD to get double the amount of GBP and EUR. We can visualize this as follows:



To prevent this, a valid transaction proposal must also achieve uniqueness consensus. Uniqueness consensus is the requirement that none of the inputs to a proposed transaction have already been consumed in another transaction.

If one or more of the inputs have already been consumed in another transaction, this is known as a *double spend*, and the transaction proposal is considered invalid.

Uniqueness consensus is provided by notaries. See [Notaries](#) for more details

Notaries

Summary

- *Notaries prevent “double-spends”*
- *Notaries may optionally also validate transactions*
- *A network can have several notaries, each running a different consensus algorithm*

Overview

A *notary* is a network service that provides **uniqueness consensus** by attesting that, for a given transaction, it has not already signed other transactions that consumes any of the proposed transaction’s input states.

Upon being sent asked to notarise a transaction, a notary will either:

- Sign the transaction if it has not already signed other transactions consuming any of the proposed transaction’s input states
- Reject the transaction and flag that a double-spend attempt has occurred otherwise

In doing so, the notary provides the point of finality in the system. Until the notary’s signature is obtained, parties cannot be sure that an equally valid, but conflicting, transaction will not be regarded as the “valid” attempt to spend a given input state. However, after the notary’s signature is obtained, we can be sure that the proposed transaction’s input states had not already been consumed by a prior transaction. Hence, notarisation is the point of finality in the system.

Every state has an appointed notary, and a notary will only notarise a transaction if it is the appointed notary of all the transaction's input states.

Consensus algorithms

Corda has “pluggable” consensus, allowing notaries to choose a consensus algorithm based on their requirements in terms of privacy, scalability, legal-system compatibility and algorithmic agility.

In particular, notaries may differ in terms of:

- **Structure** - a notary may be a single network node, a cluster of mutually-trusting nodes, or a cluster of mutually-distrusting nodes
- **Consensus algorithm** - a notary service may choose to run a high-speed, high-trust algorithm such as RAFT, a low-speed, low-trust algorithm such as BFT, or any other consensus algorithm it chooses

Validation

A notary service must also decide whether or not to provide **validity consensus** by validating each transaction before committing it. In making this decision, they face the following trade-off:

- If a transaction **is not** checked for validity, it creates the risk of “denial of state” attacks, where a node knowingly builds an invalid transaction consuming some set of existing states and sends it to the notary, causing the states to be marked as consumed
- If the transaction **is** checked for validity, the notary will need to see the full contents of the transaction and its dependencies. This leaks potentially private data to the notary

There are several further points to keep in mind when evaluating this trade-off. In the case of the non-validating model, Corda’s controlled data distribution model means that information on unconsumed states is not widely shared. Additionally, Corda’s permissioned network means that the notary can store to the identity of the party that created the “denial of state” transaction, allowing the attack to be resolved off-ledger.

In the case of the validating model, the use of anonymous, freshly-generated public keys instead of legal identities to identify parties in a transaction limit the information the notary sees.

Multiple notaries

Each Corda network can have multiple notaries, each potentially running a different consensus algorithm. This provides several benefits:

- **Privacy** - we can have both validating and non-validating notary services on the same network, each running a different algorithm. This allows nodes to choose the preferred notary on a per-transaction basis
- **Load balancing** - spreading the transaction load over multiple notaries allows higher transaction throughput for the platform overall
- **Low latency** - latency can be minimised by choosing a notary physically closer to the transacting parties

Changing notaries

Remember that a notary will only sign a transaction if it is the appointed notary of all of the transaction's input states. However, there are cases in which we may need to change a state's appointed notary. These include:

- When a single transaction needs to consume several states that have different appointed notaries
-
- When a node would prefer to use a different notary for a given transaction due to privacy or efficiency concerns
-

Before these transactions can be created, the states must first be repointed to all have the same notary. This is achieved using a special notary-change transaction that takes:

- A single input state
- An output state identical to the input state, except that the appointed notary has been changed

The input state's appointed notary will sign the transaction if it doesn't constitute a double-spend, at which point a state will enter existence that has all the properties of the old state, but has a different appointed notary.

Time-windows

Summary

- *If a transaction includes a time-window, it can only be committed during that window*
- *The notary is the timestamping authority, refusing to commit transactions outside of that window*
- *Time-windows can have a start and end time, or be open at either end*

Time in a distributed system

A notary also act as the *timestamping authority*, verifying that a transaction occurred during a specific time-window before notarising it.

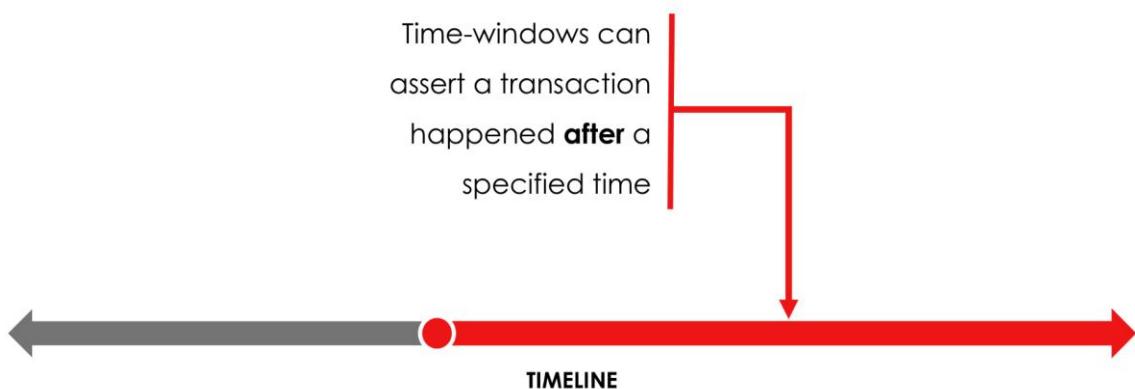
For a time-window to be meaningful, its implications must be binding on the party requesting it. A party can obtain a time-window signature in order to prove that some event happened *before*, *on*, or *after* a particular point in time. However, if the party is not also compelled to commit to the associated transaction, it has a choice of whether or not to reveal this fact until some point in the future. As a result, we need to ensure that the notary either has to also sign the transaction within some time tolerance, or perform timestamping *and* notarisation at the same time. The latter is the chosen behaviour for this model.

There will never be exact clock synchronisation between the party creating the transaction and the notary. This is not only due to issues of physics and network latency, but also because between inserting the command and getting the notary to sign there may be many other steps (e.g. sending the transaction to other parties involved in the trade, requesting human sign-off...). Thus the time at

which the transaction is sent for notarisation may be quite different to the time at which the transaction was created.

Time-windows

For this reason, times in transactions are specified as time *windows*, not absolute times. In a distributed system there can never be “true time”, only an approximation of it. Time windows can be open-ended (i.e. specify only one of “before” and “after”) or they can be fully bounded.



In this way, we express the idea that the *true value* of the fact “the current time” is actually unknowable. Even when both a before and an after time are included, the transaction could have occurred at any point within that time-window.

By creating a range that can be either closed or open at one end, we allow all of the following situations to be modelled:

- A transaction occurring at some point after the given time (e.g. after a maturity event)
- A transaction occurring at any time before the given time (e.g. before a bankruptcy event)
- A transaction occurring at some point roughly around the given time (e.g. on a specific day)

If a time window needs to be converted to an absolute time (e.g. for display purposes), there is a utility method to calculate the mid point.

Note

It is assumed that the time feed for a notary is GPS/NaviStar time as defined by the atomic clocks at the US Naval Observatory. This time feed is extremely accurate and available globally for free.

[Next](#) [Previous](#)

Oracles

Summary

- *A fact can be included in a transaction as part of a command*
- *An oracle is a service that will only sign the transaction if the included fact is true*

Overview

In many cases, a transaction's contractual validity depends on some external piece of data, such as the current exchange rate. However, if we were to let each participant evaluate the transaction's validity based on their own view of the current exchange rate, the contract's execution would be non-deterministic: some signers would consider the transaction valid, while others would consider it invalid. As a result, disagreements would arise over the true state of the ledger.

Corda addresses this issue using *oracles*. Oracles are network services that, upon request, provide commands that encapsulate a specific fact (e.g. the exchange rate at time x) and list the oracle as a required signer.

If a node wishes to use a given fact in a transaction, they request a command asserting this fact from the oracle. If the oracle considers the fact to be true, they send back the required command. The node then includes the command in their transaction, and the oracle will sign the transaction to assert that the fact is true.

If they wish to monetize their services, oracles can choose to only sign a transaction and attest to the validity of the fact it contains for a fee.

Transaction tear-offs

To sign a transaction, the only information the oracle needs to see is their embedded command. Providing any additional transaction data to the oracle would constitute a privacy leak. Similarly, a non-validating notary only needs to see a transaction's input states.

To combat this, the transaction proposer(s) uses a Merkle tree to "tear off" any parts of the transaction that the oracle/notary doesn't need to see before presenting it to them for signing. A Merkle tree is a well-known cryptographic scheme that is commonly used to provide proofs of inclusion and data integrity. Merkle trees are widely used in peer-to-peer networks, blockchain systems and git.

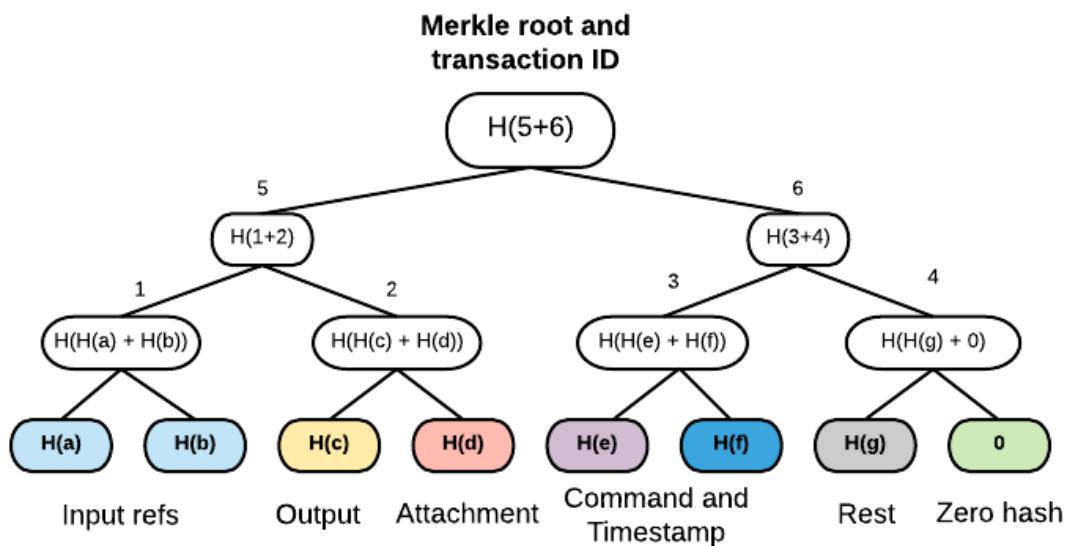
The advantage of a Merkle tree is that the parts of the transaction that were torn off when presenting the transaction to the oracle cannot later be changed without also invalidating the oracle's digital signature.

Transaction Merkle trees

A Merkle tree is constructed from a transaction by splitting the transaction into leaves, where each leaf contains either an input, an output, a command, or an

attachment. The Merkle tree also contains the other fields of the [WireTransaction](#), such as the timestamp, the notary, the type and the signers.

Next, the Merkle tree is built in the normal way by hashing the concatenation of nodes' hashes below the current one together. It's visible on the example image below, where H denotes sha256 function, "+" - concatenation.

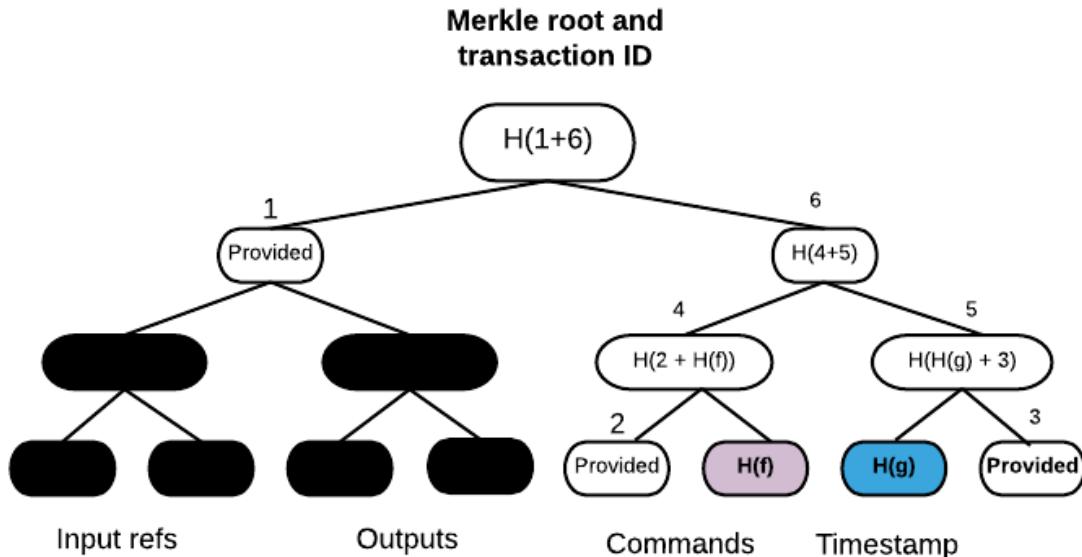


The transaction has two input states, one output state, one attachment, one command and a timestamp. For brevity we didn't include all leaves on the diagram (type, notary and signers are presented as one leaf labelled Rest - in reality they are separate leaves). Notice that if a tree is not a full binary tree, leaves are padded to the nearest power of 2 with zero hash (since finding a pre-image of $\text{sha256}(x) == 0$ is hard computational task) - marked light green above. Finally, the hash of the root is the identifier of the transaction, it's also used for signing and verification of data integrity. Every change in transaction on a leaf level will change its identifier.

Hiding data

Hiding data and providing the proof that it formed a part of a transaction is done by constructing Partial Merkle Trees (or Merkle branches). A Merkle branch is a set of hashes, that given the leaves' data, is used to calculate the root's hash.

Then that hash is compared with the hash of a whole transaction and if they match it means that data we obtained belongs to that particular transaction.



In the example above, the node $H(f)$ is the one holding command data for signing by Oracle service. Blue leaf $H(g)$ is also included since it's holding timestamp information. Nodes labelled Provided form the Partial Merkle Tree, black ones are omitted. Having timestamp with the command that should be in a violet node place and branch we are able to calculate root of this tree and compare it with original transaction identifier - we have a proof that this command and timestamp belong to this transaction.

[Next](#) [Previous](#)

Nodes

Summary

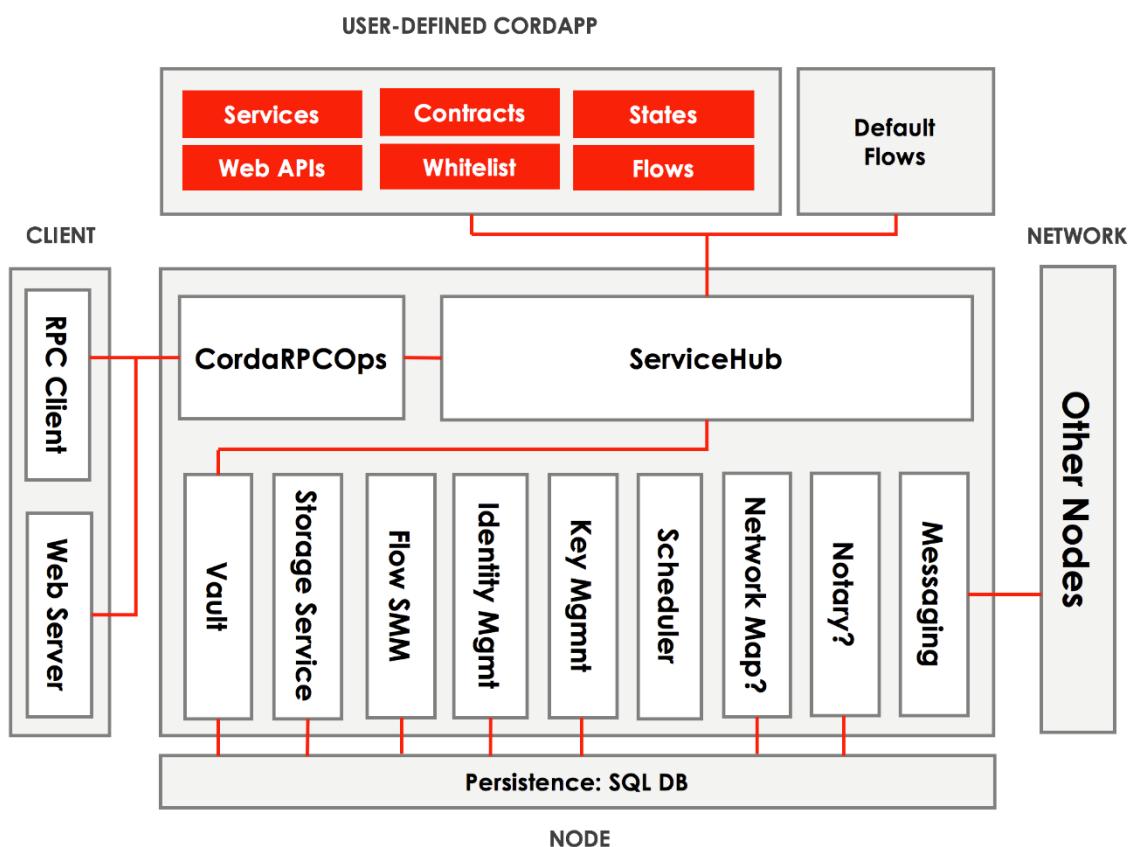
- A node is JVM run-time with a unique network identity running the Corda software
- The node has two interfaces with the outside world:
 - A network layer, for interacting with other nodes
 - RPC, for interacting with the node's owner

- The node's functionality is extended by installing CorDapps in the plugin registry

Node architecture

A Corda node is a JVM run-time environment with a unique identity on the network that hosts Corda services and CorDapps.

We can visualize the node's internal architecture as follows:



The core elements of the architecture are:

- A persistence layer for storing data
- A network interface for interacting with other nodes
- An RPC interface for interacting with the node's owner
- A service hub for allowing the node's flows to call upon the node's other services
- A cordapp interface and provider for extending the node by installing CorDapps

Persistence layer

The persistence layer has two parts:

- The **vault**, where the node stores any relevant current and historic states
- The **storage service**, where it stores transactions, attachments and flow checkpoints

The node's owner can query the node's storage using the RPC interface (see below).

Network interface

All communication with other nodes on the network is handled by the node itself, as part of running a flow. The node's owner does not interact with other network nodes directly.

RPC interface

The node's owner interacts with the node via remote procedure calls (RPC). The key RPC operations the node exposes are documented in [API: RPC operations](#).

The service hub

Internally, the node has access to a rich set of services that are used during flow execution to coordinate ledger updates. The key services provided are:

- Information on other nodes on the network and the services they offer
- Access to the contents of the vault and the storage service
- Access to, and generation of, the node's public-private keypairs
- Information about the node itself

- The current time, as tracked by the node

The CorDapp provider

The CorDapp provider is where new CorDapps are installed to extend the behavior of the node.

The node also has several CorDapps installed by default to handle common tasks such as:

- Retrieving transactions and attachments from counterparties
- Upgrading contracts
- Broadcasting agreed ledger updates for recording by counterparties

Draining mode

In order to operate a clean shutdown of a node, it is important than no flows are in-flight, meaning no checkpoints should be persisted. The node is able to be put in a Flows Draining Mode, during which:

- Commands requiring to start new flows through RPC will be rejected.
- Scheduled flows due will be ignored.
- Initial P2P session messages will not be processed, meaning peers will not be able to initiate new flows involving the node.
- All other activities will proceed as usual, ensuring that the number of in-flight flows will strictly diminish.

As their number - which can be monitored through RPC - reaches zero, it is safe to shut the node down. This property is durable, meaning that restarting the node will not reset it to its default value and that a RPC command is required.

[Next](#) [Previous](#)

Tradeoffs

Summary

- *Permissioned networks are better suited for financial use-cases*
- *Point-to-point communication allows information to be shared need-to-know*
- *A UTXO model allows for more transactions-per-second*

Permissioned vs. permissionless

Traditional blockchain networks are *permissionless*. The parties on the network are anonymous, and can join and leave at will.

By contrast, Corda networks are *permissioned*. Each party on the network has a known identity that they use when communicating with counterparties, and network access is controlled by a doorman. This has several benefits:

- Anonymous parties are inappropriate for most scenarios involving regulated financial institutions
- Knowing the identity of your counterparties allows for off-ledger resolution of conflicts using existing legal systems
- Sybil attacks are averted without the use of expensive mechanisms such as proof-of-work

Point-to-point vs. global broadcasts

Traditional blockchain networks broadcast every message to every participant. The reason for this is two-fold:

- Counterparty identities are not known, so a message must be sent to every participant to ensure it reaches its intended recipient
- Making every participant aware of every transaction allows the network to prevent double-spends

The downside is that all participants see everyone else's data. This is unacceptable for many use-cases.

In Corda, each message is instead addressed to a specific counterparty, and is not seen by any uninvolved third parties. The developer has full control over what messages are sent, to whom, and in what order. As a result, **data is shared on a need-to-know basis only**. To prevent double-spends in this system, we employ notaries as an alternative to proof-of-work.

Corda also uses several other techniques to maximize privacy on the network:

- **Transaction tear-offs:** Transactions are structured in a way that allows them to be digitally signed without disclosing the transaction's contents. This is

achieved using a data structure called a Merkle tree. You can read more about this technique in [merkle-trees](#).

- **Key randomisation:** The parties to a transaction are identified only by their public keys, and fresh keypairs are generated for each transaction. As a result, an onlooker cannot identify which parties were involved in a given transaction.

UTXO vs. account model

Corda uses a *UTXO* (unspent transaction output) model. Each transaction consumes a set of existing states to produce a set of new states.

The alternative would be an *account* model. In an account model, stateful objects are stored on-ledger, and transactions take the form of requests to update the current state of these objects.

The main advantage of the UTXO model is that transactions with different inputs can be applied in parallel, vastly increasing the network's potential transactions-per-second. In the account model, the number of transactions-per-second is

limited by the fact that updates to a given object must be applied sequentially.

Code-is-law vs. existing legal systems

Financial institutions need the ability to resolve conflicts using the traditional legal system where required. Corda is designed to make this possible by:

- Having permissioned networks, meaning that participants are aware of who they are dealing with in every single transaction
- All code contracts are backed by a legal document describing the contract's intended behavior which can be relied upon to resolve conflicts

Build vs. re-use

Wherever possible, Corda re-uses existing technologies to make the platform more robust overall. For example, Corda re-uses:

- Standard JVM programming languages for the development of CorDapps
- Existing SQL databases
- Existing message queue implementations

-

[Next](#)

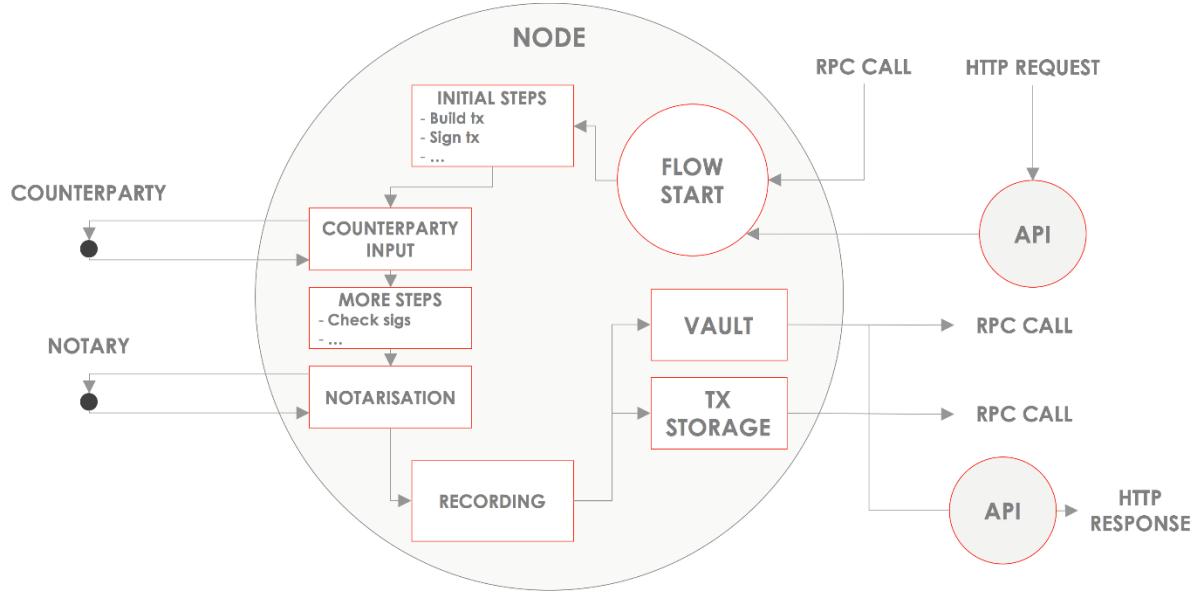
CorDapps

- [What is a CorDapp?](#)
- [Writing a CorDapp](#)
- [Debugging a CorDapp](#)
- [Upgrading a CorDapp to a new platform version](#)
- [Upgrading a CorDapp \(outside of platform version upgrades\)](#)
- [Building a CorDapp](#)
- [Building against Master](#)
- [Corda API](#)
- [Object serialization](#)
- [Secure coding guidelines](#)
- [Flow cookbook](#)
- [Cheat sheet](#)
- [CorDapp samples](#)

[Next](#) [Previous](#)

What is a CorDapp?

CorDapps (Corda Distributed Applications) are distributed applications that run on the Corda platform. The goal of a CorDapp is to allow nodes to reach agreement on updates to the ledger. They achieve this goal by defining flows that Corda node owners can invoke through RPC calls:



CorDapps are made up of the following key components:

- States, defining the facts over which agreement is reached (see [Key Concepts - States](#))
- Contracts, defining what constitutes a valid ledger update (see [Key Concepts - Contracts](#))
- Services, providing long-lived utilities within the node
- Serialisation whitelists, restricting what types your node will receive off the wire

Each CorDapp is installed at the level of the individual node, rather than on the network itself. For example, a node owner may choose to install the Bond Trading CorDapp, with the following components:

- A `BondState`, used to represent bonds as shared facts on the ledger
- A `BondContract`, used to govern which ledger updates involving `BondState` states are valid
- Three flows:
 - An `IssueBondFlow`, allowing new `BondState` states to be issued onto the ledger
 - A `TradeBondFlow`, allowing existing `BondState` states to be bought and sold on the ledger
 - An `ExitBondFlow`, allowing existing `BondState` states to be exited from the ledger

After installing this CorDapp, the node owner will be able to use the flows defined by the CorDapp to agree ledger updates related to issuance, sale, purchase and exit of bonds.

[Next](#) [Previous](#)

- Writing a CorDapp
 - [View page source](#)
-

Writing a CorDapp

Contents

- [Writing a CorDapp](#)
 - [Overview](#)
 - [Web content and RPC clients](#)
 - [Structure](#)
 - [Module one - cordapp-contracts-states](#)
 - [Module two - cordapp](#)
 - [Resources](#)

Overview

CorDapps can be written in either Java, Kotlin, or a combination of the two. Each CorDapp component takes the form of a JVM class that subclasses or implements a Corda library type:

- Flows subclass `FlowLogic`
- States implement `ContractState`
- Contracts implement `Contract`
- Services subclass `SingletonSerializationToken`
- Serialisation whitelists implement `SerializationWhitelist`

Web content and RPC clients

For testing purposes, CorDapps may also include:

- **APIs and static web content:** These are served by Corda's built-in webserver. This webserver is not production-ready, and should be used for testing purposes only
- **RPC clients:** These are programs that automate the process of interacting with a node via RPC

In production, a production-ready webserver should be used, and these files should be moved into a different module or project so that they do not bloat the CorDapp at build time.

Structure

You should base the structure of your project on the Java or Kotlin templates:

- [Java Template CorDapp](#)
- [Kotlin Template CorDapp](#)

The project should be split into two modules:

- A `cordapp-contracts-states` module containing classes such as contracts and states that will be sent across the wire as part of a flow
- A `cordapp` module containing the remaining classes

Each module will be compiled into its own CorDapp. This minimises the size of the JAR that has to be sent across the wire when nodes are agreeing ledger updates.

Module one - cordapp-contracts-states

Here is the structure of the `src` directory for the `cordapp-contracts-states` module:

```

└── main
    └── java
        └── com
            └── template
                ├── TemplateContract.java
                └── TemplateState.java

```

The directory only contains two class definitions:

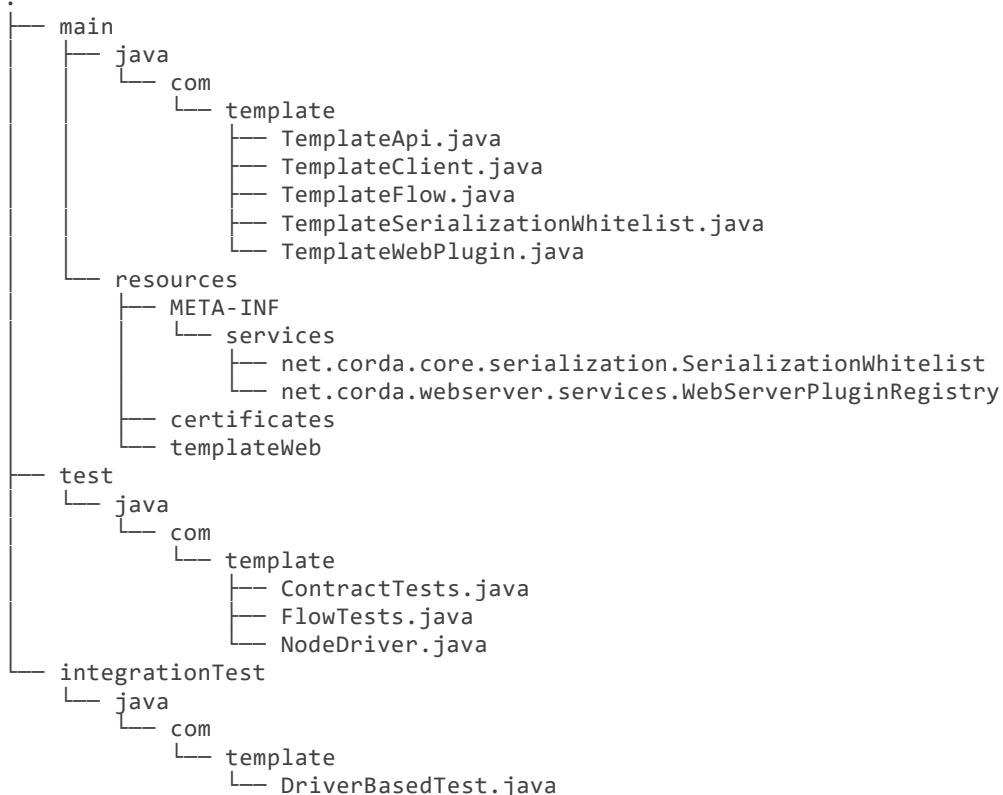
- `TemplateContract`

- `TemplateState`

These are definitions for classes that we expect to have to send over the wire. They will be compiled into their own CorDapp.

Module two - cordapp

Here is the structure of the `src` directory for the `cordapp` module:



The `src` directory is structured as follows:

- `main` contains the source of the CorDapp
- `test` contains example unit tests, as well as a node driver for running the CorDapp from IntelliJ
- `integrationTest` contains an example integration test

Within `main`, we have the following directories:

- `resources/META-INF/services` contains registries of the CorDapp's serialisation whitelists and web plugins
- `resources/certificates` contains dummy certificates for test purposes

- `resources/templateWeb` contains a dummy front-end
- `java` (or `kotlin` in the Kotlin template), which includes the source-code for our CorDapp

The source-code for our CorDapp breaks down as follows:

- `TemplateFlow.java`, which contains a dummy `FlowLogic` subclass
- `TemplateState.java`, which contains a dummy `ContractState` implementation
- `TemplateContract.java`, which contains a dummy `Contract` implementation
- `TemplateSerializationWhitelist.java`, which contains a dummy `SerializationWhitelist` implementation

In developing your CorDapp, you should start by modifying these classes to define the components of your CorDapp. A single CorDapp can define multiple flows, states, and contracts.

The template also includes a web API and RPC client:

- `TemplateApi.java`
- `TemplateClient.java`
- `TemplateWebPlugin.java`

These are for testing purposes and would be removed in a production CorDapp.

Resources

In writing a CorDapp, you should consult the following resources:

- [Getting Set Up](#) to set up your development environment
- [The Hello, World! tutorial](#) to write your first CorDapp
- [Building a CorDapp](#) to build and run your CorDapp
- The API docs to read about the API available in developing CorDapps
 - There is also a [cheatsheet](#) recapping the key types
- The [Flow cookbook](#) to see code examples of how to perform common flow tasks
- [Sample CorDapps](#) showing various parts of Corda's functionality

[Next](#) [Previous](#)

- Debugging a CorDapp

- [View page source](#)
-

Debugging a CorDapp

Contents

- [Debugging a CorDapp](#)
 - [Using a MockNetwork](#)
 - [Using the node driver](#)
 - [With the nodes in-process](#)
 - [With remote debugging](#)
 - [By enabling remote debugging on a node](#)

There are several ways to debug your CorDapp.

Using a MockNetwork

You can attach the [IntelliJ IDEA debugger](#) to a [MockNetwork](#) to debug your CorDapp:

- Define your flow tests as per [API: Testing](#)
 - In your [MockNetwork](#), ensure that [threadPerNode](#) is set to [false](#)
- Set your breakpoints
- Run the flow tests using the debugger. When the tests hit a breakpoint, execution will pause

Using the node driver

You can also attach the [IntelliJ IDEA debugger](#) to nodes running via the node driver to debug your CorDapp.

With the nodes in-process

1. Define a network using the node driver as per [Integration testing](#)
 - In your [DriverParameters](#), ensure that [startNodesInProcess](#) is set to [true](#)
2. Run the driver using the debugger
3. Set your breakpoints
4. Interact with your nodes. When execution hits a breakpoint, execution will pause

- The nodes' webservers always run in a separate process, and cannot be attached to by the debugger

With remote debugging

1. Define a network using the node driver as per Integration testing
 - In your `DriverParameters`, ensure that `startNodesInProcess` is set to `false` and `isDebug` is set to `true`
2. Run the driver. The remote debug ports for each node will be automatically generated and printed to the terminal. For example:


```
[INFO ] 11:39:55,471 [driver-pool-thread-0] (DriverDSLImpl.kt:814)
internal.DriverDSLImpl.startOutOfProcessNode -
Starting out-of-process Node PartyA, debug port is 5008, jolokia monitoring port is not
enabled {}
```
3. Attach the debugger to the node of interest on its debug port:
 - In IntelliJ IDEA, create a new run/debug configuration of type `Remote`
 - Set the run/debug configuration's `Port` to the debug port
 - Start the run/debug configuration in debug mode
4. Set your breakpoints
5. Interact with your node. When execution hits a breakpoint, execution will pause
 - The nodes' webservers always run in a separate process, and cannot be attached to by the debugger

By enabling remote debugging on a node

- Upgrading a CorDapp to a new platform version
 - [View page source](#)
-

Upgrading a CorDapp to a new platform version

These notes provide instructions for upgrading your CorDapps from previous versions, starting with the upgrade from our first public Beta (Milestone 12), to V1.0.

Contents

- [Upgrading a CorDapp to a new platform version](#)
 - [General rules](#)
 - [V3.2 to v3.3](#)
 - [v3.1 to v3.2](#)
 - [Gradle Plugin Version](#)
 - [Database schema changes](#)
 - [v3.0 to v3.1](#)
 - [Gradle Plugin Version](#)
 - [V2.0 to V3.0](#)
 - [Gradle Plugin Version](#)
 - [Network Map Service](#)
 - [Corda Plugins](#)
 - [AMQP](#)
 - [Database schema changes](#)
 - [Configuration](#)
 - [Testing](#)
 - [V1.0 to V2.0](#)
 - [Public Beta \(M12\) to V1.0](#)
 - [Build](#)
 - [Configuration](#)
 - [Missing imports](#)
 - [Core data structures](#)
 - [Flow framework](#)
 - [Node services \(ServiceHub\)](#)
 - [RPC Client](#)
 - [Testing](#)
 - [Finance](#)
 - [Miscellaneous](#)
 - [Gotchas](#)
 - [Core data structures](#)
 - [Node services \(ServiceHub\)](#)
 - [Configuration](#)
 - [Identity](#)
 - [Testing](#)
 - [Core data structures](#)
 - [Build](#)
 - [Node services \(ServiceHub\)](#)

- [Finance](#)

General rules

Always remember to update the version identifiers in your project's gradle file:

```
ext.corda_release_version = '1.0.0'  
ext.corda_gradle_plugins_version = '1.0.0'
```

It may be necessary to update the version of major dependencies:

```
ext.kotlin_version = '1.1.4'  
ext.quasar_version = '0.7.9'
```

Please consult the relevant release notes of the release in question. If not specified, you may assume the versions you are currently using are still in force.

We also strongly recommend cross referencing with the [Changelog](#) to confirm changes.

V3.2 to v3.3

- Update the Corda Release version

The `corda_release_version` identifier in your projects gradle file will need changing as follows:

```
ext.corda_release_version = '3.3-corda'
```

v3.1 to v3.2

Gradle Plugin Version

You will need to update the `corda_release_version` identifier in your project gradle file.

```
ext.corda_release_version = '3.2-corda'
```

Database schema changes

- Database upgrade - a typo has been corrected in the `NODE_ATTACHMENTS_CONTRACTS` table name.

When upgrading from versions 3.0 or 3.1, run the following command:

```
ALTER TABLE [schema].NODE_ATTACHMENTS_CONTRACTS RENAME TO NODE_ATTACHMENTS_CONTRACTS;
```

Schema name is optional, run SQL when the node is not running.

- Postgres database upgrade - Change the type of the `checkpoint_value` column to `bytea`.

This will address the issue that the `vacuum` function is unable to clean up deleted checkpoints as they are still referenced from the `pg_shdepend` table.

```
ALTER TABLE node_checkpoints ALTER COLUMN checkpoint_value set data type bytea using null;
```

Note

This change will also need to be run when migrating from version 3.0.

Important

The Corda node will fail on startup if the database was not updated with the above commands.

v3.0 to v3.1

Gradle Plugin Version

Corda 3.1 uses version 3.1.0 of the gradle plugins and your `build.gradle` file should be updated to reflect this.

```
ext.corda_gradle_plugins_version = '3.1.0'
```

You will also need to update the `corda_release_version` identifier in your project gradle file.

```
ext.corda_release_version = '3.1-corda'
```

V2.0 to V3.0

Gradle Plugin Version

Corda 3.0 uses version 3.0.9 of the gradle plugins and your `build.gradle` file should be updated to reflect this.

```
ext.corda_gradle_plugins_version = '3.0.9'
```

You will also need to update the `corda_release_version` identifier in your project gradle file.

```
ext.corda_release_version = 'corda-3.0'
```

Network Map Service

With the re-designed network map service the following changes need to be made:

- The network map is no longer provided by a node and thus the `networkMapService` config is ignored. Instead the network map is either provided by the compatibility zone (CZ) operator (who operates the doorman) and available using the `compatibilityZoneURL` config, or is provided using signed node info files which are copied locally. See [Network Map](#) for more details, and [setting-up-a-corda-network](#) on how to use the network bootstrapper for deploying a local network.
- Configuration for a notary has been simplified. `extraAdvertisedServiceIds`, `notaryNodeAddress`, `notaryClusterAddresses` and `bftSMaRT` configs have been replaced by a single `notary` config object. See [Node configuration](#) for more details.
- The advertisement of the notary to the rest of the network, and its validation type, is no longer determined by the `extraAdvertisedServiceIds` config. Instead it has been moved to the control of the network operator via the introduction of network parameters. The network bootstrapper automatically includes the configured notaries when generating the network parameters file for a local deployment.
- Any nodes defined in a `deployNodes` gradle task performing the function of the network map can be removed, or the `NetworkMap` parameter can be removed for any “controller” node which is both the network map and a notary.
- For registering a node with the doorman the `certificateSigningService` config has been replaced by `compatibilityZoneURL`.

Corda Plugins

- Corda plugins have been modularised further so the following additional gradle entries are necessary: For example:

The plugin needs to be applied in all gradle build files where there is a dependency on Corda using any of: `cordaCompile`, `cordaRuntime`, `cordapp`

- For existing contract ORM schemas that extend from `CommonSchemaV1.LinearState` or `CommonSchemaV1.FungibleState`, you will need to explicitly map the `participants` collection to a database table. Previously this mapping was done in the superclass, but that makes it impossible to properly configure the table name. The required changes are to:
 - Add the `override var participants: MutableSet<AbstractParty>? = null` field to your class, and
 - Add JPA mappings

For example:

```
@Entity
@Table(name = "cash_states_v2",
       indexes = arrayOf(Index(name = "ccy_code_idx2", columnList =
"ccy_code")))
class PersistentCashState {

    @ElementCollection
    @Column(name = "participants")
    @CollectionTable(name="cash_states_v2_participants", joinColumns =
arrayOf(
            JoinColumn(name = "output_index", referencedColumnName =
"output_index"),
            JoinColumn(name = "transaction_id", referencedColumnName =
"transaction_id")))
    override var participants: MutableSet<AbstractParty>? = null,
```

AMQP

Whilst the enablement of AMQP is a transparent change, as noted in the [Object serialization](#) documentation the way classes, and states in particular, should be written to work with this new library may require some alteration to your current implementation.

- With AMQP enabled Java classes must be compiled with the `-parameter` flag.
 - If they aren't, then the error message will complain about `arg<N>` being an unknown parameter.
 - If recompilation is not viable, a custom serializer can be written as per [Pluggable Serializers for CorDapps](#)
 - It is important to bear in mind that with AMQP there must be an implicit mapping between constructor parameters and properties you wish included in the serialized form of a class.
 - See [Object serialization](#) for more information
- Error messages of the form

```
Constructor parameter - "<some parameter of a constructor>" -  
doesn't refer to a property of "class <some.class.being.serialized>"
```

indicate that a class, in the above example `some.class.being.serialized`, has a parameter on its primary constructor that doesn't correlate to a property of the class. This is a problem because the Corda AMQP serialization library uses a class's constructor (default, primary, or annotated) as the means by which instances of the serialized form are reconstituted.

See the section “Mismatched Class Properties / Constructor Parameters” in the [Object serialization documentation](#)

Database schema changes

An H2 database instance (represented on the filesystem as a file called *persistence.mv.db*) used in Corda 1.0 or 2.0 cannot be directly reused with Corda 3.0 due to minor improvements and additions to stabilise the underlying schemas.

Configuration

Nodes that do not require SSL to be enabled for RPC clients now need an additional port to be specified as part of their configuration. To do this, add a block as follows to the nodes configuration:

```
rpcSettings {  
    adminAddress "localhost:10007"  
}
```

to *node.conf* files.

Also, the property *rpcPort* is now deprecated, so it would be preferable to substitute properties specified that way e.g., *rpcPort=10006* with a block as follows:

```
rpcSettings {  
    address "localhost:10006"  
    adminAddress "localhost:10007"  
}
```

Equivalent changes should be performed on classes extending *CordformDefinition*.

Testing

- The registration mechanism for CorDApps in `MockNetwork` unit tests has changed:
 - CorDApp registration is now done via the `cordappPackages` constructor parameter of `MockNetwork`. This parameter is a list of `String` values which should be the package names of the CorDApps containing the contract verification code you wish to load
 - The `unsetCordappPackages` method is now redundant and has been removed
- Many classes have been moved between packages, so you will need to update your imports

Tip

We have provided several scripts (depending upon your operating system of choice) to smooth the upgrade process for existing projects. This can be found at `tools\scripts\update-test-packages.sh` for the Bash shell and `tools/scripts/upgrade-test-packages.ps1` for Windows Power Shell users in the source tree

- `setCordappPackages` and `unsetCordappPackages` have been removed from the ledger/transaction DSL and the flow test framework, and are now set via a constructor parameter or automatically when constructing the `MockServices` or `MockNetwork` object
- Key constants e.g. `ALICE_KEY` have been removed; you can now use `TestIdentity` to make your own
- The ledger/transaction DSL must now be provided with `MockServices` as it no longer makes its own * In transaction blocks, input and output take their arguments as `ContractStates` rather than lambdas * Also in transaction blocks, command takes its arguments as `CommandDatas` rather than lambdas
- The `MockServices` API has changed; please refer to its API documentation
- `TestDependencyInjectionBase` has been retired in favour of a JUnit Rule called `SerializationEnvironmentRule` * This replaces the `initialiseSerialization` parameter of `ledger/transaction` and `verifierDriver` * The `withTestSerialization` method is obsoleted by `SerializationEnvironmentRule` and has been retired

- **MockNetwork now takes a MockNetworkParameters builder to make it more Java-friendly, like driver's DriverParameters**
 - Similarly, the MockNetwork.createNode methods now take a MockNodeParameters builder
- MockNode constructor parameters are now aggregated in MockNodeArgs for easier subclassing
- MockNetwork.Factory has been retired as you can simply use a lambda
- testNodeConfiguration has been retired, please use a mock object framework of your choice instead
- MockNetwork.createSomeNodes and IntegrationTestCategory have been retired with no replacement
- Starting a flow can now be done directly from a node object. Change calls of the form `node.getServices().startFlow(...)` to `node.startFlow(...)`
- Similarly a transaction can be executed directly from a node object. Change calls of the form `node.getDatabase().transaction({ it - ... })` to `node.transaction({() -> ... })`
- `startFlow` now returns a `CordaFuture`, there is no need to call `startFlow(...).getResultantFuture()`

V1.0 to V2.0

- You need to update the `corda_release_version` identifier in your project gradle file. The `corda_gradle_plugins_version` should remain at 1.0.0:
 - `ext.corda_release_version = '2.0.0'`
 - `ext.corda_gradle_plugins_version = '1.0.0'`

Public Beta (M12) to V1.0

Build

- MockNetwork has moved. To continue using `MockNetwork` for testing, you must add the following dependency to your `build.gradle` file:
 - `testCompile "net.corda:corda-node-driver:$corda_release_version"`

Note

You may only need `testCompile "net.corda:corda-test-utils:$corda_release_version"` if not using the Driver DSL

Configuration

- `CordaPluginRegistry` has been removed:

- The one remaining configuration item `customizeSerialisation`, which defined a optional whitelist of types for use in object serialization, has been replaced with the `SerializationWhitelist` interface which should be implemented to define a list of equivalent whitelisted classes
- You will need to rename your services resource file. ‘resources/META-INF/services/net.corda.core.node.CordaPluginRegistry’ becomes ‘resources/META-INF/services/net.corda.core.serialization.SerializationWhitelist’
- `MockNode.testPluginRegistries` was renamed to `MockNode.testSerializationWhitelists`
- In general, the `@CordaSerializable` annotation is the preferred method for whitelisting, as described in Object serialization

Missing imports

Use IntelliJ's automatic imports feature to intelligently resolve the new imports:

- Missing imports for contract types:
 - CommercialPaper and Cash are now contained within the `finance` module, as are associated helpers functions. For example:
 - `import net.corda.contracts.ICommercialPaperState` becomes `import net.corda.finance.contracts.ICommercialPaperState`
 - `import net.corda.contracts.asset.sumCashBy` becomes `import net.corda.finance.utils.sumCashBy`
 - `import net.corda.core.contracts.DOLLARS` becomes `import net.corda.finance.DOLLARS`
 - `import net.corda.core.contracts.issued by` becomes `import net.corda.finance.contracts.issued by`
 - `import net.corda.contracts.asset.Cash` becomes `import net.corda.finance.contracts.asset.Cash`
- Missing imports for utility functions:
 - Many common types and helper methods have been consolidated into `net.corda.core.utilities` package. For example:
 - `import net.corda.core.crypto.commonName` becomes `import net.corda.core.utilities.commonName`
 - `import net.corda.core.crypto.toBase58String` becomes `import net.corda.core.utilities.toBase58String`
 - `import net.corda.core.getOrThrow` becomes `import net.corda.core.utilities.getOrThrow`

- Missing flow imports:
 - In general, all reusable library flows are contained within the **core** API `net.corda.core.flows` package
 - Financial domain library flows are contained within the **finance** module `net.corda.finance.flows` package
 - Other flows that have moved include `import net.corda.core.flows.ResolveTransactionsFlow`, which becomes `import net.corda.core.internal.ResolveTransactionsFlow`

Core data structures

- Missing `Contract` override:
 - `Contract.legalContractReference` has been removed, and replaced by the optional annotation `@LegalProseReference(uri = "<URI>")`
- Unresolved reference:
 - `AuthenticatedObject` was renamed to `CommandWithParties`
- Overrides nothing:
 - `LinearState.isRelevant` was removed. Whether a node stores a `LinearState` in its vault depends on whether the node is one of the state's `participants`
 - `txBuilder.toLedgerTransaction` now requires a `ServiceHub` parameter. This is used by the new Contract Constraints functionality to validate and resolve attachments

Flow framework

- `FlowLogic` communication has been upgraded to use explicit `FlowSession` instances to communicate between nodes:
 - `FlowLogic.send` / `FlowLogic.receive` / `FlowLogic.sendAndReceive` has been replaced by `FlowSession.send` / `FlowSession.receive` / `FlowSession.sendAndReceive`. The replacement functions do not take a destination parameter, as this is defined implicitly by the session used
 - Initiated flows now take in a `FlowSession` instead of `Party` in their constructor. If you need to access the counterparty identity, it is in the `counterparty` property of the flow session
- `FinalityFlow` now returns a single `SignedTransaction`, instead of a `List<SignedTransaction>`
- `TransactionKeyFlow` was renamed to `SwapIdentitiesFlow`

- `SwapIdentitiesFlow` must be imported from the *confidential-identities* package `net.corda.confidential`

Node services (ServiceHub)

- Unresolved reference to `vaultQueryService`:
 - Replace all references to `<services>.vaultQueryService` with `<services>.vaultService`
 - Previously there were two vault APIs. Now there is a single unified API with the same functions: `VaultService`.
- `FlowLogic.ourIdentity` has been introduced as a shortcut for retrieving our identity in a flow
- `serviceHub.myInfo.legalIdentity` no longer exists
- `getAnyNotary` has been removed.
Use `serviceHub.networkMapCache.notaryIdentities[0]` instead
- `ServiceHub.networkMapUpdates` is replaced by `ServiceHub.networkMapFeed`
- `ServiceHub.partyFromX500Name` is replaced by `ServiceHub.wellKnownPartyFromX500Name`
 - A “well known” party is one that isn’t anonymous. This change was motivated by the confidential identities work

RPC Client

- Missing API methods on the `CordaRPCOps` interface:
 - `verifiedTransactionsFeed` has been replaced by `internalVerifiedTransactionsFeed`
 - `verifiedTransactions` has been replaced by `internalVerifiedTransactionsSnapshot`
 - These changes are in preparation for the planned integration of Intel SGX™, which will encrypt the transactions feed. Apps that use this API will not work on encrypted ledgers. They should generally be modified to use the vault query API instead
 - Accessing the `networkMapCache` via `services.nodeInfo().legalIdentities` returns a list of identities
 - This change is in preparation for allowing a node to host multiple separate identities in the future

Testing

Please note that `Clauses` have been removed completely as of V1.0. We will be revisiting this capability in a future release.

- CorDapps must be explicitly registered in `MockNetwork` unit tests:
 - This is done by calling `setCordappPackages`, an extension helper function in the `net.corda.testing` package, on the first line of your `@Before` method. This takes a variable number of `String` arguments which should be the package names of the CorDapps containing the contract verification code you wish to load
 - You should unset CorDapp packages in your `@After` method by using `unsetCordappPackages` after `stopNodes`
- CorDapps must be explicitly registered in `DriverDSL` and `RPCDriverDSL` integration tests:
 - You must register package names of the CorDapps containing the contract verification code you wish to load using the `extraCordappPackagesToScan: List<String>` constructor parameter of the driver DSL

Finance

- `FungibleAsset` interface simplification:
 - The `Commands` grouping interface that included the `Move`, `Issue` and `Exit` interfaces has been removed
 - The `move` function has been renamed to `withNewOwnerAndAmount` * This is for consistency with `OwnableState.withNewOwner`

Miscellaneous

- `args[0].parseNetworkHostAndPort()` becomes `NetworkHostAndPort.parse(args[0])`
- There is no longer a `NodeInfo.advertisedServices` property
 - The concept of advertised services has been removed from Corda. This is because it was vaguely defined and real-world apps would not typically select random, unknown counterparties from the network map based on self-declared capabilities
 - We will introduce a replacement for this functionality, business networks, in a future release
 - For now, services should be retrieved by legal name using `NetworkMapCache.getNodeByLegalName`

Gotchas

- Be sure to use the correct identity when issuing cash:
 - The third parameter to `CashIssueFlow` should be the *notary* (and not the *node identity*)
 -

From Milestone 13

Core data structures

- `TransactionBuilder` changes:
 - Use convenience class `StateAndContract` instead of `TransactionBuilder.withItems` for passing around a state and its contract.
- Transaction builder DSL changes:
 - When adding inputs and outputs to a transaction builder, you must also specify `ContractClassName`
 - `ContractClassName` is the name of the `Contract` subclass used to verify the transaction
- Contract verify method signature change:
 - `override fun verify(tx: TransactionForContract)` becomes `override fun verify(tx: LedgerTransaction)`
- You no longer need to override `ContractState.contract` function

Node services (ServiceHub)

- ServiceHub API method changes:
 - `services.networkMapUpdates().justSnapshot` becomes `services.networkMapSnapshot()`

Configuration

- No longer need to define `CordaPluginRegistry` and configure `requiredSchemas`:
 - Custom contract schemas are automatically detected at startup time by class path scanning
 - For testing purposes, use the `SchemaService` method to register new custom schemas
(e.g. `services.schemaService.registerCustomSchemas(setOf(YoSchemaV1))`)

Identity

- Party names are now `CordaX500Name`, not `X500Name`:

- `CordaX500Name` specifies a predefined set of mandatory (organisation, locality, country) and optional fields (common name, organisation unit, state) with validation checking
- Use new builder `CordaX500Name.build(X500Name(target))` or explicitly define the X500Name parameters using the `CordaX500Name` constructors

Testing

- MockNetwork testing:
 - Mock nodes in node tests are now of type `StartedNode<MockNode>`, rather than `MockNode`
 - `MockNetwork` now returns a `BasketOf(<StartedNode<MockNode>>)`
 - You must call internals on `StartedNode` to get `MockNode` (e.g. `a = nodes.partyNodes[0].internals`)
- Host and port changes:
 - Use string helper function `parseNetworkHostAndPort` to parse a URL on startup (e.g. `val hostAndPort = args[0].parseNetworkHostAndPort()`)
- Node driver parameter changes:
 - The node driver parameters for starting a node have been reordered
 - The node's name needs to be given as an `CordaX500Name`, instead of using `getX509Name`

From Milestone 12 (First Public Beta)

Core data structures

- Transaction building:
 - You no longer need to specify the type of a `TransactionBuilder` as `TransactionType.General`
 - `TransactionType.General.Builder(notary)` becomes `TransactionBuilder(notary)`

Build

- Gradle dependency reference changes:
 - Module names have changed to include `corda` in the artifacts' JAR names:

```
compile "net.corda:core:$corda_release_version" -> compile "net.corda:corda-core:$corda_release_version"
compile "net.corda:finance:$corda_release_version" -> compile "net.corda:corda-finance:$corda_release_version"
compile "net.corda:jackson:$corda_release_version" -> compile "net.corda:corda-jackson:$corda_release_version"
compile "net.corda:node:$corda_release_version" -> compile "net.corda:corda-node:$corda_release_version"
```

```
compile "net.corda:rpc:$corda_release_version" -> compile "net.corda:corda-rpc:$corda_release_version"
```

Node services (ServiceHub)

- `ServiceHub` API changes:
 - `services.networkMapUpdates` becomes `services.networkMapFeed`
 - `services.getCashBalances` becomes a helper method in the `finance` module contracts package (`net.corda.finance.contracts.getCashBalances`)

Finance

- Financial asset contracts (`Cash`, `CommercialPaper`, `Obligations`) are now a standalone CorDapp within the `finance` module:
 - You need to import them from their respective packages within the `finance` module (e.g. `net.corda.finance.contracts.asset.Cash`)
 - You need to import the associated asset flows from their respective packages within `finance` module. For example:
 - `net.corda.finance.flows.CashIssueFlow`
 - `net.corda.finance.flows.CashIssueAndPaymentFlow`
 - `net.corda.finance.flows.CashExitFlow`
- The `finance` gradle project files have been moved into a `net.corda.finance` package namespace:
 - Adjust imports of Cash flow references
 - Adjust the `StartFlow` permission in `gradle.build` files
 - Adjust imports of the associated flows (`Cash*Flow`, `TwoPartyTradeFlow`, `TwoPartyDealFlow`)

Corda nodes

- [Creating nodes locally](#)
- [Running nodes locally](#)
- [Deploying a node](#)
- [Node configuration](#)
- [Client RPC](#)
- [Shell](#)
- [Node database](#)
- [Node administration](#)
- [Out-of-process verification](#)

[Next](#) [Previous](#)

Creating nodes locally

Node structure

Each Corda node has the following structure:

```
.  
└── certificates          // The node's certificates  
└── corda-webserver.jar  // The built-in node webserver  
└── corda.jar             // The core Corda libraries  
└── logs                  // The node logs  
└── node.conf              // The node's configuration files  
└── persistence.mv.db     // The node's database  
└── cordapps               // The CorDApps jars installed on the node
```

The node is configured by editing its `node.conf` file. You install CorDApps on the node by dropping the CorDapp JARs into the `cordapps` folder.

In development mode (i.e. when `devMode = true`, see [Node configuration](#) for more information), the `certificates` directory is filled with pre-configured keystores if the required keystores do not exist. This ensures that developers can get the nodes working as quickly as possible. However, these pre-configured keystores are not secure. To learn more see [Network permissioning](#).

Node naming

A node's name must be a valid X.500 distinguished name. In order to be compatible with other implementations (particularly TLS implementations), we constrain the allowed X.500 name attribute types to a subset of the minimum supported set for X.509 certificates (specified in RFC 3280), plus the locality attribute:

- Organization (O)
- State (ST)
- Locality (L)
- Country (C)
- Organizational-unit (OU)
- Common name (CN)

`State` should be avoided unless required to differentiate from other `localities` with the same or similar names at the country level. For example, London (GB) would not need a `state`, but St Ives would (there are two,

one in Cornwall, one in Cambridgeshire). As legal entities in Corda are likely to be located in major cities, this attribute is not expected to be present in the majority of names, but is an option for the cases which require it.

The name must also obey the following constraints:

- The `organisation`, `locality` and `country` attributes are present
 - The `state`, `organisational-unit` and `common name` attributes are optional
- The fields of the name have the following maximum character lengths:
 - Common name: 64
 - Organisation: 128
 - Organisation unit: 64
 - Locality: 64
 - State: 64
- The `country` attribute is a valid ISO 3166-1 two letter code in upper-case
- All attributes must obey the following constraints:
 - Upper-case first letter
 - Has at least two letters
 - No leading or trailing whitespace
 - Does not include the following characters: `,`, `=`, `$`, `"`, `'`, `\`
 - Is in NFKC normalization form
 - Does not contain the null character
 - Only the latin, common and inherited unicode scripts are supported
- The `organisation` field of the name also obeys the following constraints:
 - No double-spacing
 - This is to avoid right-to-left issues, debugging issues when we can't pronounce names over the phone, and character confusability attacks

External identifiers

Mappings to external identifiers such as company registration numbers, LEI, BIC, etc. should be stored in custom X.509 certificate extensions. These values may change for operational reasons, without the identity they're associated with necessarily changing, and their inclusion in the distinguished name would cause

significant logistical complications. The OID and format for these extensions will be described in a further specification.

The Cordform task

Corda provides a gradle plugin called `Cordform` that allows you to automatically generate and configure a set of nodes for testing and demos. Here is an example `Cordform` task called `deployNodes` that creates three nodes, defined in the Kotlin CorDapp Template:

```
task deployNodes(type: net.corda.plugins.Cordform, dependsOn: ['jar']) {
    directory "./build/nodes"
    node {
        name "O=Notary,L=London,C=GB"
        // The notary will offer a validating notary service.
        notary = [validating : true]
        p2pPort 10002
        rpcSettings {
            port 10003
            adminPort 10023
        }
        // No webport property, so no webserver will be created.
        h2Port 10004
        // Includes the corda-finance CorDapp on our node.
        cordapps = ["net.corda:corda-finance:$corda_release_version"]
    }
    node {
        name "O=PartyA,L=London,C=GB"
        p2pPort 10005
        rpcSettings {
            port 10006
            adminPort 10026
        }
        webPort 10007
        h2Port 10008
        cordapps = ["net.corda:corda-finance:$corda_release_version"]
        // Grants user1 all RPC permissions.
        rpcUsers = [[ user: "user1", "password": "test", "permissions": ["ALL"]]]
    }
    node {
        name "O=PartyB,L>New York,C=US"
        p2pPort 10009
        rpcSettings {
            port 10010
            adminPort 10030
        }
        webPort 10011
        h2Port 10012
        cordapps = ["net.corda:corda-finance:$corda_release_version"]
        // Grants user1 the ability to start the MyFlow flow.
        rpcUsers = [[ user: "user1", "password": "test", "permissions": [
            "StartFlow.net.corda.flows.MyFlow"
        ]]]
    }
}
```

To extend node configuration beyond the properties defined in the `deployNodes` task use the `configFile` property with the path (relative or

absolute) set to an additional configuration file. This file should follow the standard Node configuration format, as per node.conf. The properties from this file will be appended to the generated node configuration. The path to the file can also be added while running the Gradle task via the `-PconfigFile` command line option. However, the same file will be applied to all nodes. Following the previous example `PartyB` node will have additional configuration options added from a file `none-b.conf`:

```
task deployNodes(type: net.corda.plugins.Cordform, dependsOn: ['jar']) {  
    [...]  
    node {  
        name "O=PartyB,L>New York,C=US"  
        [...]  
        // Grants user1 the ability to start the MyFlow flow.  
        rpcUsers = [[ user: "user1", "password": "test", "permissions":  
["StartFlow.net.corda.flows.MyFlow"]]]  
        configFile = "samples/trader-demo/src/main/resources/none-b.conf"  
    }  
}
```

Running this task will create three nodes in the `build/nodes` folder:

- A `Notary` node that:
 - Offers a validating notary service
 - Will not have a webserver (since `webPort` is not defined)
 - Is running the `corda-finance` CorDapp
- `PartyA` and `PartyB` nodes that:
 - Are not offering any services
 - Will have a webserver (since `webPort` is defined)
 - Are running the `corda-finance` CorDapp
 - Have an RPC user, `user1`, that can be used to log into the node via RPC

Additionally, all three nodes will include any CorDapps defined in the project's source folders, even though these CorDapps are not listed in each node's `cordapps` entry. This means that running the `deployNodes` task from the template CorDapp, for example, would automatically build and add the template CorDapp to each node.

You can extend `deployNodes` to generate additional nodes.

Warning

When adding nodes, make sure that there are no port clashes!

The ``Dockerform`` is a sister task of ``Cordform``. It has nearly the same syntax and produces very similar results - enhanced by an extra file to enable easy spin up of nodes using ``docker-compose``. Below you can find the example task from the ``IRS Demo<https://github.com/corda/corda/blob/release-V3.0/samples/irs-demo/cordapp/build.gradle#L111>`` included in the samples directory of main Corda GitHub repository:

```
def rpcUsersList = [
    ['username' : "user",
     'password' : "password",
     'permissions' : [
         "StartFlow.net.corda.irs.flows.AutoOfferFlow\$Requester",
         "StartFlow.net.corda.irs.flows.UpdateBusinessDayFlow\$Broadcast",
         "StartFlow.net.corda.irs.api.NodeInterestRates\$UploadFixesFlow",
         "InvokeRpc.vaultQueryBy",
         "InvokeRpc.networkMapSnapshot",
         "InvokeRpc.currentNodeTime",
         "InvokeRpc.wellKnownPartyFromX500Name"
     ]]
]

// (...)

task prepareDockerNodes(type: net.corda.plugins.Dockerform, dependsOn: ['jar']) {
    node {
        name "O=Notary Service,L=Zurich,C=CH"
        notary = [validating : true]
        cordapps = ["${project(":finance").group}:finance:$corda_release_version"]
        rpcUsers = rpcUsersList
        useTestClock true
    }
    node {
        name "O=Bank A,L=London,C=GB"
        cordapps = ["${project(":finance").group}:finance:$corda_release_version"]
        rpcUsers = rpcUsersList
        useTestClock true
    }
    node {
        name "O=Bank B,L=New York,C=US"
        cordapps = ["${project(":finance").group}:finance:$corda_release_version"]
        rpcUsers = rpcUsersList
        useTestClock true
    }
    node {
        name "O=Regulator,L=Moscow,C=RU"
        cordapps = ["${project.group}:finance:$corda_release_version"]
        rpcUsers = rpcUsersList
        useTestClock true
    }
}
```

There is no need to specify the ports, as every node is a separated container, so no ports conflict will occur. Running the task will create the same folders structure as described in The Cordform task with an additional ``Dockerfile`` in

each node directory, and `docker-compose.yml` in `build/nodes` directory. Every node by default exposes port 10003 which is the default one for RPC connections.

Warning

Webserver is not supported by this task!

Warning

Nodes are run without the local shell enabled!

To create the nodes defined in our `deployNodes` task, run the following command in a terminal window from the root of the project where the `deployNodes` task is defined:

- Linux/macOS: `./gradlew deployNodes`
- Windows: `gradlew.bat deployNodes`

This will create the nodes in the `build/nodes` folder. There will be a node folder generated for each node defined in the `deployNodes` task, plus a `runnodes` shell script (or batch file on Windows) to run all the nodes at once for testing and development purposes. If you make any changes to your CorDapp source or `deployNodes` task, you will need to re-run the task to see the changes take effect.

You can now run the nodes by following the instructions in [Running a node](#).

[Next](#) [Previous](#)

- [Running nodes locally](#)
- [View page source](#)

Running nodes locally

Contents

- [Running nodes locally](#)

- [Starting all nodes at once](#)
- [Starting an individual Corda node](#)
- [Enabling remote debugging](#)

Note

You should already have generated your node(s) with their CorDApps installed by following the instructions in [Creating nodes locally](#).

There are several ways to run a Corda node locally for testing purposes.

Starting all nodes at once

`runnodes` is a shell script (or batch file on Windows) that is generated by `deployNodes` to allow you to quickly start up all nodes and their web servers. `runnodes` should only be used for testing purposes.

Start the nodes with `runnodes` by running the following command from the root of the project:

- Linux/macOS: `build/nodes/runnodes`
- Windows: `call build\nodes\runnodes.bat`

If you receive an `OutOfMemoryError` exception when interacting with the nodes, you need to increase the amount of Java heap memory available to them, which you can do when running them individually. See [Starting an individual Corda node](#).

Starting an individual Corda node

Run the node by opening a terminal window in the node's folder and running:

```
java -jar corda.jar
```

By default, the node will look for a configuration file called `node.conf` and a CorDApps folder called `cordapps` in the current working directory. You can override the configuration file and workspace paths on the command line (e.g. `./corda.jar --config-file=test.conf --base-directory=/opt/corda/nodes/test`).

You can increase the amount of Java heap memory available to the node using the `-Xmx` command line argument. For example, the following would run the node with a heap size of 2048MB:

```
java -Xmx2048m -jar corda.jar
```

You should do this if you receive an `OutOfMemoryError` exception when interacting with the node.

Optionally run the node's webserver as well by opening a terminal window in the node's folder and running:

```
java -jar corda-webserver.jar
```

Warning

The node webserver is for testing purposes only and will be removed soon.

Enabling remote debugging

To enable remote debugging of the node, run the node with the following JVM arguments:

```
java -Dcapsule.jvm.args="-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005" -jar corda.jar
```

This will allow you to attach a debugger to your node on port 5005.

[Next](#) [Previous](#)

- [Deploying a node](#)
 - [View page source](#)
-

Deploying a node

Contents

- [Deploying a node](#)
 - [Linux: Installing and running Corda as a system service](#)
 - [Windows: Installing and running Corda as a Windows service](#)
 - [Testing your installation](#)

Note

These instructions are intended for people who want to deploy a Corda node to a server, whether they have developed and tested a CorDapp following the instructions in [Creating nodes locally](#) or are deploying a third-party CorDapp.

Linux: Installing and running Corda as a system service

We recommend creating system services to run a node and the optional webserver. This provides logging and service handling, and ensures the Corda service is run at boot.

Prerequisites:

- Oracle Java 8. The supported versions are listed in [Getting set up](#)

1. Add a system user which will be used to run Corda:

```
sudo adduser --system --no-create-home --group corda
```

2. Create a directory called `/opt/corda` and change its ownership to the user you want to use to run Corda:

```
mkdir /opt/corda; chown corda:corda /opt/corda
```

3. Download the [Corda jar](#) (under `/VERSION_NUMBER/corda-VERSION_NUMBER.jar`) and place it in `/opt/corda`

4. (Optional) Download the [Corda webserver jar](#) (under `/VERSION_NUMBER/corda-VERSION_NUMBER.jar`) and place it in `/opt/corda`

5. Create a directory called `cordapps` in `/opt/corda` and save your CorDapp jar file to it. Alternatively, download one of our [sample CorDapps](#) to the `cordapps` directory

6. Save the below as `/opt/corda/node.conf`. See [Node configuration](#) for a description of these options

```
7. basedir : "/opt/corda"
8. p2pAddress : "example.com:10002"
9. rpcAddress : "example.com:10003"
10. webAddress : "0.0.0.0:10004"
11. h2port : 11000
12. emailAddress : "you@example.com"
13. myLegalName : "O=Bank of Breakfast Tea, L=London, C=GB"
14. keyStorePassword : "cordacadevpass"
15. trustStorePassword : "trustpass"
16. devMode : false
17. rpcUsers=[
18.   {
19.     user=corda
20.     password=portal_password
```

```
21.     permissions=[  
22.         ALL  
23.     ]  
24. }  
25.]
```

26. Make the following changes to `/opt/corda/node.conf`:

- Change the `p2pAddress` and `rpcAddress` values to start with your server's hostname or external IP address. This is the address other nodes or RPC interfaces will use to communicate with your node
- Change the ports if necessary, for example if you are running multiple nodes on one server (see below)
- Enter an email address which will be used as an administrative contact during the registration process. This is only visible to the permissioning service
- Enter your node's desired legal name. This will be used during the issuance of your certificate and should rarely change as it should represent the legal identity of your node
 - Organization (`O=`) should be a unique and meaningful identifier (e.g. Bank of Breakfast Tea)
 - Location (`L=`) is your nearest city
 - Country (`C=`) is the [ISO 3166-1 alpha-2 code](#)
- Change the RPC username and password

Note

Ubuntu 16.04 and most current Linux distributions use SystemD, so if you are running one of these distributions follow the steps marked **SystemD**. If you are running Ubuntu 14.04, follow the instructions for **Upstart**.

8. **SystemD:** Create a `corda.service` file based on the example below and save it in the `/etc/systemd/system/` directory

```
9. [Unit]  
10. Description=Corda Node - Bank of Breakfast Tea  
11. Requires=network.target  
12.  
13. [Service]  
14. Type=simple  
15. User=corda  
16. WorkingDirectory=/opt/corda  
17. ExecStart=/usr/bin/java -Xmx2048m -jar /opt/corda/corda.jar  
18. Restart=on-failure  
19.  
20. [Install]  
21. WantedBy=multi-user.target
```

8. **Upstart:** Create a `corda.conf` file based on the example below and save it in the `/etc/init/` directory

```
9. description "Corda Node - Bank of Breakfast Tea"
10.
11. start on runlevel [2345]
12. stop on runlevel [|2345]
13.
14. respawn
15. setuid corda
16. chdir /opt/corda
17. exec java -Xmx2048m -jar /opt/corda/corda.jar
```

18. Make the following changes to `corda.service` or `corda.conf`:

- Make sure the service description is informative - particularly if you plan to run multiple nodes.
- Change the username to the user account you want to use to run Corda. **We recommend that this user account is not root**
- Set the maximum amount of memory available to the Corda process by changing the `-Xmx2048m` parameter
- **SystemD:** Make sure the `corda.service` file is owned by root with the correct permissions:
 - `sudo chown root:root /etc/systemd/system/corda.service`
 - `sudo chmod 644 /etc/systemd/system/corda.service`
- **Upstart:** Make sure the `corda.conf` file is owned by root with the correct permissions:
 - `sudo chown root:root /etc/init/corda.conf`
 - `sudo chmod 644 /etc/init/corda.conf`

Note

The Corda webserver provides a simple interface for interacting with your installed CorDapps in a browser. Running the webserver is optional.

10. **SystemD:** Create a `corda-webserver.service` file based on the example below and save it in the `/etc/systemd/system/` directory

```
11. [Unit]
12. Description=Webserver for Corda Node - Bank of Breakfast Tea
13. Requires=network.target
14.
15. [Service]
16. Type=simple
17. User=corda
18. WorkingDirectory=/opt/corda
19. ExecStart=/usr/bin/java -jar /opt/corda/corda-webserver.jar
20. Restart=on-failure
21.
```

```
22. [Install]
23. WantedBy=multi-user.target
```

10. **Upstart:** Create a `corda-webserver.conf` file based on the example below and save it in the `/etc/init/` directory

```
11. description "Webserver for Corda Node - Bank of Breakfast Tea"
12.
13. start on runlevel [2345]
14. stop on runlevel [!2345]
15.
16. respawn
17. setuid corda
18. chdir /opt/corda
19. exec java -jar /opt/corda/corda-webserver.jar
```

20. Provision the required certificates to your node. Contact the network permissioning service or see [Network permissioning](#)

21. **SystemD:** You can now start a node and its webserver and set the services to start on boot by running the following `systemctl` commands:

- `sudo systemctl daemon-reload`
- `sudo systemctl enable --now corda`
- `sudo systemctl enable --now corda-webserver`

12. **Upstart:** You can now start a node and its webserver by running the following commands:

- `sudo start corda`
- `sudo start corda-webserver`

The Upstart configuration files created above tell Upstart to start the Corda services on boot so there is no need to explicitly enable them.

You can run multiple nodes by creating multiple directories and Corda services, modifying the `node.conf` and SystemD or Upstart configuration files so they are unique.

Windows: Installing and running Corda as a Windows service

We recommend running Corda as a Windows service. This provides service handling, ensures the Corda service is run at boot, and means the Corda service stays running with no users connected to the server.

Prerequisites:

- Oracle Java 8. The supported versions are listed in [Getting set up](#)
1. Create a Corda directory and download the Corda jar.
Replace `VERSION_NUMBER` with the desired version. Here's an example using PowerShell:
 2. `mkdir C:\Corda`
 3. `wget http://jcenter.bintray.com/net/corda/corda/VERSION_NUMBER/corda-VERSION_NUMBER.jar -OutFile C:\Corda\corda.jar`
 4. Create a directory called `cordapps` in `C:\Corda\` and save your CorDapp jar file to it. Alternatively, download one of our [sample CorDapps](#) to the `cordapps` directory
 5. Save the below as `C:\Corda\node.conf`. See [Node configuration](#) for a description of these options
 6. `basedir : "C:\\Corda"`
7. `p2pAddress : "example.com:10002"`
8. `rpcAddress : "example.com:10003"`
9. `webAddress : "0.0.0.0:10004"`
10. `h2port : 11000`
11. `emailAddress: "you@example.com"`
12. `myLegalName : "O=Bank of Breakfast Tea, L=London, C=GB"`
13. `keyStorePassword : "cordacadevpass"`
14. `trustStorePassword : "trustpass"`
15. `extraAdvertisedServiceIds: [""]`
16. `devMode : false`
17. `rpcUsers=[`
18. `{`
19. `user=corda`
20. `password=portal_password`
21. `permissions=[`
22. `ALL`
23. `]`
24. `}`
25. `]`

26. Make the following changes to `C:\Corda\node.conf`:

- Change the `p2pAddress` and `rpcAddress` values to start with your server's hostname or external IP address. This is the address other nodes or RPC interfaces will use to communicate with your node
- Change the ports if necessary, for example if you are running multiple nodes on one server (see below)
- Enter an email address which will be used as an administrative contact during the registration process. This is only visible to the permissioning service

- Enter your node's desired legal name. This will be used during the issuance of your certificate and should rarely change as it should represent the legal identity of your node
 - Organization (`O=`) should be a unique and meaningful identifier (e.g. Bank of Breakfast Tea)
 - Location (`L=`) is your nearest city
 - Country (`C=`) is the ISO 3166-1 alpha-2 code
- Change the RPC username and password

27. Copy the required Java keystores to the node. See Network permissioning

28. Download the [NSSM service manager](#)

29. Unzip `nssm-2.24\win64\nssm.exe` to `C:\Corda`

30. Save the following as `C:\Corda\nssm.bat`:

```
31. nssm install cordanode1 C:\ProgramData\Oracle\Java\javapath\java.exe
32. nssm set cordanode1 AppDirectory C:\Corda
33. nssm set cordanode1 AppParameters "-Xmx2048m -jar corda.jar --config-
  file=C:\corda\node.conf"
34. nssm set cordanode1 AppStdout C:\Corda\service.log
35. nssm set cordanode1 AppStderr C:\Corda\service.log
36. nssm set cordanode1 Description Corda Node - Bank of Breakfast Tea
37. nssm set cordanode1 Start SERVICE_AUTO_START
38. sc start cordanode1
```

39. Modify the batch file:

- If you are installing multiple nodes, use a different service name (`cordanode1`) for each node
- Set the amount of Java heap memory available to this node by modifying the `-Xmx` argument
- Set an informative description

40. Run the batch file by clicking on it or from a command prompt

41. Run `services.msc` and verify that a service called `cordanode1` is present and running

42. Run `netstat -ano` and check for the ports you configured in `node.conf`

43. You may need to open the ports on the Windows firewall

Testing your installation

You can verify Corda is running by connecting to your RPC port from another host, e.g.:

```
telnet your-hostname.example.com 10002
```

If you receive the message “Escape character is ^]”, Corda is running and accessible. Press Ctrl-] and Ctrl-D to exit telnet.

- Node configuration
 - [View page source](#)
-

Node configuration

File location

The Corda all-in-one `corda.jar` file is generated by the `gradle buildCordaJAR` task and defaults to reading configuration from a `node.conf` file in the present working directory. This behaviour can be overridden using the `--config-file` command line option to target configuration files with different names, or different file location (relative paths are relative to the current working directory). Also, the `--base-directory` command line option alters the Corda node workspace location and if specified a `node.conf` configuration file is then expected in the root of the workspace.

The configuration file templates used for the `gradle deployNodes` task are to be found in the `/config/dev` folder. Also note that there is a basic set of defaults loaded from the built in resource file `/node/src/main/resources/reference.conf` of the `:node` gradle module. All properties in this can be overridden in the file configuration and for rarely changed properties this defaulting allows the property to be excluded from the configuration file.

Format

The Corda configuration file uses the HOCON format which is superset of JSON. It has several features which makes it very useful as a configuration format. Please visit their [page](#) for further details.

Examples

General node configuration file for hosting the IRSDemo services.

```
myLegalName : "O=Bank A,L=London,C=GB"
keyStorePassword : "cordacadevpass"
trustStorePassword : "trustpass"
dataSourceProperties : {
    dataSourceClassName : org.h2.jdbcx.JdbcDataSource
```

```

    "dataSource.url" : "jdbc:h2:file:"${baseDirectory}"/persistence"
    "dataSource.user" : sa
    "dataSource.password" : ""
}
p2pAddress : "my-corda-node:10002"
rpcSettings = {
    useSsl = false
    standAloneBroker = false
    address : "my-corda-node:10003"
    adminAddress : "my-corda-node:10004"
}
webAddress : "localhost:10004"
rpcUsers : [
    { username=user1, password=letmein, permissions=[StartFlow.net.corda.protocols.CashProtocol ] }
]
devMode : true

```

Simple Notary configuration file.

```

myLegalName : "O=Notary Service,OU=corda,L=London,C=GB"
keyStorePassword : "cordacadevpass"
trustStorePassword : "trustpass"
p2pAddress : "localhost:12345"
rpcSettings = {
    useSsl = false
    standAloneBroker = false
    address : "my-corda-node:10003"
    adminAddress : "my-corda-node:10004"
}
webAddress : "localhost:12347"
notary : {
    validating : false
}
devMode : true
compatibilityZoneURL : "https://cz.corda.net"

```

Fields

The available config fields are listed below. `baseDirectory` is available as a substitution value, containing the absolute path to the node's base directory.

myLegalName: The legal identity of the node acts as a human readable alias to the node's public key and several demos use this to lookup the NodeInfo.

keyStorePassword:

The password to unlock the KeyStore file (`<workspace>/certificates/sslkeystore.jks`) containing the node certificate and private key.

ote

This is the non-secret value for the development certificates automatically generated during the first node run. Longer term these keys will be managed in secure

hardware devices.

trustStorePassword:

The password to unlock the Trust store file (`<workspace>/certificates/truststore.jks`) containing the Corda network root certificate. This is the non-secret value for the development certificates automatically generated during the first node run.

ote

Longer term these keys will be managed in secure hardware devices.

Database configuration:

serverNameTablePrefix:

Prefix string to apply to all the database tables. The default is no prefix.

transactionIsolationLevel:

database:

Transaction isolation level as defined by the `TRANSACTION_` constants in `java.sql.Connection`, but without the “`TRANSACTION_`” prefix. Defaults to `REPEATABLE_READ`.

exportHibernateJMXStatistics:

Whether to export Hibernate JMX statistics (caution: expensive run-time overhead)

dataSourceProperties:

This section is used to configure the jdbc connection and database driver used for the nodes persistence. Currently the defaults in `/node/src/main/resources/reference.conf` are as shown in the first example. This is currently the only configuration that has been tested, although in the future full support for other storage layers will be validated.

messagingServerAddress:

The address of the ArtemisMQ broker instance. If not provided the node will run one locally.

The host and port on which the node is available for protocol operations over ArtemisMQ.

ote

p2pAddress:

In practice the ArtemisMQ messaging services bind to all local addresses on the specified port. However, note that the host is included as the advertised entry in the network map. As a result the value listed here must be externally accessible when running nodes across a cluster of machines. If the provided host is

unreachable, the node will try to auto-discover its public one.

rpcAddress: The address of the RPC system on which RPC requests can be made to the node. If not provided then the node will run without RPC. This is now deprecated in favour of the `rpcSettings` block.

Options for the RPC server.

useSsl: (optional) boolean, indicates whether the node should require clients to use SSL for RPC connections, defaulted to `false`.

standaloneBroker:

(optional) boolean, indicates whether the node will connect to a standalone broker for RPC, defaulted to `false`.

address: (optional) host and port for the RPC server binding, if any.

adminAddress: (optional) host and port for the RPC admin binding (only required when `useSsl` is `false`, because the node connects to Artemis using SSL to ensure admin privileges are not accessible outside the node).

rpcSettings: (optional) SSL settings for the RPC server.

keyStorePassword:

password for the key store.

trustStorePassword:

password for the trust store.

ssl:

certificatesDirectory:

directory in which the stores will be searched, unless absolute paths are provided.

sslKeystore: absolute path to the ssl key store, defaulted to `certificatesDirectory / "sslkeystore.jks"`.

trustStoreFile: absolute path to the trust store, defaulted to `certificatesDirectory / "truststore.jks"`.

security: Contains various nested fields controlling user authentication/authorization, in particular for RPC accesses. See [Client RPC](#) for details.

The host and port on which the webserver will listen if it is started. This is not used by the node itself.

webAddress:

ote

If HTTPS is enabled then the browser security checks will require that the accessing

url host name is one of either the machine name, fully qualified machine name, or server IP address to line up with the Subject Alternative Names contained within the development certificates. This is addition to requiring the `/config/dev/corda_dev_ca.cer` root certificate be installed as a Trusted CA.

Note

The driver will not automatically create a webserver instance, but the Cordformation will. If this field is present the web server will start.

Optional configuration object which if present configures the node to run as a notary. If part of a Raft or BFT SMaRt cluster then specify `raft` or `bftSMaRt` respectively as described below. If a single node notary then omit both.

validating: Boolean to determine whether the notary is a validating or non-validating one.

If part of a distributed Raft cluster specify this config object, with the following settings:

raft: **nodeAddress:** The host and port to which to bind the embedded Raft server. Note that the Raft cluster uses a separate transport layer for communication that does not integrate with ArtemisMQ messaging services.

clusterAddresses:

notary: Must list the addresses of all the members in the cluster. At least one of the members must be active and be able to communicate with the cluster leader for the node to join the cluster. If empty, a new cluster will be bootstrapped.

If part of a distributed BFT-SMaRt cluster specify this config object, with the following settings:

bftSMaRt: **replicaId:** The zero-based index of the current replica. All replicas must specify a unique replica id.

clusterAddresses:

Must list the addresses of all the members in the cluster. At least one of the members must be active and be able to communicate with the cluster leader for the node to join the cluster. If empty, a new cluster will be bootstrapped.

custom: If *true*, will load and install a notary service from a CorDapp. See [Writing a custom notary service \(experimental\)](#).

Only one of `raft`, `bftSMaRt` or `custom` configuration values may be specified.

rpcUsers: A list of users who are authorised to access the RPC system. Each user in the list is

a config object with the following fields:

username: Username consisting only of word characters (a-z, A-Z, 0-9 and _)

password: The password

A list of permissions for starting flows via RPC. To give the user the permission to start the flow `foo.bar.FlowClass`, add the

permissions: string `StartFlow.foo.bar.FlowClass` to the list. If the list contains the string `ALL`, the user can start any flow via RPC. This value is intended for administrator users and for development.

devMode: This flag sets the node to run in development mode. On startup, if the keystore `<workspace>/certificates/sslkeystore.jks` does not exist, a developer keystore will be used if `devMode` is true. The node will exit if `devMode` is false and the keystore does not exist. `devMode` also turns on background checking of flow checkpoints to shake out any bugs in the checkpointing process. Also, if `devMode` is true, Hibernate will try to automatically create the schema required by Corda or update an existing schema in the SQL database; if `devMode` is false, Hibernate will simply validate an existing schema failing on node start if this schema is either not present or not compatible.

detectPublicIp: This flag toggles the auto IP detection behaviour, it is enabled by default. On startup the node will attempt to discover its externally visible IP address first by looking for any public addresses on its network interfaces, and then by sending an IP discovery request to the network map service. Set to `false` to disable.

compatibilityZoneURL:

The root address of Corda compatibility zone network management services, it is used by the Corda node to register with the network and obtain Corda node certificate, (See [Network permissioning](#) for more information.) and also used by the node to obtain network map information. Cannot be set at the same time as the `networkServices` option.

networkServices:

If the Corda compatibility zone services, both network map and registration (doorman), are not running on the same endpoint and thus have different URLs then this option should be used in place of the `compatibilityZoneURL` setting.

doormanURL: Root address of the network registration service.

networkMapURL: Root address of the network map service.

Note

Only one of `compatibilityZoneURL` or `networkServices` should be used.

jvmArgs: An optional list of JVM args, as strings, which replace those inherited from the

command line when launching via `corda.jar` only. e.g. `jvmArgs = ["-Xmx220m", "-Xms220m", "-XX:+UseG1GC"]`

systemProperties:

An optional map of additional system properties to be set when launching via `corda.jar` only. Keys and values of the map should be strings.

e.g. `systemProperties = { visualvm.display.name = FooBar }`

jarDirs: An optional list of file system directories containing JARs to include in the classpath when launching via `corda.jar` only. Each should be a string. Only the JARs in the directories are added, not the directories themselves. This is useful for including JDBC drivers and the like. e.g. `jarDirs = ['lib']`

sshd: If provided, node will start internal SSH server which will provide a management shell. It uses the same credentials and permissions as RPC subsystem. It has one required parameter.

port: The port to start SSH server on

exportJMXTo: If set to `http`, will enable JMX metrics reporting via the Jolokia HTTP/JSON agent. Default Jolokia access url is <http://127.0.0.1:7005/jolokia/>

transactionCacheSizeMegabytes:

Optionally specify how much memory should be used for caching of ledger transactions in memory. Otherwise defaults to 8MB plus 5% of all heap memory above 300MB.

attachmentContentCacheSizeMegabytes:

Optionally specify how much memory should be used to cache attachment contents in memory. Otherwise defaults to 10MB

attachmentCacheBound:

Optionally specify how many attachments should be cached locally. Note that this includes only the key and metadata, the content is cached separately and can be loaded lazily. Defaults to 1024.

- Client RPC
- [View page source](#)

Client RPC

Contents

- [Client RPC](#)
 - [Overview](#)
 - [RPC permissions](#)
 - [Granting flow permissions](#)
 - [Granting other RPC permissions](#)
 - [Granting all permissions](#)
 - [RPC security management](#)
 - [Observables](#)
 - [Futures](#)
 - [Versioning](#)
 - [Thread safety](#)
 - [Error handling](#)
 - [Wire protocol](#)
 - [Wire security](#)
 - [Whitelisting classes with the Corda node](#)

Overview

Corda provides a client library that allows you to easily write clients in a JVM-compatible language to interact with a running node. The library connects to the node using a message queue protocol and then provides a simple RPC interface to interact with the node. You make calls on a Java object as normal, and the marshalling back and forth is handled for you.

The starting point for the client library is the [CordaRPCClient](#) class. [CordaRPCClient](#) provides a `start` method that returns a [CordaRPCConnection](#). A [CordaRPCConnection](#) allows you to access an implementation of the [CordaRPCOps](#) interface with `proxy` in Kotlin or `getProxy()` in Java. The observables that are returned by RPC operations can be subscribed to in order to receive an ongoing stream of updates from the node. More detail on this functionality is provided in the docs for the `proxy` method.

Warning

The returned [CordaRPCConnection](#) is somewhat expensive to create and consumes a small amount of server side resources. When you're done with it, call `close` on it. Alternatively you may use the `use` method on [CordaRPCClient](#) which cleans up automatically after the passed in lambda

finishes. Don't create a new proxy for every call you make - reuse an existing one.

For a brief tutorial on using the RPC API, see [Using the client RPC API](#).

RPC permissions

For a node's owner to interact with their node via RPC, they must define one or more RPC users. Each user is authenticated with a username and password, and is assigned a set of permissions that control which RPC operations they can perform.

RPC users are created by adding them to the `rpcUsers` list in the node's `node.conf` file:

```
rpcUsers=[  
    {  
        username=exampleUser  
        password=examplePass  
        permissions=[]  
    }  
    ...  
]
```

By default, RPC users are not permissioned to perform any RPC operations.

Granting flow permissions

You provide an RPC user with the permission to start a specific flow using the syntax `StartFlow.<fully qualified flow name>`:

```
rpcUsers=[  
    {  
        username=exampleUser  
        password=examplePass  
        permissions=[  
            "StartFlow.net.corda.flows.ExampleFlow1",  
            "StartFlow.net.corda.flows.ExampleFlow2"  
        ]  
    }  
    ...  
]
```

You can also provide an RPC user with the permission to start any flow using the syntax `InvokeRpc.startFlow`:

```
rpcUsers=[  
    {  
        username=exampleUser  
        password=examplePass  
        permissions=[  
            "InvokeRpc.startFlow"  
        ]  
    }  
    ...  
]
```

Granting other RPC permissions

You provide an RPC user with the permission to perform a specific RPC operation using the syntax `InvokeRpc.<rpc method name>`:

```
rpcUsers=[  
    {  
        username=exampleUser  
        password=examplePass  
        permissions=[  
            "InvokeRpc.nodeInfo",  
            "InvokeRpc.networkMapSnapshot"  
        ]  
    }  
    ...  
]
```

Granting all permissions

You can provide an RPC user with the permission to perform any RPC operation (including starting any flow) using the `ALL` permission:

```
rpcUsers=[  
    {  
        username=exampleUser  
        password=examplePass  
        permissions=[  
            "ALL"  
        ]  
    }  
    ...  
]
```

RPC security management

Setting `rpcUsers` provides a simple way of granting RPC permissions to a fixed set of users, but has some obvious shortcomings. To support use cases aiming for higher security and flexibility, Corda offers additional security features such as:

- Fetching users credentials and permissions from an external data source (e.g.: a remote RDBMS), with optional in-memory caching. In particular, this allows credentials and permissions to be updated externally without requiring nodes to be restarted.
- Password stored in hash-encrypted form. This is regarded as must-have when security is a concern. Corda currently supports a flexible password hash format conforming to the Modular Crypt Format provided by the [Apache Shiro framework](#)

These features are controlled by a set of options nested in the `security` field of `node.conf`. The following example shows how to configure retrieval of users credentials and permissions from a remote database with passwords in hash-encrypted format and enable in-memory caching of users data:

```
security = {
    authService = {
        dataSource = {
            type = "DB",
            passwordEncryption = "SHIRO_1_CRYPT",
            connection = {
                jdbcUrl = "<jdbc connection string>"
                username = "<db username>"
                password = "<db user password>"
                driverClassName = "<JDBC driver>"
            }
        }
        options = {
            cache = {
                expireAfterSecs = 120
                maxEntries = 10000
            }
        }
    }
}
```

It is also possible to have a static list of users embedded in the `security` structure by specifying a `dataSource` of `INMEMORY` type:

```
security = {
    authService = {
        dataSource = {
            type = "INMEMORY",
            users = [
                {
                    username = "<username>",
                    password = "<password>",
                    permissions = ["<permission 1>", "<permission 2>", ...]
                },
                ...
            ]
        }
    }
}
```

Warning

A valid configuration cannot specify both the `rpcUsers` and `security` fields. Doing so will trigger an exception at node startup.

The `dataSource` structure defines the data provider supplying credentials and permissions for users. There exist two supported types of such data source, identified by the `dataSource.type` field:

INMEMORY: A static list of user credentials and permissions specified by the `users` field.

An external RDBMS accessed via the JDBC connection described by `connection`. Note that, unlike the `INMEMORY` case, in a user database permissions are assigned to *roles* rather than individual users. The current implementation expects the database to store data according to the following schema:

- Table `users` containing columns `username` and `password`.
The `username` column *must have unique values*.
- Table `user_roles` containing columns `username` and `role_name` associating a user to a set of *roles*.
- Table `roles_permissions` containing columns `role_name` and `permission` associating a role to a set of permission strings.

DB:

Note

There is no prescription on the SQL type of each column (although our tests were conducted on `username` and `role_name` declared of SQL type `VARCHAR` and `password` of `TEXT` type). It is also possible to have extra columns in each table alongside the expected ones.

Storing passwords in plain text is discouraged in applications where security is critical. Passwords are assumed to be in plain format by default, unless a different format is specified by the `passwordEncryption` field, like:

```
passwordEncryption = SHIRO_1_CRYPT
```

`SHIRO_1_CRYPT` identifies the [Apache Shiro fully reversible Modular Crypt Format](#), it is currently the only non-plain password hash-encryption format supported. Hash-encrypted passwords in this format can be produced by using the [Apache Shiro Hasher command line tool](#).

A cache layer on top of the external data source of users credentials and permissions can significantly improve performances in some cases, with the disadvantage of causing a (controllable) delay in picking up updates to the underlying data. Caching is disabled by default, it can be enabled by defining the `options.cache` field in `security.authService`, for example:

```
options = {
    cache = {
        expireAfterSecs = 120
        maxEntries = 10000
    }
}
```

This will enable a non-persistent cache contained in the node's memory with maximum number of entries set to `maxEntries` where entries are expired and refreshed after `expireAfterSecs` seconds.

Observables

The RPC system handles observables in a special way. When a method returns an observable, whether directly or as a sub-object of the response object graph, an observable is created on the client to match the one on the server. Objects emitted by the server-side observable are pushed onto a queue which is then drained by the client. The returned observable may even emit object graphs with even more observables in them, and it all works as you would expect.

This feature comes with a cost: the server must queue up objects emitted by the server-side observable until you download them. Note that the server side observation buffer is bounded, once it fills up the client is considered slow and kicked. You are expected to subscribe to all the observables returned, otherwise client-side memory starts filling up as observations come in. If you don't want an observable then subscribe then unsubscribe immediately to clear the client-side buffers and to stop the server from streaming. If your app quits then server side resources will be freed automatically.

Warning

If you leak an observable on the client side and it gets garbage collected, you will get a warning printed to the logs and the observable will be unsubscribed for you. But don't rely on this, as garbage collection is non-deterministic.

Futures

A method can also return a `ListenableFuture` in its object graph and it will be treated in a similar manner to observables. Calling the `cancel` method on the future will unsubscribe it from any future value and release any resources.

Versioning

The client RPC protocol is versioned using the node's Platform Version (see [Versioning](#)). When a proxy is created the server is queried for its version, and you can specify your minimum requirement. Methods added in later versions are tagged with the `@RPCSinceVersion` annotation. If you try to use a method that the server isn't advertising support of, an `UnsupportedOperationException` is thrown.

If you want to know the version of the server, just use the `protocolVersion` property (i.e. `getProtocolVersion` in Java).

Thread safety

A proxy is thread safe, blocking, and allows multiple RPCs to be in flight at once. Any observables that are returned and you subscribe to will have objects emitted in order on a background thread pool. Each Observable stream is tied to a single thread, however note that two separate Observables may invoke their respective callbacks on different threads.

Error handling

If something goes wrong with the RPC infrastructure itself, an `RPCException` is thrown. If you call a method that requires a higher version of the protocol than the server supports, `UnsupportedOperationException` is thrown. Otherwise, if the server implementation throws an exception, that exception is serialised and rethrown on the client side as if it was thrown from inside the called RPC method. These exceptions can be caught as normal.

Wire protocol

The client RPC wire protocol is defined and documented in [`net/corda/client/rpc/RPCApi.kt`](#).

Wire security

`CordaRPCClient` has an optional constructor parameter of type `SSLConfiguration`, defaulted to `null`, which allows communication with the node using SSL. Default `null` value means no SSL used in the context of RPC.

Whitelisting classes with the Corda node

CorDapps must whitelist any classes used over RPC with Corda's serialization framework, unless they are whitelisted by default in `DefaultWhitelist`. The whitelisting is done either via the plugin architecture or by using the `@CordaSerializable` annotation. See [Object serialization](#). An example is shown in [Using the client RPC API](#).

- Shell
 - [View page source](#)
-

Shell

Contents

- [Shell](#)
 - [The shell via the local terminal](#)
 - [The shell via SSH](#)
 - [Enabling SSH access](#)
 - [Authentication](#)
 - [Connecting to the shell](#)
 - [Linux and MacOS](#)
 - [Windows](#)
 - [Permissions](#)
 - [Interacting with the node via the shell](#)
 - [Flow commands](#)
 - [Parameter syntax](#)
 - [Creating an instance of a class](#)
 - [Mappings from strings to types](#)
 - [Amount](#)
 - [SecureHash](#)
 - [OpaqueBytes](#)
 - [PublicKey and CompositeKey](#)

- [Party](#)
- [NodeInfo](#)
- [AnonymousParty](#)
- [NetworkHostAndPort](#)
- [Instant and Date](#)
- [Examples](#)
 - [Starting a flow](#)
 - [Querying the vault](#)
- [Attachments](#)
- [Getting help](#)
- [Extending the shell](#)
- [Limitations](#)

The Corda shell is an embedded command line that allows an administrator to control and monitor a node. It is based on the [CRaSH](#) shell and supports many of the same features. These features include:

- Invoking any of the node's RPC methods
- Viewing a dashboard of threads, heap usage, VM properties
- Uploading and downloading attachments
- Issuing SQL queries to the underlying database
- Viewing JMX metrics and monitoring exports
- UNIX style pipes for both text and objects, an [egrep](#) command and a command for working with columnar data

The shell via the local terminal

In development mode, the shell will display in the node's terminal window. It may be disabled by passing the [--no-local-shell](#) flag when running the node.

The shell via SSH

The shell is also accessible via SSH.

Enabling SSH access

By default, the SSH server is *disabled*. To enable it, a port must be configured in the node's [node.conf](#) file:

```
sshd {
```

```
    port = 2222  
}
```

Authentication

Users log in to shell via SSH using the same credentials as for RPC. This is because the shell actually communicates with the node using RPC calls. No RPC permissions are required to allow the connection and log in.

The host key is loaded from the `<node root directory>/sshkey/hostkey.pem` file. If this file does not exist, it is generated automatically. In development mode, the seed may be specified to give the same results on the same computer in order to avoid host-checking errors.

Connecting to the shell

Linux and MacOS

Run the following command from the terminal:

```
ssh -p [portNumber] [host] -l [user]
```

Where:

- `[portNumber]` is the port number specified in the `node.conf` file
- `[host]` is the node's host (e.g. `localhost` if running the node locally)
- `[user]` is the RPC username

The RPC password will be requested after a connection is established.

In development mode, restarting a node frequently may cause the host key to be regenerated. SSH usually saves trusted hosts and will refuse to connect in case of a change.
note: This check can be disabled using the `-o StrictHostKeyChecking=no` flag. This option should never be used in production environment!

Windows

Windows does not provide a built-in SSH tool. An alternative such as PuTTY should be used.

Permissions

When accessing the shell via SSH, some additional RPC permissions are required:

- Watching flows (`flow watch`) requires `InvokeRpc.stateMachinesFeed`
- Starting flows requires `InvokeRpc.startTrackedFlowDynamic` and `InvokeRpc.registeredFlows`, as well as a permission for the flow being started

Interacting with the node via the shell

The shell interacts with the node by issuing RPCs (remote procedure calls). You make an RPC from the shell by typing `run` followed by the name of the desired RPC method. For example, you'd see a list of the registered flows on your node by running:

```
run registeredFlows
```

Some RPCs return a stream of events that will be shown on screen until you press Ctrl-C.

You can find a list of the available RPC methods [here](#).

Flow commands

The shell also has special commands for working with flows:

- `flow list` lists the flows available on the node
- `flow watch` shows all the flows currently running on the node with result (or error) information
- `flow start` starts a flow. The `flow start` command takes the name of a flow class, or *any unambiguous substring* thereof, as well as the data to be passed to the flow constructor. If there are several matches for a given substring, the possible matches will be printed out. If a flow has multiple constructors then the names and types of the arguments will be used to try and automatically determine which one to use. If the match against available constructors is unclear, the reasons each available constructor failed to match will be printed out. In the case of an ambiguous match, the first applicable constructor will be used

Parameter syntax

Parameters are passed to RPC or flow commands using a syntax called [Yaml](#) (yet another markup language), a simple JSON-like language. The key features of Yaml are:

- Parameters are separated by commas
- Each parameter is specified as a `key: value` pair
 - There **MUST** be a space after the colon, otherwise you'll get a syntax error
- Strings do not need to be surrounded by quotes unless they contain commas, colons or embedded quotes
- Class names must be fully-qualified (e.g. `java.lang.String`)

Note

If your CorDapp is written in Java, named arguments won't work unless you compiled the node using the `-parameters` argument to javac. See [Creating nodes locally](#) for how to specify it via Gradle.

Creating an instance of a class

Class instances are created using curly-bracket syntax. For example, if we have a `Campaign` class with the following constructor:

```
data class Campaign(val name: String, val target: Int)
```

Then we could create an instance of this class to pass as a parameter as follows:

```
newCampaign: { name: Roger, target: 1000 }
```

Where `newCampaign` is a parameter of type `Campaign`.

Mappings from strings to types

In addition to the types already supported by Jackson, several parameter types can automatically be mapped from strings. We cover the most common types here.

Amount

A parameter of type `Amount<Currency>` can be written as either:

- A dollar (\$), pound (£) or euro (€) symbol followed by the amount as a decimal
- The amount as a decimal followed by the ISO currency code (e.g. "100.12 CHF")

SecureHash

A parameter of type `SecureHash` can be written as a hexadecimal

string: `F69A7626ACC27042FEEAE187E6BFF4CE666E6F318DC2B32BE9FAF87DF687930C`

OpaqueBytes

A parameter of type `OpaqueBytes` can be provided as a UTF-8 string.

PublicKey and CompositeKey

A parameter of type `PublicKey` can be written as a Base58 string of its encoded

format: `GfHq2tTVk9z4eXgyQXzegw6wNsZfHcDhw8oTt6fCHySFGp3g7XHPAyC2o6D`. `net.corda.core.`

`utilities.EncodingUtils.toBase58String` will convert a `PublicKey` to this string format.

Party

A parameter of type `Party` can be written in several ways:

- By using the full name: `"O=Monogram Bank,L=Sao Paulo,C=GB"`
- By specifying the organisation name only: `"Monogram Bank"`
- By specifying any other non-ambiguous part of the name: `"Sao Paulo"` (if only one network node is located in Sao Paulo)
- By specifying the public key (see above)

NodeInfo

A parameter of type `NodeInfo` can be written in terms of one of its identities

(see `Party` above)

AnonymousParty

A parameter of type `AnonymousParty` can be written in terms of its `PublicKey` (see above)

NetworkHostAndPort

A parameter of type `NetworkHostAndPort` can be written as a “host:port”

string: `"localhost:1010"`

Instant and Date

A parameter of `Instant` and `Date` can be written as an ISO-8601 string: `"2017-12-22T00:00:00Z"`

Examples

Starting a flow

We would start the `CashIssueFlow` flow as follows:

```
flow start CashIssueFlow amount: $1000, issuerBankPartyRef: 1234, notary: "O=Controll  
er, L=London, C=GB"
```

This breaks down as follows:

- `flow start` is a shell command for starting a flow
- `CashIssueFlow` is the flow we want to start
- Each `name: value` pair after that is a flow constructor argument

This command invokes the following `CashIssueFlow` constructor:

Kotlin

```
class CashIssueFlow(val amount: Amount<Currency>,  
                   val issuerBankPartyRef: OpaqueBytes,  
                   val recipient: Party,  
                   val notary: Party) : AbstractCashFlow(progressTracker)
```

Querying the vault

We would query the vault for `IOUState` states as follows:

```
run vaultQuery contractStateType: com.template.IOUState
```

This breaks down as follows:

- `run` is a shell command for making an RPC call
- `vaultQuery` is the RPC call we want to make
- `contractStateType: com.template.IOUState` is the fully-qualified name of the state type we are querying for

Attachments

The shell can be used to upload and download attachments from the node. To learn more, see the tutorial “[Using attachments](#)”.

Getting help

You can type `help` in the shell to list the available commands, and `man` to get interactive help on many commands. You can also pass the `--help` or `-h` flags to a command to get info about what switches it supports.

Commands may have subcommands, in the same style as `git`. In that case, running the command by itself will list the supported subcommands.

Extending the shell

The shell can be extended using commands written in either Java or [Groovy](#) (a Java-compatible scripting language). These commands have full access to the node's internal APIs and thus can be used to achieve almost anything.

A full tutorial on how to write such commands is out of scope for this documentation. To learn more, please refer to the [CRaSH](#) documentation. New commands are placed in the `shell-commands` subdirectory in the node directory.

Edits to existing commands will be used automatically, but currently commands added after the node has started won't be automatically detected. Commands must have names all in lower-case with either a `.java` or `.groovy` extension.

Warning

Commands written in Groovy ignore Java security checks, so have unrestricted access to node and JVM internals regardless of any sandboxing that may be in place. Don't allow untrusted users to edit files in the shell-commands directory!

Limitations

The shell will be enhanced over time. The currently known limitations include:

- There is no command completion for flows or RPCs
- Command history is not preserved across restarts
- The `jdb` command requires you to explicitly log into the database first
- Commands placed in the `shell-commands` directory are only noticed after the node is restarted
- The `jul` command advertises access to logs, but it doesn't work with the logging framework we're using

[Next](#) [Previous](#)

- Node database
- View page source

Node database

Currently, nodes store their data in an H2 database. In the future, we plan to support a wide range of databases.

You can connect directly to a running node's database to see its stored states, transactions and attachments as follows:

- Download the h2 platform-independent zip, unzip the zip, and navigate in a terminal window to the unzipped folder
- Change directories to the bin folder:

```
cd h2/bin
```

- Run the following command to open the h2 web console in a web browser tab:
 - Unix: `sh h2.sh`
 - Windows: `h2.bat`
- Find the node's JDBC connection string. Each node outputs its connection string in the terminal window as it starts up. In a terminal window where a node is running, look for the following string:

```
Database connection URL is : jdbc:h2:tcp://10.18.0.150:56736/node
```

- Paste this string into the JDBC URL field and click `Connect`, using the default username and password.

You will be presented with a web interface that shows the contents of your node's storage and vault, and provides an interface for you to query them using SQL.

[Next](#) [Previous](#)

- Node administration
- [View page source](#)

Node administration

When a node is running, it exposes an RPC interface that lets you monitor it, upload and download attachments, and so on.

Logging

By default the node log files are stored to the `logs` subdirectory of the working directory and are rotated from time to time. You can have logging printed to the console as well by passing the `--log-to-console` command line flag. The default logging level is `INFO` which can be adjusted by the `--logging-level` command line argument. This configuration option will affect all modules.

It may be the case that you require to amend the log level of a particular subset of modules (e.g. if you'd like to take a closer look at hibernate activity). So, for more bespoke logging configuration, the logger settings can be completely overridden with a [Log4j 2](#) configuration file assigned to the `log4j.configurationFile` system property.

Example

Create a file `sql.xml` in the current working directory. Add the following text :

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
        </Console>
    </Appenders>
    <Loggers>
        <Logger name="org.hibernate" level="debug" additivity="false">
            <AppenderRef ref="Console"/>
        </Logger>
        <Root level="error">
            <AppenderRef ref="Console"/>
        </Root>
    </Loggers>
</Configuration>
```

Note the addition of a logger named `org.hibernate` that has set this particular logger level to `debug`.

Now start the node as usual but with the additional parameter `log4j.configurationFile` set to the filename as above, e.g.

```
java <Your existing startup options here> -Dlog4j.configurationFile=sql.xml -
jar corda.jar
```

To determine the name of the logger, for Corda objects, use the fully qualified name (e.g. to look at node output in more detail, use `net.corda.node.internal.Node` although be aware that as we have marked this class `internal` we reserve the right to move and rename it as it's not part of the public API as yet). For other libraries, refer to their logging name construction. If you can't find what you need to refer to, use the `--logging-level` option as above and then determine the logging module name from the console output.

SSH access

Node can be configured to run SSH server. See [Shell](#) for details.

Database access

The node exposes its internal database over a socket which can be browsed using any tool that can use JDBC drivers. The JDBC URL is printed during node startup to the log and will typically look like this:

```
jdbc:h2:tcp://192.168.0.31:31339/node
```

The username and password can be altered in the [Node configuration](#) but default to username “sa” and a blank password.

Any database browsing tool that supports JDBC can be used, but if you have IntelliJ Ultimate edition then there is a tool integrated with your IDE. Just open the database window and add an H2 data source with the above details. You will now be able to browse the tables and row data within them.

Monitoring your node

Like most Java servers, the node exports various useful metrics and management operations via the industry-standard [JMX infrastructure](#). JMX is a standard API for registering so-called *MBeans* ... objects whose properties and methods are intended for server management. It does not require any particular network protocol for export. So this data can be exported from the node in various ways: some monitoring systems provide a “Java Agent”, which is essentially a JVM plugin that finds all the MBeans and sends them out to a statistics collector over the network. For those systems, follow the instructions provided by the vendor.

Warning

As of Corda M11, Java serialisation in the Corda node has been restricted, meaning MBeans access via the JMX port will no longer work. Please use java agents instead, you can find details on how to use Jolokia JVM agent [here](#).

[Jolokia](#) allows you to access the raw data and operations without connecting to the JMX port directly. The nodes export the data over HTTP on the `/jolokia` HTTP endpoint, Jolokia defines the JSON and REST formats for accessing MBeans, and provides client libraries to work with that protocol as well.

Here are a few ways to build dashboards and extract monitoring data for a node:

- [hawtio](#) is a web based console that connects directly to JVM's that have been instrumented with a jolokia agent. This tool provides a nice JMX dashboard very similar to the traditional JVisualVM / JConsole MBbeans original.
- [JMX2Graphite](#) is a tool that can be pointed to `/monitoring/json` and will scrape the statistics found there, then insert them into the Graphite monitoring tool on a regular basis. It runs in Docker and can be started with a single command.
- [JMXTans](#) is another tool for Graphite, this time, it's got its own agent (JVM plugin) which reads a custom config file and exports only the named data. It's more configurable than JMX2Graphite and doesn't require a separate process, as the JVM will write directly to Graphite.
- Cloud metrics services like New Relic also understand JMX, typically, by providing their own agent that uploads the data to their service on a regular schedule.
- [Telegraf](#) is a tool to collect, process, aggregate, and write metrics. It can bridge any data input to any output using their plugin system, for example, Telegraf can be configured to collect data from Jolokia and write to DataDog web api.

The Node configuration parameter `exportJMXTTo` should be set to `http` to ensure a Jolokia agent is instrumented with the JVM run-time.

The following JMX statistics are exported:

- Corda specific metrics: flow information (total started, finished, in-flight; flow duration by flow type), attachments (count)

- Apache Artemis metrics: queue information for P2P and RPC services
- JVM statistics: classloading, garbage collection, memory, runtime, threading, operating system
- Hibernate statistics (only when node is started-up in `devMode` due to expensive run-time costs)

When starting Corda nodes using Cordformation runner (see [Running nodes locally](#)), you should see a startup message similar to the following: **Jolokia: Agent started with URL `http://127.0.0.1:7005/jolokia/`**

When starting Corda nodes using the *DriverDSL*, you should see a startup message in the logs similar to the following: **Starting out-of-process Node USA Bank Corp, debug port is not enabled, jolokia monitoring port is 7005 {}**

Several Jolokia policy based security configuration files ([jolokia-access.xml](#)) are available for dev, test, and prod environments under [/config/<env>](#).

The following diagram illustrates Corda flow metrics visualized using [hawtio](#):

The screenshot shows the Hawtio JMX interface with the following details:

- Toolbar:** Container, Dashboard, Diagnostics, JMX, Runtime, Threads.
- Address Bar:** localhost:8080/hawtio/jmx/attributes?con=local&nid=root-net.corda-FlowDuration-Success.net.corda.flows.CashIssueAndPaymentFlow
- Left Sidebar (Tree View):**
 - co.paralleluniverse
 - Fibers
 - ForkJoinPool
 - MonitoringServices
 - com.sun.management
 - java.lang
 - java.nio
 - java.util.logging
 - JMImplementation
 - jmx4perl
 - jolokia
 - net.corda
 - Attachments
 - FlowDuration
 - Failure.net.corda.flows.CashIssueAndPaymentFlow
 - Failure.net.corda.flows.CashPaymentFlow
 - Success.net.corda.confidential.SwapIdentitiesHandler
 - Success.net.corda.flows.CashExitFlow
 - Success.net.corda.flows.CashIssueAndPaymentFlow
 - Success.net.corda.flows.CashPaymentFlow
 - Success.net.corda.node.services.FinalityHandler
 - Flows
 - Checkpointing Rate
 - Finished
 - InFlight
 - Started
 - org.apache.activemq.artemis
 - org.apache.logging.log4j2
- Right Panel (Table View):**

Property	Value
50th percentile	246.35247299999997
75th percentile	362.42470099999997
95th percentile	751.67354
98th percentile	877.629994
99th percentile	2009.5112259999999
999th percentile	2009.5112259999999
Count	99
Duration unit	milliseconds
Fifteen minute rate	0.8035079691286605
Five minute rate	0.5536463660205365
Max	2009.5112259999999
Mean	293.12339201078424
Mean rate	0.32339542967230134
Min	75.603246
Object Name	net.corda:type=FlowDuration,name=Success.net.corda.flows.CashIssueAndPaymentFlow

Memory usage and tuning

All garbage collected programs can run faster if you give them more memory, as they need to collect less frequently. As a default JVM will happily consume all the memory on your system if you let it, Corda is configured with a relatively small

200mb Java heap by default. When other overheads are added, this yields a total memory usage of about 500mb for a node (the overheads come from things like compiled code, metadata, off-heap buffers, thread stacks, etc).

If you want to make your node go faster and profiling suggests excessive GC overhead is the cause, or if your node is running out of memory, you can give it more by running the node like this:

```
java -Xmx1024m -jar corda.jar
```

The example command above would give a 1 gigabyte Java heap.

Note

Unfortunately the JVM does not let you limit the total memory usage of Java program, just the heap size.

[Next](#) [Previous](#)

- [Out-of-process verification](#)
 - [View page source](#)
-

Out-of-process verification

A Corda node does transaction verification through `ServiceHub.transactionVerifierService`. This is by default an `InMemoryTransactionVerifierService` which just verifies transactions in-process.

Corda may be configured to use out of process verification. Any number of verifiers may be started connecting to a node through the node's exposed artemis SSL port. The messaging layer takes care of load balancing.

Note

We plan to introduce kernel level sandboxing around the out of process verifiers as an additional line of defence in case of inner sandbox escapes.

To configure a node to use out of process verification specify the `verifierType` option in your node.conf:

```
myLegalName : "O=Bank A,L=London,C=GB"  
p2pAddress : "my-corda-node:10002"  
webAddress : "localhost:10003"  
verifierType: "OutOfProcess"
```

You can build a verifier jar using `./gradlew verifier:standaloneJar`.

And run it with `java -jar verifier/build/libs/corda-verifier.jar <PATH_TO_VERIFIER_BASE_DIR>`.

`PATH_TO_VERIFIER_BASE_DIR` should contain a `certificates` folder akin to the one in a node directory, and a `verifier.conf` containing the following:

```
nodeHostAndPort: "my-corda-node:10002"  
keyStorePassword : "cordacadevpass"  
trustStorePassword : "trustpass"
```

[Next](#) [Previous](#)

Corda networks

- [Corda networks](#)
- [Network permissioning](#)
- [Network Map](#)
- [Versioning](#)

[Next](#) [Previous](#)

- Corda networks
- [View page source](#)

Corda networks

A Corda network consists of a number of machines running nodes. These nodes communicate using persistent protocols in order to create and validate transactions.

There are three broader categories of functionality one such node may have. These pieces of functionality are provided as services, and one node may run several of them.

- Notary: Nodes running a notary service witness state spends and have the final say in whether a transaction is a double-spend or not
- Oracle: Network services that link the ledger to the outside world by providing facts that affect the validity of transactions
- Regular node: All nodes have a vault and may start protocols communicating with other nodes, notaries and oracles and evolve their private ledger

Bootstrap your own test network

Certificates

Every node in a given Corda network must have an identity certificate signed by the network's root CA. See [Network permissioning](#) for more information.

Configuration

A node can be configured by adding/editing `node.conf` in the node's directory. For details see [Node configuration](#).

An example configuration:

```
myLegalName : "O=Bank A,L=London,C=GB"
keyStorePassword : "cordacadevpass"
trustStorePassword : "trustpass"
dataSourceProperties : {
    dataSourceClassName : org.h2.jdbcx.JdbcDataSource
    "dataSource.url" : "jdbc:h2:file:${baseDirectory}/persistence"
    "dataSource.user" : sa
    "dataSource.password" : ""
}
p2pAddress : "my-corda-node:10002"
rpcSettings = {
    useSsl = false
    standAloneBroker = false
    address : "my-corda-node:10003"
    adminAddress : "my-corda-node:10004"
}
webAddress : "localhost:10004"
rpcUsers : [
    { username=user1, password=letmein, permissions=[StartFlow.net.corda.protocols.CashProtocol ] }
]
devMode : true
```

The most important fields regarding network configuration are:

- `p2pAddress`: This specifies a host and port to which Artemis will bind for messaging with other nodes. Note that the address bound will **NOT** be `my-corda-node`, but rather `::` (all addresses on all network interfaces). The

hostname specified is the hostname *that must be externally resolvable by other nodes in the network*. In the above configuration this is the resolvable name of a machine in a VPN.

- `rpcAddress`: The address to which Artemis will bind for RPC calls.
- `webAddress`: The address the webserver should bind. Note that the port must be distinct from that of `p2pAddress` and `rpcAddress` if they are on the same machine.

Starting the nodes

You will first need to create the local network by bootstrapping it with the bootstrapper. Details of how to do that can be found in [Network Bootstrapper](#).

Once that's done you may now start the nodes in any order. You should see a banner, some log lines and eventually `Node started up and registered`, indicating that the node is fully started.

In terms of process management there is no prescribed method. You may start the jars by hand or perhaps use systemd and friends.

Logging

Only a handful of important lines are printed to the console. For details/diagnosing problems check the logs.

Logging is standard [log4j2](#) and may be configured accordingly. Logs are by default redirected to files in `NODE_DIRECTORY/logs/`.

Connecting to the nodes

Once a node has started up successfully you may connect to it as a client to initiate protocols/query state etc. Depending on your network setup you may need to tunnel to do this remotely.

See the [Using the client RPC API](#) on how to establish an RPC link.

Sidenote: A client is always associated with a single node with a single identity, which only sees their part of the ledger.

- Network permissioning
 - [View page source](#)
-

Network permissioning

Contents

- [Network permissioning](#)
 - [Certificate hierarchy](#)
 - [Keypair and certificate formats](#)
 - [Certificate role extension](#)
 - [Creating the root and doorman CAs](#)
 - [Creating the root network CA's keystore and truststore](#)
 - [Creating the doorman CA's keystore](#)
 - [Creating the node CA keystores and TLS keystores](#)
 - [Creating the node CA keystores](#)
 - [Creating the node TLS keystores](#)
 - [Installing the certificates on the nodes](#)
 - [Connecting to a compatibility zone](#)

Corda networks are *permissioned*. To connect to a network, a node needs three keystores in its `<workspace>/certificates/` folder:

- `truststore.jks`, which stores trusted public keys and certificates (in our case, those of the network root CA)
- `nodekeystore.jks`, which stores the node's identity keypairs and certificates
- `sslkeystore.jks`, which stores the node's TLS keypairs and certificates

Production deployments require a secure certificate authority. Most production deployments will use an existing certificate authority or construct one using software that will be made available in the coming months. Until then, the documentation below can be used to create your own certificate authority.

Note

If you are looking for information on how to connect to the existing compatibility zone go to the section: [Connecting to a compatibility zone](#)

Certificate hierarchy

A Corda network has four types of certificate authorities (CAs):

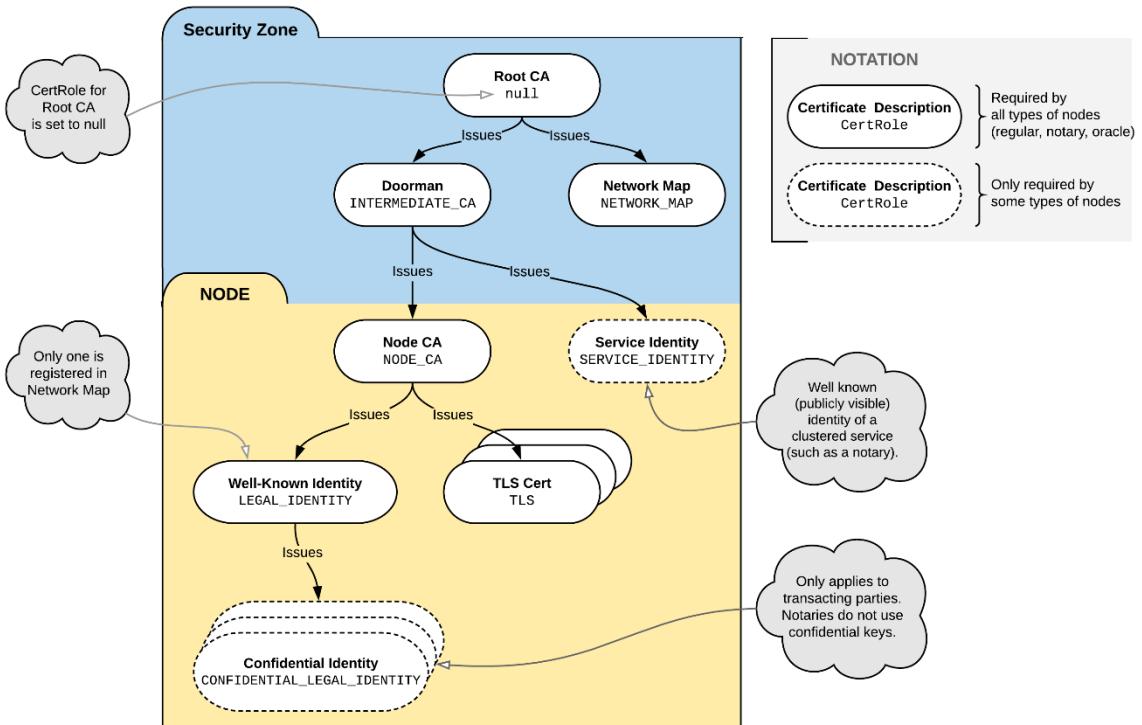
- The **root network CA**
- The **doorman CA**
 - The doorman CA is used instead of the root network CA for day-to-day key signing to reduce the risk of the root network CA's private key being compromised
- The **node CAs**
 - Each node serves as its own CA in issuing the child certificates that it uses to sign its identity keys and TLS certificates
- The **legal identity CAs**
 - Node's well-known legal identity, apart from signing transactions, can also issue certificates for confidential legal identities

The following constraints are also imposed:

- Doorman certificates are issued by a network root which certificate doesn't contain the extension
- Well-known service identity certificates are issued by an entity with a Doorman certificate
- Node CA certificates are issued by an entity with a Doorman certificate
- Well known legal identity/TLS certificates are issued by a certificate marked as node CA
- Confidential legal identity certificates are issued by a certificate marked as well known legal identity
- Party certificates are marked as either a well known identity or a confidential identity
- The structure of certificates above Doorman/Network map is intentionally left untouched, as they are not relevant to the identity service and therefore there is no advantage in enforcing a specific structure on those certificates. The certificate hierarchy consistency checks are required because nodes can issue their own certificates and can set their own role flags on certificates, and it's important to verify that these are set consistently with the certificate hierarchy design. As a side-effect this also acts as a secondary depth restriction on issued certificates

All the certificates must be issued with the custom role extension (see below).

We can visualise the permissioning structure as follows:



Keypair and certificate formats

You can use any standard key tools or Corda's [X509Utilities](#) (which uses Bouncy Castle) to create the required public/private keypairs and certificates. The keypairs and certificates should obey the following restrictions:

- The certificates must follow the [X.509 standard](#)
 - We recommend X.509 v3 for forward compatibility
- The TLS certificates must follow the [TLS v1.2 standard](#)
- The root network CA, doorman CA and node CA keys, as well as the node TLS keys, must follow one of the following schemes:
 - ECDSA using the NIST P-256 curve (secp256r1)
 - RSA with 3072-bit key size

Certificate role extension

Corda certificates have a custom X.509 v3 extension that specifies the role the certificate relates to. This extension has the OID [1.3.6.1.4.1.50530.1.1](#) and is non-critical, so implementations outside of Corda nodes can safely ignore it. The

extension contains a single ASN.1 integer identifying the identity type the certificate is for:

1. Doorman
2. Network map
3. Service identity (currently only used as the shared identity in distributed notaries)
4. Node certificate authority (from which the TLS and well-known identity certificates are issued)
5. Transport layer security
6. Well-known legal identity
7. Confidential legal identity

In a typical installation, node administrators needn't be aware of these. However, when node certificates are managed by external tools (such as an existing PKI solution deployed within an organisation), it is important to understand these constraints.

Certificate path validation is extended so that a certificate must contain the extension if the extension was present in the certificate of the issuer.

Creating the root and doorman CAs

Creating the root network CA's keystore and truststore

1. Create a new keypair
 - This will be used as the root network CA's keypair
2. Create a self-signed certificate for the keypair. The basic constraints extension must be set to `true`
 - This will be used as the root network CA's certificate
3. Create a new keystore and store the root network CA's keypair and certificate in it for later use
 - This keystore will be used by the root network CA to sign the doorman CA's certificate
4. Create a new Java keystore named `truststore.jks` and store the root network CA's certificate in it using the alias `cordarootca`
 - This keystore must then be provisioned to the individual nodes later so they can store it in their `certificates` folder

Warning

The root network CA's private key should be protected and kept safe.

Creating the doorman CA's keystore

1. Create a new keypair
 - This will be used as the doorman CA's keypair
2. Obtain a certificate for the keypair signed with the root network CA key. The basic constraints extension must be set to `true`
 - This will be used as the doorman CA's certificate
3. Create a new keystore and store the doorman CA's keypair and certificate chain (i.e. the doorman CA certificate *and* the root network CA certificate) in it for later use
 - This keystore will be used by the doorman CA to sign the nodes' identity certificates

Creating the node CA keystores and TLS keystores

Creating the node CA keystores

1. For each node, create a new keypair
2. Obtain a certificate for the keypair signed with the doorman CA key. The basic constraints extension must be set to `true`
3. Create a new Java keystore named `nodekeystore.jks` and store the keypair in it using the alias `cordaclientca`
 - The node will store this keystore locally to sign its identity keys and anonymous keys

Creating the node TLS keystores

1. For each node, create a new keypair
2. Create a certificate for the keypair signed with the node CA key. The basic constraints extension must be set to `false`
3. Create a new Java keystore named `sslkeystore.jks` and store the key and certificates in it using the alias `cordaclienttls`
 - The node will store this keystore locally to sign its TLS certificates

Installing the certificates on the nodes

For each node, copy the following files to the node's certificate directory (`<workspace>/certificates/`):

1. The node's `nodekeystore.jks` keystore
2. The node's `sslkeystore.jks` keystore
3. The root network CA's `truststore.jks` keystore

Connecting to a compatibility zone

To connect to a compatibility zone you need to register with their certificate signing authority (doorman) by submitting a certificate signing request (CSR) to obtain a valid identity for the zone.

Before you can register, you must first have received the trust store file containing the root certificate from the zone operator. Then run the following command:

```
java -jar corda.jar --initial-registration --network-root-truststore-
password <trust store password>
```

By default it will expect the trust store file to be in the location `certificates/network-root-truststore.jks`. This can be overridden with the additional `--network-root-truststore` flag.

The certificate signing request will be created based on node information obtained from the node configuration. The following information from the node configuration file is needed to generate the request.

- **myLegalName** Your company's legal name as an X.500 string. X.500 allows differentiation between entities with the same name as the legal name needs to be unique on the network. If another node has already been permissioned with this name then the permissioning server will automatically reject the request. The request will also be rejected if it violates legal name rules, see [Node naming](#) for more information.
- **emailAddress** e.g. "admin@company.com"
- **devMode** must be set to false
- **networkServices or compatibilityZoneURL** The Corda compatibility zone services must be configured. This must be either:

- **compatibilityZoneURL** The Corda compatibility zone network management service root URL.
- **networkServices** Replaces the `compatibilityZoneURL` when the Doorman and Network Map services are configured to operate on different URL endpoints. The `doorman` entry is used for registration.

A new pair of private and public keys generated by the Corda node will be used to create the request.

The utility will submit the request to the doorman server and poll for a result periodically to retrieve the certificates. Once the request has been approved and the certificates downloaded from the server, the node will create the keystore and trust store using the certificates and the generated private key.

Note

You can exit the utility at any time if the approval process is taking longer than expected. The request process will resume on restart.

This process only is needed when the node connects to the network for the first time, or when the certificate expires.

[Next](#) [Previous](#)

- Network Map
 - [View page source](#)
-

Network Map

The network map is a collection of signed `NodeInfo` objects. Each `NodeInfo` is signed by the node it represents and thus cannot be tampered with. It forms the set of reachable nodes in a compatibility zone. A node can receive these objects from two sources:

1. A network map server that speaks a simple HTTP based protocol.
2. The `additional-node-infos` directory within the node's directory.

The network map server also distributes the parameters file that define values for various settings that all nodes need to agree on to remain in sync.

Note

In Corda 3 no implementation of the HTTP network map server is provided. This is because the details of how a compatibility zone manages its membership (the databases, ticketing workflows, HSM hardware etc) is expected to vary between operators, so we provide a simple REST based protocol for uploading/downloading NodeInfos and managing network parameters. A future version of Corda may provide a simple “stub” implementation for running test zones. In Corda 3 the right way to run a test network is through distribution of the relevant files via your own mechanisms. We provide a tool to automate the bulk of this task (see below).

HTTP network map protocol

If the node is configured with the `compatibilityZoneURL` config then it first uploads its own signed `NodeInfo` to the server (and each time it changes on startup) and then proceeds to download the entire network map. The network map consists of a list of `NodeInfo` hashes. The node periodically polls for the network map (based on the HTTP cache expiry header) and any new entries are downloaded and cached. Entries which no longer exist are deleted from the node’s cache.

The set of REST end-points for the network map service are as follows.

Request method	Path	Description
POST	/network-map/publish	For the node to upload its signed <code>NodeInfo</code> object to the network map.
POST	/network-map/ack-parameters	For the node operator to acknowledge network map that new parameters were accepted for future update.
GET	/network-map	Retrieve the current signed network map object. The entire object is signed with the network map certificate which is also attached.
GET	/network-map/node-info/{hash}	Retrieve a signed <code>NodeInfo</code> as specified in the network map object.
GET	/network-map/network-parameters/{hash}	Retrieve the signed network parameters (see below). The entire object is signed with the network map certificate which is also attached.

HTTP is used for the network map service instead of Corda's own AMQP based peer to peer messaging protocol to enable the server to be placed behind caching content delivery networks like Cloudflare, Akamai, Amazon Cloudfront and so on. By using industrial HTTP cache networks the map server can be shielded from DoS attacks more effectively. Additionally, for the case of distributing small files that rarely change, HTTP is a well understood and optimised protocol. Corda's own protocol is designed for complex multi-way conversations between authenticated identities using signed binary messages separated into parallel and nested flows, which isn't necessary for network map distribution.

The `additional-node-infos` directory

Alongside the HTTP network map service, or as a replacement if the node isn't connected to one, the node polls the contents of the `additional-node-infos` directory located in its base directory. Each file is expected to be the same signed `NodeInfo` object that the network map service vends. These are automatically added to the node's cache and can be used to supplement or replace the HTTP network map. If the same node is advertised through both mechanisms then the latest one is taken.

On startup the node generates its own signed node info file, filename of the format `nodeInfo-${hash}`. It can also be generated using the `--just-generate-node-info` command line flag without starting the node. To create a simple network without the HTTP network map service simply place this file in the `additional-node-infos` directory of every node that's part of this network. For example, a simple way to do this is to use rsync.

Usually, test networks have a structure that is known ahead of time. For the creation of such networks we provide a `network-bootstrapper` tool. This tool pre-generates node configuration directories if given the IP addresses/domain names of each machine in the network. The generated node directories contain the `NodeInfos` for every other node on the network, along with the network parameters file and identity certificates. Generated nodes do not need to all be online at once - an offline node that isn't being interacted with doesn't impact the network in any way. So a test cluster generated like this can be sized for the maximum size you may need, and then scaled up and down as necessary.

More information can be found in [setting-up-a-corda-network](#).

Network parameters

Network parameters are a set of values that every node participating in the zone needs to agree on and use to correctly interoperate with each other. They can be thought of as an encapsulation of all aspects of a Corda deployment on which reasonable people may disagree. Whilst other blockchain/DLT systems typically require a source code fork to alter various constants (like the total number of coins in a cryptocurrency, port numbers to use etc), in Corda we have refactored these sorts of decisions out into a separate file and allow “zone operators” to make decisions about them. The operator signs a data structure that contains the values and they are distributed along with the network map. Tools are provided to gain user opt-in consent to a new version of the parameters and ensure everyone switches to them at the same time.

If the node is using the HTTP network map service then on first startup it will download the signed network parameters, cache it in a `network-parameters` file and apply them on the node.

Warning

If the `network-parameters` file is changed and no longer matches what the network map service is advertising then the node will automatically shutdown. Resolution to this is to delete the incorrect file and restart the node so that the parameters can be downloaded again.

If the node isn't using a HTTP network map service then it's expected the signed file is provided by some other means. For such a scenario there is the network bootstrapper tool which in addition to generating the network parameters file also distributes the node info files to the node directories.

The current set of network parameters:

minimumPlatformVersion:

The minimum platform version that the nodes must be running. Any node which is below this will not start.

notaries: List of identity and validation type (either validating or non-validating) of the notaries which are permitted in the compatibility zone.

maxMessageSize: (This is currently ignored. However, it will be wired up in a future release.)

maxTransactionSize:

Maximum allowed size in bytes of a transaction. This is the size of the transaction object and its attachments.

modifiedTime: The time when the network parameters were last modified by the compatibility zone operator.

epoch: Version number of the network parameters. Starting from 1, this will always increment whenever any of the parameters change.

whitelistedContractImplementations:

List of whitelisted versions of contract code. For each contract class there is a list of hashes of the approved CorDapp jar versions containing that contract. Read more about *Zone constraints* here [API: Contract Constraints](#)

eventHorizon: Time after which nodes are considered to be unresponsive and removed from network map. Nodes republish their `NodeInfo` on a regular interval. Network map treats that as a heartbeat from the node.

More parameters will be added in future releases to regulate things like allowed port numbers, how long a node can be offline before it is evicted from the zone, whether or not IPv6 connectivity is required for zone members, required cryptographic algorithms and rollout schedules (e.g. for moving to post quantum cryptography), parameters related to SGX and so on.

Network parameters update process

In case of the need to change network parameters Corda zone operator will start the update process. There are many reasons that may lead to this decision: adding a notary, setting new fields that were added to enable smooth network interoperability, or a change of the existing compatibility constants is required, for example.

Note

A future release may support the notion of phased rollout of network parameter changes.

To synchronize all nodes in the compatibility zone to use the new set of the network parameters two RPC methods are provided. The process requires human interaction and approval of the change, so node operators can review the differences before agreeing to them.

When the update is about to happen the network map service starts to advertise the additional information with the usual network map data. It includes new network parameters hash, description of the change and the update deadline. Nodes query the network map server for the new set of parameters.

The fact a new set of parameters is being advertised shows up in the node logs with the message “Downloaded new network parameters”, and programs connected via RPC can receive `ParametersUpdateInfo` by using the `CordaRPCOps.networkParametersFeed` method. Typically a zone operator would also email node operators to let them know about the details of the impending change, along with the justification, how to object, deadlines and so on.

```
/**  
 * Data class containing information about the scheduled network parameters update.  
 * The info is emitted every time node  
 * receives network map with [ParametersUpdate] which wasn't seen before. For more  
 * information see: [CordaRPCOps.networkParametersFeed] and  
 * [CordaRPCOps.acceptNewNetworkParameters].  
 * @property hash new [NetworkParameters] hash  
 * @property parameters new [NetworkParameters] data structure  
 * @property description description of the update  
 * @property updateDeadline deadline for accepting this update using  
 * [CordaRPCOps.acceptNewNetworkParameters]  
 */  
@CordaSerializable  
data class ParametersUpdateInfo(  
    val hash: SecureHash,  
    val parameters: NetworkParameters,  
    val description: String,  
    val updateDeadline: Instant  
)
```

The node administrator can review the change and decide if they are going to accept it. The approval should be done before the update Deadline. Nodes that don't approve before the deadline will likely be removed from the network map by the zone operator, but that is a decision that is left to the operator's discretion. For example the operator might also choose to change the deadline instead.

If the network operator starts advertising a different set of new parameters then that new set overrides the previous set. Only the latest update can be accepted.

To send back parameters approval to the zone operator, the RPC method `fun acceptNewNetworkParameters(parametersHash: SecureHash)` has to be called with `parametersHash` from the update. Note that approval cannot be undone. You can do this via the Corda shell (see [Shell](#)):

```
run acceptNewNetworkParameters parametersHash: "ba19fc1b9e9c1c7cbea712efda5f78b53ae4e  
5d123c89d02c9da44ec50e9c17d"
```

If the administrator does not accept the update then next time the node polls network map after the deadline, the advertised network parameters will be the updated ones. The previous set of parameters will no longer be valid. At this point the node will automatically shutdown and will require the node operator to bring it back again.

[Next](#) [Previous](#)

- [Versioning](#)
 - [View page source](#)
-

Versioning

As the Corda platform evolves and new features are added it becomes important to have a versioning system which allows its users to easily compare versions and know what feature are available to them. Each Corda release uses the standard semantic versioning scheme of `major.minor.patch`. This is useful when making releases in the public domain but is not friendly for a developer working on the platform. It first has to be parsed and then they have three separate segments on which to determine API differences. The release version is still useful and every MQ message the node sends attaches it to the `release-version` header property for debugging purposes.

It is much easier to use a single incrementing integer value to represent the API version of the Corda platform, which is called the Platform Version. It is similar to Android's [API Level](#). It starts at 1 and will increment by exactly 1 for each release which changes any of the publicly exposed APIs in the entire platform. This includes public APIs on the node itself, the RPC system, messaging, serialisation, etc. API backwards compatibility will always be maintained, with the use of deprecation to migrate away from old APIs. In rare situations APIs may have to be removed, for example due to security issues. There is no relationship between the Platform Version and the release version - a change in the major, minor or patch values may or may not increase the Platform Version.

The Platform Version is part of the node's `NodeInfo` object, which is available from the `ServiceHub`. This enables a CorDapp to find out which version it's running on and determine whether a desired feature is available. When a node registers with the Network Map Service it will use the node's Platform Version to enforce a minimum version requirement for the network.

Note

A future release may introduce the concept of a target platform version, which would be similar to Android's `targetSdkVersion`, and would provide a means of maintaining behavioural compatibility for the cases where the platform's behaviour has changed.

[Next](#) [Previous](#)

- [Tutorials](#)
 - [View page source](#)
-

Tutorials

This section is split into two parts.

The Hello, World tutorials should be followed in sequence and show how to extend the Java or Kotlin CorDapp Template into a full CorDapp.

- [Hello, World!](#)
- [Hello, World! Pt.2 - Contract constraints](#)

The remaining tutorials cover individual platform features in isolation. They don't depend on the code from the Hello, World tutorials, and can be read in any order.

- [Writing a contract](#)
- [Writing a contract test](#)
- [Upgrading contracts](#)
- [Integration testing](#)
- [Using the client RPC API](#)
- [Building transactions](#)
- [Writing flows](#)
- [Writing flow tests](#)

- Running a notary service
- Writing oracle services
- Writing a custom notary service (experimental)
- Transaction tear-offs
- Using attachments
- Event scheduling
- Observer nodes

[Next](#) [Previous](#)

- Hello, World!
 - [View page source](#)
-

Hello, World!

- [The CorDapp Template](#)
- [Writing the state](#)
- [Writing the flow](#)
- [Running our CorDapp](#)

By this point, your dev environment should be set up, you've run your first CorDapp, and you're familiar with Corda's key concepts. What comes next?

If you're a developer, the next step is to write your own CorDapp. CorDapps are plugins that are installed on one or more Corda nodes, and give the nodes' owners the ability to make their node conduct some new process - anything from issuing a debt instrument to making a restaurant booking.

Our use-case

Our CorDapp will model IOUs on-ledger. An IOU – short for “I O(we) (yo)U” – records the fact that one person owes another person a given amount of money. Clearly this is sensitive information that we'd only want to communicate on a need-to-know basis between the lender and the borrower. Fortunately, this is one of the areas where Corda excels. Corda makes it easy to allow a small set of parties to agree on a shared fact without needing to share this fact with everyone else on the network, as is the norm in blockchain platforms.

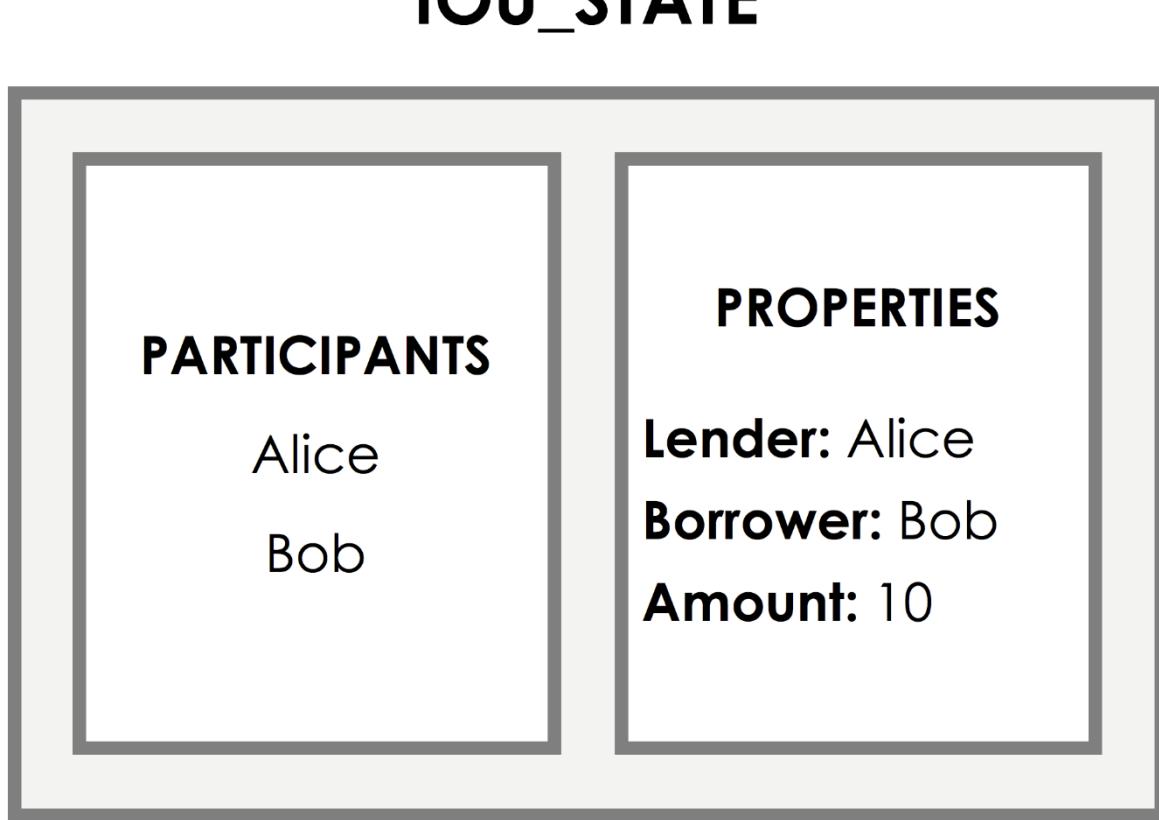
To serve any useful function, our CorDapp will need at least two things:

- **States**, the shared facts that Corda nodes reach consensus over and are then stored on the ledger
- **Flows**, which encapsulate the procedure for carrying out a specific ledger update

Our IOU CorDapp is no exception. It will define both a state and a flow:

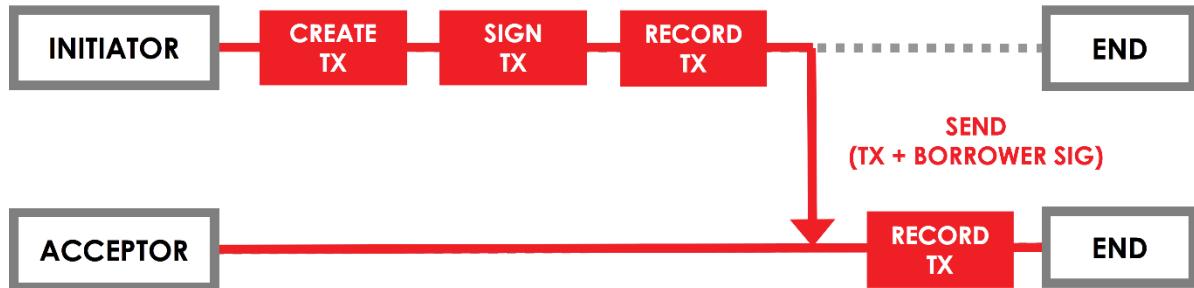
The IOUState

Our state will be the `IOUState`. It will store the value of the IOU, as well as the identities of the lender and the borrower. We can visualize `IOUState` as follows:



The IOUFlow

Our flow will be the `IOUFlow`. This flow will completely automate the process of issuing a new IOU onto a ledger. It is composed of the following steps:



In traditional distributed ledger systems, where all data is broadcast to every network participant, you don't need to think about data flows – you simply package up your ledger update and send it to everyone else on the network. But in Corda, where privacy is a core focus, flows allow us to carefully control who sees what during the process of agreeing a ledger update.

Progress so far

We've sketched out a simple CorDapp that will allow nodes to confidentially issue new IOUs onto a ledger.

Next, we'll be taking a look at the template project we'll be using as the basis for our CorDapp.

[Next](#) [Previous](#)

- The CorDapp Template
 - [View page source](#)
-

The CorDapp Template

When writing a new CorDapp, you'll generally want to base it on the standard templates:

- The [Java Cordapp Template](#)
- The [Kotlin Cordapp Template](#)

The Cordapp templates provide the required boilerplate for developing a CorDapp, and allow you to quickly deploy your CorDapp onto a local test network of dummy nodes to test its functionality.

CorDapps can be written in both Java and Kotlin, and will be providing the code in both languages in this tutorial.

Note that there's no need to download and install Corda itself. Corda's required libraries will be downloaded automatically from an online Maven repository.

Downloading the template

To download the template, open a terminal window in the directory where you want to download the CorDapp template, and run the following command:

```
git clone https://github.com/corda/cordapp-template-java.git ; cd cordapp-template-java  
*or*  
git clone https://github.com/corda/cordapp-template-kotlin.git ; cd cordapp-template-kotlin
```

Opening the template in IntelliJ

Once the template is download, open it in IntelliJ by following the instructions here: <https://docs.corda.net/tutorial-cordapp.html#opening-the-example-cordapp-in-intellij>.

Template structure

The template has a number of files, but we can ignore most of them. We will only be modifying the following files:

KotlinJava

```
// 1. The state  
cordapp-contracts-states/src/main/kotlin/com/template/StatesAndContracts.kt  
  
// 2. The flow  
cordapp/src/main/kotlin/com/template/Flows.kt
```

Progress so far

We now have a template that we can build upon to define our IOU CorDapp. Let's start by defining the `IOUState`.

[Next](#) [Previous](#)

- Writing the state
- [View page source](#)

Writing the state

In Corda, shared facts on the ledger are represented as states. Our first task will be to define a new state type to represent an IOU.

The ContractState interface

A Corda state is any instance of a class that implements the `ContractState` interface. The `ContractState` interface is defined as follows:

Kotlin

```
interface ContractState {  
    // The list of entities considered to have a stake in this state.  
    val participants: List<AbstractParty>  
}
```

We can see that the `ContractState` interface has a single field, `participants`. `participants` is a list of the entities for which this state is relevant.

Beyond this, our state is free to define any fields, methods, helpers or inner classes it requires to accurately represent a given type of shared fact on the ledger.

Note

The first thing you'll probably notice about the declaration of `ContractState` is that its not written in Java or another common language. The core Corda platform, including the interface declaration above, is entirely written in Kotlin.

Learning some Kotlin will be very useful for understanding how Corda works internally, and usually only takes an experienced Java developer a day or so to pick up. However, learning Kotlin isn't essential. Because Kotlin code compiles to JVM bytecode, CorDapps written in other JVM languages such as Java can interoperate with Corda.

If you do want to dive into Kotlin, there's an official [getting started guide](#), and a series of [Kotlin Koans](#).

Modelling IOUs

How should we define the `IOUState` representing IOUs on the ledger? Beyond implementing the `ContractState` interface, our `IOUState` will also need properties to track the relevant features of the IOU:

- The value of the IOU
- The lender of the IOU
- The borrower of the IOU

There are many more fields you could include, such as the IOU's currency, but let's ignore those for now. Adding them later is often as simple as adding an additional property to your class definition.

Defining `IOUState`

Let's get started by opening `TemplateState.java` (for Java) or `StatesAndContracts.kt` (for Kotlin) and updating `TemplateState` to define an `IOUState`:

KotlinJava

```
// Add these imports:  
import net.corda.core.identity.Party  
  
// Replace TemplateState's definition with:  
class IOUState(val value: Int,  
               val lender: Party,  
               val borrower: Party) : ContractState {  
    override val participants get() = listOf(lender, borrower)  
}
```

If you're following along in Java, you'll also need to rename `TemplateState.java` to `IOUState.java`.

To define `IOUState`, we've made the following changes:

- We've renamed the `TemplateState` class to `IOUState`
- We've added properties for `value`, `lender` and `borrower`, along with the required getters and setters in Java:
 - `value` is of type `int` (in Java)/`Int` (in Kotlin)
 - `lender` and `borrower` are of type `Party`
 - `Party` is a built-in Corda type that represents an entity on the network
- We've overridden `participants` to return a list of the `lender` and `borrower`
 - `participants` is a list of all the parties who should be notified of the creation or consumption of this state

The IOUs that we issue onto a ledger will simply be instances of this class.

Progress so far

We've defined an `IOUState` that can be used to represent IOUs as shared facts on a ledger. As we've seen, states in Corda are simply classes that implement the `ContractState` interface. They can have any additional properties and methods you like.

All that's left to do is write the `IOUFlow` that will allow a node to orchestrate the creation of a new `IOUState` on the ledger, while only sharing information on a need-to-know basis.

What about the contract?

If you've read the white paper or Key Concepts section, you'll know that each state has an associated contract that imposes invariants on how the state evolves over time. Including a contract isn't crucial for our first CorDapp, so we'll just use the empty `TemplateContract` and `TemplateContract.Commands.Action` command defined by the template for now. In the next tutorial, we'll implement our own contract and command.

[Next](#) [Previous](#)

- Writing the flow
 - [View page source](#)
-

Writing the flow

A flow encodes a sequence of steps that a node can perform to achieve a specific ledger update. By installing new flows on a node, we allow the node to handle new business processes. The flow we define will allow a node to issue an `IOUState` onto the ledger.

Flow outline

The goal of our flow will be to orchestrate an IOU issuance transaction. Transactions in Corda are the atomic units of change that update the ledger. Each transaction is a proposal to mark zero or more existing states as historic (the inputs), while creating zero or more new states (the outputs).

The process of creating and applying this transaction to a ledger will be conducted by the IOU's lender, and will require the following steps:

1. Building the transaction proposal for the issuance of a new IOU onto a ledger
2. Signing the transaction proposal
3. Recording the transaction
4. Sending the transaction to the IOU's borrower so that they can record it too

At this stage, we do not require the borrower to approve and sign IOU issuance transactions. We will be able to impose this requirement when we look at contracts in the next tutorial.

Subflows

Tasks like recording a transaction or sending a transaction to a counterparty are very common in Corda. Instead of forcing each developer to reimplement their own logic to handle these tasks, Corda provides a number of library flows to handle these tasks. We call these flows that are invoked in the context of a larger flow to handle a repeatable task *subflows*.

In our case, we can automate steps 3 and 4 of the IOU issuance flow using `FinalityFlow`.

FlowLogic

All flows must subclass `FlowLogic`. You then define the steps taken by the flow by overriding `FlowLogic.call`.

Let's define our `IOUFlow` in either `Initiator.java` or `Flows.kt`. Delete the two existing flows in the template (`Initiator` and `Responder`), and replace them with the following:

KotlinJava

```
// Add these imports:  
import net.corda.core.contracts.Command  
import net.corda.core.identity.Party  
import net.corda.core.transactions.TransactionBuilder  
import net.corda.core.utilities.ProgressTracker  
  
// Replace TemplateFlow's definition with:  
@InitiatingFlow  
@StartableByRPC  
class IOUFlow(val iouValue: Int,  
             val otherParty: Party) : FlowLogic<Unit>() {  
  
    /** The progress tracker provides checkpoints indicating the progress of the flow  
     * to observers. */  
    override val progressTracker = ProgressTracker()  
  
    /** The flow logic is encapsulated within the call() method. */  
    @Suspendable  
    override fun call() {  
        // We retrieve the notary identity from the network map.  
        val notary = serviceHub.networkMapCache.notaryIdentities[0]  
  
        // We create the transaction components.  
        val outputState = IOUState(iouValue, ourIdentity, otherParty)  
        val cmd = Command(TemplateContract.Commands.Action(), ourIdentity.owningKey)  
  
        // We create a transaction builder and add the components.  
        val txBuilder = TransactionBuilder(notary = notary)  
            .addOutputState(outputState, TemplateContract.ID)  
            .addCommand(cmd)  
  
        // We sign the transaction.  
        val signedTx = serviceHub.signInitialTransaction(txBuilder)  
  
        // We finalise the transaction.  
        subFlow(FinalityFlow(signedTx))  
    }  
}
```

If you're following along in Java, you'll also need to rename `Initiator.java` to `IOUFlow.java`. Let's walk through this code step-by-step.

We've defined our own `FlowLogic` subclass that overrides `FlowLogic.call`. `FlowLogic.call` has a return type that must match the type parameter passed to `FlowLogic` - this is type returned by running the flow.

`FlowLogic` subclasses can optionally have constructor parameters, which can be used as arguments to `FlowLogic.call`. In our case, we have two:

- `iouValue`, which is the value of the IOU being issued
- `otherParty`, the IOU's borrower (the node running the flow is the lender)

`FlowLogic.call` is annotated `@Suspendable` - this allows the flow to be check-pointed and serialised to disk when it encounters a long-running operation, allowing your

node to move on to running other flows. Forgetting this annotation out will lead to some very weird error messages!

There are also a few more annotations, on the `FlowLogic` subclass itself:

- `@InitiatingFlow` means that this flow can be started directly by the node
- `@StartableByRPC` allows the node owner to start this flow via an RPC call

Let's walk through the steps of `FlowLogic.call` itself. This is where we actually describe the procedure for issuing the `IOUState` onto a ledger.

Choosing a notary

Every transaction requires a notary to prevent double-spends and serve as a timestamping authority. The first thing we do in our flow is retrieve the a notary from the node's `ServiceHub`. `ServiceHub.networkMapCache` provides information about the other nodes on the network and the services that they offer.

Note

Whenever we need information within a flow - whether it's about our own node's identity, the node's local storage, or the rest of the network - we generally obtain it via the node's `ServiceHub`.

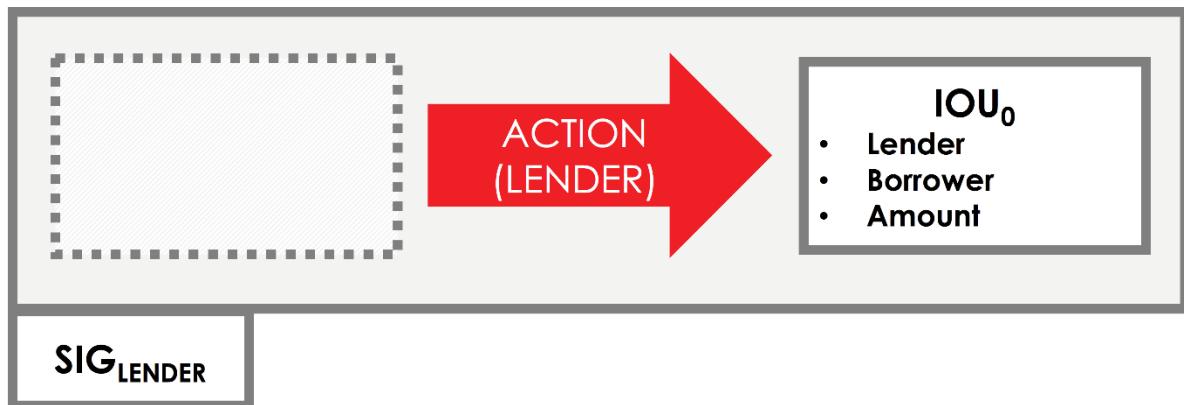
Building the transaction

We'll build our transaction proposal in two steps:

- Creating the transaction's components
- Adding these components to a transaction builder

Transaction items

Our transaction will have the following structure:



- The output `IOUState` on the right represents the state we will be adding to the ledger. As you can see, there are no inputs - we are not consuming any existing ledger states in the creation of our IOU
- An `Action` command listing the IOU's lender as a signer

We've already talked about the `IOUState`, but we haven't looked at commands yet. Commands serve two functions:

- They indicate the intent of a transaction - issuance, transfer, redemption, revocation. This will be crucial when we discuss contracts in the next tutorial
- They allow us to define the required signers for the transaction. For example, IOU creation might require signatures from the lender only, whereas the transfer of an IOU might require signatures from both the IOU's borrower and lender

Each `Command` contains a command type plus a list of public keys. For now, we use the pre-defined `TemplateContract.Action` as our command type, and we list the lender as the only public key. This means that for the transaction to be valid, the lender is required to sign the transaction.

Creating a transaction builder

To actually build the proposed transaction, we need a `TransactionBuilder`. This is a mutable transaction class to which we can add inputs, outputs, commands, and any other items the transaction needs. We create a `TransactionBuilder` that uses the notary we retrieved earlier.

Once we have the `TransactionBuilder`, we add our components:

- The command is added directly using `TransactionBuilder.addCommand`
- The output `IOUState` is added using `TransactionBuilder.addOutputState`. As well as the output state itself, this method takes a reference to the contract that will govern the evolution of the state over time. Here, we are passing in a reference to the `TemplateContract`, which imposes no constraints. We will define a contract imposing real constraints in the next tutorial

Signing the transaction

Now that we have a valid transaction proposal, we need to sign it. Once the transaction is signed, no-one will be able to modify the transaction without invalidating this signature. This effectively makes the transaction immutable.

We sign the transaction using `ServiceHub.signInitialTransaction`, which returns a `SignedTransaction`. A `SignedTransaction` is an object that pairs a transaction with a list of signatures over that transaction.

Finalising the transaction

We now have a valid signed transaction. All that's left to do is to have it recorded by all the relevant parties. By doing so, it will become a permanent part of the ledger. As discussed, we'll handle this process automatically using a built-in flow called `FinalityFlow`. `FinalityFlow` completely automates the process of:

- Notarising the transaction if required (i.e. if the transaction contains inputs and/or a time-window)
- Recording it in our vault
- Sending it to the other participants (i.e. the lender) for them to record as well

Progress so far

Our flow, and our CorDapp, are now ready! We have now defined a flow that we can start on our node to completely automate the process of issuing an IOU onto the ledger. All that's left is to spin up some nodes and test our CorDapp.

[Next](#) [Previous](#)

- Running our CorDapp
 - [View page source](#)
-

Running our CorDapp

Now that we've written a CorDapp, it's time to test it by running it on some real Corda nodes.

Deploying our CorDapp

Let's take a look at the nodes we're going to deploy. Open the project's `build.gradle` file and scroll down to the `task deployNodes` section. This section defines three nodes. There are two standard nodes (`PartyA` and `PartyB`), plus a special network map/notary node that is running the network map service and advertises a validating notary service.

```
task deployNodes(type: net.corda.plugins.Cordform, dependsOn: ['jar']) {
    directory "./build/nodes"
    node {
        name "O=Notary,L=London,C=GB"
        notary = [validating : true]
        p2pPort 10002
        rpcPort 10003
        cordapps = ["net.corda:corda-finance:$corda_release_version"]
    }
    node {
        name "O=PartyA,L=London,C=GB"
        p2pPort 10005
        rpcPort 10006
        webPort 10007
        cordapps = ["net.corda:corda-finance:$corda_release_version"]
        rpcUsers = [[user: "user1", "password": "test", "permissions": ["ALL"]]]
    }
    node {
        name "O=PartyB,L>New York,C=US"
        p2pPort 10008
        rpcPort 10009
        webPort 10010
        sshdPort 10024
        cordapps = ["net.corda:corda-finance:$corda_release_version"]
        rpcUsers = [[user: "user1", "password": "test", "permissions": ["ALL"]]]
    }
}
```

We can run this `deployNodes` task using Gradle. For each node definition, Gradle will:

- Package the project's source files into a CorDapp jar
- Create a new node in `build/nodes` with our CorDapp already installed

We can do that now by running the following commands from the root of the project:

```
// On Windows  
gradlew clean deployNodes
```

```
// On Mac  
.gradlew clean deployNodes
```

Running the nodes

Running `deployNodes` will build the nodes under `build/nodes`. If we navigate to one of these folders, we'll see the three node folders. Each node folder has the following structure:

```
.  
|__ corda.jar // The runnable node  
|__ corda-webserver.jar // The node's webserver (The notary doesn't need a  
web server)  
|__ node.conf // The node's configuration file  
|__ cordapps  
|__ java/kotlin-source-0.1.jar // Our IOU CorDapp
```

Let's start the nodes by running the following commands from the root of the project:

```
// On Windows  
build/nodes/runnodes.bat
```

```
// On Mac  
build/nodes/runnodes
```

This will start a terminal window for each node, and an additional terminal window for each node's webserver - five terminal windows in all. Give each node a moment to start - you'll know it's ready when its terminal windows displays the message, "Welcome to the Corda interactive shell.".



What you can buy for a dollar these days is absolute non-cents! 💰

```
--- Corda Open Source 0.12.1 (da47f1c) ---
 New! Training now available worldwide, see https://corda.net/corda-training/
Logs can be found in : /Users/joeldudley/Desktop/tutorial/cordapp-tutorial
Database connection url is : jdbc:h2:tcp://10.163.199.132:62696/node
Listening on address : 127.0.0.1:10005
RPC service listening on address : localhost:10006
Node for "NodeA" started up and registered in 29.96 sec

Welcome to the Corda interactive shell.
Useful commands include 'help' to see what is available, and 'bye' to shut down the node.
Thu Jun 15 10:08:34 BST 2017>>> [REDACTED]
```

Interacting with the nodes

Now that our nodes are running, let's order one of them to create an IOU by kicking off our `IOUFlow`. In a larger app, we'd generally provide a web API sitting on top of our node. Here, for simplicity, we'll be interacting with the node via its built-in CRaSH shell.

Go to the terminal window displaying the CRaSH shell of PartyA.

Typing `help` will display a list of the available commands.

Note

Local terminal shell is available only in a development mode. In production environment SSH server can be enabled. More about SSH and how to connect can be found on the `Shell` page.

We want to create an IOU of 99 with PartyB. We start the `IOUFlow` by typing:

Kotlin

```
start IOUFlow iouValue: 99, otherParty: "O=PartyB,L>New York,C=US"
```

This single command will cause PartyA and PartyB to automatically agree an IOU. This is one of the great advantages of the flow framework - it allows you to reduce complex negotiation and update processes into a single function call.

If the flow worked, it should have recorded a new IOU in the vaults of both PartyA and PartyB. Let's check.

We can check the contents of each node's vault by running:

```
run vaultQuery contractStateType: com.template.IOUState
```

The vaults of PartyA and PartyB should both display the following output:

```
states:
- state:
  data:
    value: 99
    lender: "C=GB,L=London,O=PartyA"
    borrower: "C=US,L>New York,O=PartyB"
    participants:
      - "C=GB,L=London,O=PartyA"
      - "C=US,L>New York,O=PartyB"
    contract: "com.template.contract.IOUContract"
    notary: "C=GB,L=London,O=Notary"
    encumbrance: null
    constraint:
      attachmentId: "F578320232CAB87BB1E919F3E5DB9D81B7346F9D7EA6D9155DC0F7BA8E472552"
    ref:
      txhash: "5CED068E790A347B0DD1C6BB5B2B463406807F95E080037208627565E6A2103B"
      index: 0
  statesMetadata:
- ref:
  txhash: "5CED068E790A347B0DD1C6BB5B2B463406807F95E080037208627565E6A2103B"
  index: 0
  contractStateClassName: "com.template.state.IOUState"
  recordedTime: 1506415268.875000000
  consumedTime: null
  status: "UNCONSUMED"
  notary: "C=GB,L=London,O=Notary"
  lockId: null
  lockUpdateTime: 1506415269.548000000
  totalStatesAvailable: -1
  stateTypes: "UNCONSUMED"
  otherResults: []
```

This is the transaction issuing our `IOUState` onto a ledger.

Conclusion

We have written a simple CorDapp that allows IOUs to be issued onto the ledger. Our CorDapp is made up of two key parts:

- The `IOUState`, representing IOUs on the ledger
- The `IOUFlow`, orchestrating the process of agreeing the creation of an IOU on-ledger

After completing this tutorial, your CorDapp should look like this:

- Java: <https://github.com/corda/corda-tut1-solution-java>
- Kotlin: <https://github.com/corda/corda-tut1-solution-kotlin>

Next steps

There are a number of improvements we could make to this CorDapp:

- We should add unit tests, using the contract-test and flow-test frameworks
- We should change `IOUState.value` from an integer to a proper amount of a given currency
- We could add an API, to make it easier to interact with the CorDapp

But for now, the biggest priority is to add an `IOUContract` imposing constraints on the evolution of each `IOUState` over time. This will be the focus of our next tutorial.

[Next](#) [Previous](#)

- Hello, World! Pt.2 - Contract constraints
- [View page source](#)

Hello, World! Pt.2 - Contract constraints

- Writing the contract
- Updating the flow

Note

This tutorial extends the CorDapp built during the [Hello, World tutorial](#).

In the Hello, World tutorial, we built a CorDapp allowing us to model IOUs on ledger. Our CorDapp was made up of two elements:

- An `IOUState`, representing IOUs on the ledger
- An `IOUFlow`, orchestrating the process of agreeing the creation of an IOU on-ledger

However, our CorDapp did not impose any constraints on the evolution of IOUs on the ledger over time. Anyone was free to create IOUs of any value, between any party.

In this tutorial, we'll write a contract to impose rules on how an `IOUState` can change over time. In turn, this will require some small changes to the flow we defined in the previous tutorial.

We'll start by writing the contract.

[Next](#) [Previous](#)

- Writing the contract
 - [View page source](#)
-

Writing the contract

It's easy to imagine that most CorDapps will want to impose some constraints on how their states evolve over time:

- A cash CorDapp will not want to allow users to create transactions that generate money out of thin air (at least without the involvement of a central bank or commercial bank)
- A loan CorDapp might not want to allow the creation of negative-valued loans
- An asset-trading CorDapp will not want to allow users to finalise a trade without the agreement of their counterparty

In Corda, we impose constraints on how states can evolve using contracts.

Note

Contracts in Corda are very different to the smart contracts of other distributed ledger platforms. They are not stateful objects representing the current state of the world. Instead, like a real-world contract, they simply impose rules on what kinds of transactions are allowed.

Every state has an associated contract. A transaction is invalid if it does not satisfy the contract of every input and output state in the transaction.

The Contract interface

Just as every Corda state must implement the `ContractState` interface, every contract must implement the `Contract` interface:

Kotlin

```
interface Contract {  
    // Implements the contract constraints in code.  
    @Throws(IllegalArgumentException::class)  
    fun verify(tx: LedgerTransaction)  
}
```

We can see that `Contract` expresses its constraints through a `verify` function that takes a transaction as input, and:

- Throws an `IllegalArgumentException` if it rejects the transaction proposal
- Returns silently if it accepts the transaction proposal

Controlling IOU evolution

What would a good contract for an `IOUState` look like? There is no right or wrong answer - it depends on how you want your CorDapp to behave.

For our CorDapp, let's impose the constraint that we only want to allow the creation of IOUs. We don't want nodes to transfer them or redeem them for cash. One way to enforce this behaviour would be by imposing the following constraints:

- A transaction involving IOUs must consume zero inputs, and create one output of type `IOUState`
- The transaction should also include a `Create` command, indicating the transaction's intent (more on commands shortly)

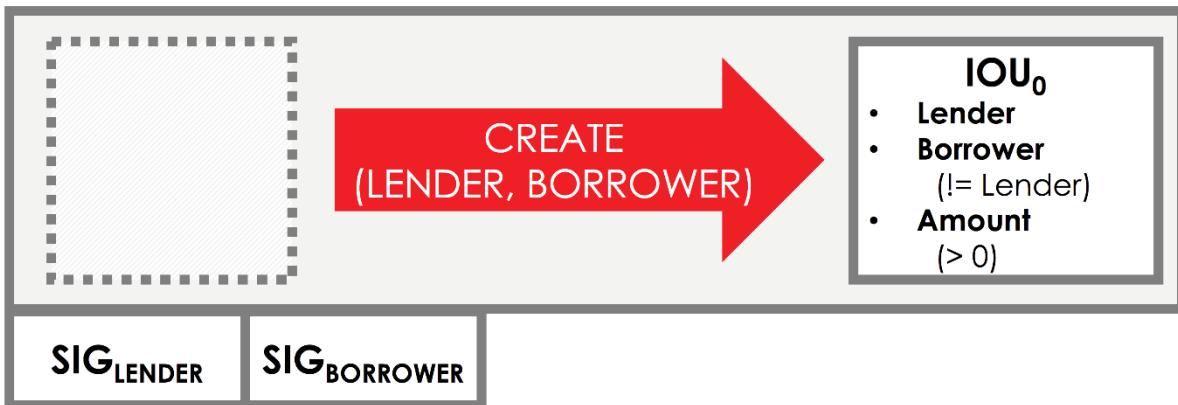
We might also want to impose some constraints on the properties of the issued `IOUState`:

- Its value must be non-negative
- The lender and the borrower cannot be the same entity

And finally, we'll want to impose constraints on who is required to sign the transaction:

- The IOU's lender must sign
- The IOU's borrower must sign

We can picture this transaction as follows:



Defining IOUContract

Let's write a contract that enforces these constraints. We'll do this by modifying either `TemplateContract.java` or `StatesAndContracts.kt` and updating `TemplateContract` to define an `IOUContract`:

KotlinJava

```
// Add these imports:
import net.corda.core.contracts.*

// Replace IOUContract's contract ID and definition with:
class IOUContract : Contract {
    companion object {
        val ID = "com.template.IOUContract"
    }

    // Our Create command.
    class Create : CommandData

    override fun verify(tx: LedgerTransaction) {
        val command = tx.commands.requireSingleCommand<Create>()

        requireThat {
            // Constraints on the shape of the transaction.
            "No inputs should be consumed when issuing an IOU." using
(tx.inputs.isEmpty())
            "There should be one output state of type IOUState." using
(tx.outputs.size == 1)

            // IOU-specific constraints.
            val out = tx.outputsOfType<IOUState>().single()
            "The IOU's value must be non-negative." using (out.value > 0)
            "The lender and the borrower cannot be the same entity." using (out.lender
!= out.borrower)

            // Constraints on the signers.
            "There must be two signers." using (command.signers.toSet().size == 2)
            "The borrower and lender must be signers." using
(command.signers.containsAll(listOf(
                out.borrower.owningKey, out.lender.owningKey)))
        }
    }
}
```

If you're following along in Java, you'll also need to rename `TemplateContract.java` to `IOUContract.java`.

Let's walk through this code step by step.

The Create command

The first thing we add to our contract is a *command*. Commands serve two functions:

- They indicate the transaction's intent, allowing us to perform different verification for different types of transaction. For example, a transaction proposing the creation of an IOU could have to meet different constraints to one redeeming an IOU
- They allow us to define the required signers for the transaction. For example, IOU creation might require signatures from the lender only, whereas the transfer of an IOU might require signatures from both the IOU's borrower and lender

Our contract has one command, a `Create` command. All commands must implement the `CommandData` interface.

The `CommandData` interface is a simple marker interface for commands. In fact, its declaration is only two words long (Kotlin interfaces do not require a body):

```
Kotlin  
interface CommandData
```

The verify logic

Our contract also needs to define the actual contract constraints by implementing `verify`. Our goal in writing the `verify` function is to write a function that, given a transaction:

- Throws an `IllegalArgumentException` if the transaction is considered invalid
- Does **not** throw an exception if the transaction is considered valid

In deciding whether the transaction is valid, the `verify` function only has access to the contents of the transaction:

- `tx.inputs`, which lists the inputs
- `tx.outputs`, which lists the outputs
- `tx.commands`, which lists the commands and their associated signers

As well as to the transaction's attachments and time-window, which we won't use here.

Based on the constraints enumerated above, we need to write a `verify` function that rejects a transaction if any of the following are true:

- The transaction doesn't include a `Create` command
- The transaction has inputs
- The transaction doesn't have exactly one output
- The IOU itself is invalid
- The transaction doesn't require the lender's signature

Command constraints

Our first constraint is around the transaction's commands. We use Corda's `requireSingleCommand` function to test for the presence of a single `Create` command.

If the `Create` command isn't present, or if the transaction has multiple `Create` commands, an exception will be thrown and contract verification will fail.

Transaction constraints

We also want our transaction to have no inputs and only a single output - an issuance transaction.

To impose this and the subsequent constraints, we are using Corda's built-in `requireThat` block. `requireThat` provides a terse way to write the following:

- If the condition on the right-hand side doesn't evaluate to true...
- ...throw an `IllegalArgumentException` with the message on the left-hand side

As before, the act of throwing this exception causes the transaction to be considered invalid.

IOU constraints

We want to impose two constraints on the `IOUState` itself:

- Its value must be non-negative
- The lender and the borrower cannot be the same entity

We impose these constraints in the same `requireThat` block as before.

You can see that we're not restricted to only writing constraints in the `requireThat` block. We can also write other statements - in this case, extracting the transaction's single `IOUState` and assigning it to a variable.

Signer constraints

Finally, we require both the lender and the borrower to be required signers on the transaction. A transaction's required signers is equal to the union of all the signers listed on the commands. We therefore extract the signers from the `Create` command we retrieved earlier.

This is an absolutely essential constraint - it ensures that no `IOUState` can ever be created on the ledger without the express agreement of both the lender and borrower nodes.

Progress so far

We've now written an `IOUContract` constraining the evolution of each `IOUState` over time:

- An `IOUState` can only be created, not transferred or redeemed
- Creating an `IOUState` requires an issuance transaction with no inputs, a single `IOUState` output, and a `Create` command
- The `IOUState` created by the issuance transaction must have a non-negative value, and the lender and borrower must be different entities

Next, we'll update the `IOUFlow` so that it obeys these contract constraints when issuing an `IOUState` onto the ledger.

[Next](#) [Previous](#)

- Updating the flow
 - [View page source](#)
-

Updating the flow

We now need to update our flow to achieve three things:

- Verifying that the transaction proposal we build fulfills the `IOUContract` constraints
- Updating the lender's side of the flow to request the borrower's signature
- Creating a response flow for the borrower that responds to the signature request from the lender

We'll do this by modifying the flow we wrote in the previous tutorial.

Verifying the transaction

In `IOUFlow.java` / `App.kt`, change the imports block to the following:

KotlinJava

```
import co.paralleluniverse.fibers.Suspendable
import net.corda.core.contracts.Command
import net.corda.core.contracts.StateAndContract
import net.corda.core.flows.*
import net.corda.core.identity.Party
import net.corda.core.messaging.CordaRPCOps
import net.corda.core.serialization.SerializationWhitelist
import net.corda.core.transactions.TransactionBuilder
import net.corda.core.utilities.ProgressTracker
import java.util.function.Function
import javax.ws.rs.GET
import javax.ws.rs.Path
import javax.ws.rs.Produces
import javax.ws.rs.core.MediaType
import javax.ws.rs.core.Response
```

And update `IOUflow.call` by changing the code following the retrieval of the notary's identity from the network as follows:

KotlinJava

```
// We create a transaction builder.
val txBuilder = TransactionBuilder(notary = notary)

// We create the transaction components.
val outputState = IOUState(iouValue, ourIdentity, otherParty)
val outputContractAndState = StateAndContract(outputState, IOUContract.ID)
val cmd = Command(IOUContract.Create(), listOf(ourIdentity.owningKey,
otherParty.owningKey))
```

```

// We add the items to the builder.
txBuilder.withItems(outputContractAndState, cmd)

// Verifying the transaction.
txBuilder.verify(serviceHub)

// Signing the transaction.
val signedTx = serviceHub.signInitialTransaction(txBuilder)

// Creating a session with the other party.
val otherpartySession = initiateFlow(otherParty)

// Obtaining the counterparty's signature.
val fullySignedTx = subFlow(CollectSignaturesFlow(signedTx, listOf(otherpartySession),
CollectSignaturesFlow.tracker()))

// Finalising the transaction.
subFlow(FinalityFlow(fullySignedTx))

```

In the original CorDapp, we automated the process of notarising a transaction and recording it in every party’s vault by invoking a built-in flow called `FinalityFlow` as a subflow. We’re going to use another pre-defined flow, `CollectSignaturesFlow`, to gather the borrower’s signature.

First, we need to update the command. We are now using `IOUContract.Create`, rather than `TemplateContract.Commands.Action`. We also want to make the borrower a required signer, as per the contract constraints. This is as simple as adding the borrower’s public key to the transaction’s command.

We also need to add the output state to the transaction using a reference to the `IOUContract`, instead of to the old `TemplateContract`.

Now that our state is governed by a real contract, we’ll want to check that our transaction proposal satisfies these requirements before kicking off the signing process. We do this by calling `TransactionBuilder.verify` on our transaction proposal before finalising it by adding our signature.

Requesting the borrower’s signature

We now need to communicate with the borrower to request their signature over the transaction. Whenever you want to communicate with another party in the context of a flow, you first need to establish a flow session with them. If the counterparty has a `FlowLogic` registered to respond to the `FlowLogic` initiating the session, a session will be established. All communication between the two `FlowLogic` instances will then place as part of this session.

Once we have a session with the borrower, we gather the borrower's signature using `CollectSignaturesFlow`, which takes:

- A transaction signed by the flow initiator
- A list of flow-sessions between the flow initiator and the required signers

And returns a transaction signed by all the required signers.

We can then pass this fully-signed transaction into `FinalityFlow`.

Creating the borrower's flow

On the lender's side, we used `CollectSignaturesFlow` to automate the collection of signatures. To allow the lender to respond, we need to write a response flow as well. In a new `IOUFlowResponder.java` file in Java, or within the `App.kt` file in Kotlin, add the following class:

KotlinJava

```
// Add these imports:  
import net.corda.core.contracts.requireThat  
import net.corda.core.transactions.SignedTransaction  
  
// Define IOUFlowResponder:  
@InitiatedBy(IOUFlow::class)  
class IOUFlowResponder(val otherPartySession: FlowSession) : FlowLogic<Unit>() {  
    @Suspendable  
    override fun call() {  
        val signTransactionFlow = object : SignTransactionFlow(otherPartySession,  
SignTransactionFlow.tracker()) {  
            override fun checkTransaction(stx: SignedTransaction) = requireThat {  
                val output = stx.tx.outputs.single().data  
                "This must be an IOU transaction." using (output is IOUState)  
                val iou = output as IOUState  
                "The IOU's value can't be too high." using (iou.value < 100)  
            }  
        }  
        subFlow(signTransactionFlow)  
    }  
}
```

As with the `IOUFlow`, our `IOUFlowResponder` flow is a `FlowLogic` subclass where we've overridden `FlowLogic.call`.

The flow is annotated with `InitiatedBy(IOUFlow.class)`, which means that your node will invoke `IOUFlowResponder.call` when it receives a message from a instance of `Initiator` running on another node. What will this message from the `IOUFlow` be? If we look at the definition of `CollectSignaturesFlow`, we can see

that we'll be sent a `SignedTransaction`, and are expected to send back our signature over that transaction.

We could write our own flow to handle this process. However, there is also a pre-defined flow called `SignTransactionFlow` that can handle the process automatically. The only catch is that `SignTransactionFlow` is an abstract class - we must subclass it and override `SignTransactionFlow.checkTransaction`.

CheckTransactions

`SignTransactionFlow` will automatically verify the transaction and its signatures before signing it. However, just because a transaction is contractually valid doesn't mean we necessarily want to sign. What if we don't want to deal with the counterparty in question, or the value is too high, or we're not happy with the transaction's structure?

Overriding `SignTransactionFlow.checkTransaction` allows us to define these additional checks. In our case, we are checking that:

- The transaction involves an `IOUState` - this ensures that `IOUContract` will be run to verify the transaction
- The IOU's value is less than some amount (100 in this case)

If either of these conditions are not met, we will not sign the transaction - even if the transaction and its signatures are contractually valid.

Once we've defined the `SignTransactionFlow` subclass, we invoke it using `FlowLogic.subFlow`, and the communication with the borrower's and the lender's flow is conducted automatically.

Conclusion

We have now updated our flow to verify the transaction and gather the lender's signature, in line with the constraints defined in `IOUContract`. We can now re-run our updated CorDapp, using the same instructions as before.

Our CorDapp now imposes restrictions on the issuance of IOUs. Most importantly, IOU issuance now requires agreement from both the lender and the borrower before an IOU can be created on the ledger. This prevents either the

lender or the borrower from unilaterally updating the ledger in a way that only benefits themselves.

After completing this tutorial, your CorDapp should look like this:

- Java: <https://github.com/corda/corda-tut2-solution-java>
- Kotlin: <https://github.com/corda/corda-tut2-solution-kotlin>

You should now be ready to develop your own CorDapps. You can also find a list of sample CorDapps [here](#). As you write CorDapps, you'll also want to learn more about the Corda API.

If you get stuck at any point, please reach out on [Slack](#) or [Stack Overflow](#).

[Next](#) [Previous](#)

- Writing a contract
 - [View page source](#)
-

Writing a contract

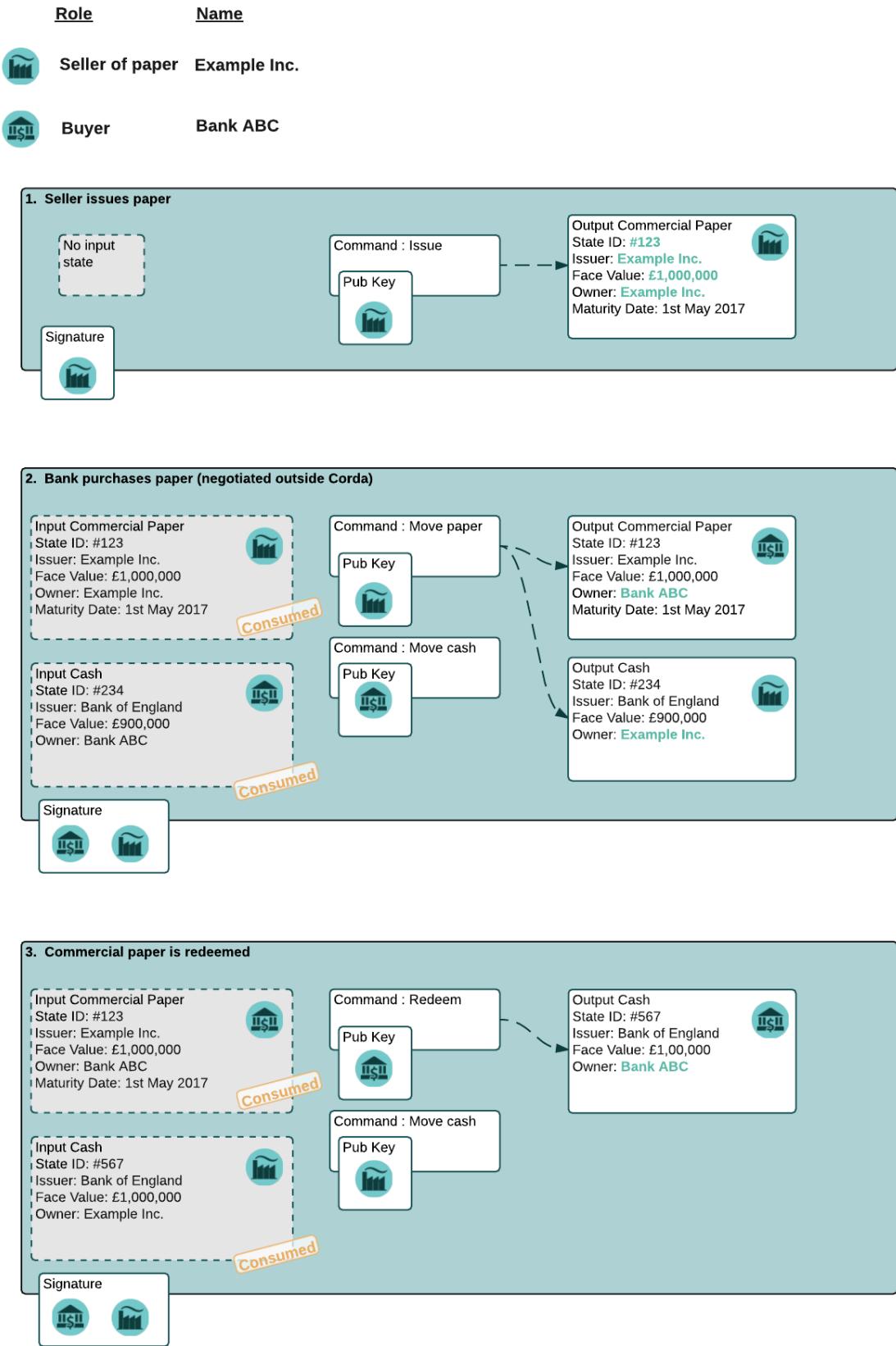
This tutorial will take you through writing a contract, using a simple commercial paper contract as an example. Smart contracts in Corda have three key elements:

- Executable code (validation logic)
- State objects
- Commands

The core of a smart contract is the executable code which validates changes to state objects in transactions. State objects are the data held on the ledger, which represent the current state of an instance of a contract, and are used as inputs and outputs of transactions. Commands are additional data included in transactions to describe what is going on, used to instruct the executable code on how to verify the transaction. For example an `Issue` command may indicate that the validation logic should expect to see an output which does not exist as an input, issued by the same entity that signed the command.

The first thing to think about with a new contract is the lifecycle of contract states, how are they issued, what happens to them after they are issued, and how are they destroyed (if applicable). For the commercial paper contract, states are issued by a legal entity which wishes to create a contract to pay money in the future (the maturity date), in return for a lesser payment now. They are then transferred (moved) to another owner as part of a transaction where the issuer receives funds in payment, and later (after the maturity date) are destroyed (redeemed) by paying the owner the face value of the commercial paper.

This lifecycle for commercial paper is illustrated in the diagram below:



Starting the commercial paper class

A smart contract is a class that implements the `Contract` interface. This can be either implemented directly, as done here, or by subclassing an abstract contract such as `OnLedgerAsset`. The heart of any contract in Corda is the `verify` function, which determines whether a given transaction is valid. This example shows how to write a `verify` function from scratch.

The code in this tutorial is available in both Kotlin and Java. You can quickly switch between them to get a feeling for Kotlin's syntax.

KotlinJava

```
class CommercialPaper : Contract {
    override fun verify(tx: LedgerTransaction) {
        TODO()
    }
}
```

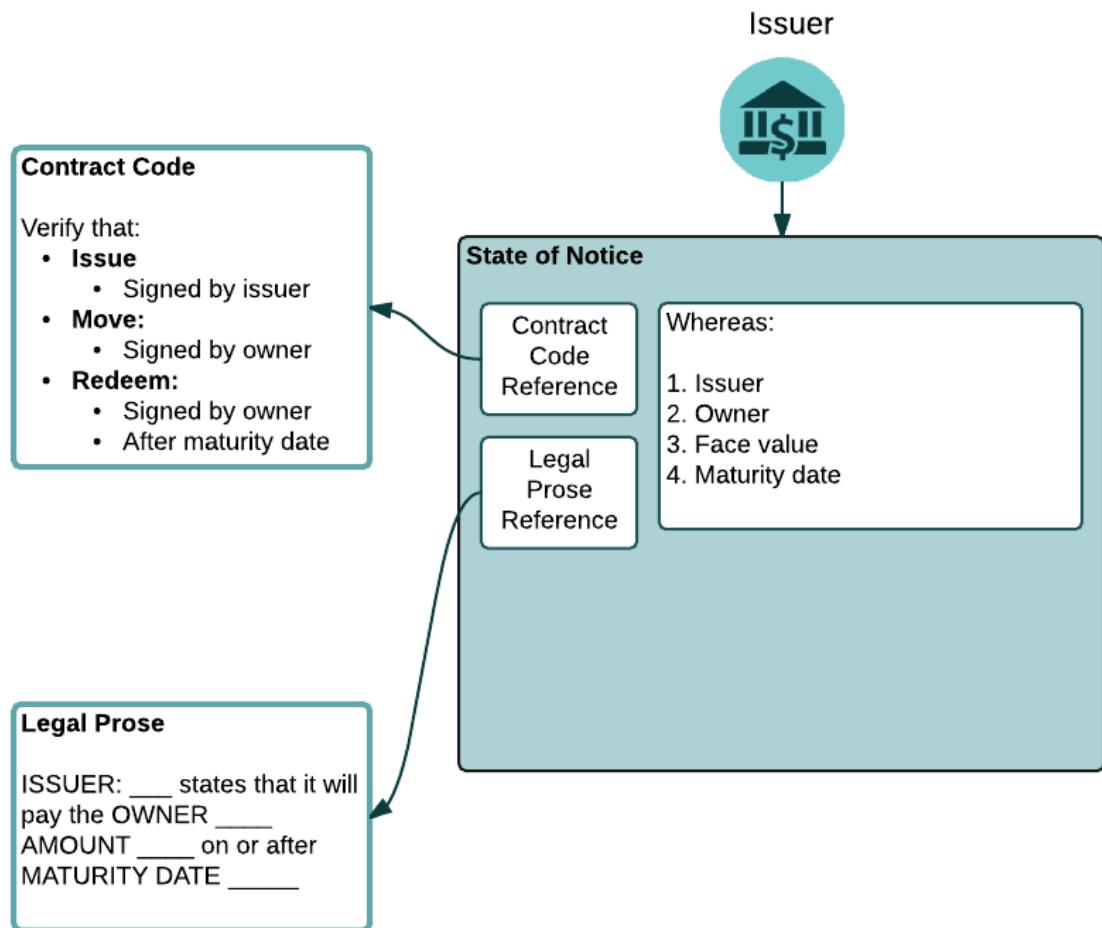
Every contract must have at least a `verify` method. The verify method returns nothing. This is intentional: the function either completes correctly, or throws an exception, in which case the transaction is rejected.

So far, so simple. Now we need to define the commercial paper *state*, which represents the fact of ownership of a piece of issued paper.

States

A state is a class that stores data that is checked by the contract. A commercial paper state is structured as below:

Commercial Paper



KotlinJava

```
data class State(
    val issuance: PartyAndReference,
    override val owner: AbstractParty,
    val faceValue: Amount<Issued<Currency>>,
    val maturityDate: Instant
) : OwnableState {
    override val participants = listOf(owner)

    fun withoutOwner() = copy(owner = AnonymousParty(NullKeys.NullPublicKey))
    override fun withNewOwner(newOwner: AbstractParty) =
        CommandAndState(CommercialPaper.Commands.Move(), copy(owner = newOwner))
}
```

We define a class that implements the `ContractState` interface.

We have four fields in our state:

- `issuance`, a reference to a specific piece of commercial paper issued by some party.
- `owner`, the public key of the current owner. This is the same concept as seen in Bitcoin: the public key has no attached identity and is expected to be one-time-use for privacy reasons. However, unlike in Bitcoin, we model ownership at the level of individual states rather than as a platform-level concept as we envisage many (possibly most) contracts on the platform will not represent “owner/issuer” relationships, but “party/party” relationships such as a derivative contract.
- `faceValue`, an `Amount<Issued<Currency>>`, which wraps an integer number of pennies and a currency that is specific to some issuer (e.g. a regular bank, a central bank, etc). You can read more about this very common type in [API: Core types](#).
- `maturityDate`, an `Instant`, which is a type from the Java 8 standard time library. It defines a point on the timeline.

States are immutable, and thus the class is defined as immutable as well. The `data` modifier in the Kotlin version causes the compiler to generate the equals/hashCode/toString methods automatically, along with a copy method that can be used to create variants of the original object. Data classes are similar to case classes in Scala, if you are familiar with that language. The `withoutOwner` method uses the auto-generated copy method to return a version of the state with the owner public key blanked out: this will prove useful later.

The Java code compiles to almost identical bytecode as the Kotlin version, but as you can see, is much more verbose.

Commands

The validation logic for a contract may vary depending on what stage of a state’s lifecycle it is automating. So it can be useful to pass additional data into the contract code that isn’t represented by the states which exist permanently in the ledger, in order to clarify intent of a transaction.

For this purpose we have commands. Often they don’t need to contain any data at all, they just need to exist. A command is a piece of data associated with some *signatures*. By the time the contract runs the signatures have already been checked, so from the contract code’s perspective, a command is simply a data

structure with a list of attached public keys. Each key had a signature proving that the corresponding private key was used to sign. Because of this approach contracts never actually interact or work with digital signatures directly.

Let's define a few commands now:

KotlinJava

```
interface Commands : CommandData {  
    class Move : TypeOnlyCommandData(), Commands  
    class Redeem : TypeOnlyCommandData(), Commands  
    class Issue : TypeOnlyCommandData(), Commands  
}
```

We define a simple grouping interface or static class, this gives us a type that all our commands have in common, then we go ahead and create three commands: `Move`, `Redeem`, `Issue`. `TypeOnlyCommandData` is a helpful utility for the case when there's no data inside the command; only the existence matters. It defines equals and hashCode such that any instances always compare equal and hash to the same value.

The verify function

The heart of a smart contract is the code that verifies a set of state transitions (a *transaction*). The function is simple: it's given a class representing the transaction, and if the function returns then the transaction is considered acceptable. If it throws an exception, the transaction is rejected.

Each transaction can have multiple input and output states of different types. The set of contracts to run is decided by taking the code references inside each state. Each contract is run only once. As an example, a contract that includes 2 cash states and 1 commercial paper state as input, and has as output 1 cash state and 1 commercial paper state, will run two contracts one time each: Cash and CommercialPaper.

KotlinJava

```
override fun verify(tx: LedgerTransaction) {  
    // Group by everything except owner: any modification to the CP at all is  
    // considered changing it fundamentally.  
    val groups = tx.groupStates(State::withoutOwner)  
  
    // There are two possible things that can be done with this CP. The first is  
    // trading it. The second is redeeming  
    // it for cash on or after the maturity date.  
    val command = tx.commands.requireSingleCommand<CommercialPaper.Commands>()
```

We start by using the `groupStates` method, which takes a type and a function.

State grouping is a way of ensuring your contract can handle multiple unrelated states of the same type in the same transaction, which is needed for splitting/merging of assets, atomic swaps and so on. More on this next.

The second line does what the code suggests: it searches for a command object that inherits from the `CommercialPaper.Commands` supertype, and either returns it, or throws an exception if there's zero or more than one such command.

Using state groups

The simplest way to write a smart contract would be to say that each transaction can have a single input state and a single output state of the kind covered by that contract. This would be easy for the developer, but would prevent many important use cases.

The next easiest way to write a contract would be to iterate over each input state and expect it to have an output state. Now you can build a single transaction that, for instance, moves two different cash states in different currencies simultaneously. But it gets complicated when you want to issue or exit one state at the same time as moving another.

Things get harder still once you want to split and merge states. We say states are *fungible* if they are treated identically to each other by the recipient, despite the fact that they aren't quite identical. Dollar bills are fungible because even though one may be worn/a bit dirty and another may be crisp and new, they are still both worth exactly \$1. Likewise, ten \$1 bills are almost exactly equivalent to one \$10 bill. On the other hand, \$10 and £10 are not fungible: if you tried to pay for something that cost £20 with \$10+£10 notes your trade would not be accepted.

To make all this easier the contract API provides a notion of groups. A group is a set of input states and output states that should be checked for validity together.

Consider the following simplified currency trade transaction:

- **Input:** \$12,000 owned by Alice (A)
- **Input:** \$3,000 owned by Alice (A)

- **Input:** £10,000 owned by Bob (B)
- **Output:** £10,000 owned by Alice (B)
- **Output:** \$15,000 owned by Bob (A)

In this transaction Alice and Bob are trading \$15,000 for £10,000. Alice has her money in the form of two different inputs e.g. because she received the dollars in two payments. The input and output amounts do balance correctly, but the cash smart contract must consider the pounds and the dollars separately because they are not fungible: they cannot be merged together. So we have two groups: A and B.

The `LedgerTransaction.groupStates` method handles this logic for us: firstly, it selects only states of the given type (as the transaction may include other types of state, such as states representing bond ownership, or a multi-sig state) and then it takes a function that maps a state to a grouping key. All states that share the same key are grouped together. In the case of the cash example above, the grouping key would be the currency.

In this kind of contract we don't want CP to be fungible: merging and splitting is (in our example) not allowed. So we just use a copy of the state minus the owner field as the grouping key.

Here are some code examples:

KotlinJava

```
// Type of groups is List<InOutGroup<State, Pair<PartyReference, Currency>>>
val groups = tx.groupStates { it: Cash.State -> it.amount.token }
for ((inputs, outputs, key) in groups) {
    // Either inputs or outputs could be empty.
    val (deposit, currency) = key
    ...
}
```

The `groupStates` call uses the provided function to calculate a “grouping key”. All states that have the same grouping key are placed in the same group. A grouping key can be anything that implements equals/hashCode, but it's always an aggregate of the fields that shouldn't change between input and output. In the above example we picked the fields we wanted and packed them into a `Pair`. It returns a list of `InOutGroup`, which is just a holder for the inputs, outputs and the key that was used to define the group. In the Kotlin version we unpack these

using destructuring to get convenient access to the inputs, the outputs, the deposit data and the currency. The Java version is more verbose, but equivalent.

The rules can then be applied to the inputs and outputs as if it were a single transaction. A group may have zero inputs or zero outputs: this can occur when issuing assets onto the ledger, or removing them.

In this example, we do it differently and use the state class itself as the aggregator. We just blank out fields that are allowed to change, making the grouping key be “everything that isn’t that”:

KotlinJava

```
val groups = tx.groupStates(State::withoutOwner)
```

For large states with many fields that must remain constant and only one or two that are really mutable, it’s often easier to do things this way than to specifically name each field that must stay the same. The `withoutOwner` function here simply returns a copy of the object but with the `owner` field set to `NullPublicKey`, which is just a public key of all zeros. It’s invalid and useless, but that’s OK, because all we’re doing is preventing the field from mattering in `equals` and `hashCode`.

Checking the requirements

After extracting the command and the groups, we then iterate over each group and verify it meets the required business logic.

KotlinJava

```
val timeWindow: TimeWindow? = tx.timeWindow

for ((inputs, outputs, _) in groups) {
    when (command.value) {
        is Commands.Move -> {
            val input = inputs.single()
            requireThat {
                "the transaction is signed by the owner of the CP" using
                (input.owner.owningKey in command.signers)
                "the state is propagated" using (outputs.size == 1)
                // Don't need to check anything else, as if outputs.size == 1 then the
                output is equal to
                // the input ignoring the owner field due to the grouping.
            }
        }

        is Commands.Redeem -> {
            // Redemption of the paper requires movement of on-Ledger cash.
            val input = inputs.single()
            val received = tx.outputs.map { it.data }.sumCashBy(input.owner)
            val time = timeWindow?.fromTime ?: throw
            IllegalArgumentException("Redemptions must be timestamped")
        }
    }
}
```

```

        requireThat {
            "the paper must have matured" using (time >= input.maturityDate)
            "the received amount equals the face value" using (received ==
input.faceValue)
            "the paper must be destroyed" using outputs.isEmpty()
            "the transaction is signed by the owner of the CP" using
(input.owner.owningKey in command.signers)
        }
    }

    is Commands.Issue -> {
        val output = outputs.single()
        val time = timeWindow?.untilTime ?: throw
IllegalArgumentException("Issuances must be timestamped")
        requireThat {
            // Don't allow people to issue commercial paper under other entities
identities.
            "output states are issued by a command signer" using
(output.issuance.party.owningKey in command.signers)
            "output values sum to more than the inputs" using
(output.faceValue.quantity > 0)
            "the maturity date is not in the past" using (time <
output.maturityDate)
            // Don't allow an existing CP state to be replaced by this issuance.
            "can't reissue an existing state" using inputs.isEmpty()
        }
    }

    else -> throw IllegalArgumentException("Unrecognised command")
}
}

```

This loop is the core logic of the contract.

The first line simply gets the timestamp out of the transaction. Timestamping of transactions is optional, so a time may be missing here. We check for it being null later.

Warning

In the Kotlin version as long as we write a comparison with the transaction time first the compiler will verify we didn't forget to check if it's missing. Unfortunately due to the need for smooth Java interop, this check won't happen if we write e.g. `someDate > time`, it has to be `time < someDate`. So it's good practice to always write the transaction timestamp first.

Next, we take one of three paths, depending on what the type of the command object is.

If the command is a ``Move`` command:

The first line (first three lines in Java) impose a requirement that there be a single piece of commercial paper in this group. We do not allow multiple units of

CP to be split or merged even if they are owned by the same owner.

The `single()` method is a static *extension method* defined by the Kotlin standard library: given a list, it throws an exception if the list size is not 1, otherwise it returns the single item in that list. In Java, this appears as a regular static method of the type familiar from many FooUtils type singleton classes and we have statically imported it here. In Kotlin, it appears as a method that can be called on any JDK list. The syntax is slightly different but behind the scenes, the code compiles to the same bytecodes.

Next, we check that the transaction was signed by the public key that's marked as the current owner of the commercial paper. Because the platform has already verified all the digital signatures before the contract begins execution, all we have to do is verify that the owner's public key was one of the keys that signed the transaction. The Java code is straightforward: we are simply using the `Preconditions.checkState` method from Guava. The Kotlin version looks a little odd: we have a *requireThat* construct that looks like it's built into the language. In fact *requireThat* is an ordinary function provided by the platform's contract API. Kotlin supports the creation of *domain specific languages* through the intersection of several features of the language, and we use it here to support the natural listing of requirements. To see what it compiles down to, look at the Java version. Each `"string" using (expression)` statement inside a `requireThat` turns into an assertion that the given expression is true, with an `IllegalArgumentException` being thrown that contains the string if not. It's just another way to write out a regular assertion, but with the English-language requirement being put front and center.

Next, we simply verify that the output state is actually present: a move is not allowed to delete the CP from the ledger. The grouping logic already ensured that the details are identical and haven't been changed, save for the public key of the owner.

If the command is a ``Redeem`` command, then the requirements are more complex:

1. We still check there is a CP input state.
2. We want to see that the face value of the CP is being moved as a cash claim against some party, that is, the issuer of the CP is really paying back the face value.

3. The transaction must be happening after the maturity date.
4. The commercial paper must *not* be propagated by this transaction: it must be deleted, by the group having no output state. This prevents the same CP being considered redeemable multiple times.

To calculate how much cash is moving, we use the `sumCashBy` utility function.

Again, this is an extension function, so in Kotlin code it appears as if it was a method on the `List<Cash.State>` type even though JDK provides no such method.

In Java we see its true nature: it is actually a static method named `CashKt.sumCashBy`. This method simply returns an `Amount` object containing the sum of all the cash states in the transaction outputs that are owned by that given public key, or throws an exception if there were no such states *or* if there were different currencies represented in the outputs! So we can see that this contract imposes a limitation on the structure of a redemption transaction: you are not allowed to move currencies in the same transaction that the CP does not involve. This limitation could be addressed with better APIs, if it were to be a real limitation.

Finally, we support an ``Issue`` command, to create new instances of commercial paper on the ledger.

It likewise enforces various invariants upon the issuance, such as, there must be one output CP state, for instance.

This contract is simple and does not implement all the business logic a real commercial paper lifecycle management program would. For instance, there is no logic requiring a signature from the issuer for redemption: it is assumed that any transfer of money that takes place at the same time as redemption is good enough. Perhaps that is something that should be tightened. Likewise, there is no logic handling what happens if the issuer has gone bankrupt, if there is a dispute, and so on.

As the prototype evolves, these requirements will be explored and this tutorial updated to reflect improvements in the contracts API.

How to test your contract

Of course, it is essential to unit test your new nugget of business logic to ensure that it behaves as you expect. As contract code is just a regular Java function

you could write out the logic entirely by hand in the usual manner. But this would be inconvenient, and then you'd get bored of writing tests and that would be bad: you might be tempted to skip a few.

To make contract testing more convenient Corda provides a language-like API for both Kotlin and Java that lets you easily construct chains of transactions and verify that they either pass validation, or fail with a particular error message.

Testing contracts with this domain specific language is covered in the separate tutorial, [Writing a contract test](#).

Adding a generation API to your contract

Contract classes **must** provide a verify function, but they may optionally also provide helper functions to simplify their usage. A simple class of functions most contracts provide are *generation functions*, which either create or modify a transaction to perform certain actions (an action is normally mappable 1:1 to a command, but doesn't have to be so).

Generation may involve complex logic. For example, the cash contract has a `generateSpend` method that is given a set of cash states and chooses a way to combine them together to satisfy the amount of money that is being sent. In the immutable-state model that we are using ledger entries (states) can only be created and deleted, but never modified. Therefore to send \$1200 when we have only \$900 and \$500 requires combining both states together, and then creating two new output states of \$1200 and \$200 back to ourselves. This latter state is called the *change* and is a concept that should be familiar to anyone who has worked with Bitcoin.

As another example, we can imagine code that implements a netting algorithm may generate complex transactions that must be signed by many people. Whilst such code might be too big for a single utility method (it'd probably be sized more like a module), the basic concept is the same: preparation of a transaction using complex logic.

For our commercial paper contract however, the things that can be done with it are quite simple. Let's start with a method to wrap up the issuance process:

Kotlin

```

fun generateIssue(issuance: PartyAndReference, faceValue: Amount<Issued<Currency>>,
maturityDate: Instant,
    notary: Party): TransactionBuilder {
    val state = State(issuance, issuance.party, faceValue, maturityDate)
    val stateAndContract = StateAndContract(state, CP_PROGRAM_ID)
    return TransactionBuilder(notary = notary).withItems(stateAndContract,
Command(Command.Commands.Issue(), issuance.party.owningKey))
}

```

We take a reference that points to the issuing party (i.e. the caller) and which can contain any internal bookkeeping/reference numbers that we may require. The reference field is an ideal place to put (for example) a join key. Then the face value of the paper, and the maturity date. It returns a `TransactionBuilder`.

A `TransactionBuilder` is one of the few mutable classes the platform provides. It allows you to add inputs, outputs and commands to it and is designed to be passed around, potentially between multiple contracts.

Note

Generation methods should ideally be written to compose with each other, that is, they should take a `TransactionBuilder` as an argument instead of returning one, unless you are sure it doesn't make sense to combine this type of transaction with others. In this case, issuing CP at the same time as doing other things would just introduce complexity that isn't likely to be worth it, so we return a fresh object each time: instead, an issuer should issue the CP (starting out owned by themselves), and then sell it in a separate transaction.

The function we define creates a `CommercialPaper.State` object that mostly just uses the arguments we were given, but it fills out the owner field of the state to be the same public key as the issuing party.

We then combine the `CommercialPaper.State` object with a reference to the `CommercialPaper` contract, which is defined inside the contract itself

KotlinJava

```

companion object {
    const val CP_PROGRAM_ID: ContractClassName =
"net.corda.finance.contracts.CommercialPaper"
}

```

This value, which is the fully qualified class name of the contract, tells the Corda platform where to find the contract code that should be used to validate a transaction containing an output state of this contract type. Typically the contract code will be included in the transaction as an attachment (see [Using attachments](#)).

The returned partial transaction has a `Command` object as a parameter. This is a container for any object that implements the `CommandData` interface, along with a list of keys that are expected to sign this transaction. In this case, issuance requires that the issuing party sign, so we put the key of the party there.

The `TransactionBuilder` has a convenience `withItems` method that takes a variable argument list. You can pass in any `StateAndRef` (input), `StateAndContract` (output) or `Command` objects and it'll build up the transaction for you.

There's one final thing to be aware of: we ask the caller to select a *notary* that controls this state and prevents it from being double spent. You can learn more about this topic in the [Notaries](#) article.

Note

For now, don't worry about how to pick a notary. More infrastructure will come later to automate this decision for you.

What about moving the paper, i.e. reassigning ownership to someone else?

Kotlin

```
fun generateMove(tx: TransactionBuilder, paper: StateAndRef<State>, newOwner: AbstractParty) {
    tx.addInputState(paper)
    val outputState = paper.state.data.withNewOwner(newOwner).ownableState
    tx.addOutputState(outputState, CP_PROGRAM_ID)
    tx.addCommand(Command.Commands.Move(), paper.state.data.owner.owningKey))
}
```

Here, the method takes a pre-existing `TransactionBuilder` and adds to it. This is correct because typically you will want to combine a sale of CP atomically with the movement of some other asset, such as cash. So both generate methods should operate on the same transaction. You can see an example of this being done in the unit tests for the commercial paper contract.

The paper is given to us as a `StateAndRef<CommercialPaper.State>` object. This is exactly what it sounds like: a small object that has a (copy of a) state object, and also the `(txhash, index)` that indicates the location of this state on the ledger.

We add the existing paper state as an input, the same paper state with the owner field adjusted as an output, and finally a move command that has the old owner's public key: this is what forces the current owner's signature to be present on the transaction, and is what's checked by the contract.

Finally, we can do redemption.

Kotlin

```
@Throws(InsufficientBalanceException::class)
fun generateRedeem(tx: TransactionBuilder, paper: StateAndRef<State>, services: ServiceHub) {
    // Add the cash movement using the states in our vault.
    Cash.generateSpend(
        services = services,
        tx = tx,
        amount = paper.state.data.faceValue.withoutIssuer(),
        ourIdentity = services.myInfo.singleIdentityAndCert(),
        to = paper.state.data.owner
    )
    tx.addInputState(paper)
    tx.addCommand(CommandCommands.Redeem(), paper.state.data.owner.owningKey))
}
```

Here we can see an example of composing contracts together. When an owner wishes to redeem the commercial paper, the issuer (i.e. the caller) must gather cash from its vault and send the face value to the owner of the paper.

Note

This contract has no explicit concept of rollover.

The *vault* is a concept that may be familiar from Bitcoin and Ethereum. It is simply a set of states (such as cash) that are owned by the caller. Here, we use the vault to update the partial transaction we are handed with a movement of cash from the issuer of the commercial paper to the current owner. If we don't have enough quantity of cash in our vault, an exception is thrown. Then we add the paper itself as an input, but, not an output (as we wish to remove it from the ledger). Finally, we add a Redeem command that should be signed by the owner of the commercial paper.

Warning

The amount we pass to the `Cash.generateSpend` function has to be treated first with `withoutIssuer`. This reflects the fact that the way we handle issuer constraints is still evolving; the commercial paper contract requires payment in the form of a currency issued by a specific party (e.g. the central bank, or the issuers own bank perhaps). But the vault wants to assemble spend transactions using cash states from any issuer, thus we must strip it here. This represents a design mismatch that we will resolve in future versions with a more complete way to express issuer constraints.

A `TransactionBuilder` is not by itself ready to be used anywhere, so first, we must convert it to something that is recognised by the network. The most important next step is for the participating entities to sign it. Typically, an initiating flow will create an initial partially signed `SignedTransaction` by calling the `serviceHub.toSignedTransaction` method. Then the frozen `SignedTransaction` can be passed to other nodes by the flow, these can sign using `serviceHub.createSignature` and distribute. The `CollectSignaturesFlow` provides a generic implementation of this process that can be used as a `subFlow`.

You can see how transactions flow through the different stages of construction by examining the commercial paper unit tests.

How multi-party transactions are constructed and transmitted

OK, so now we know how to define the rules of the ledger, and we know how to construct transactions that satisfy those rules ... and if all we were doing was maintaining our own data that might be enough. But we aren't: Corda is about keeping many different parties all in sync with each other.

In a classical blockchain system all data is transmitted to everyone and if you want to do something fancy, like a multi-party transaction, you're on your own. In Corda data is transmitted only to parties that need it and multi-party transactions are a way of life, so we provide lots of support for managing them.

You can learn how transactions are moved between peers and taken through the build-sign-notarise-broadcast process in a separate tutorial, [Writing flows](#).

Non-asset-oriented smart contracts

Although this tutorial covers how to implement an owned asset, there is no requirement that states and code contracts *must* be concerned with ownership of an asset. It is better to think of states as representing useful facts about the world, and (code) contracts as imposing logical relations on how facts combine to produce new facts. Alternatively you can imagine that states are like rows in a relational database and contracts are like stored procedures and relational constraints.

When writing a contract that handles deal-like entities rather than asset-like entities, you may wish to refer to “Interest rate swaps” and the accompanying source code. Whilst all the concepts are the same, deals are typically not splittable or mergeable and thus you don’t have to worry much about grouping of states.

Making things happen at a particular time

It would be nice if you could program your node to automatically redeem your commercial paper as soon as it matures. Corda provides a way for states to advertise scheduled events that should occur in future. Whilst this information is by default ignored, if the corresponding *Cordapp* is installed and active in your node, and if the state is considered relevant by your vault (e.g. because you own it), then the node can automatically begin the process of creating a transaction and taking it through the life cycle. You can learn more about this in the article “Event scheduling”.

Encumbrances

All contract states may be *encumbered* by up to one other state, which we call an **encumbrance**.

The encumbrance state, if present, forces additional controls over the encumbered state, since the encumbrance state contract will also be verified during the execution of the transaction. For example, a contract state could be encumbered with a time-lock contract state; the state is then only processable in a transaction that verifies that the time specified in the encumbrance time-lock has passed.

The encumbered state refers to its encumbrance by index, and the referred encumbrance state is an output state in a particular position on the same transaction that created the encumbered state. Note that an encumbered state that is being consumed must have its encumbrance consumed in the same transaction, otherwise the transaction is not valid.

The encumbrance reference is optional in the `ContractState` interface:

KotlinJava

```
val encumbrance: Int? get() = null
```

The time-lock contract mentioned above can be implemented very simply:

Kotlin

```
class TestTimeLock : Contract {
    ...
    override fun verify(tx: LedgerTransaction) {
        val time = tx.timestamp.before ?: throw IllegalStateException(...)
        ...
        requireThat {
            "the time specified in the time-lock has passed" by
                (time >=
            tx.inputs.filterIsInstance<TestTimeLock.State>().single().validFrom)
        }
    }
    ...
}
```

We can then set up an encumbered state:

Kotlin

```
val encumberedState = Cash.State(amount = 1000.DOLLARS `issued by` defaultIssuer,
owner = DUMMY_PUBKEY_1, encumbrance = 1)
val fourPmTimelock = TestTimeLock.State(Instant.parse("2015-04-17T16:00:00.00Z"))
```

When we construct a transaction that generates the encumbered state, we must place the encumbrance in the corresponding output position of that transaction. And when we subsequently consume that encumbered state, the same encumbrance state must be available somewhere within the input set of states.

In future, we will consider the concept of a *covenant*. This is where the encumbrance travels alongside each iteration of the encumbered state. For example, a cash state may be encumbered with a *domicile* encumbrance, which checks the domicile of the identity of the owner that the cash state is being moved to, in order to uphold sanction screening regulations, and prevent cash being paid to parties domiciled in e.g. North Korea. In this case, the encumbrance should be permanently attached to the all future cash states stemming from this one.

We will also consider marking states that are capable of being encumbrances as such. This will prevent states being used as encumbrances inadvertently. For example, the time-lock above would be usable as an encumbrance, but it makes no sense to be able to encumber a cash state with another one.

[Next](#) [Previous](#)

- Writing a contract test
- [View page source](#)

Writing a contract test

This tutorial will take you through the steps required to write a contract test using Kotlin and Java.

The testing DSL allows one to define a piece of the ledger with transactions referring to each other, and ways of verifying their correctness.

Testing single transactions

We start with the empty ledger:

```
KotlinJava
class CommercialPaperTest{
    @Test
    fun emptyLedger() {
        ledger {
        }
    }
    ...
}
```

The DSL keyword `ledger` takes a closure that can build up several transactions and may verify their overall correctness. A ledger is effectively a fresh world with no pre-existing transactions or services within it.

We will start with defining helper function that returns a `CommercialPaper` state:

```
KotlinJava
private val ledgerServices = MockServices(
    // A List of packages to scan for cordapps
    cordappPackages = listOf("net.corda.finance.contracts"),
    // The identity represented by this set of mock services. Defaults to a test
    identity.
    // You can also use the alternative parameter initialIdentityName which
    accepts a
    // [CordaX500Name]
    initialIdentity = megaCorp,
    // An implementation of IdentityService, which contains a list of all
    identities known
    // to the node. Use [makeTestIdentityService] which returns an implementation
    of
    // [InMemoryIdentityService] with the given identities
    identityService = makeTestIdentityService(megaCorp.identity)
)
```

It's a `CommercialPaper` issued by `MEGA_CORP` with face value of \$1000 and maturity date in 7 days.

Let's add a `CommercialPaper` transaction:

```
KotlinJava
@Test
fun simpleCPDoesntCompile() {
    val inState = getPaper()
    ledger {
        transaction {
            input(CommercialPaper.CP_PROGRAM_ID) { inState }
        }
    }
}
```

We can add a transaction to the ledger using the `transaction` primitive. The transaction in turn may be defined by specifying `input`-S, `output`-S, `command`-S and `attachment`-S.

The above `input` call is a bit special; transactions don't actually contain input states, just references to output states of other transactions. Under the hood the above `input` call creates a dummy transaction in the ledger (that won't be verified) which outputs the specified state, and references that from this transaction.

The above code however doesn't compile:

```
KotlinJava
Error:(29, 17) Kotlin: Type mismatch: inferred type is Unit but EnforceVerifyOrFail
was expected
```

This is deliberate: The DSL forces us to specify either `verifies()` or ``fails with`("some text")` on the last line of `transaction`:

```
KotlinJava
// This example test will fail with this exception.
@Test(expected = IllegalStateException::class)
fun simpleCP() {
    val inState = getPaper()
    ledgerServices.ledger(dummyNotary.party) {
        transaction {
            attachments(CP_PROGRAM_ID)
            input(CP_PROGRAM_ID, inState)
            verifies()
        }
    }
}
```

Let's take a look at a transaction that fails.

```
KotlinJava
// This example test will fail with this exception.
@Test(expected = TransactionVerificationException.ContractRejection::class)
```

```

fun simpleCPMove() {
    val inState = getPaper()
    ledgerServices.ledger(dummyNotary.party) {
        transaction {
            input(CP_PROGRAM_ID, inState)
            command(megaCorp.publicKey, CommercialPaper.Commands.Move())
            attachments(CP_PROGRAM_ID)
            verifies()
        }
    }
}

```

When run, that code produces the following error:

KotlinJava

```

net.corda.core.contracts.TransactionVerificationException$ContractRejection:
java.lang.IllegalArgumentException: Failed requirement: the state is propagated

```

The transaction verification failed, because we wanted to move paper but didn't specify an output - but the state should be propagated. However we can specify that this is an intended behaviour by changing `verifies()` to ``fails with`("the state is propagated")`:

KotlinJava

```

@Test
fun simpleCPMoveFails() {
    val inState = getPaper()
    ledgerServices.ledger(dummyNotary.party) {
        transaction {
            input(CP_PROGRAM_ID, inState)
            command(megaCorp.publicKey, CommercialPaper.Commands.Move())
            attachments(CP_PROGRAM_ID)
            `fails with`("the state is propagated")
        }
    }
}

```

We can continue to build the transaction until it `verifies`:

KotlinJava

```

@Test
fun simpleCPMoveFailureAndSuccess() {
    val inState = getPaper()
    ledgerServices.ledger(dummyNotary.party) {
        transaction {
            input(CP_PROGRAM_ID, inState)
            command(megaCorp.publicKey, CommercialPaper.Commands.Move())
            attachments(CP_PROGRAM_ID)
            `fails with`("the state is propagated")
            output(CP_PROGRAM_ID, "alice's paper", inState.withOwner(alice.party))
            verifies()
        }
    }
}

```

`output` specifies that we want the input state to be transferred to `ALICE` and `command` adds the `Move` command itself, signed by the current owner of the input state, `MEGA_CORP_PUBKEY`.

We constructed a complete signed commercial paper transaction and verified it. Note how we left in the `fails with` line - this is fine, the failure will be tested on the partially constructed transaction.

What should we do if we wanted to test what happens when the wrong party signs the transaction? If we simply add a `command` it will permanently ruin the transaction... Enter `tweak`:

KotlinJava

```
@Test
fun `simple issuance with tweak`() {
    ledgerServices.ledger(dummyNotary.party) {
        transaction {
            output(CP_PROGRAM_ID, "paper", getPaper()) // Some CP is issued onto the
ledger by MegaCorp.
            attachments(CP_PROGRAM_ID)
            tweak {
                // The wrong pubkey.
                command(bigCorp.publicKey, CommercialPaper.Commands.Issue())
                timeWindow(TEST_TX_TIME)
                `fails with`("output states are issued by a command signer")
            }
            command(megaCorp.publicKey, CommercialPaper.Commands.Issue())
            timeWindow(TEST_TX_TIME)
            verifies()
        }
    }
}
```

`tweak` creates a local copy of the transaction. This makes possible to locally “ruin” the transaction while not modifying the original one, allowing testing of different error conditions.

We now have a neat little test that tests a single transaction. This is already useful, and in fact testing of a single transaction in this way is very common. There is even a shorthand top-level `transaction` primitive that creates a ledger with a single transaction:

KotlinJava

```
@Test
fun `simple issuance with tweak and top level transaction`() {
    ledgerServices.transaction(dummyNotary.party) {
        output(CP_PROGRAM_ID, "paper", getPaper()) // Some CP is issued onto the
ledger by MegaCorp.
        attachments(CP_PROGRAM_ID)
        tweak {
```

```

        // The wrong pubkey.
        command(bigCorp.publicKey, CommercialPaper.Commands.Issue())
        timeWindow(TEST_TX_TIME)
        `fails with`("output states are issued by a command signer")
    }
    command(megaCorp.publicKey, CommercialPaper.Commands.Issue())
    timeWindow(TEST_TX_TIME)
    verifies()
}
}

```

Chaining transactions

Now that we know how to define a single transaction, let's look at how to define a chain of them:

KotlinJava

```

@Test
fun `chain commercial paper`() {
    val issuer = megaCorp.party.ref(123)
    ledgerServices.ledger(dummyNotary.party) {
        unverifiedTransaction {
            attachments(Cash.PROGRAM_ID)
            output(Cash.PROGRAM_ID, "alice's $900", 900.DOLLARS.CASH issuedBy issuer
ownedBy alice.party)
        }

        // Some CP is issued onto the ledger by MegaCorp.
        transaction("Issuance") {
            output(CP_PROGRAM_ID, "paper", getPaper())
            command(megaCorp.publicKey, CommercialPaper.Commands.Issue())
            attachments(CP_PROGRAM_ID)
            timeWindow(TEST_TX_TIME)
            verifies()
        }

        transaction("Trade") {
            input("paper")
            input("alice's $900")
            output(Cash.PROGRAM_ID, "borrowed $900", 900.DOLLARS.CASH issuedBy issuer
ownedBy megaCorp.party)
            output(CP_PROGRAM_ID, "alice's paper",
"paper".output<ICommercialPaperState>().withOwner(alice.party))
            command(alice.publicKey, Cash.Commands.Move())
            command(megaCorp.publicKey, CommercialPaper.Commands.Move())
            verifies()
        }
    }
}

```

In this example we declare that `ALICE` has \$900 but we don't care where from. For this we can use `unverifiedTransaction`. Note how we don't need to specify `verifies()`.

Notice that we labelled output with `"alice's $900"`, also in transaction named `"Issuance"` we labelled a commercial paper with `"paper"`. Now we can

subsequently refer to them in other transactions, e.g.

by `input("alice's $900")` or `"paper".output<IClaimableState>()`.

The last transaction named `"Trade"` exemplifies simple fact of selling the `CommercialPaper` to Alice for her \$900, \$100 less than the face value at 10% interest after only 7 days.

We can also test whole ledger calling `verifies()` and `fails()` on the ledger level. To do so let's create a simple example that uses the same input twice:

KotlinJava

```
@Test
fun `chain commercial paper double spend`() {
    val issuer = megaCorp.party.ref(123)
    ledgerServices.ledger(dummyNotary.party) {
        unverifiedTransaction {
            attachments(Cash.PROGRAM_ID)
            output(Cash.PROGRAM_ID, "alice's $900", 900.DOLLARS.CASH issuedBy issuer
ownedBy alice.party)
        }

        // Some CP is issued onto the ledger by MegaCorp.
        transaction("Issuance") {
            output(CP_PROGRAM_ID, "paper", getPaper())
            command(megaCorp.publicKey, CommercialPaper.Commands.Issue())
            attachments(CP_PROGRAM_ID)
            timeWindow(TEST_TX_TIME)
            verifies()
        }

        transaction("Trade") {
            input("paper")
            input("alice's $900")
            output(Cash.PROGRAM_ID, "borrowed $900", 900.DOLLARS.CASH issuedBy issuer
ownedBy megaCorp.party)
            output(CP_PROGRAM_ID, "alice's paper",
"paper".output<IClaimableState>().withOwner(alice.party))
            command(alice.publicKey, Cash.Commands.Move())
            command(megaCorp.publicKey, CommercialPaper.Commands.Move())
            verifies()
        }

        transaction {
            input("paper")
            // We moved a paper to another pubkey.
            output(CP_PROGRAM_ID, "bob's paper",
"paper".output<IClaimableState>().withOwner(bob.party))
            command(megaCorp.publicKey, CommercialPaper.Commands.Move())
            verifies()
        }

        fails()
    }
}
```

The transactions `verifies()` individually, however the state was spent twice! That's why we need the global ledger verification (`fails()` at the end). As in previous examples we can use `tweak` to create a local copy of the whole ledger:

KotlinJava

```
@Test
fun `chain commercial tweak`() {
    val issuer = megaCorp.party.ref(123)
    ledgerServices.ledger(dummyNotary.party) {
        unverifiedTransaction {
            attachments(Cash.PROGRAM_ID)
            output(Cash.PROGRAM_ID, "alice's $900", 900.DOLLARS.CASH issuedBy issuer
ownedBy alice.party)
        }

        // Some CP is issued onto the ledger by MegaCorp.
        transaction("Issuance") {
            output(CP_PROGRAM_ID, "paper", getPaper())
            command(megaCorp.publicKey, CommercialPaper.Commands.Issue())
            attachments(CP_PROGRAM_ID)
            timeWindow(TEST_TX_TIME)
            verifies()
        }

        transaction("Trade") {
            input("paper")
            input("alice's $900")
            output(Cash.PROGRAM_ID, "borrowed $900", 900.DOLLARS.CASH issuedBy issuer
ownedBy megaCorp.party)
            output(CP_PROGRAM_ID, "alice's paper",
"paper".output<ICommercialPaperState>().withOwner(alice.party))
            command(alice.publicKey, Cash.Commands.Move())
            command(megaCorp.publicKey, CommercialPaper.Commands.Move())
            verifies()
        }

        tweak {
            transaction {
                input("paper")
                // We moved a paper to another pubkey.
                output(CP_PROGRAM_ID, "bob's paper",
"paper".output<ICommercialPaperState>().withOwner(bob.party))
                command(megaCorp.publicKey, CommercialPaper.Commands.Move())
                verifies()
            }
            fails()
        }
        verifies()
    }
}
```

[Next](#) [Previous](#)

- [Upgrading contracts](#)
 - [View page source](#)
-

Upgrading contracts

While every care is taken in development of contract code, inevitably upgrades will be required to fix bugs (in either design or implementation). Upgrades can involve a substitution of one version of the contract code for another or changing to a different contract that understands how to migrate the existing state objects. When state objects are added as outputs to transactions, they are linked to the contract code they are intended for via the `StateAndContract` type. Changing a state's contract only requires substituting one `ContractClassName` for another.

Workflow

Here's the workflow for contract upgrades:

1. Banks A and B negotiate a trade, off-platform
2. Banks A and B execute a flow to construct a state object representing the trade, using contract X, and include it in a transaction (which is then signed and sent to the consensus service)
3. Time passes
4. The developer of contract X discovers a bug in the contract code, and releases a new version, contract Y. The developer will then notify all existing users (e.g. via a mailing list or CorDapp store) to stop their nodes from issuing further states with contract X
5. Banks A and B review the new contract via standard change control processes and identify the contract states they agree to upgrade (they may decide not to upgrade some contract states as these might be needed for some other obligation contract)
6. Banks A and B instruct their Corda nodes (via RPC) to be willing to upgrade state objects with contract X to state objects with contract Y using the agreed upgrade path
7. One of the parties (the `Initiator`) initiates a flow to replace state objects referring to contract X with new state objects referring to contract Y
8. A proposed transaction (the `Proposal`), with the old states as input and the reissued states as outputs, is created and signed with the node's private key
9. The `Initiator` node sends the proposed transaction, along with details of the new contract upgrade path that it is proposing, to all participants of the state object

10. Each counterparty (the ``Acceptor``'s) verifies the proposal, signs or rejects the state reissuance accordingly, and sends a signature or rejection notification back to the initiating node
11. If signatures are received from all parties, the `Initiator` assembles the complete signed transaction and sends it to the notary

Authorising an upgrade

Each of the participants in the state for which the contract is being upgraded will have to instruct their node that they agree to the upgrade before the upgrade can take place. The `ContractUpgradeFlow` is used to manage the authorisation process. Each node administrator can use RPC to trigger either an `Authorise` or a `Deauthorise` flow for the state in question.

```
@StartableByRPC
class Authorise(
    val stateAndRef: StateAndRef<*>,
    private val upgradedContractClass: Class<out UpgradedContract<*, *>>
) : FlowLogic<Void?>() {

@StartableByRPC
class Deauthorise(val stateRef: StateRef) : FlowLogic<Void?>() {
    @Suspendable
    override fun call(): Void? {
```

Proposing an upgrade

After all parties have authorised the contract upgrade for the state, one of the contract participants can initiate the upgrade process by triggering the `ContractUpgradeFlow.Initiate` flow. `Initiate` creates a transaction including the old state and the updated state, and sends it to each of the participants. Each participant will verify the transaction, create a signature over it, and send the signature back to the initiator. Once all the signatures are collected, the transaction will be notarised and persisted to every participant's vault.

Example

Suppose Bank A has entered into an agreement with Bank B which is represented by the state object `DummyContractState` and governed by the contract code `DummyContract`. A few days after the exchange of contracts, the developer of the contract code discovers a bug in the contract code.

Bank A and Bank B decide to upgrade the contract to `DummyContractV2`:

- The developer creates a new contract `DummyContractV2` extending the `UpgradedContract` class, and a new state object `DummyContractV2.State` referencing the new contract.

```
class DummyContractV2 : UpgradedContractWithLegacyConstraint<DummyContract.State, DummyContractV2.State> {
    companion object {
        const val PROGRAM_ID: ContractClassName =
            "net.corda.testing.contracts.DummyContractV2"
    }

    override val legacyContract: String = DummyContract::class.java.name
    override val legacyContractConstraint: AttachmentConstraint =
        AlwaysAcceptAttachmentConstraint

    data class State(val magicNumber: Int = 0, val owners: List<AbstractParty>) : ContractState {
        override val participants: List<AbstractParty> = owners
    }

    interface Commands : CommandData {
        class Create : TypeOnlyCommandData(), Commands
        class Move : TypeOnlyCommandData(), Commands
    }

    override fun upgrade(state: DummyContract.State): State {
        return State(state.magicNumber, state.participants)
    }

    override fun verify(tx: LedgerTransaction) {
        // Other verifications.
    }
}
```

- Bank A instructs its node to accept the contract upgrade to `DummyContractV2` for the contract state.

Kotlin

```
val rpcClient : CordaRPCClient = << Bank A's Corda RPC Client >>
val rpcA = rpcClient.proxy()
rpcA.startFlow(ContractUpgradeFlow.Authorise(<<StateAndRef of the contract state>>, DummyContractV2::class.java))
```

- Bank B initiates the upgrade flow, which will send an upgrade proposal to all contract participants. Each of the participants of the contract state will sign and return the contract state upgrade proposal once they have validated and agreed with the upgrade. The upgraded transaction will be recorded in every participant's node by the flow.

Kotlin

```
val rpcClient : CordaRPCClient = << Bank B's Corda RPC Client >>
val rpcB = rpcClient.proxy()
rpcB.startFlow({ stateAndRef, upgrade -> ContractUpgradeFlow(stateAndRef, upgrade) },
    <<StateAndRef of the contract state>>, DummyContractV2::class.java)
```

Note

See [ContractUpgradeFlowTest](#) for more detailed code examples.

[Next](#) [Previous](#)

- »

- [Integration testing](#)
 - [View page source](#)
-

Integration testing

Integration testing involves bringing up nodes locally and testing invariants about them by starting flows and inspecting their state.

In this tutorial we will bring up three nodes - Alice, Bob and a notary. Alice will issue cash to Bob, then Bob will send this cash back to Alice. We will see how to test some simple deterministic and nondeterministic invariants in the meantime.

Note

This example where Alice is self-issuing cash is purely for demonstration purposes, in reality, cash would be issued by a bank and subsequently passed around.

In order to spawn nodes we will use the Driver DSL. This DSL allows one to start up node processes from code. It manages a network map service and safe shutting down of nodes in the background.

```
driver(DriverParameters(startNodesInProcess = true,
    extraCordappPackagesToScan = listOf("net.corda.finance.contracts.asset",
"net.corda.finance.schemas")))
    val aliceUser = User("aliceUser", "testPassword1", permissions = setOf(
        startFlow<CashIssueFlow>(),
        startFlow<CashPaymentFlow>(),
        invokeRpc("vaultTrackBy"),
        invokeRpc(CordaRPCOps::notaryIdentities),
        invokeRpc(CordaRPCOps::networkMapFeed)
    ))
    val bobUser = User("bobUser", "testPassword2", permissions = setOf(
        startFlow<CashPaymentFlow>(),
        invokeRpc("vaultTrackBy"),
        invokeRpc(CordaRPCOps::networkMapFeed)
    ))
    val (alice, bob) = listOf(
        startNode(providedName = ALICE_NAME, rpcUsers = listOf(aliceUser)),
```

```
    startNode(providedName = BOB_NAME, rpcUsers = listOf(bobUser))
).transpose().getOrThrow()
```

The above code starts three nodes:

- Alice, who has user permissions to start the `CashIssueFlow` and `CashPaymentFlow` flows
- Bob, who only has user permissions to start the `CashPaymentFlow`
- A notary that offers a `ValidatingNotaryService`. We won't connect to the notary directly, so there's no need to provide a `User`

The `startNode` function returns a future that completes once the node is fully started. This allows starting of the nodes to be parallel. We wait on these futures as we need the information returned; their respective `NodeHandles`s.

```
val aliceClient = CordaRPCClient(alice.rpcAddress)
val aliceProxy = aliceClient.start("aliceUser", "testPassword1").proxy

val bobClient = CordaRPCClient(bob.rpcAddress)
val bobProxy = bobClient.start("bobUser", "testPassword2").proxy
```

After getting the handles we wait for both parties to register with the network map to ensure we don't have race conditions with network map registration. Next we connect to Alice and Bob respectively from the test process using the test user we created. Then we establish RPC links that allow us to start flows and query state.

```
val bobVaultUpdates = bobProxy.vaultTrackBy<Cash.State>().updates
val aliceVaultUpdates = aliceProxy.vaultTrackBy<Cash.State>().updates
```

We will be interested in changes to Alice's and Bob's vault, so we query a stream of vault updates from each.

Now that we're all set up we can finally get some cash action going!

```
val issueRef = OpaqueBytes.of(0)
val notaryParty = aliceProxy.notaryIdentities().first()
(1..10).map { i ->
    aliceProxy.startFlow(::CashIssueFlow,
        i.DOLLARS,
        issueRef,
        notaryParty
    ).returnValue
}.transpose().getOrThrow()
// We wait for all of the issuances to run before we start making payments
(1..10).map { i ->
    aliceProxy.startFlow(::CashPaymentFlow,
        i.DOLLARS,
        bob.nodeInfo.singleIdentity(),
```

```

        true
    ).returnValue
}.transpose().getOrThrow()

bobVaultUpdates.expectEvents {
    parallel(
        (1..10).map { i ->
            expect(
                match = { update: Vault.Update<Cash.State> ->
                    update.produced.first().state.data.amount.quantity == i *
100L
                }
            ) { update ->
                println("Bob vault update of $update")
            }
        }
    )
}

```

The first loop creates 10 threads, each starting a `CashFlow` flow on the Alice node. We specify that we want to issue `i` dollars to Bob, setting our notary as the notary responsible for notarising the created states. Note that no notarisation will occur yet as we're not spending any states, only creating new ones on the ledger.

We started the flows from different threads for the sake of the tutorial, to demonstrate how to test non-determinism, which is what the `expectEvents` block does.

The Expect DSL allows ordering constraints to be checked on a stream of events. The above code specifies that we are expecting 10 updates to be emitted on the `bobVaultUpdates` stream in unspecified order (this is what the `parallel` construct does). We specify an (otherwise optional) `match` predicate to identify specific updates we are interested in, which we then print.

If we run the code written so far we should see 4 nodes starting up (Alice, Bob, the notary and an implicit Network Map service), then 10 logs of Bob receiving 1,2,...10 dollars from Alice in some unspecified order.

Next we want Bob to send this cash back to Alice.

```

for (i in 1..10) {
    bobProxy.startFlow(::CashPaymentFlow, i.DOLLARS,
alice.nodeInfo.singleIdentity()).returnValue.getOrThrow()
}

aliceVaultUpdates.expectEvents {
    sequence(
        (1..10).map { i ->
            expect { update: Vault.Update<Cash.State> ->

```

```

        println("Alice got vault update of $update")
        assertEquals(update.produced.first().state.data.amount.quantity, i
* 100L)
    }
}
}

```

This time we'll do it sequentially. We make Bob pay 1,2,..10 dollars to Alice in order. We make sure that a the `CashFlow` has finished by waiting on `startFlow`'s `returnValue`.

Then we use the Expect DSL again, this time using `sequence` to test for the updates arriving in the order we expect them to.

Note that `parallel` and `sequence` may be nested into each other arbitrarily to test more complex scenarios.

That's it! We saw how to start up several corda nodes locally, how to connect to them, and how to test some simple invariants about `CashFlow`.

To run the complete test you can open `example-code/src/integration-test/kotlin/net/corda/docs/IntegrationTestingTutorial.kt` from IntelliJ and run the test, or alternatively use gradle:

```
# Run example-code integration tests
./gradlew docs/source/example-code:integrationTest -i
```

[Next](#) [Previous](#)

- Using the client RPC API
 - [View page source](#)
-

Using the client RPC API

In this tutorial we will build a simple command line utility that connects to a node, creates some cash transactions and dumps the transaction graph to the standard output. We will then put some simple visualisation on top. For an explanation on how RPC works in Corda see [Client RPC](#).

We start off by connecting to the node itself. For the purposes of the tutorial we will use the Driver to start up a notary and a Alice node that can issue, move and exit cash.

Here's how we configure the node to create a user that has the permissions to start the `CashIssueFlow`, `CashPaymentFlow`, and `CashExitFlow`:

```
enum class PrintOrVisualise {
    Print,
    Visualise
}

fun main(args: Array<String>) {
    require(args.isNotEmpty()) { "Usage: <binary> [Print|Visualise]" }
    val printOrVisualise = PrintOrVisualise.valueOf(args[0])

    val baseDirectory = Paths.get("build/rpc-api-tutorial")
    val user = User("user", "password", permissions =
setOf(startFlow<CashIssueFlow>(),
      startFlow<CashPaymentFlow>(),
      startFlow<CashExitFlow>(),
      invokeRpc(CordaRPCOps::nodeInfo)
))
    driver(DriverParameters(driverDirectory = baseDirectory,
extraCordappPackagesToScan = listOf("net.corda.finance"), waitForAllNodesToFinish =
true)) {
        val node = startNode(providedName = ALICE_NAME, rpcUsers = listOf(user)).get()
    }
}
```

Now we can connect to the node itself using a valid RPC user login and start generating transactions in a different thread using `generateTransactions` (to be defined later):

```
val client = CordaRPCClient(node.rpcAddress)
val proxy = client.start("user", "password").proxy

thread {
    generateTransactions(proxy)
}
```

`proxy` exposes the full RPC interface of the node:

```
/**
 * Returns the RPC protocol version, which is the same the node's Platform
Version. Exists since version 1 so guaranteed
 * to be present.
 */
override val protocolVersion: Int get() = nodeInfo().platformVersion

/** Returns a list of currently in-progress state machine infos. */
fun stateMachinesSnapshot(): List<StateMachineInfo>

/**
 * Returns a data feed of currently in-progress state machine infos and an
observable of
 * future state machine adds/removes.
 */
@RPCReturnsObservables
```

```

    fun stateMachinesFeed(): DataFeed<List<StateMachineInfo>, StateMachineUpdate>

    /**
     * Returns a snapshot of vault states for a given query criteria (and optional
     * order and paging specification)
     *
     * Generic vault query function which takes a [QueryCriteria] object to define
     * filters,
     * optional [PageSpecification] and optional [Sort] modification criteria (default
     * unsorted),
     * and returns a [Vault.Page] object containing the following:
     * 1. states as a List of <StateAndRef> (page number and size defined by
     * [PageSpecification])
     * 2. states metadata as a List of [Vault.StateMetadata] held in the Vault States
     * table.
     * 3. total number of results available if [PageSpecification] supplied
     * (otherwise returns -1)
     * 4. status types used in this query: UNCONSUMED, CONSUMED, ALL
     * 5. other results (aggregate functions with/without using value groups)
     *
     * @throws VaultQueryException if the query cannot be executed for any reason
     * (missing criteria or parsing error, paging errors, unsupported query,
     * underlying database error)
     *
     * Notes
     * If no [PageSpecification] is provided, a maximum of [DEFAULT_PAGE_SIZE]
     * results will be returned.
     * API users must specify a [PageSpecification] if they are expecting more than
     * [DEFAULT_PAGE_SIZE] results,
     * otherwise a [VaultQueryException] will be thrown alerting to this condition.
     * It is the responsibility of the API user to request further pages and/or
     * specify a more suitable [PageSpecification].
     */
    // DOCSTART VaultQueryByAPI
    @RPCReturnsObservables
    fun <T : ContractState> vaultQueryBy(criteria: QueryCriteria,
                                             paging: PageSpecification,
                                             sorting: Sort,
                                             contractStateType: Class<out T>):
    Vault.Page<T>
    // DOCEND VaultQueryByAPI

    // Note: cannot apply @JvmOverloads to interfaces nor interface implementations
    // Java Helpers

    // DOCSTART VaultQueryAPIHelpers
    fun <T : ContractState> vaultQuery(contractStateType: Class<out T>): Vault.Page<T>

    fun <T : ContractState> vaultQueryByCriteria(criteria: QueryCriteria,
                                                 contractStateType: Class<out T>): Vault.Page<T>

    fun <T : ContractState> vaultQueryByWithPagingSpec(contractStateType: Class<out T>,
                                                       criteria: QueryCriteria, paging: PageSpecification): Vault.Page<T>

    fun <T : ContractState> vaultQueryByWithSorting(contractStateType: Class<out T>,
                                                   criteria: QueryCriteria, sorting: Sort): Vault.Page<T>
    // DOCEND VaultQueryAPIHelpers

    /**
     * Returns a snapshot (as per queryBy) and an observable of future updates to the
     * vault for the given query criteria.
     *
     * Generic vault query function which takes a [QueryCriteria] object to define
     * filters,
     * optional [PageSpecification] and optional [Sort] modification criteria (default
     * unsorted),

```

```

        * and returns a [DataFeed] object containing
        * 1) a snapshot as a [Vault.Page] (described previously in
[CordaRPCOps.vaultQueryBy])
        * 2) an [Observable] of [Vault.Update]
        *
        * Notes: the snapshot part of the query adheres to the same behaviour as the
[CordaRPCOps.vaultQueryBy] function.
        *          the [QueryCriteria] applies to both snapshot and deltas (streaming
updates).
        */
    // DOCSTART VaultTrackByAPI
    @RPCReturnsObservables
    fun <T : ContractState> vaultTrackBy(criteria: QueryCriteria,
                                            paging: PageSpecification,
                                            sorting: Sort,
                                            contractStateType: Class<out T>):
DataFeed<Vault.Page<T>, Vault.Update<T>>
    // DOCEND VaultTrackByAPI

    // Note: cannot apply @JvmOverloads to interfaces nor interface implementations
    // Java Helpers

    // DOCSTART VaultTrackAPIHelpers
    fun <T : ContractState> vaultTrack(contractStateType: Class<out T>):
DataFeed<Vault.Page<T>, Vault.Update<T>>

        fun <T : ContractState> vaultTrackByCriteria(contractStateType: Class<out T>,
criteria: QueryCriteria): DataFeed<Vault.Page<T>, Vault.Update<T>>

        fun <T : ContractState> vaultTrackByWithPagingSpec(contractStateType: Class<out T>,
criteria: QueryCriteria, paging: PageSpecification): DataFeed<Vault.Page<T>,
Vault.Update<T>>

        fun <T : ContractState> vaultTrackByWithSorting(contractStateType: Class<out T>,
criteria: QueryCriteria, sorting: Sort): DataFeed<Vault.Page<T>, Vault.Update<T>>
    // DOCEND VaultTrackAPIHelpers

    /**
     * @suppress Returns a list of all recorded transactions.
     *
     * TODO This method should be removed once SGX work is finalised and the design of
the corresponding API using [FilteredTransaction] can be started
     */
    @Deprecated("This method is intended only for internal use and will be removed
from the public API soon.")
    fun internalVerifiedTransactionsSnapshot(): List<SignedTransaction>

    /**
     * @suppress Returns a data feed of all recorded transactions and an observable of
future recorded ones.
     *
     * TODO This method should be removed once SGX work is finalised and the design of
the corresponding API using [FilteredTransaction] can be started
     */
    @Deprecated("This method is intended only for internal use and will be removed
from the public API soon.")
    @RPCReturnsObservables
    fun internalVerifiedTransactionsFeed(): DataFeed<List<SignedTransaction>,
SignedTransaction>

    /** Returns a snapshot list of existing state machine id - recorded transaction
hash mappings. */
    fun stateMachineRecordedTransactionMappingSnapshot():
List<StateMachineTransactionMapping>

    /**

```

```

        * Returns a snapshot list of existing state machine id - recorded transaction
hash mappings, and a stream of future
        * such mappings as well.
    */
    @RPCReturnsObservables
    fun stateMachineRecordedTransactionMappingFeed(): DataFeed<List<StateMachineTransactionMapping>, StateMachineTransactionMapping>

        /** Returns all parties currently visible on the network with their advertised
services. */
    fun networkMapSnapshot(): List<NodeInfo>

    /**
        * Returns all parties currently visible on the network with their advertised
services and an observable of
        * future updates to the network.
    */
    @RPCReturnsObservables
    fun networkMapFeed(): DataFeed<List<NodeInfo>, NetworkMapCache.MapChange>

    /**
        * Returns [DataFeed] object containing information on currently scheduled
parameters update (null if none are currently scheduled)
        * and observable with future update events. Any update that occurs before the
deadline automatically cancels the current one.
        * Only the latest update can be accepted.
        * Note: This operation may be restricted only to node administrators.
    */
    // TODO This operation should be restricted to just node admins.
    @RPCReturnsObservables
    fun networkParametersFeed(): DataFeed<ParametersUpdateInfo?, ParametersUpdateInfo>

    /**
        * Accept network parameters with given hash, hash is obtained through
[networkParametersFeed] method.
        * Information is sent back to the zone operator that the node accepted the
parameters update - this process cannot be
        * undone.
        * Only parameters that are scheduled for update can be accepted, if different
hash is provided this method will fail.
        * Note: This operation may be restricted only to node administrators.
        * @param parametersHash hash of network parameters to accept
        * @throws IllegalArgumentException if network map advertises update with
different parameters hash then the one accepted by node's operator.
        * @throws IOException if failed to send the approval to network map
    */
    // TODO This operation should be restricted to just node admins.
    fun acceptNewNetworkParameters(parametersHash: SecureHash)

    /**
        * Start the given flow with the given arguments. [logicType] must be annotated
        * with [net.corda.core.flows.StartableByRPC].
    */
    @RPCReturnsObservables
    fun <T> startFlowDynamic(logicType: Class<out FlowLogic<T>>, vararg args: Any?): FlowHandle<T>

    /**
        * Start the given flow with the given arguments, returning an [Observable] with a
single observation of the
        * result of running the flow. [logicType] must be annotated with
[net.corda.core.flows.StartableByRPC].
    */
    @RPCReturnsObservables
    fun <T> startTrackedFlowDynamic(logicType: Class<out FlowLogic<T>>, vararg args: Any?): FlowProgressHandle<T>

```

```

    /** Returns Node's NodeInfo, assuming this will not change while the node is
running. */
    fun nodeInfo(): NodeInfo

    /**
     * Returns network's notary identities, assuming this will not change while the
node is running.
     *
     * Note that the identities are sorted based on legal name, and the ordering might
change once new notaries are introduced.
     */
    fun notaryIdentities(): List<Party>

    /** Add note(s) to an existing Vault transaction. */
    fun addVaultTransactionNote(txnId: SecureHash, txnNote: String)

    /** Retrieve existing note(s) for a given Vault transaction. */
    fun getVaultTransactionNotes(txnId: SecureHash): Iterable<String>

    /** Checks whether an attachment with the given hash is stored on the node. */
    fun attachmentExists(id: SecureHash): Boolean

    /** Download an attachment JAR by ID. */
    fun openAttachment(id: SecureHash): InputStream

    /** Uploads a jar to the node, returns it's hash. */
    fun uploadAttachment(jar: InputStream): SecureHash

    /** Uploads a jar including metadata to the node, returns it's hash. */
    fun uploadAttachmentWithMetadata(jar: InputStream, uploader: String, filename:
String): SecureHash

    /** Queries attachments metadata */
    fun queryAttachments(query: AttachmentQueryCriteria, sorting: AttachmentSort?): List<AttachmentId>

    /** Returns the node's current time. */
    fun currentNodeTime(): Instant

    /**
     * Returns a [CordaFuture] which completes when the node has registered with the
network map service. It can also
     * complete with an exception if it is unable to.
     */
    @RPCReturnsObservables
    fun waitUntilNetworkReady(): CordaFuture<Void?>

    // TODO These need rethinking. Instead of these direct calls we should have a way
of replicating a subset of
    // the node's state locally and query that directly.
    /**
     * Returns the well known identity from an abstract party. This is intended to
resolve the well known identity
     * from a confidential identity, however it transparently handles returning the
well known identity back if
     * a well known identity is passed in.
     *
     * @param party identity to determine well known identity for.
     * @return well known identity, if found.
     */
    fun wellKnownPartyFromAnonymous(party: AbstractParty): Party?

    /** Returns the [Party] corresponding to the given key, if found. */
    fun partyFromKey(key: PublicKey): Party?

```

```

/** Returns the [Party] with the X.500 principal as it's [Party.name]. */
fun wellKnownPartyFromX500Name(x500Name: CordaX500Name): Party?

/**
 * Get a notary identity by name.
 *
 * @return the notary identity, or null if there is no notary by that name. Note
that this will return null if there
 * is a peer with that name but they are not a recognised notary service.
 */
fun notaryPartyFromX500Name(x500Name: CordaX500Name): Party?

/**
 * Returns a list of candidate matches for a given string, with optional
fuzzy(ish) matching. Fuzzy matching may
 * get smarter with time e.g. to correct spelling errors, so you should not hard-
code indexes into the results
 * but rather show them via a user interface and let the user pick the one they
wanted.
 *
 * @param query The string to check against the X.500 name components
 * @param exactMatch If true, a case sensitive match is done against each
component of each X.500 name.
 */
fun partiesFromName(query: String, exactMatch: Boolean): Set<Party>

/** Enumerates the class names of the flows that this node knows about. */
fun registeredFlows(): List<String>

/**
 * Returns a node's info from the network map cache, where known.
 * Notice that when there are more than one node for a given name (in case of
distributed services) first service node
 * found will be returned.
 *
 * @return the node info if available.
 */
fun nodeInfoFromParty(party: AbstractParty): NodeInfo?

/** Clear all network map data from local node cache. */
fun clearNetworkMapCache()

/** Sets the value of the node's flows draining mode.
 * If this mode is [enabled], the node will reject new flows through RPC, ignore
scheduled flows, and do not process
 * initial session messages, meaning that P2P counterparties will not be able to
initiate new flows involving the node.
 *
 * @param enabled whether the flows draining mode will be enabled.
 * */
fun setFlowsDrainingModeEnabled(enabled: Boolean)

/**
 * Returns whether the flows draining mode is enabled.
 *
 * @see setFlowsDrainingModeEnabled
 */
fun isFlowsDrainingModeEnabled(): Boolean

```

The RPC operation we need in order to dump the transaction graph is `internalVerifiedTransactionsFeed`. The type signature tells us that the RPC operation will return a list of transactions and an `Observable` stream. This is a general pattern, we query some data and the node will return the current

snapshot and future updates done to it. Observables are described in further detail in [Client RPC](#)

```
val (transactions: List<SignedTransaction>, futureTransactions: Observable<SignedTransaction>) = proxy.internalVerifiedTransactionsFeed()
```

The graph will be defined as follows:

- Each transaction is a vertex, represented by printing `NODE <txhash>`
- Each input-output relationship is an edge, represented by printing `EDGE <txhash> <txhash>`

```
when (printOrVisualise) {  
    PrintOrVisualise.Print -> {  
        futureTransactions.startWith(transactions).subscribe { transaction ->  
            println("NODE ${transaction.id}")  
            transaction.tx.inputs.forEach { (txhash) ->  
                println("EDGE $txhash ${transaction.id}")  
            }  
        }  
    }  
}
```

Now we just need to create the transactions themselves!

```
fun generateTransactions(proxy: CordaRPCOps) {  
    val vault = proxy.vaultQueryBy<Cash.State>().states  
  
    var ownedQuantity = vault.fold(0L) { sum, state ->  
        sum + state.state.data.amount.quantity  
    }  
    val issueRef = OpaqueBytes.of(0)  
    val notary = proxy.notaryIdentities().first()  
    val me = proxy.nodeInfo().legalIdentities.first()  
    while (true) {  
        Thread.sleep(1000)  
        val random = SplittableRandom()  
        val n = random.nextDouble()  
        if (ownedQuantity > 10000 && n > 0.8) {  
            val quantity = Math.abs(random.nextLong()) % 2000  
            proxy.startFlow(::CashExitFlow, Amount(quantity, USD), issueRef)  
            ownedQuantity -= quantity  
        } else if (ownedQuantity > 1000 && n < 0.7) {  
            val quantity = Math.abs(random.nextLong()) % Math.min(ownedQuantity, 2000)  
            proxy.startFlow(::CashPaymentFlow, Amount(quantity, USD), me)  
        } else {  
            val quantity = Math.abs(random.nextLong()) % 1000  
            proxy.startFlow(::CashIssueFlow, Amount(quantity, USD), issueRef, notary)  
            ownedQuantity += quantity  
        }  
    }  
}
```

We utilise several RPC functions here to query things like the notaries in the node cluster or our own vault. These RPC functions also return `Observable` objects so that the node can send us updated values. However, we don't need updates here and so we mark these observables as `notUsed` (as a

rule, you should always either subscribe to an `Observable` or mark it as not used. Failing to do so will leak resources in the node).

Then in a loop we generate randomly either an Issue, a Pay or an Exit transaction.

The RPC we need to initiate a cash transaction is `startFlow` which starts an arbitrary flow given sufficient permissions to do so.

Finally we have everything in place: we start a couple of nodes, connect to them, and start creating transactions while listening on successfully created ones, which are dumped to the console. We just need to run it!

```
# Build the example
./gradlew docs/source/example-code:installDist
# Start it
./docs/source/example-code/build/install/docs/source/example-code/bin/client-rpc-
tutorial Print
```

Now let's try to visualise the transaction graph. We will use a graph drawing library called [graphstream](#).

```
PrintOrVisualise.Visualise -> {
    val graph = MultiGraph("transactions")
    transactions.forEach { transaction ->
        graph.addNode<Node>("${transaction.id}")
    }
    transactions.forEach { transaction ->
        transaction.tx.inputs.forEach { ref ->
            graph.addEdge<Edge>("$ref", "${ref.txhash}", "${transaction.id}")
        }
    }
    futureTransactions.subscribe { transaction ->
        graph.addNode<Node>("${transaction.id}")
        transaction.tx.inputs.forEach { ref ->
            graph.addEdge<Edge>("$ref", "${ref.txhash}", "${transaction.id}")
        }
    }
    graph.display()
}
```

If we run the client with `Visualise` we should see a simple random graph being drawn as new transactions are being created.

Whitelisting classes from your CorDapp with the Corda node

As described in [Client RPC](#), you have to whitelist any additional classes you add that are needed in RPC requests or responses with the Corda node. Here's an example of both ways you can do this for a couple of example classes.

```
// Not annotated, so need to whitelist manually.  
data class ExampleRPCValue(val foo: String)  
  
// Annotated, so no need to whitelist manually.  
@CordaSerializable  
data class ExampleRPCValue2(val bar: Int)  
  
class ExampleRPCSerializationWhitelist : SerializationWhitelist {  
    // Add classes like this.  
    override val whitelist = listOf(ExampleRPCValue::class.java)  
}
```

See more on plugins in [Running nodes locally](#).

Security

RPC credentials associated with a Client must match the permission set configured on the server node. This refers to both authentication (username and password) and role-based authorisation (a permissioned set of RPC operations an authenticated user is entitled to run).

Note

Permissions are represented as *String*'s to allow RPC implementations to add their own permissioning. Currently the only permission type defined is *StartFlow*, which defines a list of whitelisted flows an authenticated user may execute. An administrator user (or a developer) may also be assigned the *ALL* permission, which grants access to any flow.

In the instructions above the server node permissions are configured programmatically in the driver code:

```
driver(driverDirectory = baseDirectory) {  
    val user = User("user", "password", permissions = setOf(startFlow<CashFlow>()))  
    val node = startNode("CN=Alice Corp,O=Alice Corp,L=London,C=GB", rpcUsers =  
listOf(user)).get()
```

When starting a standalone node using a configuration file we must supply the RPC credentials as follows:

```
rpcUsers : [  
    { username=user, password=password, permissions=[  
        StartFlow.net.corda.finance.flows.CashFlow ] }  
]
```

When using the gradle Cordformation plugin to configure and deploy a node you must supply the RPC credentials in a similar manner:

```
rpcUsers = [
    ['username' : "user",
     'password' : "password",
     'permissions' : ["StartFlow.net.corda.finance.flows.CashFlow"]]
]
```

You can then deploy and launch the nodes (Notary and Alice) as follows:

```
# to create a set of configs and installs under ``docs/source/example-
code/build/nodes`` run
./gradlew docs/source/example-code:deployNodes
# to open up two new terminals with the two nodes run
./docs/source/example-code/build/nodes/runnodes
# followed by the same commands as before:
./docs/source/example-code/build/install/docs/source/example-code/bin/client-rpc-
tutorial Print
./docs/source/example-code/build/install/docs/source/example-code/bin/client-rpc-
tutorial Visualise
```

With regards to the start flow RPCs, there is an extra layer of security whereby the flow to be executed has to be annotated with `@StartableByRPC`. Flows without this annotation cannot execute using RPC.

See more on security in [Secure coding guidelines](#), [node configuration in Node configuration](#) and [Cordformation in Running nodes locally](#).

[Next](#) [Previous](#)

- Building transactions
 - [View page source](#)
-

Building transactions

Introduction

Understanding and implementing transactions in Corda is key to building and implementing real world smart contracts. It is only through construction of valid Corda transactions containing appropriate data that nodes on the ledger can map real world business objects into a shared digital view of the data in the Corda ledger. More importantly as the developer of new smart contracts it is the code which determines what data is well formed and what data should be rejected as mistakes, or to prevent malicious activity. This document details

some of the considerations and APIs used to when constructing transactions as part of a flow.

The Basic Lifecycle Of Transactions

Transactions in Corda contain a number of elements:

1. A set of Input state references that will be consumed by the final accepted transaction
2. A set of Output states to create/replace the consumed states and thus become the new latest versions of data on the ledger
3. A set of `Attachment` items which can contain legal documents, contract code, or private encrypted sections as an extension beyond the native contract states
4. A set of `Command` items which indicate the type of ledger transition that is encoded in the transaction. Each command also has an associated set of signer keys, which will be required to sign the transaction
5. A signers list, which is the union of the signers on the individual Command objects
6. A notary identity to specify which notary node is tracking the state consumption (if the transaction's input states are registered with different notary nodes the flow will have to insert additional `NotaryChange` transactions to migrate the states across to a consistent notary node before being allowed to mutate any states)
7. Optionally a timestamp that can be used by the notary to bound the period during which the proposed transaction can be committed to the ledger

A transaction is built by populating a `TransactionBuilder`. Typically, the `TransactionBuilder` will need to be exchanged back and forth between parties before it is fully populated. This is an immediate consequence of the Corda privacy model, in which the input states are likely to be unknown to the other node.

Once the builder is fully populated, the flow should freeze the `TransactionBuilder` by signing it to create a `SignedTransaction`. This is key to the ledger agreement process - once a flow has attached a node's signature to a transaction, it has effectively stated that it accepts all the details of the transaction.

It is best practice for flows to receive back the `TransactionSignature` of other parties rather than a full `SignedTransaction` objects, because otherwise we have to separately check that this is still the same `SignedTransaction` and not a malicious substitute.

The final stage of committing the transaction to the ledger is to notarise the `SignedTransaction`, distribute it to all appropriate parties and record the data into the ledger. These actions are best delegated to the `FinalityFlow`, rather than calling the individual steps manually. However, do note that the final broadcast to the other nodes is asynchronous, so care must be used in unit testing to correctly await the vault updates.

Gathering Inputs

One of the first steps to forming a transaction is gathering the set of input references. This process will clearly vary according to the nature of the business process being captured by the smart contract and the parameterised details of the request. However, it will generally involve searching the vault via the `VaultService` interface on the `ServiceHub` to locate the input states.

To give a few more specific details consider two simplified real world scenarios. First, a basic foreign exchange cash transaction. This transaction needs to locate a set of funds to exchange. A flow modelling this is implemented in `FxTransactionBuildTutorial.kt`. Second, a simple business model in which parties manually accept or reject each other's trade proposals, which is implemented in `WorkflowTransactionBuildTutorial.kt`. To run and explore these examples using the IntelliJ IDE one can run/step through the respective unit tests in `FxTransactionBuildTutorialTest.kt` and `WorkflowTransactionBuildTutorialTest.kt`, which drive the flows as part of a simulated in-memory network of nodes.

Note

Before creating the IntelliJ run configurations for these unit tests go to Run -> Edit Configurations -> Defaults -> JUnit, add `-javaagent:lib/quasar.jar` to the VM options, and set Working directory to `$PROJECT_DIR$` so that the `Quasar` instrumentation is correctly configured.

For the cash transaction, let's assume we are using the standard `CashState` in the `:financial` Gradle module. The `Cash` contract uses `FungibleAsset` states to model holdings of interchangeable assets and allow the splitting, merging and

summing of states to meet a contractual obligation. We would normally use the `Cash.generateSpend` method to gather the required amount of cash into a `TransactionBuilder`, set the outputs and generate the `Move` command. However, to make things clearer, the example flow code shown here will manually carry out the input queries by specifying relevant query criteria filters to the `tryLockFungibleStatesForSpending` method of the `VaultService`.

```
// This is equivalent to the Cash.generateSpend
// Which is brought here to make the filtering logic more visible in the example
private fun gatherOurInputs(serviceHub: ServiceHub,
    lockId: UUID,
    amountRequired: Amount<Issued<Currency>>,
    notary: Party?): Pair<List<StateAndRef<Cash.State>>, Long>
{
    // extract our identity for convenience
    val ourKeys = serviceHub.keyManagementService.keys
    val ourParties = ourKeys.map { serviceHub.identityService.partyFromKey(it) ?: throw IllegalStateException("Unable to resolve party from key") }
    val fungibleCriteria = QueryCriteria.FungibleAssetQueryCriteria(owner = ourParties)

    val notaries = notary ?: serviceHub.networkMapCache.notaryIdentities.first()
    val vaultCriteria: QueryCriteria = QueryCriteria.VaultQueryCriteria(notary = listOf(notaries as AbstractParty))

    val logicalExpression = builder {
        CashSchemaV1.PersistentCashState::currency.equal(amountRequired.token.product.currencyCode)
    }
    val cashCriteria = QueryCriteria.VaultCustomQueryCriteria(logicalExpression)

    val fullCriteria = fungibleCriteria.and(vaultCriteria).and(cashCriteria)

    val eligibleStates =
        serviceHub.vaultService.tryLockFungibleStatesForSpending(lockId, fullCriteria, amountRequired.withoutIssuer(), Cash.State::class.java)

    check(eligibleStates.isNotEmpty()) { "Insufficient funds" }
    val amount = eligibleStates.fold(0L) { tot, (state) -> tot + state.data.amount.quantity }
    val change = amount - amountRequired.quantity

    return Pair(eligibleStates, change)
}
```

This is a foreign exchange transaction, so we expect another set of input states of another currency from a counterparty. However, the Corda privacy model means we are not aware of the other node's states. Our flow must therefore ask the other node to carry out a similar query and return the additional inputs to the transaction (see the `ForeignExchangeFlow` for more details of the exchange). We now have all the required input `StateRef` items, and can turn to gathering the outputs.

For the trade approval flow we need to implement a simple workflow pattern. We start by recording the unconfirmed trade details in a state object implementing

the `LinearState` interface. One field of this record is used to map the business workflow to an enumerated state. Initially the initiator creates a new state object which receives a new `UniqueId` in its `linearId` property and a starting workflow state of `NEW`. The `Contract.verify` method is written to allow the initiator to sign this initial transaction and send it to the other party. This pattern ensures that a permanent copy is recorded on both ledgers for audit purposes, but the state is prevented from being maliciously put in an approved state. The subsequent workflow steps then follow with transactions that consume the state as inputs on one side and output a new version with whatever state updates, or amendments match to the business process, the `linearId` being preserved across the changes. Attached `Command` objects help the verify method restrict changes to appropriate fields and signers at each step in the workflow. In this it is typical to have both parties sign the change transactions, but it can be valid to allow unilateral signing, if for instance one side could block a rejection.

Commonly the manual initiator of these workflows will query the Vault for states of the right contract type and in the right workflow state over the RPC interface. The RPC will then initiate the relevant flow using `StateRef`, or `linearId` values as parameters to the flow to identify the states being operated upon. Thus code to gather the latest input state for a given `StateRef` would use the `VaultService` as follows:

```
val criteria = VaultQueryCriteria(stateRefs = listOf(ref))
val latestRecord =
    serviceHub.vaultService.queryBy<TradeApprovalContract.State>(criteria).states.single()
```

Generating Commands

For the commands that will be added to the transaction, these will need to correctly reflect the task at hand. These must match because inside the `Contract.verify` method the command will be used to select the validation code path. The `Contract.verify` method will then restrict the allowed contents of the transaction to reflect this context. Typical restrictions might include that the input cash amount must equal the output cash amount, or that a workflow step is only allowed to change the status field. Sometimes, the command may capture some data too e.g. the foreign exchange rate, or the identity of one party, or the StateRef of the specific input that originates the command in a bulk operation. This data will be used to further aid the `Contract.verify`, because to ensure consistent, secure and reproducible behaviour in a distributed environment the `Contract.verify`, transaction is the only allowed to use the content of the transaction to decide validity.

Another essential requirement for commands is that the correct set of `PublicKey` objects are added to the `Command` on the builder, which will be used to form the set of required signers on the final validated transaction. These must correctly align with the expectations of the `Contract.verify` method, which should be written to defensively check this. In particular, it is expected that at minimum the owner of an asset would have to be signing to permission transfer of that asset. In addition, other signatories will often be required e.g. an Oracle identity for an Oracle command, or both parties when there is an exchange of assets.

Generating Outputs

Having located a `StateAndRefs` set as the transaction inputs, the flow has to generate the output states. Typically, this is a simple call to the Kotlin `copy` method to modify the few fields that will transitioned in the transaction. The contract code may provide a `generateXXX` method to help with this process if the task is more complicated. With a workflow state a slightly modified copy state is usually sufficient, especially as it is expected that we wish to preserve the `linearId` between state revisions, so that Vault queries can find the latest revision.

For fungible contract states such as `cash` it is common to distribute and split the total amount e.g. to produce a remaining balance output state for the original owner when breaking up a large amount input state. Remember that the result of a successful transaction is always to fully consume/spend the input states, so this is required to conserve the total cash. For example from the demo code:

```
// Gather our inputs. We would normally use VaultService.generateSpend
// to carry out the build in a single step. To be more explicit
// we will use query manually in the helper function below.
// Putting this into a non-suspendable function also prevents issues when
// the flow is suspended.
val (inputs, residual) = gatherOurInputs(serviceHub, lockId, sellAmount,
request.notary)

// Build and an output state for the counterparty
val transferredFundsOutput = Cash.State(sellAmount, request.counterparty)

val outputs = if (residual > 0L) {
    // Build an output state for the residual change back to us
    val residualAmount = Amount(residual, sellAmount.token)
    val residualOutput = Cash.State(residualAmount,
serviceHub.myInfo.singleIdentity())
    listOf(transferredFundsOutput, residualOutput)
} else {
    listOf(transferredFundsOutput)
}
return Pair(inputs, outputs)
```

Building the SignedTransaction

Having gathered all the components for the transaction we now need to use a `TransactionBuilder` to construct the full `SignedTransaction`. We instantiate a `TransactionBuilder` and provide a notary that will be associated with the output states. Then we keep adding inputs, outputs, commands and attachments to complete the transaction.

Once the transaction is fully formed, we call `ServiceHub.signInitialTransaction` to sign the `TransactionBuilder` and convert it into a `SignedTransaction`.

Examples of this process are:

```
// Modify the state field for new output. We use copy, to ensure no other
// modifications.
// It is especially important for a LinearState that the linearId is copied across,
// not accidentally assigned a new random id.
val newState = latestRecord.state.data.copy(state = verdict)

// We have to use the original notary for the new transaction
val notary = latestRecord.state.notary

// Get and populate the new TransactionBuilder
// To destroy the old proposal state and replace with the new completion state.
// Also add the Completed command with keys of all parties to signal the Tx purpose
// to the Contract verify method.
val tx = TransactionBuilder(notary).
    withItems(
        latestRecord,
        StateAndContract(newState, TRADE_APPROVAL_PROGRAM_ID),
        Command(TradeApprovalContract.Commands.Completed(),
            listOf(ourIdentity.owningKey,
latestRecord.state.data.source.owningKey)))
tx.setTimeWindow(serviceHub.clock.instant(), 60.seconds)
// We can sign this transaction immediately as we have already checked all the fields
and the decision
// is ultimately a manual one from the caller.
// As a SignedTransaction we can pass the data around certain that it cannot be
modified,
// although we do require further signatures to complete the process.
val selfSignedTx = serviceHub.signInitialTransaction(tx)

private fun buildTradeProposal(ourInputStates: List<StateAndRef<Cash.State>>,
                               ourOutputState: List<Cash.State>,
                               theirInputStates: List<StateAndRef<Cash.State>>,
                               theirOutputState: List<Cash.State>): SignedTransaction
{
    // This is the correct way to create a TransactionBuilder,
    // do not construct directly.
    // We also set the notary to match the input notary
    val builder = TransactionBuilder(ourInputStates.first().state.notary)

    // Add the move commands and key to indicate all the respective owners and need to
sign
    val ourSigners = ourInputStates.map { it.state.data.owner.owningKey }.toSet()
    val theirSigners = theirInputStates.map { it.state.data.owner.owningKey }.toSet()
    builder.addCommand(Cash.Commands.Move(), (ourSigners + theirSigners).toList())
}
```

```

    // Build and add the inputs and outputs
    builder.withItems(*ourInputStates.toTypedArray())
    builder.withItems(*theirInputStates.toTypedArray())
    builder.withItems(*ourOutputState.map { StateAndContract(it, Cash.PROGRAM_ID)
}.toTypedArray())
    builder.withItems(*theirOutputState.map { StateAndContract(it, Cash.PROGRAM_ID)
}.toTypedArray())

    // We have already validated their response and trust our own data
    // so we can sign. Note the returned SignedTransaction is still not fully signed
    // and would not pass full verification yet.
    return serviceHub.signInitialTransaction(builder, ourSigners.single())
}

```

Completing the SignedTransaction

Having created an initial `TransactionBuilder` and converted this to a `SignedTransaction`, the process of verifying and forming a full `SignedTransaction` begins and then completes with the notarisation. In practice this is a relatively stereotypical process, because assuming the `SignedTransaction` is correctly constructed the verification should be immediate. However, it is also important to recheck the business details of any data received back from an external node, because a malicious party could always modify the contents before returning the transaction. Each remote flow should therefore check as much as possible of the initial `SignedTransaction` inside the `unwrap` of the receive before agreeing to sign. Any issues should immediately throw an exception to abort the flow. Similarly the originator, should always apply any new signatures to its original proposal to ensure the contents of the transaction has not been altered by the remote parties.

The typical code therefore checks the received `SignedTransaction` using the `verifySignaturesExcept` method, excluding itself, the notary and any other parties yet to apply their signature. The contents of the `SignedTransaction` should be fully verified further by expanding with `toLedgerTransaction` and calling `verify`. Further context specific and business checks should then be made, because the `Contract.verify` is not allowed to access external context. For example, the flow may need to check that the parties are the right ones, or that the `Command` present on the transaction is as expected for this specific flow. An example of this from the demo code is:

```

// First we receive the verdict transaction signed by their single key
val completeTx = sourceSession.receive<SignedTransaction>().unwrap {
    // Check the transaction is signed apart from our own key and the notary
    it.verifySignaturesExcept(ourIdentity.owningKey, it.tx.notary!!.owningKey)
    // Check the transaction data is correctly formed
    val ltx = it.toLedgerTransaction(serviceHub, false)
    ltx.verify()
}

```

```

// Confirm that this is the expected type of transaction
require(ltx.commands.single().value is TradeApprovalContract.Commands.Completed) {
    "Transaction must represent a workflow completion"
}
// Check the context dependent parts of the transaction as the
// Contract verify method must not use serviceHub queries.
val state = ltx.outRef<TradeApprovalContract.State>(0)
require(serviceHub.myInfo.isLegalIdentity(state.state.data.source)) {
    "Proposal not one of our original proposals"
}
require(state.state.data.counterparty == sourceSession.counterparty) {
    "Proposal not sent from correct source"
}
it
}

```

After verification the remote flow will return its signature to the originator. The originator should apply that signature to the starting `SignedTransaction` and recheck the signatures match.

Committing the Transaction

Once all the signatures are applied to the `SignedTransaction`, the final steps are notarisation and ensuring that all nodes record the fully-signed transaction. The code for this is standardised in the `FinalityFlow`:

```
// Notarise and distribute the completed transaction.
subFlow(FinalityFlow(allPartySignedTx, setOf(newState.source)))
```

Partially Visible Transactions

The discussion so far has assumed that the parties need full visibility of the transaction to sign. However, there may be situations where each party needs to store private data for audit purposes, or for evidence to a regulator, but does not wish to share that with the other trading partner. The tear-off/Merkle tree support in Corda allows flows to send portions of the full transaction to restrict visibility to remote parties. To do this one can use

the `SignedTransaction.buildFilteredTransaction` extension method to produce a `FilteredTransaction`. The elements of the `SignedTransaction` which we wish to be hide will be replaced with their secure hash. The overall transaction id is still provable from the `FilteredTransaction` preventing change of the private data, but we do not expose that data to the other node directly. A full example of this can be found in the `NodeInterestRates` Oracle code from the `irs-demo` project which interacts with the `RatesFixFlow` flow. Also, refer to the merkle-trees documentation.

- Writing flows
 - [View page source](#)
-

Writing flows

This article explains our approach to modelling business processes and the lower level network protocols that implement them. It explains how the platform's flow framework is used, and takes you through the code for a simple 2-party asset trading flow which is included in the source.

Introduction

Shared distributed ledgers are interesting because they allow many different, mutually distrusting parties to share a single source of truth about the ownership of assets. Digitally signed transactions are used to update that shared ledger, and transactions may alter many states simultaneously and atomically.

Blockchain systems such as Bitcoin support the idea of building up a finished, signed transaction by passing around partially signed invalid transactions outside of the main network, and by doing this you can implement *delivery versus payment* such that there is no chance of settlement failure, because the movement of cash and the traded asset are performed atomically by the same transaction. To perform such a trade involves a multi-step flow in which messages are passed back and forth privately between parties, checked, signed and so on.

Despite how useful these flows are, platforms such as Bitcoin and Ethereum do not assist the developer with the rather tricky task of actually building them. That is unfortunate. There are many awkward problems in their implementation that a good platform would take care of for you, problems like:

- Avoiding “callback hell” in which code that should ideally be sequential is turned into an unreadable mess due to the desire to avoid using up a thread for every flow instantiation.
- Surviving node shutdowns/restarts that may occur in the middle of the flow without complicating things. This implies that the state of the flow must be persisted to disk.

- Error handling.
- Message routing.
- Serialisation.
- Catching type errors, in which the developer gets temporarily confused and expects to receive/send one type of message when actually they need to receive/send another.
- Unit testing of the finished flow.

Actor frameworks can solve some of the above but they are often tightly bound to a particular messaging layer, and we would like to keep a clean separation. Additionally, they are typically not type safe, and don't make persistence or writing sequential code much easier.

To put these problems in perspective, the *payment channel protocol* in the bitcoinj library, which allows bitcoins to be temporarily moved off-chain and traded at high speed between two parties in private, consists of about 7000 lines of Java and took over a month of full time work to develop. Most of that code is concerned with the details of persistence, message passing, lifecycle management, error handling and callback management. Because the business logic is quite spread out the code can be difficult to read and debug.

As small contract-specific trading flows are a common occurrence in finance, we provide a framework for the construction of them that automatically handles many of the concerns outlined above.

Theory

A *continuation* is a suspended stack frame stored in a regular object that can be passed around, serialised, unserialised and resumed from where it was suspended. This concept is sometimes referred to as "fibers". This may sound abstract but don't worry, the examples below will make it clearer. The JVM does not natively support continuations, so we implement them using a library called Quasar which works through behind-the-scenes bytecode rewriting. You don't have to know how this works to benefit from it, however.

We use continuations for the following reasons:

- It allows us to write code that is free of callbacks, that looks like ordinary sequential code.

- A suspended continuation takes far less memory than a suspended thread. It can be as low as a few hundred bytes. In contrast a suspended Java thread stack can easily be 1mb in size.
- It frees the developer from thinking (much) about persistence and serialisation.

A *state machine* is a piece of code that moves through various *states*. These are not the same as states in the data model (that represent facts about the world on the ledger), but rather indicate different stages in the progression of a multi-stage flow. Typically writing a state machine would require the use of a big switch statement and some explicit variables to keep track of where you're up to. The use of continuations avoids this hassle.

A two party trading flow

We would like to implement the “hello world” of shared transaction building flows: a seller wishes to sell some asset (e.g. some commercial paper) in return for *cash*. The buyer wishes to purchase the asset using his cash. They want the trade to be atomic so neither side is exposed to the risk of settlement failure. We assume that the buyer and seller have found each other and arranged the details on some exchange, or over the counter. The details of how the trade is arranged isn’t covered in this article.

Our flow has two parties (B and S for buyer and seller) and will proceed as follows:

1. S sends a `StateAndRef` pointing to the state they want to sell to B, along with info about the price they require B to pay.
2. B sends to S a `SignedTransaction` that includes two inputs (the state owned by S, and cash owned by B) and three outputs (the state now owned by B, the cash now owned by S, and any change cash still owned by B).
The `SignedTransaction` has a single signature from B but isn’t valid because it lacks a signature from S authorising movement of the asset.
3. S signs the transaction and sends it back to B.
4. B *finalises* the transaction by sending it to the notary who checks the transaction for validity, recording the transaction in B’s local vault, and then sending it on to S who also checks it and commits the transaction to S’s local vault.

You can find the implementation of this flow in the file [finance/src/main/kotlin/net/corda/finance/TwoPartyTradeFlow.kt](#).

Assuming no malicious termination, they both end the flow being in possession of a valid, signed transaction that represents an atomic asset swap.

Note that it's the *seller* who initiates contact with the buyer, not vice-versa as you might imagine.

We start by defining two classes that will contain the flow definition. We also pick what data will be used by each side.

Note

The code samples in this tutorial are only available in Kotlin, but you can use any JVM language to write them and the approach is the same.

Kotlin

```
object TwoPartyTradeFlow {
    class UnacceptablePriceException(givenPrice: Amount<Currency>) :
        FlowException("Unacceptable price: $givenPrice")
    class AssetMismatchException(val expectedTypeName: String, val typeName: String) :
        FlowException() {
        override fun toString() = "The submitted asset didn't match the expected type: $expectedTypeName vs $typeName"
    }

    /**
     * This object is serialised to the network and is the first flow message the seller sends to the buyer.
     *
     * @param payToIdentity anonymous identity of the seller, for payment to be sent to.
     */
    @CordaSerializable
    data class SellerTradeInfo(
        val price: Amount<Currency>,
        val payToIdentity: PartyAndCertificate
    )

    open class Seller(private val otherSideSession: FlowSession,
                     private val assetToSell: StateAndRef<OwnableState>,
                     private val price: Amount<Currency>,
                     private val myParty: PartyAndCertificate,
                     override val progressTracker: ProgressTracker =
            TwoPartyTradeFlow.Seller.tracker()): FlowLogic<SignedTransaction>() {

        companion object {
            fun tracker() = ProgressTracker()
        }

        @Suspendable
        override fun call(): SignedTransaction {
            TODO()
        }
    }
}
```

```

    }

    open class Buyer(private val sellerSession: FlowSession,
                    private val notary: Party,
                    private val acceptablePrice: Amount<Currency>,
                    private val typeToBuy: Class<out OwnableState>,
                    private val anonymous: Boolean) : FlowLogic<SignedTransaction>()
    {

        @Suspendable
        override fun call(): SignedTransaction {
            TODO()
        }
    }
}

```

This code defines several classes nested inside the main `TwoPartyTradeFlow` singleton. Some of the classes are simply flow messages or exceptions. The other two represent the buyer and seller side of the flow.

Going through the data needed to become a seller, we have:

- `otherSideSession: FlowSession` - a flow session for communication with the buyer
- `assetToSell: StateAndRef<OwnableState>` - a pointer to the ledger entry that represents the thing being sold
- `price: Amount<Currency>` - the agreed on price that the asset is being sold for (without an issuer constraint)
- `myParty: PartyAndCertificate` - the certificate representing the party that controls the asset being sold

And for the buyer:

- `sellerSession: FlowSession` - a flow session for communication with the seller
- `notary: Party` - the entry in the network map for the chosen notary. See “Notaries” for more information on notaries
- `acceptablePrice: Amount<Currency>` - the price that was agreed upon out of band. If the seller specifies a price less than or equal to this, then the trade will go ahead
- `typeToBuy: Class<out OwnableState>` - the type of state that is being purchased. This is used to check that the sell side of the flow isn’t trying to sell us the wrong thing, whether by accident or on purpose
- `anonymous: Boolean` - whether to generate a fresh, anonymous public key for the transaction

Alright, so using this flow shouldn't be too hard: in the simplest case we can just create a Buyer or Seller with the details of the trade, depending on who we are. We then have to start the flow in some way. Just calling the `call` function ourselves won't work: instead we need to ask the framework to start the flow for us. More on that in a moment.

Suspendable functions

The `call` function of the buyer/seller classes is marked with the `@Suspendable` annotation. What does this mean?

As mentioned above, our flow framework will at points suspend the code and serialise it to disk. For this to work, any methods on the call stack must have been pre-marked as `@Suspendable` so the bytecode rewriter knows to modify the underlying code to support this new feature. A flow is suspended when calling either `receive`, `send` or `sendAndReceive` which we will learn more about below. For now, just be aware that when one of these methods is invoked, all methods on the stack must have been marked. If you forget, then in the unit test environment you will get a useful error message telling you which methods you didn't mark. The fix is simple enough: just add the annotation and try again.

Note

Java 9 is likely to remove this pre-marking requirement completely.

Whitelisted classes with the Corda node

For security reasons, we do not want Corda nodes to be able to just receive instances of any class on the classpath via messaging, since this has been exploited in other Java application containers in the past. Instead, we require every class contained in messages to be whitelisted. Some classes are whitelisted by default (see `DefaultWhitelist`), but others outside of that set need to be whitelisted either by using the annotation `@CordaSerializable` or via the plugin framework. See [Object serialization](#). You can see above that the `SellerTradeInfo` has been annotated.

Starting your flow

The `StateMachineManager` is the class responsible for taking care of all running flows in a node. It knows how to register handlers with the messaging system

(see “Networking and messaging”) and iterate the right state machine when messages arrive. It provides the send/receive/sendAndReceive calls that let the code request network interaction and it will save/restore serialised versions of the fiber at the right times.

Flows can be invoked in several ways. For instance, they can be triggered by scheduled events (in which case they need to be annotated with `@SchedulableFlow`), see “Event scheduling” to learn more about this. They can also be triggered directly via the node’s RPC API from your app code (in which case they need to be annotated with `StartableByRPC`). It’s possible for a flow to be of both types.

You request a flow to be invoked by using the `CordaRPCOps.startFlowDynamic` method. This takes a Java reflection `Class` object that describes the flow class to use (in this case, either `Buyer` or `Seller`). It also takes a set of arguments to pass to the constructor. Because it’s possible for flow invocations to be requested by untrusted code (e.g. a state that you have been sent), the types that can be passed into the flow are checked against a whitelist, which can be extended by apps themselves at load time. There are also a series of inlined Kotlin extension functions of the form `CordaRPCOps.startFlow` which help with invoking flows in a type safe manner.

The process of starting a flow returns a `FlowHandle` that you can use to observe the result, and which also contains a permanent identifier for the invoked flow in the form of the `StateMachineRunId`. Should you also wish to track the progress of your flow (see Progress tracking) then you can invoke your flow instead using `CordaRPCOps.startTrackedFlowDynamic` or any of its corresponding `CordaRPCOps.startTrackedFlow` extension functions. These will return a `FlowProgressHandle`, which is just like a `FlowHandle` except that it also contains an observable `progress` field.

Note

The developer *must* then either subscribe to this `progress` observable or invoke the `notUsed()` extension function for it. Otherwise the unused observable will waste resources back in the node.

Implementing the seller

Let’s implement the `Seller.call` method that will be run when the flow is invoked.

Kotlin

```
@Suspendable
override fun call(): SignedTransaction {
    progressTracker.currentStep = AWAITING_PROPOSAL
    // Make the first message we'll send to kick off the flow.
    val hello = SellerTradeInfo(price, myParty)
    // What we get back from the other side is a transaction that *might* be valid and
    acceptable to us,
    // but we must check it out thoroughly before we sign!
    // SendTransactionFlow allows seller to access our data to resolve the
    transaction.
    subFlow(SendStateAndRefFlow(otherSideSession, listOf(assetToSell)))
    otherSideSession.send(hello)

    // Verify and sign the transaction.
    progressTracker.currentStep = VERIFYING_AND_SIGNING

    // DOCSTART 07
    // Sync identities to ensure we know all of the identities involved in the
    transaction we're about to
    // be asked to sign
    subFlow(IdentitySyncFlow.Receive(otherSideSession))
    // DOCEND 07

    // DOCSTART 5
    val signTransactionFlow = object : SignTransactionFlow(otherSideSession,
    VERIFYING_AND_SIGNING.childProgressTracker()) {
        override fun checkTransaction(stx: SignedTransaction) {
            // Verify that we know who all the participants in the transaction are
            val states: Iterable<ContractState> =
                serviceHub.loadStates(stx.tx.inputs.toSet()).map { it.state.data } +
                stx.tx.outputs.map { it.data }
            states.forEach { state ->
                state.participants.forEach { anon ->

                    require(serviceHub.identityService.wellKnownPartyFromAnonymous(anon) != null) {
                        "Transaction state $state involves unknown participant $anon"
                    }
                }
            }

            if (stx.tx.outputStates.sumCashBy(myParty.party).withoutIssuer() != price)
                throw FlowException("Transaction is not sending us the right amount of
cash")
        }
    }

    val txId = subFlow(signTransactionFlow).id
    // DOCEND 5

    return waitForLedgerCommit(txId)
}
```

We start by sending information about the asset we wish to sell to the buyer. We fill out the initial flow message with the trade info, and then call `otherSideSession.send`, which takes two arguments:

- The party we wish to send the message to
- The payload being sent

`otherSideSession.send` will serialise the payload and send it to the other party automatically.

Next, we call a *subflow* called `IdentitySyncFlow.Receive` (see Sub-flows). `IdentitySyncFlow.Receive` ensures that our node can de-anonymise any confidential identities in the transaction it's about to be asked to sign.

Next, we call another subflow called `SignTransactionFlow`. `SignTransactionFlow` automates the process of:

- Receiving a proposed trade transaction from the buyer, with the buyer's signature attached.
- Checking that the proposed transaction is valid.
- Calculating and attaching our own signature so that the transaction is now signed by both the buyer and the seller.
- Sending the transaction back to the buyer.

The transaction then needs to be finalized. This is the the process of sending the transaction to a notary to assert (with another signature) that the timestamp in the transaction (if any) is valid and there are no double spends. In this flow, finalization is handled by the buyer, so we just wait for the signed transaction to appear in our transaction storage. It will have the same ID as the one we started with but more signatures.

Implementing the buyer

OK, let's do the same for the buyer side:

Kotlin

```
@Suspendable
override fun call(): SignedTransaction {
    // Wait for a trade request to come in from the other party.
    progressTracker.currentStep = RECEIVING
    val (assetForSale, tradeRequest) = receiveAndValidateTradeRequest()

    // Create the identity we'll be paying to, and send the counterparty proof we own
    // the identity
    val buyerAnonymousIdentity = if (anonymous)
        serviceHub.keyManagementService.freshKeyAndCert(ourIdentityAndCert, false)
    else
        ourIdentityAndCert
    // Put together a proposed transaction that performs the trade, and sign it.
    progressTracker.currentStep = SIGNING
    val (ptx, cashSigningPubKeys) = assembleSharedTX(assetForSale, tradeRequest,
buyerAnonymousIdentity)
```

```

// DOCSTART 6
// Now sign the transaction with whatever keys we need to move the cash.
val partSignedTx = serviceHub.signInitialTransaction(ptx, cashSigningPubKeys)

// Sync up confidential identities in the transaction with our counterparty
subFlow(IdentitySyncFlow.Send(sellerSession, ptx.toWireTransaction(serviceHub)))

// Send the signed transaction to the seller, who must then sign it themselves and
commit
// it to the ledger by sending it to the notary.
progressTracker.currentStep = COLLECTING_SIGNATURES
val sellerSignature = subFlow(CollectSignatureFlow(partSignedTx, sellerSession,
sellerSession.counterparty.owningKey))
val twiceSignedTx = partSignedTx + sellerSignature
// DOCEND 6

// Notarise and record the transaction.
progressTracker.currentStep = RECORDING
return subFlow(FinalityFlow(twiceSignedTx))
}

@Suspendable
private fun receiveAndValidateTradeRequest(): Pair<StateAndRef<OwnableState>,
SellerTradeInfo> {
    val assetForSale =
subFlow(ReceiveStateAndRefFlow<OwnableState>(sellerSession)).single()
    return assetForSale to sellerSession.receive<SellerTradeInfo>().unwrap {
        progressTracker.currentStep = VERIFYING
        // What is the seller trying to sell us?
        val asset = assetForSale.state.data
        val assetTypeName = asset.javaClass.name

        // The asset must either be owned by the well known identity of the
counterparty, or we must be able to
        // prove the owner is a confidential identity of the counterparty.
        val assetForSaleIdentity =
serviceHub.identityService.wellKnownPartyFromAnonymous(asset.owner)
        require(assetForSaleIdentity == sellerSession.counterparty)

        // Register the identity we're about to send payment to. This shouldn't be the
same as the asset owner
        // identity, so that anonymity is enforced.
        val wellKnownPayToIdentity =
serviceHub.identityService.verifyAndRegisterIdentity(it.payToIdentity) ?:
it.payToIdentity
        require(wellKnownPayToIdentity.party == sellerSession.counterparty) { "Well
known identity to pay to must match counterparty identity" }

        if (it.price > acceptablePrice)
            throw UnacceptablePriceException(it.price)
        if (!typeToBuy.isInstance(asset))
            throw AssetMismatchException(typeToBuy.name, assetTypeName)

        it
    }
}

@Suspendable
private fun assembleSharedTX(assetForSale: StateAndRef<OwnableState>, tradeRequest:
SellerTradeInfo, buyerAnonymousIdentity: PartyAndCertificate): SharedTx {
    val ptx = TransactionBuilder(notary)

    // Add input and output states for the movement of cash, by using the Cash
contract to generate the states
    val (tx, cashSigningPubKeys) = Cash.generateSpend(serviceHub, ptx,
tradeRequest.price, ourIdentityAndCert, tradeRequest.payToIdentity.party)

```

```

// Add inputs/outputs/a command for the movement of the asset.
tx.addInputState(assetForSale)

    val (command, state) =
assetForSale.state.data.withNewOwner(buyerAnonymousIdentity.party)
    tx.addOutputState(state, assetForSale.state.contract, assetForSale.state.notary)
    tx.addCommand(command, assetForSale.state.data.owner.owningKey)

    // We set the transaction's time-window: it may be that none of the contracts need
this!
    // But it can't hurt to have one.
    val currentTime = serviceHub.clock.instant()
    tx.setTimeWindow(currentTime, 30.seconds)

    return SharedTx(tx, cashSigningPubKeys)
}

```

This code is longer but no more complicated. Here are some things to pay attention to:

1. We do some sanity checking on the proposed trade transaction received from the seller to ensure we're being offered what we expected to be offered.
2. We create a cash spend using `Cash.generateSpend`. You can read the vault documentation to learn more about this.
3. We access the *service hub* as needed to access things that are transient and may change or be recreated whilst a flow is suspended, such as the wallet or the network map.
4. We call `CollectSignaturesFlow` as a subflow to send the unfinished, still-invalid transaction to the seller so they can sign it and send it back to us.
5. Last, we call `FinalityFlow` as a subflow to finalize the transaction.

As you can see, the flow logic is straightforward and does not contain any callbacks or network glue code, despite the fact that it takes minimal resources and can survive node restarts.

Flow sessions

It will be useful to describe how flows communicate with each other. A node may have many flows running at the same time, and perhaps communicating with the same counterparty node but for different purposes. Therefore flows need a way to segregate communication channels so that concurrent conversations between flows on the same set of nodes do not interfere with each other.

To achieve this in order to communicate with a counterparty one needs to first initiate such a session with a `Party` using `initiateFlow`, which returns

a `FlowSession` object, identifying this communication. Subsequently the first actual communication will kick off a counter-flow on the other side, receiving a “reply” session object. A session ends when either flow ends, whether as expected or pre-maturely. If a flow ends pre-maturely then the other side will be notified of that and they will also end, as the whole point of flows is a known sequence of message transfers. Flows end pre-maturely due to exceptions, and as described above, if that exception is `FlowException` or a sub-type then it will propagate to the other side. Any other exception will not propagate.

Taking a step back, we mentioned that the other side has to accept the session request for there to be a communication channel. A node accepts a session request if it has registered the flow type (the fully-qualified class name) that is making the request - each session initiation includes the initiating flow type. The *initiated* (server) flow must name the *initiating* (client) flow using the `@InitiatedBy` annotation and passing the class name that will be starting the flow session as the annotation parameter.

Sub-flows

Flows can be composed via nesting. Invoking a sub-flow looks similar to an ordinary function call:

```
KotlinJava
@suspendable
fun call() {
    val unnotarisedTransaction = ...
    subFlow(FinalityFlow(unnotarisedTransaction))
}
```

Let's take a look at the three subflows we invoke in this flow.

FinalityFlow

On the buyer side, we use `FinalityFlow` to finalise the transaction. It will:

- Send the transaction to the chosen notary and, if necessary, satisfy the notary that the transaction is valid.
- Record the transaction in the local vault, if it is relevant (i.e. involves the owner of the node).
- Send the fully signed transaction to the other participants for recording as well.

Warning

If the seller stops before sending the finalised transaction to the buyer, the seller is left with a valid transaction but the buyer isn't, so they can't spend the asset they just purchased! This sort of thing is not always a risk (as the seller may not gain anything from that sort of behaviour except a lawsuit), but if it is, a future version of the platform will allow you to ask the notary to send you the transaction as well, in case your counterparty does not. This is not the default because it reveals more private info to the notary.

We simply create the flow object via its constructor, and then pass it to the `subFlow` method which returns the result of the flow's execution directly.

Behind the scenes all this is doing is wiring up progress tracking (discussed more below) and then running the object's `call` method. Because the sub-flow might suspend, we must mark the method that invokes it as suspendable.

Within `FinalityFlow`, we use a further sub-flow called `ReceiveTransactionFlow`. This is responsible for downloading and checking all the dependencies of a transaction, which in Corda are always retrievable from the party that sent you a transaction that uses them. This flow returns a list of `LedgerTransaction` objects.

Note

Transaction dependency resolution assumes that the peer you got the transaction from has all of the dependencies itself. It must do, otherwise it could not have convinced itself that the dependencies were themselves valid. It's important to realise that requesting only the transactions we require is a privacy leak, because if we don't download a transaction from the peer, they know we must have already seen it before. Fixing this privacy leak will come later.

CollectSignaturesFlow/SignTransactionFlow

We also invoke two other subflows:

- `CollectSignaturesFlow`, on the buyer side
- `SignTransactionFlow`, on the seller side

These flows communicate to gather all the required signatures for the proposed transaction. `CollectSignaturesFlow` will:

- Verify any signatures collected on the transaction so far

- Verify the transaction itself
- Send the transaction to the remaining required signers and receive back their signatures
- Verify the collected signatures

`SignTransactionFlow` responds by:

- Receiving the partially-signed transaction off the wire
- Verifying the existing signatures
- Resolving the transaction's dependencies
- Verifying the transaction itself
- Running any custom validation logic
- Sending their signature back to the buyer
- Waiting for the transaction to be recorded in their vault

We cannot instantiate `SignTransactionFlow` itself, as it's an abstract class. Instead, we need to subclass it and override `checkTransaction()` to add our own custom validation logic:

Kotlin

```
val signTransactionFlow = object : SignTransactionFlow(otherSideSession,
VERIFYING_AND_SIGNING.childProgressTracker()) {
    override fun checkTransaction(stx: SignedTransaction) {
        // Verify that we know who all the participants in the transaction are
        val states: Iterable<ContractState> =
            serviceHub.loadStates(stx.tx.inputs.toSet()).map { it.state.data } +
            stx.tx.outputs.map { it.data }
        states.forEach { state ->
            state.participants.forEach { anon ->
                require(serviceHub.identityService.wellKnownPartyFromAnonymous(anon)
!= null) {
                    "Transaction state $state involves unknown participant $anon"
                }
            }
        }
        if (stx.tx.outputStates.sumCashBy(myParty.party).withoutIssuer() != price)
            throw FlowException("Transaction is not sending us the right amount of
cash")
    }
}

val txId = subFlow(signTransactionFlow).id
```

In this case, our custom validation logic ensures that the amount of cash outputs in the transaction equals the price of the asset.

Persisting flows

If you look at the code

for `FinalityFlow`, `CollectSignaturesFlow` and `SignTransactionFlow`, you'll see calls to both `receive` and `sendAndReceive`. Once either of these methods is called, the `call` method will be suspended into a continuation and saved to persistent storage. If the node crashes or is restarted, the flow will effectively continue as if nothing had happened. Your code may remain blocked inside such a call for seconds, minutes, hours or even days in the case of a flow that needs human interaction!

Note

There are a couple of rules you need to bear in mind when writing a class that will be used as a continuation. The first is that anything on the stack when the function is suspended will be stored into the heap and kept alive by the garbage collector. So try to avoid keeping enormous data structures alive unless you really have to. You can always use private methods to keep the stack uncluttered with temporary variables, or to avoid objects that Kryo is not able to serialise correctly.

The second is that as well as being kept on the heap, objects reachable from the stack will be serialised. The state of the function call may be resurrected much later! Kryo doesn't require objects be marked as serialisable, but even so, doing things like creating threads from inside these calls would be a bad idea. They should only contain business logic and only do I/O via the methods exposed by the flow framework.

It's OK to keep references around to many large internal node services though: these will be serialised using a special token that's recognised by the platform, and wired up to the right instance when the continuation is loaded off disk again.

`receive` and `sendAndReceive` return a simple wrapper class, `UntrustworthyData<T>`, which is just a marker class that reminds us that the data came from a potentially malicious external source and may have been tampered with or be unexpected in other ways. It doesn't add any functionality, but acts as a reminder to "scrub" the data before use.

Exception handling

Flows can throw exceptions to prematurely terminate their execution. The flow framework gives special treatment to `FlowException` and its subtypes. These

exceptions are treated as error responses of the flow and are propagated to all counterparties it is communicating with. The receiving flows will throw the same exception the next time they do a `receive` or `sendAndReceive` and thus end the flow session. If the receiver was invoked via `subFlow` then the exception can be caught there enabling re-invocation of the sub-flow.

If the exception thrown by the erroring flow is not a `FlowException` it will still terminate but will not propagate to the other counterparties. Instead they will be informed the flow has terminated and will themselves be terminated with a generic exception.

Note

A future version will extend this to give the node administrator more control on what to do with such erroring flows.

Throwing a `FlowException` enables a flow to reject a piece of data it has received back to the sender. This is typically done in the `unwrap` method of the received `UntrustworthyData`. In the above example the seller checks the price and throws `FlowException` if it's invalid. It's then up to the buyer to either try again with a better price or give up.

Progress tracking

Not shown in the code snippets above is the usage of the `ProgressTracker` API. Progress tracking exports information from a flow about where it's got up to in such a way that observers can render it in a useful manner to humans who may need to be informed. It may be rendered via an API, in a GUI, onto a terminal window, etc.

A `ProgressTracker` is constructed with a series of `Step` objects, where each step is an object representing a stage in a piece of work. It is therefore typical to use singletons that subclass `Step`, which may be defined easily in one line when using Kotlin. Typical steps might be "Waiting for response from peer", "Waiting for signature to be approved", "Downloading and verifying data" etc.

A flow might declare some steps with code inside the flow class like this:

KotlinJava

```
object RECEIVING : ProgressTracker.Step("Waiting for seller trading info")

object VERIFYING : ProgressTracker.Step("Verifying seller assets")
object SIGNING : ProgressTracker.Step("Generating and signing transaction proposal")
object COLLECTING_SIGNATURES : ProgressTracker.Step("Collecting signatures from other
parties") {
    override fun childProgressTracker() = CollectSignaturesFlow.tracker()
}

object RECORDING : ProgressTracker.Step("Recording completed transaction") {
    // TODO: Currently triggers a race condition on Team City. See
    // https://github.com/corda/corda/issues/733.
    // override fun childProgressTracker() = FinalityFlow.tracker()
}

override val progressTracker = ProgressTracker(RECEIVING, VERIFYING, SIGNING,
COLLECTING_SIGNATURES, RECORDING)
```

Each step exposes a label. By defining your own step types, you can export progress in a way that's both human readable and machine readable.

Progress trackers are hierarchical. Each step can be the parent for another tracker. By setting `Step.childProgressTracker`, a tree of steps can be created. It's allowed to alter the hierarchy at runtime, on the fly, and the progress renderers will adapt to that properly. This can be helpful when you don't fully know ahead of time what steps will be required. If you *do* know what is required, configuring as much of the hierarchy ahead of time is a good idea, as that will help the users see what is coming up. You can pre-configure steps by overriding the `Step` class like this:

KotlinJava

```
object VERIFYING_AND_SIGNING : ProgressTracker.Step("Verifying and signing transaction
proposal") {
    override fun childProgressTracker() = SignTransactionFlow.tracker()
}
```

Every tracker has not only the steps given to it at construction time, but also the singleton `ProgressTracker.UNSTARTED` step and the `ProgressTracker.DONE` step. Once a tracker has become `DONE` its position may not be modified again (because e.g. the UI may have been removed/cleaned up), but until that point, the position can be set to any arbitrary set both forwards and backwards. Steps may be skipped, repeated, etc. Note that rolling the current step backwards will delete any progress trackers that are children of the steps being reversed, on the assumption that those subtasks will have to be repeated.

Trackers provide an `Rx observable` which streams changes to the hierarchy. The top level observable exposes all the events generated by its children as well. The

changes are represented by objects indicating whether the change is one of position (i.e. progress), structure (i.e. new subtasks being added/removed) or some other aspect of rendering (i.e. a step has changed in some way and is requesting a re-render).

The flow framework is somewhat integrated with this API. Each `FlowLogic` may optionally provide a tracker by overriding the `progressTracker` property (`getProgressTracker` method in Java). If the `FlowLogic.subFlow` method is used, then the tracker of the sub-flow will be made a child of the current step in the parent flow automatically, if the parent is using tracking in the first place. The framework will also automatically set the current step to `DONE` for you, when the flow is finished.

Because a flow may sometimes wish to configure the children in its progress hierarchy *before* the sub-flow is constructed, for sub-flows that always follow the same outline regardless of their parameters it's conventional to define a companion object/static method (for Kotlin/Java respectively) that constructs a tracker, and then allow the sub-flow to have the tracker it will use be passed in as a parameter. This allows all trackers to be built and linked ahead of time.

In future, the progress tracking framework will become a vital part of how exceptions, errors, and other faults are surfaced to human operators for investigation and resolution.

Future features

The flow framework is a key part of the platform and will be extended in major ways in future. Here are some of the features we have planned:

- Exception management, with a “flow hospital” tool to manually provide solutions to unavoidable problems (e.g. the other side doesn’t know the trade)
- Being able to interact with people, either via some sort of external ticketing system, or email, or a custom UI. For example to implement human transaction authorisations
- A standard library of flows that can be easily sub-classed by local developers in order to integrate internal reporting logic, or anything else that might be required as part of a communications lifecycle

- Writing flow tests
 - View page source
-

Writing flow tests

A flow can be a fairly complex thing that interacts with many services and other parties over the network. That means unit testing one requires some infrastructure to provide lightweight mock implementations. The MockNetwork provides this testing infrastructure layer; you can find this class in the test-utils module.

A good example to examine for learning how to unit test flows is the `ResolveTransactionsFlow` tests. This flow takes care of downloading and verifying transaction graphs, with all the needed dependencies. We start with this basic skeleton:

```
class ResolveTransactionsFlowTest {
    private lateinit var mockNet: InternalMockNetwork
    private lateinit var notaryNode: StartedNode<MockNode>
    private lateinit var megaCorpNode: StartedNode<MockNode>
    private lateinit var miniCorpNode: StartedNode<MockNode>
    private lateinit var megaCorp: Party
    private lateinit var miniCorp: Party
    private lateinit var notary: Party

    @Before
    fun setup() {
        mockNet = InternalMockNetwork(cordappPackages =
listOf("net.corda.testing.contracts", "net.corda.core.internal"))
        notaryNode = mockNet.defaultNotaryNode
        megaCorpNode = mockNet.createPartyNode(CordaX500Name("MegaCorp", "London",
"GB"))
        miniCorpNode = mockNet.createPartyNode(CordaX500Name("MiniCorp", "London",
"GB"))
        notary = mockNet.defaultNotaryIdentity
        megaCorp = megaCorpNode.info.singleIdentity()
        miniCorp = miniCorpNode.info.singleIdentity()
    }

    @After
    fun tearDown() {
        mockNet.stopNodes()
    }
}
```

We create a mock network in our `@Before` setup method and create a couple of nodes. We also record the identity of the notary in our test network, which will come in handy later. We also tidy up when we're done.

Next, we write a test case:

```

@Test
fun `resolve from two hashes`() {
    val (stx1, stx2) = makeTransactions()
    val p = TestFlow(setOf(stx2.id), megaCorp)
    val future = miniCorpNode.services.startFlow(p)
    mockNet.runNetwork()
    val results = future.resultFuture.getOrThrow()
    assertEquals(listOf(stx1.id, stx2.id), results.map { it.id })
    miniCorpNode.database.transaction {
        assertEquals(stx1,
            miniCorpNode.services.validatedTransactions.getTransaction(stx1.id))
        assertEquals(stx2,
            miniCorpNode.services.validatedTransactions.getTransaction(stx2.id))
    }
}

```

We'll take a look at the `makeTransactions` function in a moment. For now, it's enough to know that it returns two `SignedTransaction` objects, the second of which spends the first. Both transactions are known by `MegaCorpNode` but not `MiniCorpNode`.

The test logic is simple enough: we create the flow, giving it `MegaCorpNode`'s identity as the target to talk to. Then we start it on `MiniCorpNode` and use the `mockNet.runNetwork()` method to bounce messages around until things have settled (i.e. there are no more messages waiting to be delivered). All this is done using an in memory message routing implementation that is fast to initialise and use. Finally, we obtain the result of the flow and do some tests on it. We also check the contents of `MiniCorpNode`'s database to see that the flow had the intended effect on the node's persistent state.

Here's what `makeTransactions` looks like:

```

private fun makeTransactions(signFirstTX: Boolean = true, withAttachment: SecureHash? = null): Pair<SignedTransaction, SignedTransaction> {
    // Make a chain of custody of dummy states and insert into node A.
    val dummy1: SignedTransaction = DummyContract.generateInitial(0, notary,
        megaCorp.ref(1)).let {
        if (withAttachment != null)
            it.addAttachment(withAttachment)
        when (signFirstTX) {
            true -> {
                val ptx = megaCorpNode.services.signInitialTransaction(it)
                notaryNode.services.addSignature(ptx, notary.owningKey)
            }
            false -> {
                notaryNode.services.signInitialTransaction(it, notary.owningKey)
            }
        }
    }
    val dummy2: SignedTransaction = DummyContract.move(dummy1.tx.outRef(0),
        miniCorp).let {
        val ptx = megaCorpNode.services.signInitialTransaction(it)
        notaryNode.services.addSignature(ptx, notary.owningKey)
    }
}

```

```

    megaCorpNode.database.transaction {
        megaCorpNode.services.recordTransactions(dummy1, dummy2)
    }
    return Pair(dummy1, dummy2)
}

```

We're using the `DummyContract`, a simple test smart contract which stores a single number in its states, along with ownership and issuer information. You can issue such states, exit them and re-assign ownership (move them). It doesn't do anything else. This code simply creates a transaction that issues a dummy state (the issuer is `MEGA_CORP`, a pre-defined unit test identity), signs it with the test notary and MegaCorp keys and then converts the builder to the final `SignedTransaction`. It then does so again, but this time instead of issuing it re-assigns ownership instead. The chain of two transactions is finally committed to MegaCorpNode by sending them directly to the `megaCorpNode.services.recordTransaction` method (note that this method doesn't check the transactions are valid) inside a `database.transaction`. All node flows run within a database transaction in the nodes themselves, but any time we need to use the database directly from a unit test, you need to provide a database transaction as shown here.

[Next](#) [Previous](#)

- [Running a notary service](#)
 - [View page source](#)
-

Running a notary service

At present we have several prototype notary implementations:

1. `SimpleNotaryService` (single node) – commits the provided transaction input states without any validation.
2. `ValidatingNotaryService` (single node) – retrieves and validates the whole transaction history (including the given transaction) before committing.
3. `RaftNonValidatingNotaryService` (distributed) – functionally equivalent to `SimpleNotaryService`, but stores the committed states in a distributed collection replicated and persisted in a Raft cluster. For the consensus layer we are using the `Copycat` framework
4. `RaftValidatingNotaryService` (distributed) – as above, but performs validation on the transactions received

To have a node run a notary service, you need to set appropriate `notary` configuration before starting it (see [Node configuration](#) for reference).

For `SimpleNotaryService` the config is simply:

```
notary : { validating : false }
```

For `ValidatingNotaryService`, it is:

```
notary : { validating : true }
```

Setting up a Raft notary is currently slightly more involved and is not recommended for prototyping purposes. There is work in progress to simplify it. To see it in action, however, you can try out the notary-demo.

Use the `--bootstrap-raft-cluster` command line argument when starting the first node of a notary cluster for the first time. When the flag is set, the node will act as a seed for the cluster that other members can join.

[Next](#) [Previous](#)

- Writing oracle services
 - [View page source](#)
-

Writing oracle services

This article covers *oracles*: network services that link the ledger to the outside world by providing facts that affect the validity of transactions.

The current prototype includes an example oracle that provides an interest rate fixing service. It is used by the IRS trading demo app.

Introduction to oracles

Oracles are a key concept in the block chain/decentralised ledger space. They can be essential for many kinds of application, because we often wish to condition the validity of a transaction on some fact being true or false, but the ledger itself has a design that is essentially functional: all transactions

are *pure* and *immutable*. Phrased another way, a contract cannot perform any input/output or depend on any state outside of the transaction itself. For example, there is no way to download a web page or interact with the user from within a contract. It must be this way because everyone must be able to independently check a transaction and arrive at an identical conclusion regarding its validity for the ledger to maintain its integrity: if a transaction could evaluate to “valid” on one computer and then “invalid” a few minutes later on a different computer, the entire shared ledger concept wouldn’t work.

But transaction validity does often depend on data from the outside world - verifying that an interest rate swap is paying out correctly may require data on interest rates, verifying that a loan has reached maturity requires knowledge about the current time, knowing which side of a bet receives the payment may require arbitrary facts about the real world (e.g. the bankruptcy or solvency of a company or country), and so on.

We can solve this problem by introducing services that create digitally signed data structures which assert facts. These structures can then be used as an input to a transaction and distributed with the transaction data itself. Because the statements are themselves immutable and signed, it is impossible for an oracle to change its mind later and invalidate transactions that were previously found to be valid. In contrast, consider what would happen if a contract could do an HTTP request: it’s possible that an answer would change after being downloaded, resulting in loss of consensus.

The two basic approaches

The architecture provides two ways of implementing oracles with different tradeoffs:

1. Using commands
2. Using attachments

When a fact is encoded in a command, it is embedded in the transaction itself. The oracle then acts as a co-signer to the entire transaction. The oracle’s signature is valid only for that transaction, and thus even if a fact (like a stock price) does not change, every transaction that incorporates that fact must go back to the oracle for signing.

When a fact is encoded as an attachment, it is a separate object to the transaction and is referred to by hash. Nodes download attachments from peers at the same time as they download transactions, unless of course the node has already seen that attachment, in which case it won't fetch it again. Contracts have access to the contents of attachments when they run.

Note

Currently attachments do not support digital signing, but this is a planned feature.

As you can see, both approaches share a few things: they both allow arbitrary binary data to be provided to transactions (and thus contracts). The primary difference is whether the data is a freely reusable, standalone object or whether it's integrated with a transaction.

Here's a quick way to decide which approach makes more sense for your data source:

- Is your data *continuously changing*, like a stock price, the current time, etc? If yes, use a command.
- Is your data *commercially valuable*, like a feed which you are not allowed to resell unless it's incorporated into a business deal? If yes, use a command, so you can charge money for signing the same fact in each unique business context.
- Is your data *very small*, like a single number? If yes, use a command.
- Is your data *large, static* and *commercially worthless*, for instance, a holiday calendar? If yes, use an attachment.
- Is your data *intended for human consumption*, like a PDF of legal prose, or an Excel spreadsheet? If yes, use an attachment.

Asserting continuously varying data

Let's look at the interest rates oracle that can be found in the `NodeInterestRates` file. This is an example of an oracle that uses a command because the current interest rate fix is a constantly changing fact.

The obvious way to implement such a service is this:

1. The creator of the transaction that depends on the interest rate sends it to the oracle.
2. The oracle inserts a command with the rate and signs the transaction.
3. The oracle sends it back.

But this has a problem - it would mean that the oracle has to be the first entity to sign the transaction, which might impose ordering constraints we don't want to deal with (being able to get all parties to sign in parallel is a very nice thing). So the way we actually implement it is like this:

1. The creator of the transaction that depends on the interest rate asks for the current rate. They can abort at this point if they want to.
2. They insert a command with that rate and the time it was obtained into the transaction.
3. They then send it to the oracle for signing, along with everyone else, potentially in parallel. The oracle checks that the command has the correct data for the asserted time, and signs if so.

This same technique can be adapted to other types of oracle.

The oracle consists of a core class that implements the query/sign operations (for easy unit testing), and then a separate class that binds it to the network layer.

Here is an extract from the `NodeInterestRates.Oracle` class and supporting types:

```
/** A [FixOf] identifies the question side of a fix: what day, tenor and type of fix
("LIBOR", "EURIBOR" etc) */
@CordaSerializable
data class FixOf(val name: String, val forDay: LocalDate, val ofTenor: Tenor)

/** A [Fix] represents a named interest rate, on a given day, for a given duration. It
can be embedded in a tx. */
data class Fix(val of: FixOf, val value: BigDecimal) : CommandData

class Oracle {
    fun query(queries: List<FixOf>): List<Fix>
    fun sign(ftx: FilteredTransaction): TransactionSignature
}
```

The fix contains a timestamp (the `forDay` field) that identifies the version of the data being requested. Since there can be an arbitrary delay between a fix being requested via `query` and the signature being requested via `sign`, this timestamp

allows the Oracle to know which, potentially historical, value it is being asked to sign for. This is an important technique for continuously varying data.

Hiding transaction data from the oracle

Because the transaction is sent to the oracle for signing, ordinarily the oracle would be able to see the entire contents of that transaction including the inputs, output contract states and all the commands, not just the one (in this case) relevant command. This is an obvious privacy leak for the other participants. We currently solve this using a `FilteredTransaction`, which implements a Merkle Tree. These reveal only the necessary parts of the transaction to the oracle but still allow it to sign it by providing the Merkle hashes for the remaining parts. See [Oracles](#) for more details.

Pay-per-play oracles

Because the signature covers the transaction, and transactions may end up being forwarded anywhere, the fact itself is independently checkable. However, this approach can still be useful when the data itself costs money, because the act of issuing the signature in the first place can be charged for (e.g. by requiring the submission of a fresh `Cash.State` that has been re-assigned to a key owned by the oracle service). Because the signature covers the *transaction* and not only the *fact*, this allows for a kind of weak pseudo-DRM over data feeds. Whilst a contract could in theory include a transaction parsing and signature checking library, writing a contract in this way would be conclusive evidence of intent to disobey the rules of the service (*res ipsa loquitur*). In an environment where parties are legally identifiable, usage of such a contract would by itself be sufficient to trigger some sort of punishment.

Implementing an oracle with continuously varying data

Implement the core classes

The key is to implement your oracle in a similar way to the `NodeInterestRates.Oracle` outline we gave above with both a `query` and a `sign` method. Typically you would want one class that encapsulates the parameters to the `query` method (`FixOf`, above), and a `CommandData` implementation (`Fix`, above) that encapsulates both an instance of

that parameter class and an instance of whatever the result of the `query` is (`BigDecimal` above).

The `NodeInterestRates.Oracle` allows querying for multiple `Fix` objects but that is not necessary and is provided for the convenience of callers who need multiple fixes and want to be able to do it all in one query request.

Assuming you have a data source and can query it, it should be very easy to implement your `query` method and the parameter and `CommandData` classes.

Let's see how the `sign` method for `NodeInterestRates.Oracle` is written:

```
fun sign(ftx: FilteredTransaction): TransactionSignature {
    ftx.verify()
    // Performing validation of obtained filtered components.
    fun commandValidator(elem: Command<*>): Boolean {
        require(services.myInfo.legalIdentities.first().owningKey in elem.signers &&
        elem.value is Fix) {
            "Oracle received unknown command (not in signers or not Fix)."
        }
        val fix = elem.value as Fix
        val known = knownFixes[fix.of]
        if (known == null || known != fix)
            throw UnknownFix(fix.of)
        return true
    }

    fun check(elem: Any): Boolean {
        return when (elem) {
            is Command<*> -> commandValidator(elem)
            else -> throw IllegalArgumentException("Oracle received data of different
type than expected.")
        }
    }

    require(ftx.checkWithFun(::check))
    ftx.checkCommandVisibility(services.myInfo.legalIdentities.first().owningKey)
    // It all checks out, so we can return a signature.
    //
    // Note that we will happily sign an invalid transaction, as we are only being
presented with a filtered
    // version so we can't resolve or check it ourselves. However, that doesn't matter
much, as if we sign
    // an invalid transaction the signature is worthless.
    return services.createSignature(ftx,
        services.myInfo.legalIdentities.first().owningKey)
}
```

Here we can see that there are several steps:

1. Ensure that the transaction we have been sent is indeed valid and passes verification, even though we cannot see all of it

- Check that we only received commands as expected, and each of those commands expects us to sign for them and is of the expected type ([Fix](#) here)
- Iterate over each of the commands we identified in the last step and check that the data they represent matches exactly our data source. The final step, assuming we have got this far, is to generate a signature for the transaction and return it

Binding to the network

Note

Before reading any further, we advise that you understand the concept of flows and how to write them and use them. See [Writing flows](#). Likewise some understanding of Cordapps, plugins and services will be helpful. See [Running nodes locally](#).

The first step is to create the oracle as a service by annotating its class with [@CordaService](#). Let's see how that's done:

```
@CordaService
class Oracle(private val services: AppServiceHub) : SingletonSerializeAsToken() {
    private val mutex = ThreadBox(InnerState())

    init {
        // Set some default fixes to the Oracle, so we can smoothly run the IRS Demo
        // without uploading fixes.
        // This is required to avoid a situation where the runnodes version of the
        // demo isn't in a good state
        // upon startup.
        addDefaultFixes()
    }
}
```

The Corda node scans for any class with this annotation and initialises them. The only requirement is that the class provide a constructor with a single parameter of type [ServiceHub](#).

```
@InitiatedBy(RatesFixFlow.FixSignFlow::class)
class FixSignHandler(private val otherPartySession: FlowSession) : FlowLogic<Unit>() {
    @Suspendable
    override fun call() {
        val request = otherPartySession.receive<RatesFixFlow.SignRequest>().unwrap {
            it
        }
        val oracle = serviceHub.cordaService(Oracle::class.java)
        otherPartySession.send(oracle.sign(request.ftx))
    }
}

@InitiatedBy(RatesFixFlow.FixQueryFlow::class)
class FixQueryHandler(private val otherPartySession: FlowSession) : FlowLogic<Unit>() {
    object RECEIVED : ProgressTracker.Step("Received fix request")
    object SENDING : ProgressTracker.Step("Sending fix response")
```

```

override val progressTracker = ProgressTracker(RECEIVED, SENDING)

@Suspendable
override fun call() {
    val request = otherPartySession.receive<RatesFixFlow.QueryRequest>().unwrap {
it }
    progressTracker.currentStep = RECEIVED
    val oracle = serviceHub.cordaService(Oracle::class.java)
    val answers = oracle.query(request.queries)
    progressTracker.currentStep = SENDING
    otherPartySession.send(answers)
}
}

```

These two flows leverage the oracle to provide the querying and signing operations. They get reference to the oracle, which will have already been initialised by the node, using `ServiceHub.cordaService`. Both flows are annotated with `@InitiatedBy`. This tells the node which initiating flow (which are discussed in the next section) they are meant to be executed with.

Providing sub-flows for querying and signing

We mentioned the client sub-flow briefly above. They are the mechanism that clients, in the form of other flows, will use to interact with your oracle. Typically there will be one for querying and one for signing. Let's take a look at those for `NodeInterestRates.Oracle`.

```

@InitiatingFlow
class FixQueryFlow(val fixOf: FixOf, val oracle: Party) : FlowLogic<Fix>() {
    @Suspendable
    override fun call(): Fix {
        val oracleSession = initiateFlow(oracle)
        // TODO: add deadline to receive
        val resp =
    oracleSession.sendAndReceive<List<Fix>>(QueryRequest(listOf(fixOf)))

        return resp.unwrap {
            val fix = it.first()
            // Check the returned fix is for what we asked for.
            check(fix.of == fixOf)
            fix
        }
    }
}

@InitiatingFlow
class FixSignFlow(val tx: TransactionBuilder, val oracle: Party,
                  val partialMerkleTx: FilteredTransaction) :
FlowLogic<TransactionSignature>() {
    @Suspendable
    override fun call(): TransactionSignature {
        val oracleSession = initiateFlow(oracle)
        val resp =
    oracleSession.sendAndReceive<TransactionSignature>(SignRequest(partialMerkleTx))
        return resp.unwrap { sig ->
            check(oracleSession.counterparty.owningKey.isFulfilledBy(listOf(sig.by)))
}
}

```

```
        tx.toWireTransaction(serviceHub).checkSignature(sig)
    }
}
```

You'll note that the `FixSignFlow` requires a `FilterTransaction` instance which includes only `Fix` commands. You can find a further explanation of this in [Oracles](#). Below you will see how to build such a transaction with hidden fields.

Using an oracle

The oracle is invoked through sub-flows to query for values, add them to the transaction as commands and then get the transaction signed by the oracle. Following on from the above examples, this is all encapsulated in a sub-flow called `RatesFixFlow`. Here's the `call` method of that flow.

```
@Suspendable
override fun call(): TransactionSignature {
    progressTracker.currentStep = progressTracker.steps[1]
    val fix = subFlow(FixQueryFlow(fixOf, oracle))
    progressTracker.currentStep = WORKING
    checkFixIsNearExpected(fix)
    tx.addCommand(fix, oracle.owningKey)
    beforeSigning(fix)
    progressTracker.currentStep = SIGNING
    val mtx = tx.toWireTransaction(serviceHub).buildFilteredTransaction(Predicate {
        filtering(it) })
    return subFlow(FixSignFlow(tx, oracle, mtx))
}
```

As you can see, this:

1. Queries the oracle for the fact using the client sub-flow for querying defined above
 2. Does some quick validation
 3. Adds the command to the transaction containing the fact to be signed for by the oracle
 4. Calls an extension point that allows clients to generate output states based on the fact from the oracle
 5. Builds filtered transaction based on filtering function extended from `RatesFixFlow`
 6. Requests the signature from the oracle using the client sub-flow for signing from above

Here's an example of it in action from `FixingFlow.Fixer`.

```

        val addFixing = object : RatesFixFlow(ptx, handshake.payload.oracle, fixOf,
BigDecimal.ZERO, BigDecimal.ONE) {
    @Suspendable
    override fun beforeSigning(fix: Fix) {
        newDeal.generateFix(ptx, StateAndRef(txState, handshake.payload.ref),
fix)

        // We set the transaction's time-window: it may be that none of the
contracts need this!
        // But it can't hurt to have one.
        ptx.setTimeWindow(serviceHub.clock.instant(), 30.seconds)
    }

    @Suspendable
    override fun filtering(elem: Any): Boolean {
        return when (elem) {
            // Only expose Fix commands in which the oracle is on the list of
requested signers
            // to the oracle node, to avoid leaking privacy
            is Command<*> -> handshake.payload.oracle.owningKey in
elem.signers && elem.value is Fix
            else -> false
        }
    }
}
val sig = subFlow(addFixing)

```

Note

When overriding be careful when making the sub-class an anonymous or inner class (object declarations in Kotlin), because that kind of classes can access variables from the enclosing scope and cause serialization problems when checkpointed.

Testing

The `MockNetwork` allows the creation of `MockNode` instances, which are simplified nodes which can be used for testing (see [API: Testing](#)). When creating the `MockNetwork` you supply a list of packages to scan for CorDapps. Make sure the packages you provide include your oracle service, and it automatically be installed in the test nodes. Then you can create an oracle node on the `MockNetwork` and insert any initialisation logic you want to use. In this case, our `Oracle` service is in the `net.corda.irs.api` package, so the following test setup will install the service in each node. Then an oracle node with an oracle service which is initialised with some data is created on the mock network:

```

fun setUp() {
    mockNet = MockNetwork(cordappPackages = listOf("net.corda.finance.contracts",
"net.corda.irs"))
    aliceNode = mockNet.createPartyNode(ALICE_NAME)
    oracleNode = mockNet.createNode(MockNodeParameters(legalName = BOB_NAME)).apply {
        transaction {
            services.cordaService(NodeInterestRates.Oracle::class.java).knownFixes =
TEST_DATA
    }
}

```

```
        }
    }
}
```

You can then write tests on your mock network to verify the nodes interact with your Oracle correctly.

```
@Test
fun `verify that the oracle signs the transaction if the interest rate within allowed
limit`() {
    // Create a partial transaction
    val tx = TransactionBuilder(DUMMY_NOTARY)
        .withItems(TransactionState(1000.DOLLARS.CASH issuedBy
dummyCashIssuer.party ownedBy alice.party, Cash.PROGRAM_ID, DUMMY_NOTARY))
    // Specify the rate we wish to get verified by the oracle
    val fixOf = NodeInterestRates.parseFixOf("LIBOR 2016-03-16 1M")

    // Create a new flow for the fix
    val flow = FilteredRatesFlow(tx, oracle, fixOf, BigDecimal("0.675"),
BigDecimal("0.1"))
    // Run the mock network and wait for a result
    mockNet.runNetwork()
    val future = aliceNode.startFlow(flow)
    mockNet.runNetwork()
    future.getOrThrow()

    // We should now have a valid rate on our tx from the oracle.
    val fix = tx.toWireTransaction(aliceNode.services).commands.map { it }.first()
    assertEquals(fixOf, (fix.value as Fix).of)
    // Check that the response contains the valid rate, which is within the supplied
tolerance
    assertEquals(BigDecimal("0.678"), (fix.value as Fix).value)
    // Check that the transaction has been signed by the oracle
    assertContains(fix.signers, oracle.owningKey)
}
```

See [here](#) for more examples.

[Next](#) [Previous](#)

- Writing a custom notary service (experimental)
- View page source

Writing a custom notary service (experimental)

Warning

Customising a notary service is still an experimental feature and not recommended for most use-cases. The APIs for writing a custom notary may change in the future. Additionally, customising Raft or BFT notaries is not yet

fully supported. If you want to write your own Raft notary you will have to implement a custom database connector (or use a separate database for the notary), and use a custom configuration file.

Similarly to writing an oracle service, the first step is to create a service class in your CorDapp and annotate it with `@CordaService`. The Corda node scans for any class with this annotation and initialises them. The custom notary service class should provide a constructor with two parameters of types `AppServiceHub` and `PublicKey`.

```
@CordaService
class MyCustomValidatingNotaryService(override val services: AppServiceHub, override
val notaryIdentityKey: PublicKey) : TrustedAuthorityNotaryService() {
    override val uniquenessProvider = PersistentUniquenessProvider()

    override fun createServiceFlow(otherPartySession: FlowSession): FlowLogic<Void?> =
        MyValidatingNotaryFlow(otherPartySession, this)

    override fun start() {}
    override fun stop() {}
}
```

The next step is to write a notary service flow. You are free to copy and modify the existing built-in flows such as `ValidatingNotaryFlow`, `NonValidatingNotaryFlow`, or implement your own from scratch (following the `NotaryFlow.Service` template).

Below is an example of a custom flow for a *validating* notary service:

```
class MyValidatingNotaryFlow(otherSide: FlowSession, service:
MyCustomValidatingNotaryService) : NotaryFlow.Service(otherSide, service) {
    /**
     * The received transaction is checked for contract-validity, for which the caller
     * also has to reveal the whole
     * transaction dependency chain.
     */
    @Suspendable
    override fun receiveAndVerifyTx(): TransactionParts {
        try {
            val stx = receiveTransaction()
            val notary = stx.notary
            checkNotary(notary)
            verifySignatures(stx)
            resolveAndContractVerify(stx)
            val timeWindow: TimeWindow? = if (stx.coreTransaction is WireTransaction)
                stx.tx.timeWindow else null
            return TransactionParts(stx.id, stx.inputs, timeWindow, notary!!)
        } catch (e: Exception) {
            throw when (e) {
                is TransactionVerificationException,
                is SignatureException ->
                    NotaryInternalException(NotaryError.TransactionInvalid(e))
                else -> e
            }
        }
    }
}

@Suspendable
```

```

private fun receiveTransaction(): SignedTransaction {
    return otherSideSession.receive<NotarisationPayload>().unwrap {
        val stx = it.signedTransaction
        validateRequest(NotarisationRequest(stx.inputs, stx.id),
it.requestSignature)
        stx
    }
}

@Suspendable
private fun resolveAndContractVerify(stx: SignedTransaction) {
    subFlow(ResolveTransactionsFlow(stx, otherSideSession))
    stx.verify(serviceHub, false)
    customVerify(stx)
}

private fun verifySignatures(stx: SignedTransaction) {
    val transactionWithSignatures =
stx.resolveTransactionWithSignatures(serviceHub)
    checkSignatures(transactionWithSignatures)
}

private fun checkSignatures(tx: TransactionWithSignatures) {
    try {
        tx.verifySignaturesExcept(service.notaryIdentityKey)
    } catch (e: SignatureException) {
        throw NotaryInternalException(NotaryError.TransactionInvalid(e))
    }
}

private fun customVerify(stx: SignedTransaction) {
    // Add custom verification logic
}
}

```

To enable the service, add the following to the node configuration:

```

notary : {
    validating : true # Set to false if your service is non-validating
    custom : true
}

```

[Next](#) [Previous](#)

- [Transaction tear-offs](#)
 - [View page source](#)
-

Transaction tear-offs

Suppose we want to construct a transaction that includes commands containing interest rate fix data as in [Writing oracle services](#). Before sending the transaction to the oracle to obtain its signature, we need to filter out every part of the transaction except for the `Fix` commands.

To do so, we need to create a filtering function that specifies which fields of the transaction should be included. Each field will only be included if the filtering function returns *true* when the field is passed in as input.

```
val filtering = Predicate<Any> {
    when (it) {
        is Command<*> -> oracle.owningKey in it.signers && it.value is Fix
        else -> false
    }
}
```

We can now use our filtering function to construct a `FilteredTransaction`:

```
val ftx: FilteredTransaction = stx.buildFilteredTransaction(filtering)
```

In the Oracle example this step takes place in `RatesFixFlow` by overriding the `filtering` function. See [Using an oracle](#).

Both `WireTransaction` and `FilteredTransaction` inherit from `TraversableTransaction`, so access to the transaction components is exactly the same. Note that unlike `WireTransaction`, `FilteredTransaction` only holds data that we wanted to reveal (after filtering).

```
// Direct access to included commands, inputs, outputs, attachments etc.
val cmds: List<Command<*>> = ftx.commands
val ins: List<StateRef> = ftx.inputs
val timeWindow: TimeWindow? = ftx.timeWindow
// ...
```

The following code snippet is taken from `NodeInterestRates.kt` and implements a signing part of an Oracle.

```
fun sign(ftx: FilteredTransaction): TransactionSignature {
    ftx.verify()
    // Performing validation of obtained filtered components.
    fun commandValidator(elem: Command<*>): Boolean {
        require(services.myInfo.legalIdentities.first().owningKey in elem.signers &&
            elem.value is Fix) {
            "Oracle received unknown command (not in signers or not Fix)."
        }
        val fix = elem.value as Fix
        val known = knownFixes[fix.of]
        if (known == null || known != fix)
            throw UnknownFix(fix.of)
        return true
    }

    fun check(elem: Any): Boolean {
        return when (elem) {
            is Command<*> -> commandValidator(elem)
            else -> throw IllegalArgumentException("Oracle received data of different
type than expected.")
        }
    }
}
```

```

require(ftx.checkWithFun(::check))
ftx.checkCommandVisibility(services.myInfo.legalIdentities.first().owningKey)
// It all checks out, so we can return a signature.
//
// Note that we will happily sign an invalid transaction, as we are only being
presented with a filtered
// version so we can't resolve or check it ourselves. However, that doesn't matter
much, as if we sign
// an invalid transaction the signature is worthless.
return services.createSignature(ftx,
services.myInfo.legalIdentities.first().owningKey)
}

```

Note

The way the `FilteredTransaction` is constructed ensures that after signing of the root hash it's impossible to add or remove

components (leaves). However, it can happen that having transaction with multiple commands one party reveals only subset of them to the Oracle. As signing is done now over the Merkle root hash, the service signs all commands of given type, even though it didn't see all of them. In the case however where all of the commands should be visible to an Oracle, one can type `ftx.checkAllComponentsVisible(COMMANDS_GROUP)` before invoking `ftx.verify`.

`checkAllComponentsVisible` is using a sophisticated underlying partial Merkle tree check to guarantee that all of the components of a particular group that existed in the original `WireTransaction` are included in the received `FilteredTransaction`.

[Next](#) [Previous](#)

- Using attachments
- [View page source](#)

Using attachments

Attachments are ZIP/JAR files referenced from transaction by hash, but not included in the transaction itself. These files are automatically requested from the node sending the transaction when needed and cached locally so they are not re-requested if encountered again. Attachments typically contain:

- Contract executable code
- Metadata about a transaction, such as PDF version of an invoice being settled

- Shared information to be permanently recorded on the ledger

To add attachments the file must first be uploaded to the node, which returns a unique ID that can be added using `TransactionBuilder.addAttachment()`.

Attachments can be uploaded and downloaded via RPC and the Corda Shell.

It is encouraged that where possible attachments are reusable data, so that nodes can meaningfully cache them.

Uploading and downloading

To upload an attachment to the node, or download an attachment named by its hash, you use [Client RPC](#). This is also available for interactive use via the shell.

To **upload** run:

```
>>> run uploadAttachment jar: /path/to/the/file.jar
```

or

```
>>> run uploadAttachmentWithMetadata jar: /path/to/the/file.jar, uploader: myself, filename: original_name.jar
```

to include the metadata with the attachment which can be used to find it later on. Note, that currently both uploader and filename are just plain strings (there is no connection between uploader and the RPC users for example).

The file is uploaded, checked and if successful the hash of the file is returned. This is how the attachment is identified inside the node.

To download an attachment, you can do:

```
>>> run openAttachment id: AB7FED7663A3F195A59A0F01091932B15C22405CB727A1518418BF53C6  
E6663A
```

which will then ask you to provide a path to save the file to. To do the same thing programmatically, you can pass a simple `InputStream` or `SecureHash` to the `uploadAttachment` / `openAttachment` RPCs from a JVM client.

Searching for attachments

Attachments metadata can be used to query, in the similar manner as API: Vault Query.

`AttachmentQueryCriteria` can be used to build a query, utilizing set of operations per column, namely:

- Binary logical (AND, OR)
- Comparison (LESS_THAN, LESS_THAN_OR_EQUAL, GREATER_THAN, GREATER_THAN_OR_EQUAL)
- Equality (EQUAL, NOT_EQUAL)
- Likeness (LIKE, NOT_LIKE)
- Nullability (IS_NULL, NOT_NULL)
- Collection based (IN, NOT_IN)

`And` and `or` operators can be used to build queries of arbitrary complexity.

Example of such query:

```
assertEquals(  
    emptyList(),  
    storage.queryAttachments(  
        AttachmentQueryCriteria.AttachmentsQueryCriteria(uploaderCondition =  
Builder.equal("complexA"))  
            .and(AttachmentQueryCriteria.AttachmentsQueryCriteria(uploaderCondition =  
Builder.equal("complexB"))))  
)  
  
assertEquals(  
    listOf(hashA, hashB),  
    storage.queryAttachments(  
        AttachmentQueryCriteria.AttachmentsQueryCriteria(uploaderCondition =  
Builder.equal("complexA"))  
            .or(AttachmentQueryCriteria.AttachmentsQueryCriteria(uploaderCondition =  
Builder.equal("complexB"))))  
)  
  
val complexCondition =  
  
(uploaderCondition("complexB").and(filenamerCondition("archiveB.zip"))).or(filenamerCo  
ndition("archiveC.zip"))
```

Protocol

Normally attachments on transactions are fetched automatically via the `ReceiveTransactionFlow`. Attachments are needed in order to validate a transaction (they include, for example, the contract code), so must be fetched before the validation process can run.

Note

Future versions of Corda may support non-critical attachments that are not used for transaction verification and which are shared explicitly. These are useful for attaching and signing auditing data with a transaction that isn't used as part of the contract logic.

Attachments demo

There is a worked example of attachments, which relays a simple document from one node to another. The “two party trade flow” also includes an attachment, however it is a significantly more complex demo, and less well suited for a tutorial.

The demo code is in the file `samples/attachment-demo/src/main/kotlin/net/corda/attachmentdemo/AttachmentDemo.kt`, with the core logic contained within the two functions `recipient()` and `sender()`. The first thing it does is set up an RPC connection to node B using a demo user account (this is all configured in the gradle build script for the demo and the nodes will be created using the `deployNodes` gradle task as normal). The `CordaRPCClient.use` method is a convenience helper intended for small tools that sets up an RPC connection scoped to the provided block, and brings all the RPCs into scope. Once connected the sender/recipient functions are run with the RPC proxy as a parameter.

We'll look at the recipient function first.

The first thing it does is wait to receive a notification of a new transaction by calling the `verifiedTransactions` RPC, which returns both a snapshot and an observable of changes. The observable is made blocking and the next transaction the node verifies is retrieved. That transaction is checked to see if it has the expected attachment and if so, printed out.

```
fun recipient(rpc: CordaRPCOps, webPort: Int) {
    println("Waiting to receive transaction ...")
    val stx = rpc.internalVerifiedTransactionsFeed().updates.toBlocking().first()
    val wtx = stx.tx
    if (wtx.attachments.isNotEmpty()) {
        if (wtx.outputs.isNotEmpty()) {
            val state = wtx.outputsOfType<AttachmentContract.State>().single()
            require(rpc.attachmentExists(state.hash))

            // Download the attachment via the Web endpoint.
        }
    }
}
```

```

    val connection =
URL("http://localhost:$webPort/attachments/${state.hash}").openConnection() as
HttpURLConnection
    try {
        require(connection.responseCode == SC_OK) { "HTTP status code was
${connection.responseCode}" }
        require(connection.contentType == APPLICATION_OCTET_STREAM) {
"Content-Type header was ${connection.contentType}" }
        require(connection.getHeaderField(CONTENT_DISPOSITION) == "attachment;
filename=\"${state.hash}.zip\"")
        "Content-Disposition header was
${connection.getHeaderField(CONTENT_DISPOSITION)}"
    }

    // Write out the entries inside this jar.
    println("Attachment JAR contains these entries:")
    JarInputStream(connection.inputStream).use { it ->
        while (true) {
            val e = it.nextJarEntry ?: break
            println("Entry> ${e.name}")
            it.closeEntry()
        }
    }
} finally {
    connection.disconnect()
}
println("File received - we're happy!\n\nFinal transaction
is:\n\n${Emoji.renderIfSupported(wtx)}")
} else {
    println("Error: no output state found in ${wtx.id}")
}
} else {
    println("Error: no attachments found in ${wtx.id}")
}
}
}

```

The sender correspondingly builds a transaction with the attachment, then calls `FinalityFlow` to complete the transaction and send it to the recipient node:

```

fun sender(rpc: CordaRPCOps, numOfClearBytes: Int = 1024) { // default size 1K.
    val (inputStream, hash) =
InputStreamAndHash.createInMemoryTestZip(numOfClearBytes, 0)
    val executor = Executors.newScheduledThreadPool(2)
    try {
        sender(rpc, inputStream, hash, executor)
    } finally {
        executor.shutdown()
    }
}

private fun sender(rpc: CordaRPCOps, inputStream: InputStream, hash:
SecureHash.SHA256, executor: ScheduledExecutorService) {

    // Get the identity key of the other side (the recipient).
    val notaryFuture: CordaFuture<Party> = poll(executor,
DUMMY_NOTARY_NAME.toString()) { rpc.wellKnownPartyFromX500Name(DUMMY_NOTARY_NAME) }
    val otherSideFuture: CordaFuture<Party> = poll(executor,
DUMMY_BANK_B_NAME.toString()) { rpc.wellKnownPartyFromX500Name(DUMMY_BANK_B_NAME) }
    // Make sure we have the file in storage
    if (!rpc.attachmentExists(hash)) {
        inputStream.use {
            val avail = inputStream.available()
            val id = rpc.uploadAttachment(it)
            require(hash == id) { "Id was '$id' instead of '$hash'" }
        }
    }
}

```

```

        }
        require(rpc.attachmentExists(hash))
    }

    val flowHandle = rpc.startTrackedFlow(::AttachmentDemoFlow, otherSideFuture.get(),
notaryFuture.get(), hash)
    flowHandle.progress.subscribe(::println)
    val stx = flowHandle.returnValue.getOrThrow()
    println("Sent ${stx.id}")
}

```

This side is a bit more complex. Firstly it looks up its counterparty by name in the network map. Then, if the node doesn't already have the attachment in its storage, we upload it from a JAR resource and check the hash was what we expected. Then a trivial transaction is built that has the attachment and a single signature and it's sent to the other side using the FinalityFlow. The result of starting the flow is a stream of progress messages and a `returnValue` observable that can be used to watch out for the flow completing successfully.

[Next](#) [Previous](#)

Event scheduling

This article explains our approach to modelling time based events in code. It explains how a contract state can expose an upcoming event and what action to take if the scheduled time for that event is reached.

Introduction

Many financial instruments have time sensitive components to them. For example, an Interest Rate Swap has a schedule for when:

- Interest rate fixings should take place for floating legs, so that the interest rate used as the basis for payments can be agreed.
- Any payments between the parties are expected to take place.
- Any payments between the parties become overdue.

Each of these is dependent on the current state of the financial instrument. What payments and interest rate fixings have already happened should already be recorded in the state, for example. This means that the *next* time sensitive event is thus a property of the current contract state. By next, we mean earliest in chronological terms, that is still due. If a contract state is consumed in the UTXO model, then what *was* the next event becomes irrelevant and obsolete and the next time sensitive event is determined by any successor contract state.

Knowing when the next time sensitive event is due to occur is useful, but typically some *activity* is expected to take place when this event occurs. We already have a model for business processes in the form of *flows*, so in the platform we have introduced the concept of *scheduled activities* that can invoke flow state machines at a scheduled time. A contract state can optionally describe the next scheduled activity for itself. If it omits to do so, then nothing will be scheduled.

How to implement scheduled events

There are two main steps to implementing scheduled events:

- Have your `ContractState` implementation also implement `SchedulableState`. This requires a method named `nextScheduledActivity` to be implemented which returns an optional `ScheduledActivity` instance. `ScheduledActivity` captures what `FlowLogic` instance each node will run, to perform the activity, and when it will run is described by a `java.time.Instant`. Once your state implements this interface and is tracked by the vault, it can expect to be queried for the next activity when committed to the vault. The `FlowLogic` must be annotated with `@SchedulableFlow`.
- If nothing suitable exists, implement a `FlowLogic` to be executed by each node as the activity itself. The important thing to remember is that in the current implementation, each node that is party to the transaction will execute the same `FlowLogic`, so it needs to establish roles in the business process based on the contract state and the node it is running on. Each side will follow different but complementary paths through the business logic.

Note

The scheduler's clock always operates in the UTC time zone for uniformity, so any time zone logic must be performed by the contract, using `ZonedDateTime`.

In the short term, until we have automatic flow session set up, you will also likely need to install a network handler to help with obtaining a unique and secure random session. An example is described below.

The production and consumption of `ContractStates` is observed by the scheduler and the activities associated with any consumed states are unscheduled. Any newly produced states are then queried via the `nextScheduledActivity` method and if they do not return `null` then that activity is scheduled based on the content of

the `ScheduledActivity` object returned. Be aware that this *only* happens if the vault considers the state “relevant”, for instance, because the owner of the node also owns that state. States that your node happens to encounter but which aren’t related to yourself will not have any activities scheduled.

An example

Let’s take an example of the interest rate swap fixings for our scheduled events. The first task is to implement the `nextScheduledActivity` method on the `State`.

Kotlin

```
override fun nextScheduledActivity(thisStateRef: StateRef, flowLogicRefFactory: FlowLogicRefFactory): ScheduledActivity? {
    val nextFixingOf = nextFixingOf() ?: return null

    // This is perhaps not how we should determine the time point in the business day,
    // but instead expect the schedule to detail some of these aspects
    val instant = suggestInterestRateAnnouncementTimeWindow(index = nextFixingOf.name,
        source = floatingLeg.indexSource, date = nextFixingOf.forDay).fromTime!!
    return
    ScheduledActivity(flowLogicRefFactory.create("net.corda.irs.flows.FixingFlow\$FixingRoleDecider", thisStateRef), instant)
}
```

The first thing this does is establish if there are any remaining fixings. If there are none, then it returns `null` to indicate that there is no activity to schedule.

Otherwise it calculates the `Instant` at which the interest rate should become available and schedules an activity at that time to work out what roles each node will take in the fixing business process and to take on those roles.

That `FlowLogic` will be handed the `StateRef` for the interest rate swap `State` in question, as well as a tolerance `Duration` of how long to wait after the activity is triggered for the interest rate before indicating an error.

Note

This is a way to create a reference to the `FlowLogic` class and its constructor parameters to instantiate.

As previously mentioned, we currently need a small network handler to assist with session setup until the work to automate that is complete. See the interest rate swap specific implementation `FixingSessionInitiationHandler` which is responsible for starting a `FlowLogic` to perform one role in the fixing flow with the `sessionID` sent by the `FixingRoleDecider` on the other node which then launches the other role in the fixing flow. Currently the handler needs to be manually installed in the node.

- Observer nodes
 - [View page source](#)
-

Observer nodes

Posting transactions to an observer node is a common requirement in finance, where regulators often want to receive comprehensive reporting on all actions taken. By running their own node, regulators can receive a stream of digitally signed, de-duplicated reports useful for later processing.

Adding support for observer nodes to your application is easy. The IRS (interest rate swap) demo shows to do it.

Just define a new flow that wraps the `SendTransactionFlow/ReceiveTransactionFlow`, as follows:

Kotlin

```
@InitiatedBy(Requester::class)
class AutoOfferAcceptor(otherSideSession: FlowSession) :
    Acceptor(otherSideSession) {
    @Suspendable
    override fun call(): SignedTransaction {
        val finalTx = super.call()
        // Our transaction is now committed to the Ledger, so report it to our
        regulator. We use a custom flow
        // that wraps SendTransactionFlow to allow the receiver to customise how
        ReceiveTransactionFlow is run,
        // and because in a real life app you'd probably have more complex logic
        here e.g. describing why the report
        // was filed, checking that the reportee is a regulated entity and not
        some random node from the wrong
        // country and so on.
        val regulator = serviceHub.identityService.partiesFromName("Regulator",
true).single()
        subFlow(ReportToRegulatorFlow(regulator, finalTx))
        return finalTx
    }
}

@InitiatingFlow
class ReportToRegulatorFlow(private val regulator: Party, private val finalTx:
SignedTransaction) : FlowLogic<Unit>() {
    @Suspendable
    override fun call() {
        val session = initiateFlow(regulator)
        subFlow(SendTransactionFlow(session, finalTx))
    }
}
```

```

@InitiatedBy(ReportToRegulatorFlow::class)
class ReceiveRegulatoryReportFlow(private val otherSideSession: FlowSession) :
FlowLogic<Unit>() {
    @Suspendable
    override fun call() {
        // Start the matching side of SendTransactionFlow above, but tell it to
        record all visible states even
        // though they (as far as the node can tell) are nothing to do with us.
        subFlow(ReceiveTransactionFlow(otherSideSession, true,
StatesToRecord.ALL_VISIBLE))
    }
}

```

In this example, the `AutoOfferFlow` is the business logic, and we define two very short and simple flows to send the transaction to the regulator. There are two important aspects to note here:

1. The `ReportToRegulatorFlow` is marked as an `@InitiatingFlow` because it will start a new conversation, context free, with the regulator.
2. The `ReceiveRegulatoryReportFlow` uses `ReceiveTransactionFlow` in a special way - it tells it to send the transaction to the vault for processing, including all states even if not involving our public keys. This is required because otherwise the vault will ignore states that don't list any of the node's public keys, but in this case, we do want to passively observe states we can't change. So overriding this behaviour is required.

If the states define a relational mapping (see API: Persistence) then the regulator will be able to query the reports from their database and observe new transactions coming in via RPC.

Caveats

- Nodes which act as both observers and direct participants in the ledger are not supported at this time. In particular, coin selection may return states which you do not have the private keys to be able to sign for. Future versions of Corda may address this issue, but for now, if you wish to both participate in the ledger and also observe transactions that you can't sign for you will need to run two nodes and have two separate identities
- Nodes only record each transaction once. If a node has already recorded a transaction in non-observer mode, it cannot later re-record the same transaction as an observer. This issue is tracked here: <https://r3-cev.atlassian.net/browse/CORDA-883>

- [Tools](#)
 - [View page source](#)
-

Tools

- Corda Network Builder
- Network Bootstrapper
- Blob Inspector
- Network Simulator
- DemoBench
- Node Explorer
- Building a Corda Network on Azure Marketplace
- Building a Corda VM from the AWS Marketplace
- Load testing

[Next](#) [Previous](#)

- [Corda Network Builder](#)
 - [View page source](#)
-

Corda Network Builder

Contents

- [Corda Network Builder](#)
 - [Prerequisites](#)
 - [Creating the base nodes](#)
 - [Building a network via the command line](#)
 - [Starting the nodes](#)
 - [Quickstart Local Docker](#)
 - [Quickstart Remote Azure](#)
 - [Interacting with the nodes](#)
 - [Adding additional nodes](#)
 - [Building a network in Graphical User Mode](#)
 - [Starting the nodes](#)
 - [Interacting with the nodes](#)

- [Adding additional nodes](#)
- [Shutting down the nodes](#)

The Corda Network Builder is a tool for building Corda networks for testing purposes. It leverages Docker and containers to abstract the complexity of managing a distributed network away from the user.

Currently, the network you build will either be made up of local `docker` nodes or of nodes spread across Azure containers.

The Corda Network Builder can be downloaded from <https://ci-artifactory.corda.r3cev.com/artifactory/corda-releases/net/corda/corda-network-builder/X.Y-corda/corda-network-builder-X.Y-corda-executable.jar>, where `X` is the major Corda version and `Y` is the minor Corda version.

Prerequisites

- **Docker:** docker > 17.12.0-ce
- **Azure:** authenticated az-cli >= 2.0 (see: <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest>)

Creating the base nodes

The network builder uses a set of nodes as the base for all other operations. A node is anything that satisfies the following layout:

```
-  
-- node.conf  
-- corda.jar  
-- cordapps/
```

An easy way to build a valid set of nodes is by running `deployNodes`. In this document, we will be using the output of running `deployNodes` for the [Example CorDapp](#):

1. `git clone https://github.com/corda/cordapp-example`
2. `cd cordapp-example`
3. `./gradlew clean deployNodes`

Building a network via the command line

Starting the nodes

Quickstart Local Docker

1. `cd kotlin-source/build/nodes`
2. `java -jar <path/to/network-builder-jar> -d .`

If you run `docker ps` to see the running containers, the following output should be displayed:

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS		
406868b4ba69	node-partyc:corda-network	<code>"/run-corda.sh"</code>	17 seconds ago
Up 16 seconds	0.0.0.0:32902->10003/tcp, 0.0.0.0:32895->10005/tcp, 0.0.0.0:32898->10020/tcp, 0.0.0.0:32900->12222/tcp partyc0		
4546a2fa8de7	node-partyb:corda-network	<code>"/run-corda.sh"</code>	17 seconds ago
Up 17 seconds	0.0.0.0:32896->10003/tcp, 0.0.0.0:32899->10005/tcp, 0.0.0.0:32901->10020/tcp, 0.0.0.0:32903->12222/tcp partyb0		
c8c44c515bdb	node-partya:corda-network	<code>"/run-corda.sh"</code>	17 seconds ago
Up 17 seconds	0.0.0.0:32894->10003/tcp, 0.0.0.0:32897->10005/tcp, 0.0.0.0:32892->10020/tcp, 0.0.0.0:32893->12222/tcp partya0		
cf7ab689f493	node-notary:corda-network	<code>"/run-corda.sh"</code>	30 seconds ago
Up 31 seconds	0.0.0.0:32888->10003/tcp, 0.0.0.0:32889->10005/tcp, 0.0.0.0:32890->10020/tcp, 0.0.0.0:32891->12222/tcp notary0		

Quickstart Remote Azure

1. `cd kotlin-source/build/nodes`
2. `java -jar <path/to/network-builder-jar> -b AZURE -d .`

Note

The Azure configuration is handled by the az-cli utility. See the [Prerequisites](#).

Interacting with the nodes

You can interact with the nodes by SSHing into them on the port that is mapped to 12222. For example, to SSH into the `partya0` node, you would run:

```
ssh user1@localhost -p 32893
Password authentication
Password:
```

```
Welcome to the Corda interactive shell.
Useful commands include 'help' to see what is available, and 'bye' to shut down the
node.
```

```
>>> run networkMapSnapshot
[
  { "addresses" : [ "partya0:10020" ], "legalIdentitiesAndCerts" : [ "O=PartyA,
L=London, C=GB" ], "platformVersion" : 3, "serial" : 1532701330613 },
```

```

    { "addresses" : [ "notary0:10020" ], "legalIdentitiesAndCerts" : [ "O=Notary,
L=London, C=GB" ], "platformVersion" : 3, "serial" : 1532701305115 },
    { "addresses" : [ "partyc0:10020" ], "legalIdentitiesAndCerts" : [ "O=PartyC,
L=Paris, C=FR" ], "platformVersion" : 3, "serial" : 1532701331608 },
    { "addresses" : [ "partyb0:10020" ], "legalIdentitiesAndCerts" : [ "O=PartyB, L>New
York, C=US" ], "platformVersion" : 3, "serial" : 1532701330118 }
]

>>>

```

Adding additional nodes

It is possible to add additional nodes to the network by reusing the nodes you built earlier. For example, to add a node by reusing the existing **PartyA** node, you would run:

```
java -jar <network-builder-jar> --add "PartyA=O=PartyZ,L=London,C=GB"
```

To confirm the node has been started correctly, run the following in the previously connected SSH session:

```

Tue Jul 17 15:47:14 GMT 2018>>> run networkMapSnapshot
[
    { "addresses" : [ "partya0:10020" ], "legalIdentitiesAndCerts" : [ "O=PartyA,
L=London, C=GB" ], "platformVersion" : 3, "serial" : 1532701330613 },
    { "addresses" : [ "notary0:10020" ], "legalIdentitiesAndCerts" : [ "O=Notary,
L=London, C=GB" ], "platformVersion" : 3, "serial" : 1532701305115 },
    { "addresses" : [ "partyc0:10020" ], "legalIdentitiesAndCerts" : [ "O=PartyC,
L=Paris, C=FR" ], "platformVersion" : 3, "serial" : 1532701331608 },
    { "addresses" : [ "partyb0:10020" ], "legalIdentitiesAndCerts" : [ "O=PartyB, L>New
York, C=US" ], "platformVersion" : 3, "serial" : 1532701330118 },
    { "addresses" : [ "partya1:10020" ], "legalIdentitiesAndCerts" : [ "O=PartyZ,
L=London, C=GB" ], "platformVersion" : 3, "serial" : 1532701630861 }
]
```

Building a network in Graphical User Mode

The Corda Network Builder also provides a GUI for when automated interactions are not required. To launch it, run `java -jar <path/to/network-builder-jar> -g`.

Starting the nodes

1. Click **Open nodes ...** and select the folder where you built your nodes in **Creating the base nodes** and click **Open**
2. Select **Local Docker** or **Azure**
3. Click **Build**

Note

The Azure configuration is handled by the az-cli utility. See the **Prerequisites**.

All the nodes should eventually move to a **Status** of **INSTANTIATED**. If you run `docker ps` from the terminal to see the running containers, the following output should be displayed:

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS		
406868b4ba69	node-partyc:corda-network	<code>"/run-corda.sh"</code>	17 seconds ago
Up 16 seconds	0.0.0.0:32902->10003/tcp, 0.0.0.0:32895->10005/tcp, 0.0.0.0:32898->10020/tcp, 0.0.0.0:32900->12222/tcp	partyc0	
4546a2fa8de7	node-partyb:corda-network	<code>"/run-corda.sh"</code>	17 seconds ago
Up 17 seconds	0.0.0.0:32896->10003/tcp, 0.0.0.0:32899->10005/tcp, 0.0.0.0:32901->10020/tcp, 0.0.0.0:32903->12222/tcp	partyb0	
c8c44c515bdb	node-partya:corda-network	<code>"/run-corda.sh"</code>	17 seconds ago
Up 17 seconds	0.0.0.0:32894->10003/tcp, 0.0.0.0:32897->10005/tcp, 0.0.0.0:32892->10020/tcp, 0.0.0.0:32893->12222/tcp	partya0	
cf7ab689f493	node-notary:corda-network	<code>"/run-corda.sh"</code>	30 seconds ago
Up 31 seconds	0.0.0.0:32888->10003/tcp, 0.0.0.0:32889->10005/tcp, 0.0.0.0:32890->10020/tcp, 0.0.0.0:32891->12222/tcp	notary0	

Interacting with the nodes

See [Interacting with the nodes](#).

Adding additional nodes

It is possible to add additional nodes to the network by reusing the nodes you built earlier. For example, to add a node by reusing the existing **PartyA** node, you would:

1. Select **partya** in the dropdown
2. Click **Add Instance**
3. Specify the new node's X500 name and click **OK**

If you click on **partya** in the pane, you should see an additional instance listed in the sidebar. To confirm the node has been started correctly, run the following in the previously connected SSH session:

```
Tue Jul 17 15:47:14 GMT 2018>>> run networkMapSnapshot
[
  { "addresses" : [ "partya0:10020" ], "legalIdentitiesAndCerts" : [ "O=PartyA,
L=London, C=GB" ], "platformVersion" : 3, "serial" : 1532701330613 },
  { "addresses" : [ "notary0:10020" ], "legalIdentitiesAndCerts" : [ "O=Notary,
L=London, C=GB" ], "platformVersion" : 3, "serial" : 1532701305115 },
  { "addresses" : [ "partyc0:10020" ], "legalIdentitiesAndCerts" : [ "O=PartyC,
L=Paris, C=FR" ], "platformVersion" : 3, "serial" : 1532701331608 },
  { "addresses" : [ "partyb0:10020" ], "legalIdentitiesAndCerts" : [ "O=PartyB,
L>New York, C=US" ], "platformVersion" : 3, "serial" : 1532701330118 },
  { "addresses" : [ "partya1:10020" ], "legalIdentitiesAndCerts" : [ "O=PartyZ,
L=London, C=GB" ], "platformVersion" : 3, "serial" : 1532701630861 }
]
```

Shutting down the nodes

Run `docker kill $(docker ps -q)` to kill all running Docker processes.

[Next](#) [Previous](#)

- Network Bootstrapper
 - [View page source](#)
-

Network Bootstrapper

Test deployments

Nodes within a network see each other using the network map. This is a collection of statically signed node-info files, one for each node. Most production deployments will use a highly available, secure distribution of the network map via HTTP.

For test deployments where the nodes (at least initially) reside on the same filesystem, these node-info files can be placed directly in the node's `additional-node-infos` directory from where the node will pick them up and store them in its local network map cache. The node generates its own node-info file on startup.

In addition to the network map, all the nodes must also use the same set of network parameters. These are a set of constants which guarantee interoperability between the nodes. The HTTP network map distributes the network parameters which are downloaded automatically by the nodes. In the absence of this the network parameters must be generated locally.

For these reasons, test deployments can avail themselves of the network bootstrapper. This is a tool that scans all the node configurations from a common directory to generate the network parameters file, which is then copied to all the nodes' directories. It also copies each node's node-info file to every other node so that they can all be visible to each other.

You can find out more about network maps and network parameters from [Network Map](#).

Bootstrapping a test network

The Corda Network Bootstrapper can be downloaded from [here](#).

Create a directory containing a node config file, ending in “_node.conf”, for each node you want to create. Then run the following command:

```
java -jar network-bootstrapper-VERSION.jar <nodes-root-dir>
```

For example running the command on a directory containing these files:

```
.  
└── notary_node.conf      // The notary's node.conf file  
└── partya_node.conf     // Party A's node.conf file  
└── partyb_node.conf     // Party B's node.conf file
```

will generate directories containing three nodes: `notary`, `partya` and `partyb`. They will each use the `corda.jar` that comes with the bootstrapper. If a different version of Corda is required then simply place that `corda.jar` file alongside the configuration files in the directory.

You can also have the node directories containing their “node.conf” files already laid out. The previous example would be:

```
.  
└── notary  
    └── node.conf  
└── partya  
    └── node.conf  
└── partyb  
    └── node.conf
```

Similarly, each node directory may contain its own `corda.jar`, which the bootstrapper will use instead.

Providing CorDapps to the Network Bootstrapper

If you would like the Network Bootstrapper to include your CorDapps in each generated node, just place them in the directory alongside the config files. For example, if your directory has this structure:

```
.  
└── notary_node.conf      // The notary's node.conf file  
└── partya_node.conf     // Party A's node.conf file
```

```
└── partyb_node.conf          // Party B's node.conf file
    ├── cordapp-a.jar           // A cordapp to be installed on all nodes
    └── cordapp-b.jar           // Another cordapp to be installed on all nodes
```

The `cordapp-a.jar` and `cordapp-b.jar` will be installed in each node directory, and any contracts within them will be added to the Contract Whitelist (see below).

Whitelisting contracts

Any CorDapps provided when bootstrapping a network will be scanned for contracts which will be used to create the *Zone whitelist* (see [API: Contract Constraints](#)) for the network.

Note

If you only wish to whitelist the CorDapps but not copy them to each node then run with the `--no-copy` flag.

The CorDapp JARs will be hashed and scanned for `Contract` classes. These contract class implementations will become part of the whitelisted contracts in the network parameters
(see [NetworkParameters.whitelistedContractImplementations](#) Network Map).

By default the bootstrapper will whitelist all the contracts found in all the CorDapp JARs. To prevent certain contracts from being whitelisted, add their fully qualified class name in the `exclude_whitelist.txt`. These will instead use the more restrictive `HashAttachmentConstraint`.

For example:

```
net.corda.finance.contracts.asset.Cash
net.corda.finance.contracts.asset.CommercialPaper
```

Modifying a bootstrapped network

The network bootstrapper is provided as a development tool for setting up Corda networks for development and testing. There is some limited functionality which can be used to make changes to a network, but for anything more complicated consider using a [Network Mapserver](#).

When running the Network Bootstrapper, each `node-info` file needs to be gathered together in one directory. If the nodes are being run on different machines you need to do the following:

- Copy the node directories from each machine into one directory, on one machine
- Depending on the modification being made (see below for more information), add any new files required to the root directory
- Run the Network Bootstrapper from the root directory
- Copy each individual node's directory back to the original machine

The network bootstrapper cannot dynamically update the network if an existing node has changed something in their node-info, e.g. their P2P address. For this the new node-info file will need to be placed in the other nodes' `additional-node-infos` directory. If the nodes are located on different machines, then a utility such as `rsync` can be used so that the nodes can share node-infos.

Adding a new node to the network

Running the bootstrapper again on the same network will allow a new node to be added and its node-info distributed to the existing nodes.

As an example, if we have an existing bootstrapped network, with a Notary and PartyA and we want to add a PartyB, we can use the network bootstrapper on the following network structure:

```
.
├── notary          // existing node directories
│   ├── node.conf
│   ├── network-parameters
│   ├── node-info-notary
│   └── additional-node-infos
│       ├── node-info-notary
│       └── node-info-partya
└── partya
    ├── node.conf
    ├── network-parameters
    ├── node-info-partya
    └── additional-node-infos
        ├── node-info-notary
        └── node-info-partya
└── partyb_node.conf // the node.conf for the node to be added
```

Then run the network bootstrapper again from the root dir:

```
java -jar network-bootstrapper-VERSION.jar <nodes-root-dir>
```

Which will give the following:

```

    └── notary          // the contents of the existing nodes (keys, db's
etc...) are unchanged
        ├── node.conf
        ├── network-parameters
        ├── node-info-notary
        └── additional-node-infos
            ├── node-info-notary
            ├── node-info-partya
            └── node-info-partyb
    └── partya
        ├── node.conf
        ├── network-parameters
        ├── node-info-partya
        └── additional-node-infos
            ├── node-info-notary
            ├── node-info-partya
            └── node-info-partyb
    └── partyb          // a new node directory is created for PartyB
        ├── node.conf
        ├── network-parameters
        ├── node-info-partyb
        └── additional-node-infos
            ├── node-info-notary
            ├── node-info-partya
            └── node-info-partyb

```

The bootstrapper will generate a directory and the `node-info` file for PartyB, and will also make sure a copy of each nodes' `node-info` file is in the `additional-node-info` directory of every node. Any other files in the existing nodes, such as generated keys, will be unaffected.

Note

The bootstrapper is provided for test deployments and can only generate information for nodes collected on the same machine. If a network needs to be updated using the bootstrapper once deployed, the nodes will need collecting back together.

Updating the contract whitelist for bootstrapped networks

If the network already has a set of network parameters defined (i.e. the node directories all contain the same `network-parameters` file) then the bootstrapper can be used to append contracts from new CorDapps to the current whitelist. For example, with the following pre-generated network:

```

    .
    └── notary
        ├── node.conf
        ├── network-parameters
        └── cordapps
            └── cordapp-a.jar
    └── partya

```

```

    └── node.conf
    └── network-parameters
        └── cordapps
            └── cordapp-a.jar
    └── partyb
        └── node.conf
        └── network-parameters
            └── cordapps
                └── cordapp-a.jar
    └── cordapp-b.jar                                // The new cordapp to add to the existing nodes

```

Then run the network bootstrapper again from the root dir:

```
java -jar network-bootstrapper-VERSION.jar <nodes-root-dir>
```

To give the following:

```

.
└── notary
    ├── node.conf
    ├── network-parameters      // The contracts from cordapp-b are appended to the
        whitelist in network-parameters
    └── cordapps
        ├── cordapp-a.jar
        └── cordapp-b.jar          // The updated cordapp is placed in the nodes cordapp
            directory
└── partya
    ├── node.conf
    ├── network-parameters      // The contracts from cordapp-b are appended to the
        whitelist in network-parameters
    └── cordapps
        ├── cordapp-a.jar
        └── cordapp-b.jar          // The updated cordapp is placed in the nodes cordapp
            directory
└── partyb
    ├── node.conf
    ├── network-parameters      // The contracts from cordapp-b are appended to the
        whitelist in network-parameters
    └── cordapps
        ├── cordapp-a.jar
        └── cordapp-b.jar          // The updated cordapp is placed in the nodes cordapp
            directory

```

Note

The whitelist can only ever be appended to. Once added a contract implementation can never be removed.

[Next](#) [Previous](#)

- [Blob Inspector](#)
- [View page source](#)

Blob Inspector

There are many benefits to having a custom binary serialisation format (see [Object serialization](#) for details) but one disadvantage is the inability to view the contents in a human-friendly manner. The blob inspector tool alleviates this issue by allowing the contents of a binary blob file (or URL end-point) to be output in either YAML or JSON. It uses [JacksonSupport](#) to do this (see [JSON](#)).

The latest version of the tool can be downloaded from [here](#).

To run simply pass in the file or URL as the first parameter:

```
java -jar blob-inspector.jar <file or URL>
```

Use the [--help](#) flag for a full list of command line options.

When inspecting your custom data structures, there's no need to include the jars containing the class definitions for them in the classpath. The blob inspector (or rather the serialization framework) is able to synthesise any classes found in the blob that aren't on the classpath.

SerializedBytes

One thing to note is that the binary blob may contain embedded [SerializedBytes](#) objects. Rather than printing these out as a Base64 string, the blob inspector will first materialise them into Java objects and then output those. You will see this when dealing with classes such as [SignedData](#) or other structures that attach a signature, such as the [nodeInfo-*](#) files or the [network-parameters](#) file in the node's directory. For example, the output of a node-info file may look like:

-l-format=YAML

```
net.corda.nodeapi.internal.SignedNodeInfo
---
raw:
  class: "net.corda.core.node.NodeInfo"
  deserialized:
    addresses:
      - "localhost:10005"
    legalIdentitiesAndCerts:
      - "O=BankOfCorda, L=London, C=GB"
    platformVersion: 4
    serial: 1527851068715
  signatures:
    - !!binary |-
```

```
VFRY4frbgRDbCpK1Vo88PyUoj01vbRnMR3R0R2abTFk7yJ14901aeScX/CiEP+CDGiMRsdw01cXt\nhKSobAY7Dw==
```

-format=JSON

```
net.corda.nodeapi.internal.SignedNodeInfo
{
    "raw" : {
        "class" : "net.corda.core.node.NodeInfo",
        "deserialized" : {
            "addresses" : [ "localhost:10005" ],
            "legalIdentitiesAndCerts" : [ "O=BankOfCorda, L=London, C=GB" ],
            "platformVersion" : 4,
            "serial" : 1527851068715
        }
    },
    "signatures" : [
        "VFRY4frbgRDbCpK1Vo88PyUoj01vbRnMR3R0R2abTFk7yJ14901aeScX/CiEP+CDGiMRsdw01cXthKSobAY7Dw=="
    ]
}
```

Notice the file is actually a serialised `SignedNodeInfo` object, which has a `raw` property of type `SerializedBytes<NodeInfo>`. This property is materialised into a `NodeInfo` and is output under the `deserialized` field.

[Next](#) [Previous](#)

- Network Simulator
 - [View page source](#)
-

Network Simulator

A network simulator is provided which shows traffic between nodes through the lifecycle of an interest rate swap contract. It can optionally also show network setup, during which nodes register themselves with the network map service and are notified of the changes to the map. The network simulator is run from the command line via Gradle:

Windows:

```
gradlew.bat :samples:network-visualiser:run
```

Other:

```
./gradlew :samples:network-visualiser:run
```

You can produce a standalone JAR of the tool by using the `:samples:network-visualiser:deployVisualiser` target and then using the `samples/network-visualiser/build/libs/network-visualiser-* capsule.jar` file, where * is whatever the current Corda version is.

What it is and is not

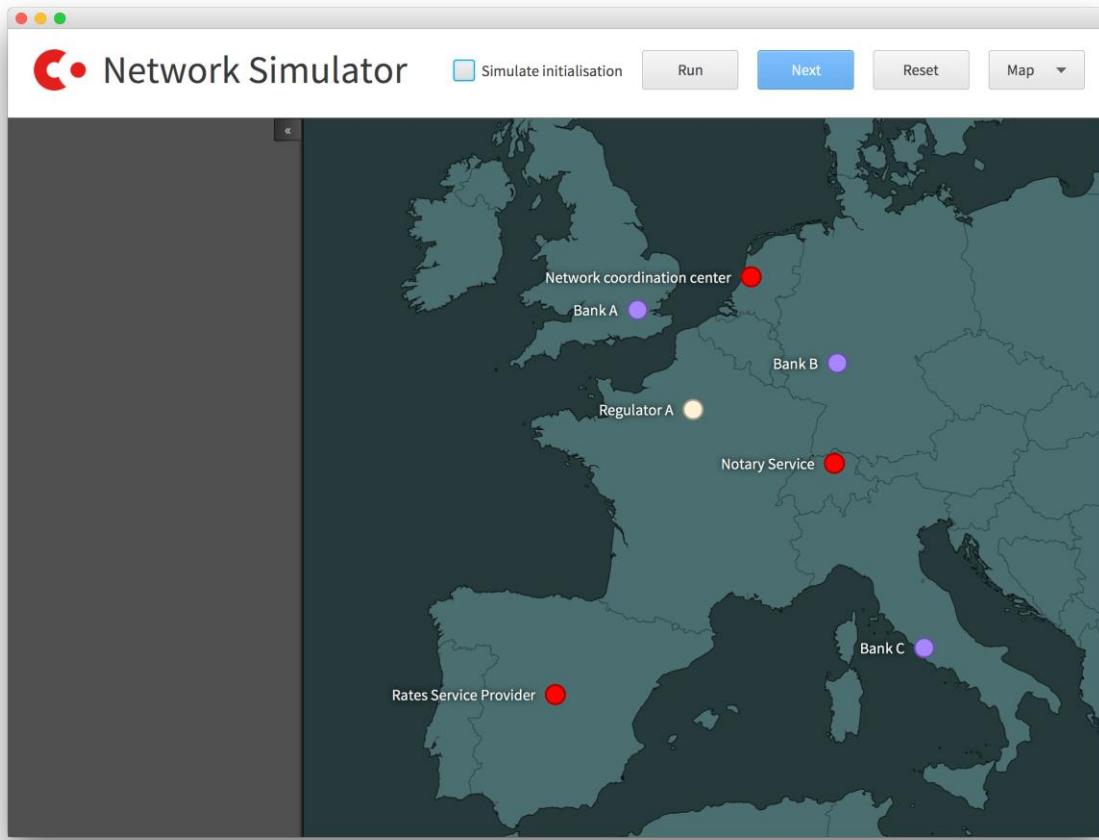
The simulator currently exists as an illustrative tool to help with explaining how Corda works in an example scenario. It utilises the `Simulator` tools that support creating a simulated Corda network and the nodes running in it within a single JVM, as an extension of the `MockNetwork` testing framework. See more about the `MockNetwork` and testing flows here: [Writing flow tests](#).

Whilst it is not yet fully generic or full featured, the intention is for the simulator to mature into the following, which it presently cannot do without writing more code:

1. A tool for visualising new CorDapps and their flows to help with debugging, presentations, explanations and tutorials, but still running as a simulation in a single JVM.
2. A tool to visualise the activity on a real Corda network deployment, with activity streams being fed from each node running in its own JVM, most likely on remote hosts.

Both of these scenarios would be fed by the standard observables in the RPC framework, rather than the local binding that the simulator uses currently. The ability to step through a flow one step at a time would obviously be restricted to single JVM simulations.

Interface



The network simulator can be run automatically, or stepped manually through each step of the interest rate swap. The options on the simulator window are:

Simulate initialisation

If checked, the nodes registering with the network map is shown. Normally this setup step is not shown, but may be of interest to understand the details of node discovery.

Run

Runs the network simulation in automatic mode, in which it progresses each step on a timed basis. Once running, the simulation can be paused in order to manually progress it, or reset.

Next

Manually progress the simulation to the next step.

Reset

Reset the simulation (only available when paused).

Map/Circle

How the nodes are shown, by default nodes are rendered on a world map, but alternatively they can be rendered in a circle layout.

While the simulation runs, details of the steps currently being executed are shown in a sidebar on the left hand side of the window.

- DemoBench
 - [View page source](#)
-

DemoBench

DemoBench is a standalone desktop application that makes it easy to configure and launch local Corda nodes. It is useful for training sessions, demos or just experimentation.

Downloading

Installers compatible with the latest Corda release can be downloaded from the [Corda website](#).

Running DemoBench

Configuring a Node

Each node must have a unique name to identify it to the network map service. DemoBench will suggest node names, nearest cities and local port numbers to use.

The first node will be a notary. Hence only notary services will be available to be selected in the **Services** list. For subsequent nodes you may also select any of Corda's other built-in services.

Press the **Start node** button to launch the Corda node with your configuration.

Running Nodes

DemoBench launches each new node in a terminal emulator.

The **View Database**, **Launch Explorer** and **Launch Web Server** buttons will all be disabled until the node has finished booting. DemoBench will then display simple statistics about the node such as its cash balance.

It is currently impossible from DemoBench to restart a node that has terminated, e.g. because the user typed “bye” at the node’s shell prompt. However, that node’s data and logs still remain in its directory.

Exiting DemoBench

When you terminate DemoBench, it will automatically shut down any nodes and explorers that it has launched and then exit.

Profiles

You can save the configurations and CorDapps for all of DemoBench's currently running nodes into a profile, which is a **ZIP** file with the following layout, e.g.:

```
notary/  
  node.conf  
  cordapps/  
banka/  
  node.conf  
  cordapps/  
bankb/  
  node.conf  
  cordapps/  
    example-cordapp.jar  
...
```

When DemoBench reloads this profile it will close any nodes that it is currently running and then launch these new nodes instead. All nodes will be created with a brand new database. Note that the **node.conf** files within each profile are JSON/HOCON format, and so can be extracted and edited as required.

DemoBench writes a log file to the following location:

MacOSX/Linux	<code>\$HOME/demobench/demobench.log</code>
Windows	<code>%USERPROFILE%\demobench\demobench.log</code>

Building the Installers

Gradle defines tasks that build DemoBench installers using JavaPackager. There are three scripts in the **tools/demobench** directory to execute these tasks:

1. `package-demobench-exe.bat` (Windows)
2. `package-demobench-dmg.sh` (MacOS)
3. `package-demobench-rpm.sh` (Fedora/Linux)

Each script can only be run on its target platform, and each expects the platform's installation tools already to be available.

1. Windows: [Inno Setup 5+](#)
2. MacOS: The packaging tools should be available automatically. The DMG contents will also be signed if the packager finds a valid [Developer ID Application](#) certificate with a private key on the keyring.

(By default, DemoBench's `build.gradle` expects the signing key's user name to be "R3CEV".) You can create such a certificate by generating a Certificate Signing Request and then asking your local "Apple team agent" to upload it to the Apple Developer portal. (See [here](#).)

Note

- Please ensure that the `/usr/bin/codesign` application always has access to your certificate's signing key. You may need to reboot your Mac after making any changes via the MacOS Keychain Access application.
- You should use JDK >= 8u152 to build DemoBench on MacOS because this version resolves a warning message that is printed to the terminal when starting each Corda node.
- Ideally, use the [JetBrains JDK](#) to build the DMG.

3. Fedora/Linux: `rpm-build` packages.

You will also need to define the environment variable `JAVA_HOME` to point to the same JDK that you use to run Gradle. The installer will be written to the `tools/demobench/build/javapackage/bundles` directory, and can be installed like any other application for your platform.

JetBrains JDK

Mac users should note that the best way to build a DemoBench DMG is with the [JetBrains JDK](#) which has [binary downloads available from BinTray](#). This JDK has some useful GUI fixes, most notably, when built with this JDK the DemoBench terminal will support emoji and as such, the nicer coloured ANSI progress renderer. It also resolves some issues with HiDPI rendering on Windows.

This JDK does not include JavaPackager, which means that you will still need to copy `$JAVA_HOME/lib/ant-javafx.jar` from an Oracle JDK into the corresponding directory within your JetBrains JDK.

Developer Notes

Developers wishing to run DemoBench *without* building a new installer each time can install it locally using Gradle:

```
$ gradlew tools:demobench:installDist  
$ cd tools/demobench/build/install/demobench  
$ bin/demobench
```

Unfortunately, DemoBench's `$CLASSPATH` may be too long for the Windows shell .

In which case you can still run DemoBench as follows:

```
> java -Djava.util.logging.config.class=net.corda.demobench.config.LoggingConfig -jar  
lib/demobench-$version.jar
```

While DemoBench *can* be executed within an IDE, it would be up to the Developer to install all of its runtime dependencies beforehand into their correct locations relative to the value of the `user.dir` system property (i.e. the current working directory of the JVM):

```
corda/  
    corda.jar  
    corda-webserver.jar  
explorer/  
    node-explorer.jar  
cordapps/  
    bank-of-corda.jar
```

[Next](#) [Previous](#)

- Node Explorer
 - [View page source](#)
-

Node Explorer

The node explorer provides views into a node's vault and transaction data using Corda's RPC framework. The user can execute cash transaction commands to issue and move cash to other parties on the network or exit cash (eg. remove from the ledger)

Running the UI

Windows:

```
gradlew.bat tools:explorer:run
```

Other:

```
./gradlew tools:explorer:run
```

Running demo nodes

A demonstration Corda network topology is configured with 5 nodes playing the following roles:

1. Notary
2. Issuer nodes, representing two fictional central banks (UK Bank Plc issuer of GBP and USA Bank Corp issuer of USD)
3. Participant nodes, representing two users (Alice and Bob)

When connected to an *Issuer* node, a user can execute cash transaction commands to issue and move cash to itself or other parties on the network or to exit cash (for itself only).

When connected to a *Participant* node a user can only execute cash transaction commands to move cash to other parties on the network.

The Demo Nodes can be started in one of two modes:

1. Normal

Fresh clean environment empty of transactions. Firstly, launch an Explorer instance to login to one of the Issuer nodes and issue some cash to the other participants (Bob and Alice). Then launch another Explorer instance to login to a participant node and start making payments (eg. move cash). You will only be able to exit (eg. redeem from the ledger) cash as an issuer node.

Windows:

```
gradlew.bat tools:explorer:runDemoNodes
```

Other:

```
./gradlew tools:explorer:runDemoNodes
```

2. Simulation

In this mode Nodes will automatically commence executing commands as part of a random generation process. The simulation start with pre-allocating chunks of cash to each of the party in 2 currencies (USD, GBP), then it enter a loop to generate random events. In each iteration, the issuers will execute a

Cash Issue or Cash Exit command (at a 9:1 ratio) and a random party will execute a move of cash to another random party.

Windows:

```
gradlew.bat tools:explorer:runSimulationNodes
```

Other:

```
./gradlew tools:explorer:runSimulationNodes
```

Note

5 Corda nodes will be created on the following port on localhost by default.

- Notary -> 20005 (Does not accept logins)
- UK Bank Plc -> 20011 (*/Issuer node*)
- USA Bank Corp -> 20008 (*/Issuer node*)
- Alice -> 20017
- Bob -> 20014

Explorer login credentials to the Issuer nodes are defaulted to `manager` and `test`.

Explorer login credentials to the Participants nodes are defaulted to `user1` and `test`. Please note you are not allowed to login to the notary.

Note

When you start the nodes in Windows using the command prompt, they might not be killed when you close the window or terminate the task. If that happens you need to manually terminate the Java processes running the nodes.

Note

Alternatively, you may start the demo nodes from within IntelliJ using either of the run configurations `Explorer - demo nodes` or `Explorer - demo nodes (simulation)`

Note

It is also possible to start the Explorer GUI from IntelliJ via `Explorer - GUI` run configuration, provided that the optional TornadoFX plugin has been installed first.

Note

Use the Explorer in conjunction with the Trader Demo and Bank of Corda samples to use other */Issuer* nodes.

Interface

Login

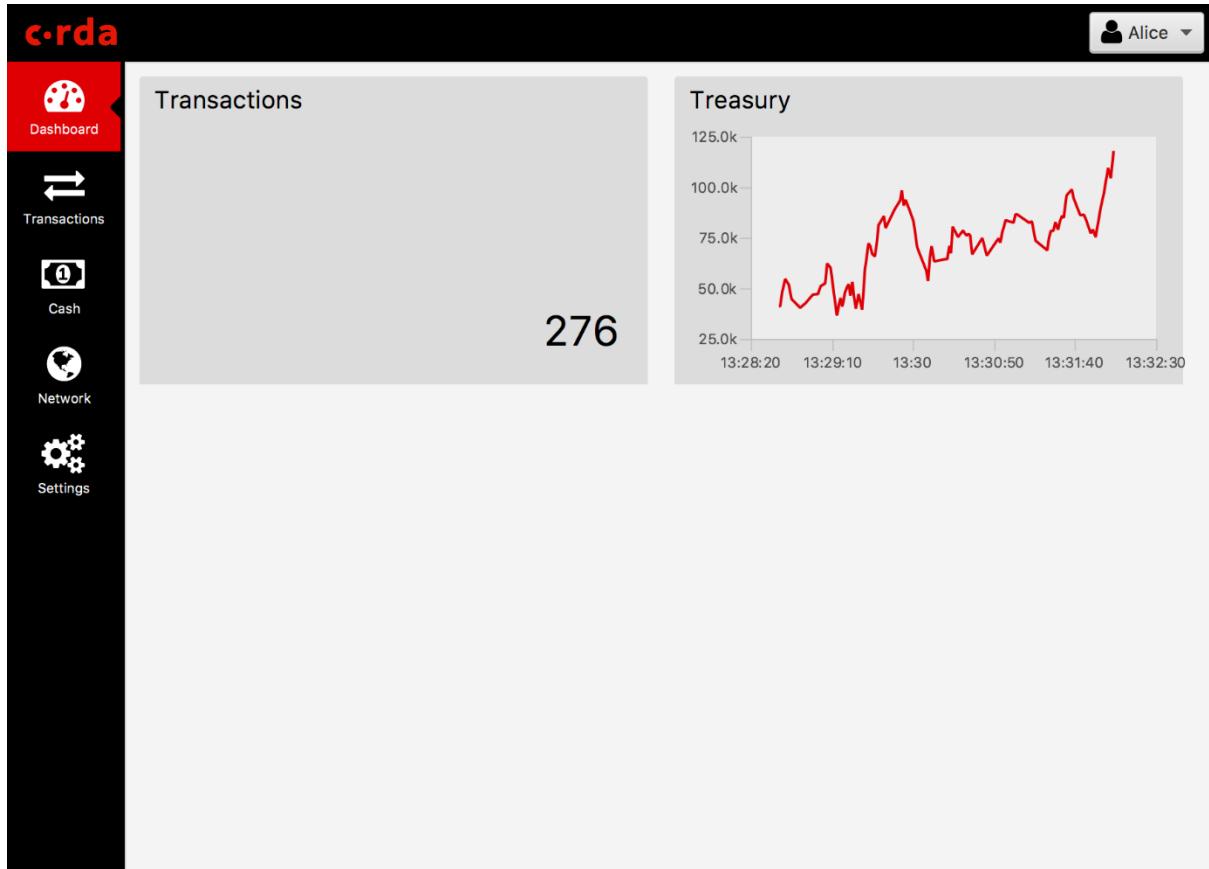
User can login to any Corda node using the explorer.

Alternatively, `gradlew explorer:runDemoNodes` can be used to start up demo nodes for testing. Corda node address, username and password are required for login, the address is defaulted to localhost:0 if leave blank. Username and password can be configured via the `rpcUsers` field in node's configuration file.



Dashboard

The dashboard shows the top level state of node and vault. Currently, it shows your cash balance and the numbers of transaction executed. The dashboard is intended to house widgets from different CordApps and provide useful information to system admin at a glance.



Cash

The cash view shows all currencies you currently own in a tree table format, it is grouped by issuer -> currency. Individual cash transactions can be viewed by clicking on the table row. The user can also use the search field to narrow down the scope.

The screenshot shows the crda application interface. On the left is a sidebar with icons for Dashboard, Transactions, Cash, Network, and Settings. The main area has a header with '+ New Transaction', a search bar ('All'), and a filter ('Filter by currency, issuer'). A dropdown menu shows 'Royal Mint'. Below this is a table with columns 'Issuer/Currency', 'Local currency', and 'USD Equiv'. Two rows are visible: one for 'Federal Reserve' (909287 USD) and one for 'Royal Mint' (291859 USD). The 'Royal Mint' row is highlighted in pink. To the right of the table is a list of 94 transactions, each with a state ID, issuer, origin date, amount, and currency. The first transaction is for the Federal Reserve.

Issuer/Currency	Local currency	USD Equiv
Federal Reserve		909287 USD
Royal Mint	291859 GBP	291859 USD
	CHF	303551 CHF
	USD	313877 USD
Royal Mint		18800 USD
	GBP	861 GBP
	CHF	7919 CHF
	USD	10020 USD

Total 2 matching issuers

>>

- State ID 4A05ECB1DEF7A1F9...[0]
Issuer Federal Reserve[00]
Originated 2016-12-09T18:30:58.322
Amount 2405 GBP
USD 2405 USD
- State ID 57865805B57888D0...[0]
Issuer Federal Reserve[01]
Originated 2016-12-09T18:30:58.334
Amount 3264 GBP
USD 3264 USD
- State ID E36643FEA7869841...[0]
Issuer Federal Reserve[01]
Originated 2016-12-09T18:30:58.334
Amount 624 GBP
USD 624 USD
- State ID 7634B84323CB65CF...[1]
Issuer Federal Reserve[00]
Originated 2016-12-09T18:30:58.335
Amount 3001 GBP
USD 3001 USD
- State ID B5D09A95570961A1...[0]

Total 94 positions

New Transactions

This is where you can create new cash transactions. The user can choose from three transaction types (issue, pay and exit) and any party visible on the network.

General nodes can only execute pay commands to any other party on the network.

The screenshot shows the corda application interface. On the left is a sidebar with icons for Dashboard, Transactions, Cash (selected), Network, and Settings. The main area has a title bar with '+ New Transaction' and search/filter options. A table lists issuers and their currency holdings in Local currency and USD Equiv. A modal dialog is open, showing a dropdown for 'Transaction Type' with 'Pay' selected, and a large empty area below it with an 'Execute' button.

Issuer/Currency	Local currency	USD Equiv
▼ Federal Reserve		5250 USD
GBP	980 GBP	980 USD
CHF	4058 CHF	4058 USD
USD	212 USD	212 USD
▼ Royal Mint		411730 USD
GBP	140379 GBP	140379 USD
CHF	183694 CHF	183694 USD
USD		

Total 2 matching issuers

Issuer Nodes

Issuer nodes can execute issue (to itself or to any other party), pay and exit transactions. The result of the transaction will be visible in the transaction screen when executed.

The screenshot shows the corda application interface. On the left, there is a sidebar with icons for Dashboard, Transactions, Cash, Network, and Settings. The main area displays a table of issuers and their currency holdings. A modal dialog is open over the table, prompting for a transaction type. The table data is as follows:

Issuer/Currency	Local currency	USD Equiv
Federal Reserve	291859 GBP 303551 CHF	909287 USD 303551 USD
Royal Mint	GBP CHF USD	

The modal dialog has a title "Transaction Type :" and a dropdown menu with three options: "Issue" (highlighted in pink), "Pay", and "Exit". A button labeled "Execute" is at the bottom right of the modal.

Transactions

The transaction view contains all transactions handled by the node in a table view. It shows basic information on the table e.g. Transaction ID, command type, USD equivalence value etc. User can expand the row by double clicking to view the inputs, outputs and the signatures details for that transaction.

The screenshot shows the corda application interface. The left sidebar has icons for Dashboard, Transactions (highlighted in red), Cash, Network, and Settings. The main area displays a table of transactions for user Alice. The table has columns: Transaction ID, Input, Output, Input Party, Output Party, Command type, and Total value.

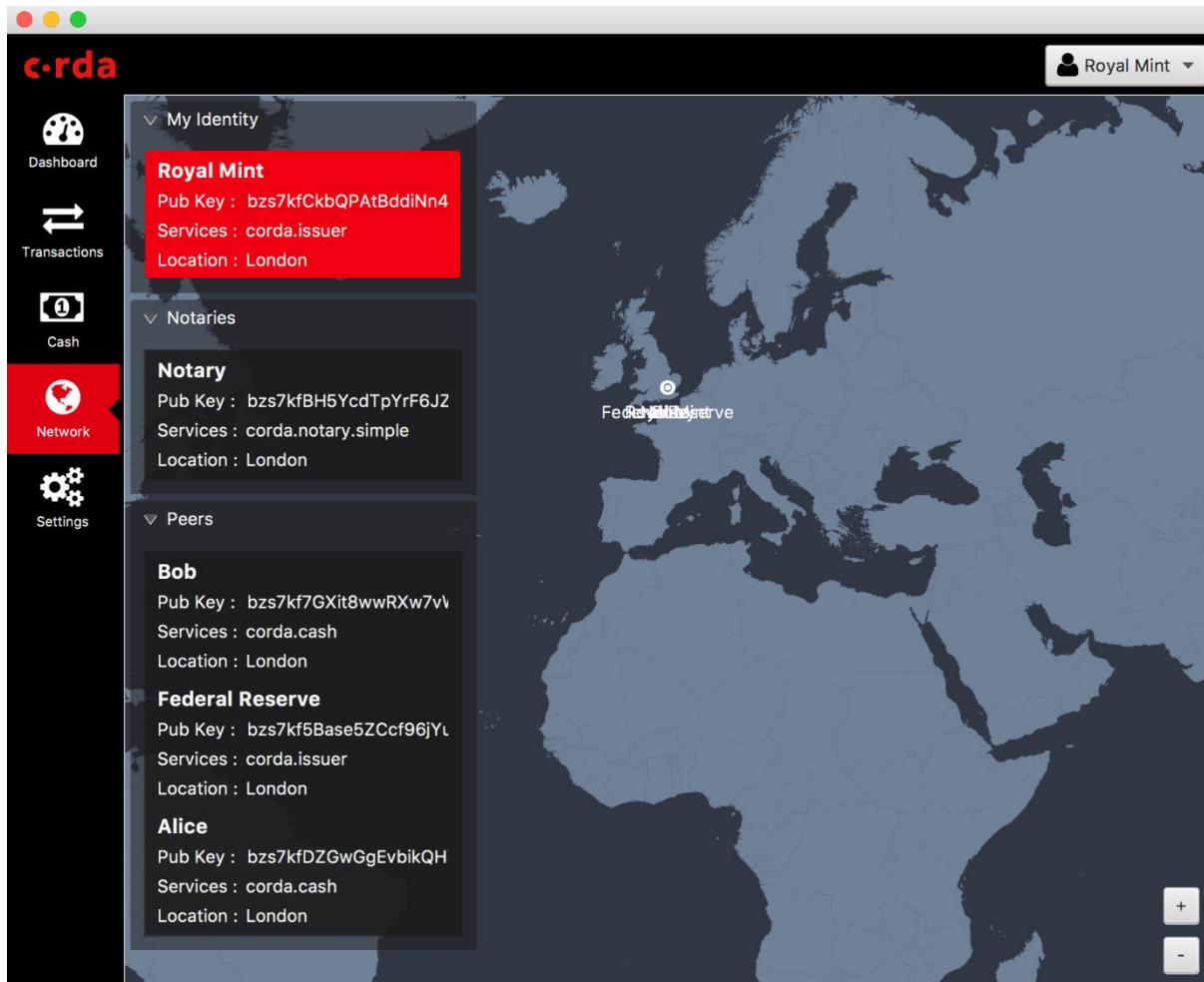
	Transaction ID	Input	Output	Input Party	Output Party	Command type	Total value
[+]	23749820B09B50AA48D...	Cash (1)		Alice	Issue	91 GBP	
[+]	BFA38D6092278AED63E...	Cash (1)		Alice	Issue	2587 GBP	
[+]	859CF7DB36151314338A...	Cash (2)	Cash (3)	Alice	Bank Of Corda, Alice	Move	-8922 GBP

Below the table, there are three sections: Input (2), Output (3), and Signatures (2). The Input section shows two cash inputs: one from Bank Of Corda[01] (amount 8154 USD) and another from Bank Of Corda[00] (amount 804 USD). The Output section shows three cash outputs: one to Alice (amount 804 USD) and two to Bank Of Corda (amounts 8118 USD and 8946 GBP). The Signatures section lists two signatures: Alice and Notary.

At the bottom, a message says "573 matching transactions".

Network

The network view shows the network information on the world map. Currently only the user's node is rendered on the map. This will be extended to other peers in a future release. The map provides an intuitive way of visualizing the Corda network and the participants.

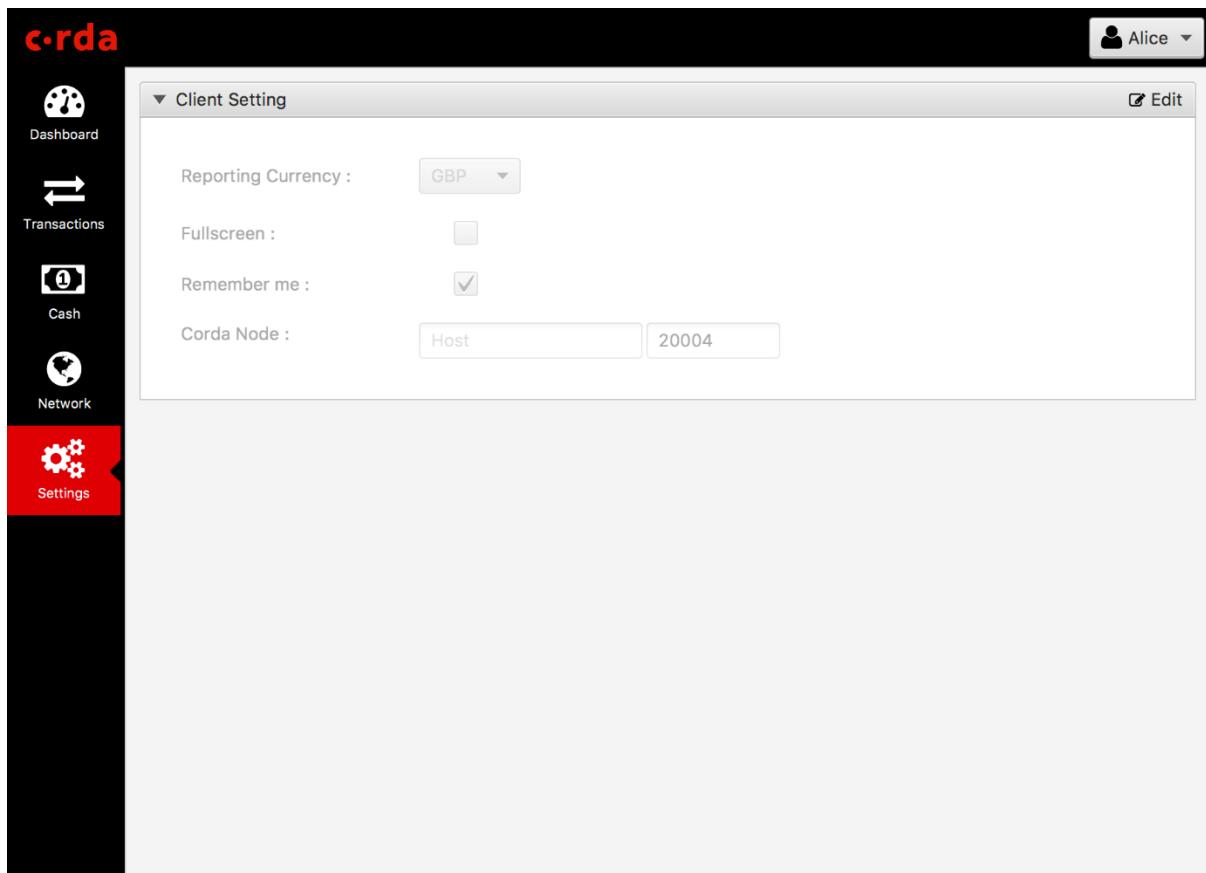


Settings

User can configure the client preference in this view.

Note

Although the reporting currency is configurable, FX conversion won't be applied to the values as we don't have an FX service yet.



[Next](#) [Previous](#)

Building a Corda Network on Azure Marketplace

To help you design, build and test applications on Corda, called CorDapps, a Corda network can be deployed on the [Microsoft Azure Marketplace](#)

This Corda network offering builds a pre-configured network of Corda nodes as Ubuntu virtual machines (VM). The network comprises of a Notary node and up

to nine Corda nodes using a version of Corda of your choosing. The following guide will also show you how to load a simple Yo! CorDapp which demonstrates the basic principles of Corda. When you are ready to go further with developing on Corda and start making contributions to the project head over to the [Corda.net](#).

Pre-requisites

- Ensure you have a registered Microsoft Azure account which can create virtual machines under your subscription(s) and you are logged on to the Azure portal ([portal.azure.com](#))
- It is recommended you generate a private-public SSH key pair (see [here](#))

Deploying the Corda Network

Browse to [portal.azure.com](#), login and search the Azure Marketplace for Corda and select ‘Corda Single Ledger Network’.

Click the ‘Create’ button.

STEP 1: Basics

Define the basic parameters which will be used to pre-configure your Corda nodes.

- **Resource prefix:** Choose an appropriate descriptive name for your Corda nodes. This name will prefix the node hostnames
- **VM user name:** This is the user login name on the Ubuntu VMs. Leave it as `azureuser` or define your own
- **Authentication type:** Select ‘SSH public key’, then paste the contents of your SSH public key file (see pre-requisites, above) into the box. Alternatively select ‘Password’ to use a password of your choice to administer the VM
- **Restrict access by IP address:** Leave this as ‘No’ to allow access from any internet host, or provide an IP address or a range of IP addresses to limit access
- **Subscription:** Select which of your Azure subscriptions you want to use
- **Resource group:** Choose to ‘Create new’ and provide a useful name of your choice
- **Location:** Select the geographical location physically closest to you

Microsoft Azure Create VM development mode > Basics

Create VM development m... Basics

1 Basics Configure basic settings

2 Network size and performance Define the number and size an...

3 Corda specific options Define Corda version and notar...

4 Summary

Basics

* Resource prefix ✓

VM user name

* Authentication type Password SSH public key

* Password ⏺

* Confirm password ⏺

Restrict access by IP address No

Subscription ✓

* Resource group Create new Use existing

Location ✓

OK

Click 'OK'

STEP 2: Network Size and Performance

Define the number of Corda nodes in your network and the size of VM.

- **Number of Network Map nodes:** There can only be one Network Map node in this network. Leave as '1'
- **Number of Notary nodes:** There can only be one Notary node in this network. Leave as '1'
- **Number of participant nodes:** This is the number of Corda nodes in your network. At least 2 nodes in your network is recommended (so you can send transactions between them). You can specific 1 participant node and use the Notary node as a second node. There is an upper limit of 9
- **Storage performance:** Leave as 'Standard'
- **Virtual machine size:** The size of the VM is automatically adjusted to suit the number of participant nodes selected. It is recommended to use the suggested values

1 Basics Done ✓

2 Network size and performance >
Define the number and size an...

3 Corda specific options >
Define Corda version and notar...

4 Summary >
Corda Single Ledger Network (...)

5 Buy >

Number of VMs
Number of Network Map nodes ⓘ
1

Number of Notary nodes ⓘ
1

Number of participant nodes. ⓘ
3

Infrastructure options
Storage performance ⓘ
Standard Premium

* Virtual machine size
5x Standard D1 v2 >

OK

Click 'OK'

STEP 3: Corda Specific Options

Define the version of Corda you want on your nodes and the type of notary.

- **Corda version (as seen in Maven Central):** Select the version of Corda you want your nodes to use from the drop down list. The version numbers can be seen in [Maven Central](#), for example 0.11.0
- **Notary type:** Select either ‘Non Validating’ (notary only checks whether a state has been previously used and marked as historic) or ‘Validating’ (notary performs transaction verification by seeing input and output states, attachments and other transaction information). More information on notaries can be found [here](#)

☰ Create Corda Single Ledger... ☐ ✎

1 Basics Done ✓

2 Network size and performance Done ✓

3 Corda specific options >
Define Corda version and notar...

4 Summary >
Corda Single Ledger Network (...)

5 Buy >

Corda Specific Options ☐ ✎

Corda Version and Notary Type

Corda Version (as seen in Maven Central) ⓘ

M11

Notary type ⓘ

Non Validating Validating

OK

Click 'OK'

STEP 4: Summary

A summary of your selections is shown.

Microsoft Azure Create VM development mode > Summary

Search ...

Create VM development m... □ X Summary

Basics

Subscription	GCL
Resource group	homer
Location	North Europe

Network size and performance

Resource prefix	doctst
VM user name	azureuser
Password	*****
Restrict access by IP address	No

Corda specific options

Number of Network Map nodes	1
Number of Notary nodes	1
Number of member nodes	2
Storage performance	Standard
Virtual machine size	Standard D1 v2

Summary

OK

Click 'OK' for your selection to be validated. If everything is ok you will see the message 'Validation passed'

Click 'OK'

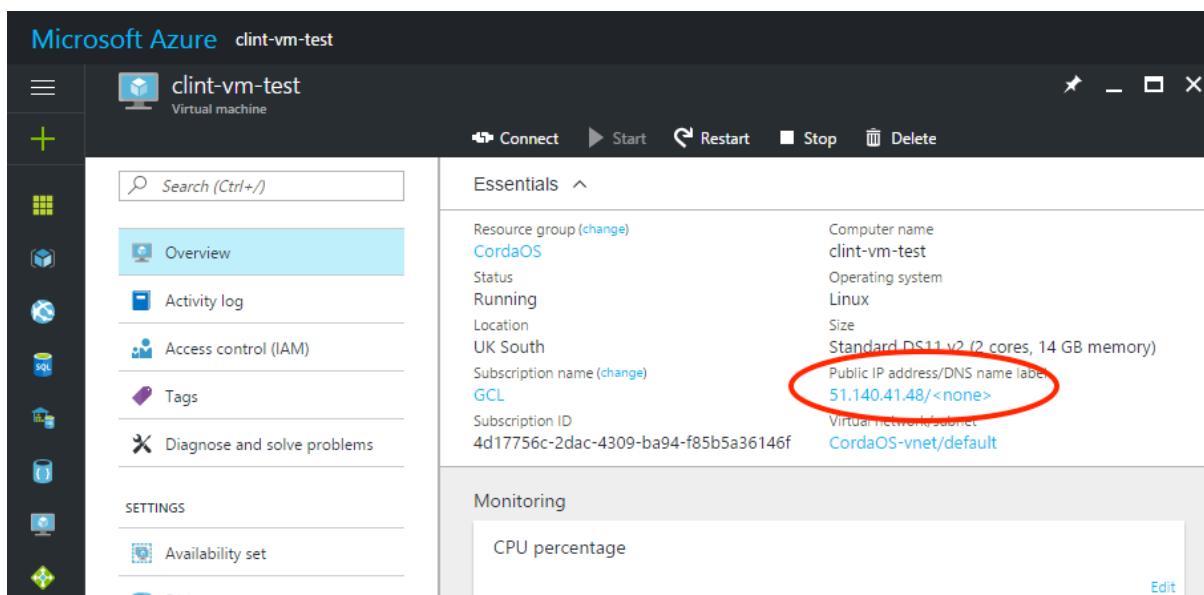
STEP 5: Buy

Review the Azure Terms of Use and Privacy Policy and click ‘Purchase’ to buy the Azure VMs which will host your Corda nodes.

The deployment process will start and typically takes 8-10 minutes to complete.

Once deployed click ‘Resources Groups’, select the resource group you defined in Step 1 above and click ‘Overview’ to see the virtual machine details. The names of your VMs will be pre-fixed with the resource prefix value you defined in Step 1 above.

The Network Map Service node is suffixed nm0. The Notary node is suffixed not0. Your Corda participant nodes are suffixed node0, node1, node2 etc. Note down the **Public IP address** for your Corda nodes. You will need these to connect to UI screens via your web browser:



The screenshot shows the Azure portal interface for a virtual machine named 'clint-vm-test'. The left sidebar has icons for various services like Storage, Network, and Compute. The main area shows the 'Overview' tab selected. On the right, under the 'Essentials' section, there is a table of details. The 'Public IP address/DNS name label' row is circled in red, showing the value '51.140.41.48/<none>'. Other details listed include Resource group (CordaOS), Status (Running), Location (UK South), Subscription name (GCL), Subscription ID (4d17756c-2dac-4309-ba94-f85b5a36146f), Computer name (clint-vm-test), Operating system (Linux), Size (Standard DS1 v2 (2 cores, 14 GB memory)), and Virtual network/Subnet (CordaOS-vnet/default).

Using the Yo! CorDapp

Loading the Yo! CordDapp on your Corda nodes lets you send simple Yo! messages to other Corda nodes on the network. A Yo! message is a very simple transaction. The Yo! CorDapp demonstrates:

- how transactions are only sent between Corda nodes which they are intended for and are not shared across the entire network by using the network map
- uses a pre-defined flow to orchestrate the ledger update automatically

- the contract imposes rules on the ledger updates
- **Loading the Yo! CorDapp onto your nodes**

The nodes you will use to send and receive Yo messages require the Yo! CorDapp jar file to be saved to their cordapps directory.

Connect to one of your Corda nodes (make sure this is not the Notary node) using an SSH client of your choice (e.g. Putty) and log into the virtual machine using the public IP address and your SSH key or username / password combination you defined in Step 1 of the Azure build process. Type the following command:

For Corda nodes running release M10

```
cd /opt/corda/cordapps
wget http://downloads.corda.net/cordapps/net/corda/yo/0.10.1/yo-0.10.1.jar
```

For Corda nodes running release M11

```
cd /opt/corda/cordapps
wget http://downloads.corda.net/cordapps/net/corda/yo/0.11.0/yo-0.11.0.jar
```

For Corda nodes running version 2

```
cd /opt/corda/plugins
wget http://ci-artifactory.corda.r3cev.com/artifactory/cordapp-showcase/yo-4.jar
```

Now restart Corda and the Corda webserver using the following commands or restart your Corda VM from the Azure portal:

```
sudo systemctl restart corda
sudo systemctl restart corda-webserver
```

Repeat these steps on other Corda nodes on your network which you want to send or receive Yo messages.

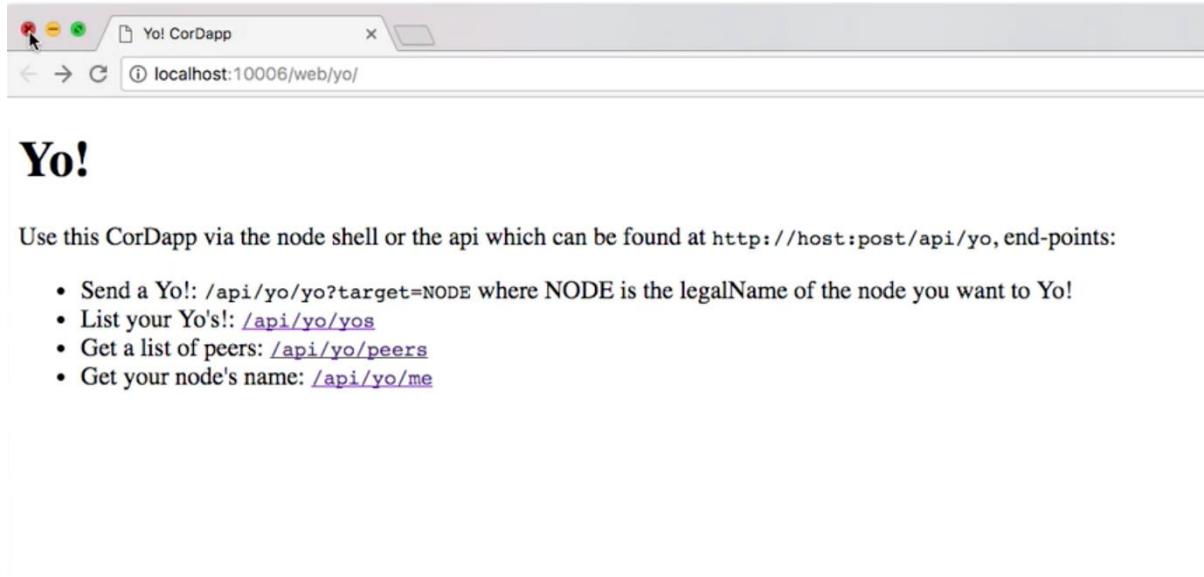
- **Verify the Yo! CorDapp is running**

Open a browser tab and browse to the following URL:

```
http://(public IP address):(port)/web/yo
```

where (public IP address) is the public IP address of one of your Corda nodes on the Azure Corda network and (port) is the web server port number for your Corda node, 10004 by default

You will now see the Yo! CordDapp web interface:



Yo!

Use this CorDapp via the node shell or the api which can be found at <http://host:post/api/yo>, end-points:

- Send a Yo!: `/api/yo/yo?target=NODE` where NODE is the legalName of the node you want to Yo!
- List your Yo's!: [/api/yo/yo](#)
- Get a list of peers: [/api/yo/peers](#)
- Get your node's name: [/api/yo/me](#)

• Sending a Yo message via the web interface

In the browser window type the following URL to send a Yo message to a target node on your Corda network:

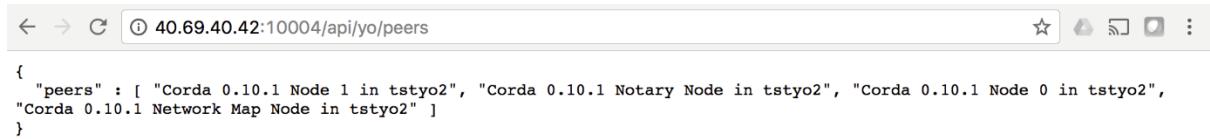
`http://(public IP address):(port)/api/yo/yo?target=(legalname of target node)`

where (public IP address) is the public IP address of one of your Corda nodes on the Azure Corda network and (port) is the web server port number for your Corda node, 10004 by default and (legalname of target node) is the Legal Name for the target node as defined in the node.conf file, for example:

`http://40.69.40.42:10004/api/yo/yo?target=Corda 0.10.1 Node 1 in tstyo2`

An easy way to see the Legal Names of Corda nodes on the network is to use the peers screen:

`http://(public IP address):(port)/api/yo/peers`



```
{ "peers" : [ "Corda 0.10.1 Node 1 in tstyo2", "Corda 0.10.1 Notary Node in tstyo2", "Corda 0.10.1 Node 0 in tstyo2", "Corda 0.10.1 Network Map Node in tstyo2" ] }
```

- **Viewing Yo messages**

To see Yo! messages sent to a particular node open a browser window and browse to the following URL:

`http://(public IP address):(port)/api/yo/yo`



```
[ { "state" : { "data" : { "origin" : "Corda 0.10.1 Node 0 in tstyo2", "target" : "Corda 0.10.1 Node 1 in tstyo2", "yo" : "Yo!", "linearId" : { "externalId" : null, "id" : "a899aaf2-382b-4c8c-824c-0afd0b190bdb" }, "contract" : { "legalContractReference" : "D11BEC2116F70B65C948DC4B12799B3AB3317327FCBF8A002D40864F437FB3EF" }, "participants" : [ "3ThWzJauCq7qLrcX4KsJvrrNYiYw8PsueJKaJtWXyz4bS17Hv6ypST4NZURng7" ] }, "notary" : "corda.notary.simple|Corda 0.10.1 Notary Node in tstyo2", "encumbrance" : null }, "ref" : { "txhash" : "A1D2F79F3EFCC472BA6F3FBCEB80A3E6358D7F93AE7B115162663C16A559154C", "index" : 0 } } ]
```

Viewing logs

Users may wish to view the raw logs generated by each node, which contain more information about the operations performed by each node.

You can access these using an SSH client of your choice (e.g. Putty) and logging into the virtual machine using the public IP address. Once logged in, navigate to the following directory for Corda logs (node-xxxxxx):

`/opt/corda/logs`

And navigate to the following directory for system logs (syslog):

/var/log

You can open log files with any text editor.


```

May 19 10:31:16 ubuntu systemd[1]: Started corda node.
May 19 10:31:19 ubuntu systemd[1]: Stopping Corda Webserver...
May 19 10:31:20 ubuntu systemd[1]: corda-webserver.service: Main process exited, code=exited, status=143/n/a
May 19 10:31:20 ubuntu systemd[1]: Stopped Corda Webserver.
May 19 10:31:20 ubuntu systemd[1]: corda-webserver.service: Unit entered failed state.
May 19 10:31:20 ubuntu systemd[1]: corda-webserver.service: Failed with result 'exit-code'.
May 19 10:31:20 ubuntu systemd[1]: Started Corda Webserver.
May 19 10:31:21 ubuntu java[62992]: 
May 19 10:31:21 ubuntu java[62992]: Whenever I go near my bank I get
May 19 10:31:21 ubuntu java[62992]: withdrawal symptoms
May 19 10:31:21 ubuntu java[62992]: 
May 19 10:31:21 ubuntu java[62992]: --- Corda Open Source 0.11.0 (d552037) -----
May 19 10:31:21 ubuntu java[62992]: New! Training now available worldwide, see https://corda.net/corda-training/
May 19 10:31:21 ubuntu java[62992]: Logs can be found in : /opt/corda/logs
May 19 10:31:27 ubuntu java[63033]: Logs can be found in /opt/corda/logs/web
May 19 10:31:29 ubuntu java[62992]: Database connection url is : jdbc:h2:tcp://10.0.0.32:11000/node
May 19 10:31:31 ubuntu java[63033]: Starting as webserver: 0.0.0.0:10004
May 19 10:31:32 ubuntu java[62992]: WARNING: The specified messaging host "10.0.0.32" is private, this node will not be reachable by any other node
s outside the private network.
May 19 10:31:42 ubuntu java[62992]: Listening on address : 10.0.0.32:10002
May 19 10:31:42 ubuntu java[62992]: RPC service listening on address : 10.0.0.32:10003
May 19 10:31:43 ubuntu java[62992]: Providing network services :
May 19 10:31:43 ubuntu java[62992]: Loaded plugins : net.corda.yo.YoPlugin
May 19 10:31:52 ubuntu java[63033]: Webserver started up in 29.09 sec
May 19 10:31:56 ubuntu java[62992]: Node for "CN=Corda 0.11.0 Node 2 in tsty01, O=Corda 0.11.0 Node 2 in tsty01, L=London, C=UK" started up and reg
istered in 38.81 sec
May 19 10:39:41 ubuntu java[63033]: 07:55 WARN: [kryo] Unable to load class net.corda.node.services.statemachine.FlowHandleImpl with kryo's ClassL
oader. Retrying with current..
May 19 10:39:50 ubuntu java[63033]: 08:04 WARN: [kryo] Unable to load class net.corda.node.services.statemachine.FlowHandleImpl with kryo's ClassL
oader. Retrying with current..
May 19 10:40:57 ubuntu java[63033]: 09:10 WARN: [kryo] Unable to load class net.corda.node.services.statemachine.FlowHandleImpl with kryo's ClassL
oader. Retrying with current..
May 19 10:45:17 ubuntu java[63033]: 13:31 WARN: [kryo] Unable to load class net.corda.node.services.statemachine.FlowHandleImpl with kryo's ClassL
oader. Retrying with current..
May 19 10:50:30 ubuntu java[63033]: 18:44 WARN: [kryo] Unable to load class net.corda.node.services.statemachine.FlowHandleImpl with kryo's ClassL
oader. Retrying with current..
May 19 11:07:38 ubuntu systemd[1]: Stopping corda node...
May 19 11:07:38 ubuntu java[62992]: Shutting down ...
May 19 11:07:39 ubuntu systemd[1]: corda.service: Main process exited, code=exited, status=143/n/a
May 19 11:07:39 ubuntu systemd[1]: Stopped corda node.
May 19 11:07:39 ubuntu systemd[1]: corda.service: Unit entered failed state.
May 19 11:07:39 ubuntu systemd[1]: corda.service: Failed with result 'exit-code'.
May 19 11:17:01 ubuntu CRON[63634]: (root) CMD ( cd / && run-parts --report /etc/cron.hourly)
May 19 11:25:01 ubuntu systemd[1]: Stopping Corda Webserver...
May 19 11:25:01 ubuntu systemd[1]: corda-webserver.service: Main process exited, code=exited, status=143/n/a
May 19 11:25:01 ubuntu systemd[1]: Stopped Corda Webserver.
May 19 11:25:01 ubuntu systemd[1]: corda-webserver.service: Unit entered failed state.
May 19 11:25:01 ubuntu systemd[1]: corda-webserver.service: Failed with result 'exit-code'.
May 19 11:25:01 ubuntu systemd[1]: Started Corda Webserver.
May 19 11:25:04 ubuntu java[63706]: Logs can be found in /opt/corda/logs/web
May 19 11:25:06 ubuntu java[63706]: Starting as webserver: 0.0.0.0:10004
May 19 11:26:36 ubuntu java[63706]: Webserver started up in 93.59 sec
May 19 11:29:38 ubuntu systemd[1]: snapd.refresh.timer: Adding 3h 18min 37.051326s random time.

```

Next Steps

Now you have built a Corda network and used a basic Corda CorDapp do go and visit the [dedicated Corda website](#)

Or to join the growing Corda community and get straight into the Corda open source codebase, head over to the [Github Corda repo](#)

[Next](#) [Previous](#)

- Building a Corda VM from the AWS Marketplace
- [View page source](#)

Building a Corda VM from the AWS Marketplace

To help you design, build and test applications on Corda, called CorDapps, a Corda network AMI can be deployed from the [AWS Marketplace](#). Instructions on running Corda nodes can be found [here](#).

This Corda network offering builds a pre-configured network of Corda nodes as Ubuntu virtual machines (VM). The network consists of a Notary node and three Corda nodes using version 1 of Corda. The following guide will also show you how to load one of four [Corda Sample apps](#) which demonstrates the basic principles of Corda. When you are ready to go further with developing on Corda and start making contributions to the project head over to the [Corda.net](#).

Pre-requisites

- Ensure you have a registered AWS account which can create virtual machines under your subscription(s) and you are logged on to the [AWS portal](#)
- It is recommended you generate a private-public SSH key pair (see [here](#))

Deploying a Corda Network

Browse to the [AWS Marketplace](#) and search for Corda.

Follow the instructions to deploy the AMI to an instance of EC2 which is in a region near to your location.

Build and Run a Sample CorDapp

Once the instance is running ssh into the instance using your keypair

```
cd ~/dev
```

There are 4 sample apps available by default

```
ubuntu@ip-xxx-xxx-xxx-xxx:~/dev$ ls -la
total 24
drwxrwxr-x  6 ubuntu ubuntu 4096 Nov 13 21:48 .
drwxr-xr-x  8 ubuntu ubuntu 4096 Nov 21 16:34 ..
drwxrwxr-x 11 ubuntu ubuntu 4096 Oct 31 19:02 cordapp-example
drwxrwxr-x  9 ubuntu ubuntu 4096 Nov 13 21:48 obligation-cordapp
drwxrwxr-x 11 ubuntu ubuntu 4096 Nov 13 21:48 oracle-example
drwxrwxr-x  8 ubuntu ubuntu 4096 Nov 13 21:48 yo-cordapp
```

cd into the Corda sample you would like to run. For example:

```
cd cordapp-example/
```

Follow instructions for the specific sample at <https://www.corda.net/samples> to build and run the Corda sample For example: with cordapp-example (IOU app) the following commands would be run:

```
./gradlew deployNodes  
./kotlin-source/build/nodes/runnodes
```

Then start the Corda webserver

```
find ~/dev/cordapp-example/kotlin-source/ -name corda-webserver.jar -execdir sh -c  
'java -jar {} &' \;
```

You can now interact with your running CorDapp. See the instructions [here](#)

Next Steps

Now you have built a Corda network and used a basic Corda Cordapp do go and visit the [dedicated Corda website](#)

Additional support is available on [Stack Overflow](#) and the [Corda Slack channel](#).

You can build and run any other [Corda samples](#) or your own custom CorDapp here.

Or to join the growing Corda community and get straight into the Corda open source codebase, head over to the [Github Corda repo](#)

[Next](#) [Previous](#)

- Load testing
 - [View page source](#)
-

Load testing

This section explains how to apply random load to nodes to stress test them. It also allows the specification of disruptions that strain different resources, allowing us to inspect the nodes' behaviour under extreme conditions.

The load-testing framework is incomplete and is not part of CI currently, but the basic pieces are there.

Configuration of the load testing cluster

The load-testing framework currently assumes the following about the node cluster:

- The nodes are managed as a systemd service
- The node directories are the same across the cluster
- The messaging ports are the same across the cluster
- The executing identity of the load-test has SSH access to all machines
- There is a single network map service node
- There is a single notary node
- Some disruptions also assume other tools (like openssl) to be present

Note that these points could and should be relaxed as needed.

The load test Main expects a single command line argument that points to a configuration file specifying the cluster hosts and optional overrides for the default configuration:

```
# nodeHosts = ["host1", "host2"]
# sshUser = "someusername", by default it uses the System property "user.name"
# executionFrequency = <number of execution per second> , optional, defaulted to 20
# flow execution per second.
# generateCount = <number of generated command> , optional, defaulted to 10000.
# parallelism = <unmber of thread used to execete the commands>, optional, defaulted to
# [ForkJoinPool] default parallelism.
localCertificatesBaseDirectory = "build/load-test/certificates"
localTunnelStartingPort = 10000
remoteNodeDirectory = "/opt/corda"
rpcPort = 10003
remoteSystemdServiceName = "corda"
rpcUser = {username = corda, password = not_blockchain, permissions = ["ALL"]}
```

Running the load tests

In order to run the loadtests you need to have an active SSH-agent running with a single identity added that has SSH access to the loadtest cluster.

You can use either IntelliJ or the gradle command line to start the tests.

To use gradle with configuration file: `./gradlew tools:loadtest:run -Ploadtest-`
`config=PATH_TO_LOADTEST_CONF`

To use gradle with system properties: `./gradlew tools:loadtest:run -`
`Dloadtest.mode=LOAD_TEST -Dloadtest.nodeHosts.0=node0.myhost.com`

Note

You can provide or override any configuration using the system properties, all properties will need to be prefixed with `loadtest.`.

To use IntelliJ simply run Main.kt with the config path supplied as an argument or system properties as vm options.

Configuration of individual load tests

The load testing configurations are not set-in-stone and are meant to be played with to see how the nodes react.

There are a couple of top-level knobs to tweak test behaviour:

```
/**
 * @param parallelism Number of concurrent threads to use to run commands. Note
that the actual parallelism may be
 *      further limited by the batches that [generate] returns.
 * @param generateCount Number of total commands to generate. Note that the actual
number of generated commands may
 *      exceed this, it is used just for cutoff.
 * @param clearDatabaseBeforeRun Indicates whether the node databases should be
cleared before running the test. May
 *      significantly slow down testing as this requires bringing the nodes down
and up again.
 * @param gatherFrequency Indicates after how many commands we should gather the
remote states.
 * @param disruptionPatterns A list of disruption-lists. The test will be run for
each such list, and the test will
 *      be interleaved with the specified disruptions.
 */
data class RunParameters(
    val parallelism: Int,
    val generateCount: Int,
    val clearDatabaseBeforeRun: Boolean,
    val executionFrequency: Int?,
    val gatherFrequency: Int,
    val disruptionPatterns: List<List<DisruptionSpec>>
)
```

The one thing of note is `disruptionPatterns`, which may be used to specify ways of disrupting the normal running of the load tests.

```
data class Disruption(
    val name: String,
    val disrupt: (NodeConnection, SplittableRandom) -> Unit
)

data class DisruptionSpec(
    val nodeFilter: (NodeConnection) -> Boolean,
    val disruption: Disruption,
    val noDisruptionWindowMs: LongRange
)
```

Disruptions run concurrently in loops on randomly chosen nodes filtered by `nodeFilter` at somewhat random intervals.

As an example take `strainCpu` which overutilises the processor:

```
fun strainCpu(parallelism: Int, durationSeconds: Int) = Disruption("Put strain on  
cpu") { connection, _ ->  
    val shell = "for c in {1..$parallelism} ; do openssl enc -aes-128-cbc -in  
/dev/urandom -pass pass: -e > /dev/null & done && JOBS=\$(jobs -p) && (sleep  
$durationSeconds && kill \$JOBS) & wait"  
    connection.runShellCommandGetOutput(shell).getResultOrThrow()  
}
```

We can use this by specifying a `DisruptionSpec` in the load test's `RunParameters`:

```
DisruptionSpec(  
    disruption = strainCpu(parallelism = 4, durationSeconds = 10),  
    nodeFilter = { true },  
    noDisruptionWindowMs = 5000L..10000L  
)
```

This means every 5-10 seconds at least one randomly chosen nodes' cores will be spinning 100% for 10 seconds.

How to write a load test

A load test is basically defined by a random datastructure generator that specifies a unit of work a node should perform, a function that performs this work, and a function that predicts what state the node should end up in by doing so:

```
data class LoadTest<T, S>(  
    val testName: String,  
    val generate: Nodes.(S, Int) -> Generator<List<T>>,  
    val interpret: (S, T) -> S,  
    val execute: Nodes.(T) -> Unit,  
    val gatherRemoteState: Nodes.(S?) -> S,  
    val isConsistent: (S) -> Boolean = { true }  
) {
```

`LoadTest` is parameterised over `T`, the unit of work, and `S`, the state type that aims to track remote node states. As an example let's look at the Self Issue test. This test simply creates Cash Issues from nodes to themselves, and then checks the vault to see if the numbers add up:

```
data class SelfIssueCommand(  
    val request: IssueAndPaymentRequest,  
    val node: NodeConnection  
)
```

```

data class SelfIssueState(
    val vaultsSelfIssued: Map<AbstractParty, Long>
) {
    fun copyVaults(): HashMap<AbstractParty, Long> {
        return HashMap(vaultsSelfIssued)
    }
}

val selfIssueTest = LoadTest<SelfIssueCommand, SelfIssueState>()

```

The unit of work `SelfIssueCommand` simply holds an Issue and a handle to a node where the issue should be submitted. The `generate` method should provide a generator for these.

The state `SelfIssueState` then holds a map from node identities to a Long that describes the sum quantity of the generated issues (we fixed the currency to be USD).

The invariant we want to hold then simply is: The sum of submitted Issues should be the sum of the quantities in the vaults.

The `interpret` function should take a `SelfIssueCommand` and update `SelfIssueState` to reflect the change we're expecting in the remote nodes. In our case this will simply be adding the issued amount to the corresponding node's Long.

The `execute` function should perform the action on the cluster. In our case it will simply take the node handle and submit an RPC request for the Issue.

The `gatherRemoteState` function should check the actual remote nodes' states and see whether they conflict with our local predictions (and should throw if they do). This function deserves its own paragraph.

```
val gatherRemoteState: Nodes.(S?) -> S,
```

`gatherRemoteState` gets as input handles to all the nodes, and the current predicted state, or null if this is the initial gathering.

The reason it gets the previous state boils down to allowing non-deterministic predictions about the nodes' remote states. Say some piece of work triggers an asynchronous notification of a node. We need to account both for the case when the node hasn't received the notification and for the case when it has. In these cases `S` should somehow represent a collection of possible states,

and `gatherRemoteState` should “collapse” the collection based on the observations it makes. Of course we don’t need this for the simple case of the Self Issue test.

The last parameter `isConsistent` is used to poll for eventual consistency at the end of a load test. This is not needed for self-issuance.

Stability Test

Stability test is one variation of the load test, instead of flooding the nodes with request, the stability test uses execution frequency limit to achieve a constant execution rate.

To run the stability test, set the load test mode to `STABILITY_TEST` (`mode=STABILITY_TEST` in config file or `-Dloadtest.mode=STABILITY_TEST` in system properties).

The stability test will first self issue cash using `StabilityTest.selfIssueTest` and after that it will randomly pay and exit cash using `StabilityTest.crossCashTest` for P2P testing, unlike the load test, the stability test will run without any disruption

Node internals

- [Node services](#)
- [Vault](#)
- [Networking and messaging](#)

[Next](#) [Previous](#)

- Node services
- [View page source](#)

Node services

This document is intended as a very brief introduction to the current service components inside the node. Whilst not at all exhaustive it is hoped that this will give some context when writing applications and code that use these services, or which are operated upon by the internal components of Corda.

Services within the node

The node services represent the various sub functions of the Corda node. Some are directly accessible to contracts and flows through the `ServiceHub`, whilst others are the framework internals used to host the node functions. Any public service interfaces are defined in the `:core` gradle project in the `src/main/kotlin/net/corda/core/node/services` folder. The `ServiceHub` interface exposes functionality suitable for flows. The implementation code for all standard services lives in the gradle `:node` project under the `src/main/kotlin/net/corda/node/services` folder. The `src/main/kotlin/net/corda/node/services/api` folder contains declarations for internal only services and for interoperation between services.

All the services are constructed in the `AbstractNode start` method (and the extension in `Node`). They may also register a shutdown handler during initialisation, which will be called in reverse order to the start registration sequence when the `Node.stop` is called.

For unit testing a number of non-persistent, memory only services are defined in the `:node` and `:test-utils` projects. The `:test-utils` project also provides an in-memory networking simulation to allow unit testing of flows and service functions.

The roles of the individual services are described below.

Key management and identity services

InMemoryIdentityService

The `InMemoryIdentityService` implements the `IdentityService` interface and provides a store of remote mappings between `CompositeKey` and remote `Parties`. It is automatically populated from the `NetworkMapCache` updates and is used when translating `CompositeKey` exposed in transactions into fully populated `Party` identities. This service is also used in the default JSON mapping of parties in the web server, thus allowing the party names to be used to refer to other nodes' legal identities. In the future the Identity service will be made persistent and extended to allow anonymised session keys to be used in flows where the well-known `CompositeKey` of nodes need to be hidden to non-involved parties.

PersistentKeyManagementService and E2ETestKeyManagementService

Typical usage of these services is to locate an appropriate `PrivateKey` to complete and sign a verified transaction as part of a flow. The normal node legal identifier keys are typically accessed via helper extension methods on the `ServiceHub`, but these ultimately delegate signing to internal `PrivateKeys` from the `KeyManagementService`. The `KeyManagementService` interface also allows other keys to be generated if anonymous keys are needed in a flow. Note that this interface works at the level of individual `PublicKey` and internally matched `PrivateKey` pairs, but the signing authority may be represented by a ``CompositeKey` on the `NodeInfo` to allow key clustering and threshold schemes.

The `PersistentKeyManagementService` is a persistent implementation of the `KeyManagementService` interface that records the key pairs to a key-value storage table in the database. `E2ETestKeyManagementService` is a simple implementation of the `KeyManagementService` that is used to track our `KeyPairs` for use in unit testing when no database is available.

Messaging and network management services

ArtemisMessagingServer

The `ArtemisMessagingServer` service is run internally by the Corda node to host the `ArtemisMQ` messaging broker that is used for reliable node communications. Although the node can be configured to disable this and connect to a remote broker by setting the `messagingServerAddress` configuration to be the remote broker address. (The `MockNode` used during testing does not use this service, and has a simplified in-memory network layer instead.) This service is not exposed to any CorDapp code as it is an entirely internal infrastructural component. However, the developer may need to be aware of this component, because the `ArtemisMessagingServer` is responsible for configuring the network ports (based upon settings in `node.conf`) and the service configures the security settings of the `ArtemisMQ` middleware and acts to form bridges between node mailbox queues based upon connection details advertised by the `NetworkMapService`.

The `ArtemisMQ` broker is configured to use TLS1.2 with a custom `TrustStore` containing a Corda root certificate and a `KeyStore` with a certificate and key signed by a chain back to this root certificate. These keystores typically reside in the `certificates` sub folder of the node workspace.

For the nodes to be able to connect to each other it is essential that the entire set of nodes are able to authenticate against each other and thus typically that they share a common root certificate. Also note that the address configuration defined for the server is the basis for the address advertised in the NetworkMapService and thus must be externally connectable by all nodes in the network.

NodeMessagingClient

The `NodeMessagingClient` is the implementation of the `MessagingService` interface operating across the `ArtemisMQ` middleware layer. It typically connects to the local `ArtemisMQ` hosted within the `ArtemisMessagingServer` service. However, the `messagingServerAddress` configuration can be set to a remote broker address if required. The responsibilities of this service include managing the node's persistent mailbox, sending messages to remote peer nodes, acknowledging properly consumed messages and deduplicating any resent messages. The service also handles the incoming requests from new RPC client sessions and hands them to the `CordaRPCOpsImpl` to carry out the requests.

InMemoryNetworkMapCache

The `InMemoryNetworkMapCache` implements the `NetworkMapCache` interface and is responsible for tracking the identities and advertised services of authorised nodes provided by the remote `NetworkMapService`. Typical use is to search for nodes hosting specific advertised services e.g. a Notary service, or an Oracle service. Also, this service allows mapping of friendly names, or `Party` identities to the full `NodeInfo` which is used in the `StateMachineManager` to convert between the `CompositeKey`, or `Party` based addressing used in the flows/contracts and the physical host and port information required for the physical `ArtemisMQ` messaging layer.

Storage and persistence related services

StorageServiceImpl

The `StorageServiceImpl` service simply hold references to the various persistence related services and provides a single grouped interface on the `ServiceHub`.

DBCheckpointStorage

The `DBCheckpointStorage` service is used from within the `StateMachineManager` code to persist the progress of flows. Thus ensuring that if the program terminates the flow can be restarted from the same point and complete the flow. This service should not be used by any CorDapp components.

DBTransactionMappingStorage and InMemoryStateMachineRecordedTransactionMappingStorage

The `DBTransactionMappingStorage` is used within the `StateMachineManager` code to relate transactions and flows. This relationship is exposed in the eventing interface to the RPC clients, thus allowing them to track the end result of a flow and map to the actual transactions/states completed. Otherwise this service is unlikely to be accessed by any CorDapps.

The `InMemoryStateMachineRecordedTransactionMappingStorage` service is available as a non-persistent implementation for unit tests with no database.

DBTransactionStorage

The `DBTransactionStorage` service is a persistent implementation of the `TransactionStorage` interface and allows flows read-only access to full transactions, plus transaction level event callbacks. Storage of new transactions must be made via the `recordTransactions` method on the `ServiceHub`, not via a direct call to this service, so that the various event notifications can occur.

NodeAttachmentService

The `NodeAttachmentService` provides an implementation of the `AttachmentStorage` interface exposed on the `ServiceHub` allowing transactions to add documents, copies of the contract code and binary data to transactions. The service is also interfaced to by the web server, which allows files to be uploaded via an HTTP post request.

Flow framework and event scheduling services

StateMachineManager

The `StateMachineManager` is the service that runs the active flows of the node whether initiated by an RPC client, the web interface, a scheduled state activity, or triggered by receipt of a message from another node.

The `StateMachineManager` wraps the flow code (extensions of the `FlowLogic` class) inside an instance of the `FlowStateMachineImpl` class, which is a `Quasar Fiber`. This allows the `StateMachineManager` to suspend flows at all key lifecycle points and persist their serialized state to the database via the `DBCheckpointStorage` service. This process uses the facilities of the `Quasar Fibers` library to manage this process and hence the requirement for the node to run the `Quasar` java instrumentation agent in its JVM.

In operation the `StateMachineManager` is typically running an active flow on its server thread until it encounters a blocking, or externally visible operation, such as sending a message, waiting for a message, or initiating a `subFlow`. The fiber is then suspended and its stack frames serialized to the database, thus ensuring that if the node is stopped, or crashes at this point the flow will restart with exactly the same action again. To further ensure consistency, every event which resumes a flow opens a database transaction, which is committed during this suspension process ensuring that the database modifications e.g. state commits stay in sync with the mutating changes of the flow. Having recorded the fiber state the `StateMachineManager` then carries out the network actions as required (internally one flow message exchanged may actually involve several physical session messages to authenticate and invoke registered flows on the remote nodes). The flow will stay suspended until the required message is returned and the scheduler will resume processing of other activated flows. On receipt of the expected response message from the network layer the `StateMachineManager` locates the appropriate flow, resuming it immediately after the blocking step with the received message. Thus from the perspective of the flow the code executes as a simple linear progression of processing, even if there were node restarts and possibly message resends (the messaging layer deduplicates messages based on an id that is part of the checkpoint).

The `StateMachineManager` service is not directly exposed to the flows, or contracts themselves.

NodeSchedulerService

The `NodeSchedulerService` implements the `SchedulerService` interface and monitors the Vault updates to track any new states that implement the `SchedulableState` interface and require automatic scheduled flow initiation. At the scheduled due time the `NodeSchedulerService` will create a new flow instance passing it a reference to the state that triggered the event. The flow can then

begin whatever action is required. Note that the scheduled activity occurs in all nodes holding the state in their Vault, it may therefore be required for the flow to exit early if the current node is not the intended initiator.

Notary flow implementation services

PersistentUniquenessProvider, InMemoryUniquenessProvider and RaftUniquenessProvider

These variants of `UniquenessProvider` service are used by the notary flows to track consumed states and thus reject double-spend scenarios.

The `InMemoryUniquenessProvider` is for unit testing only, the default being the `PersistentUniquenessProvider` which records the changes to the DB. When the Raft based notary is active the states are tracked by the whole cluster using a `RaftUniquenessProvider`. Outside of the notary flows themselves this service should not be accessed by any CorDapp components.

NotaryService (SimpleNotaryService, ValidatingNotaryService, RaftValidatingNotaryService)

The `NotaryService` is an abstract base class for the various concrete implementations of the Notary server flow. By default, a node does not run any `NotaryService` server component. For that you need to specify the `notary` config. The node may then participate in controlling state uniqueness when contacted by nodes using the `NotaryFlow.Client subFlow`.

The `SimpleNotaryService` only offers protection against double spend, but does no further verification. The `ValidatingNotaryService` checks that proposed transactions are correctly signed by all keys listed in the commands and runs the contract verify to ensure that the rules of the state transition are being followed. The `RaftValidatingNotaryService` further extends the flow to operate against a cluster of nodes running shared consensus state across the RAFT protocol (note this requires the additional configuration of the `notaryClusterAddresses` property).

Vault related services

NodeVaultService

The `NodeVaultService` implements the `VaultService` interface to allow access to the node's own set of unconsumed states. The service does this by tracking update notifications from the `TransactionStorage` service and processing relevant updates to delete consumed states and insert new states. The resulting update is then persisted to the database. The `VaultService` then exposes query and event notification APIs to flows and CorDapp services to allow them to respond to updates, or query for states meeting various conditions to begin the formation of new transactions consuming them. The equivalent services are also forwarded to RPC clients, so that they may show updating views of states held by the node.

NodeSchemaService and HibernateObserver

The `HibernateObserver` runs within the node framework and listens for vault state updates, the `HibernateObserver` then uses the mapping services of the `NodeSchemaService` to record the states in auxiliary database tables. This allows Corda state updates to be exposed to external legacy systems by insertion of unpacked data into existing tables. To enable these features the contract state must implement the `QueryableState` interface to define the mappings.

Corda Web Server

A simple web server is provided that embeds the Jetty servlet container. The Corda web server is not meant to be used for real, production-quality web apps. Instead it shows one example way of using Corda RPC in web apps to provide a REST API on top of the Corda native RPC mechanism.

Note

The Corda web server may be removed in future and replaced with sample specific webapps using a standard framework like Spring Boot.

[Next](#) [Previous](#)

- [Vault](#)
- [View page source](#)

Vault

The vault contains data extracted from the ledger that is considered relevant to the node's owner, stored in a relational model that can be easily queried and worked with.

The vault keeps track of both unconsumed and consumed states:

- Unconsumed (or unspent) states represent fungible states available for spending (including spend-to-self transactions) and linear states available for evolution (eg. in response to a lifecycle event on a deal) or transfer to another party.
- Consumed (or spent) states represent ledger immutable state for the purpose of transaction reporting, audit and archival, including the ability to perform joins with app-private data (like customer notes)

By fungible we refer to assets of measurable quantity (eg. a cash currency, units of stock) which can be combined together to represent a single ledger state.

Like with a cryptocurrency wallet, the Corda vault can create transactions that send value (eg. transfer of state) to someone else by combining fungible states and possibly adding a change output that makes the values balance (this process is usually referred to as 'coin selection'). Vault spending ensures that transactions respect the fungibility rules in order to ensure that the issuer and reference data is preserved as the assets pass from hand to hand.

A feature called **soft locking** provides the ability to automatically or explicitly reserve states to prevent multiple transactions within the same node from trying to use the same output simultaneously. Whilst this scenario would ultimately be detected by a notary, *soft locking* provides a mechanism of early detection for such unwarranted and invalid scenarios. [Soft Locking](#) describes this feature in detail.

Note

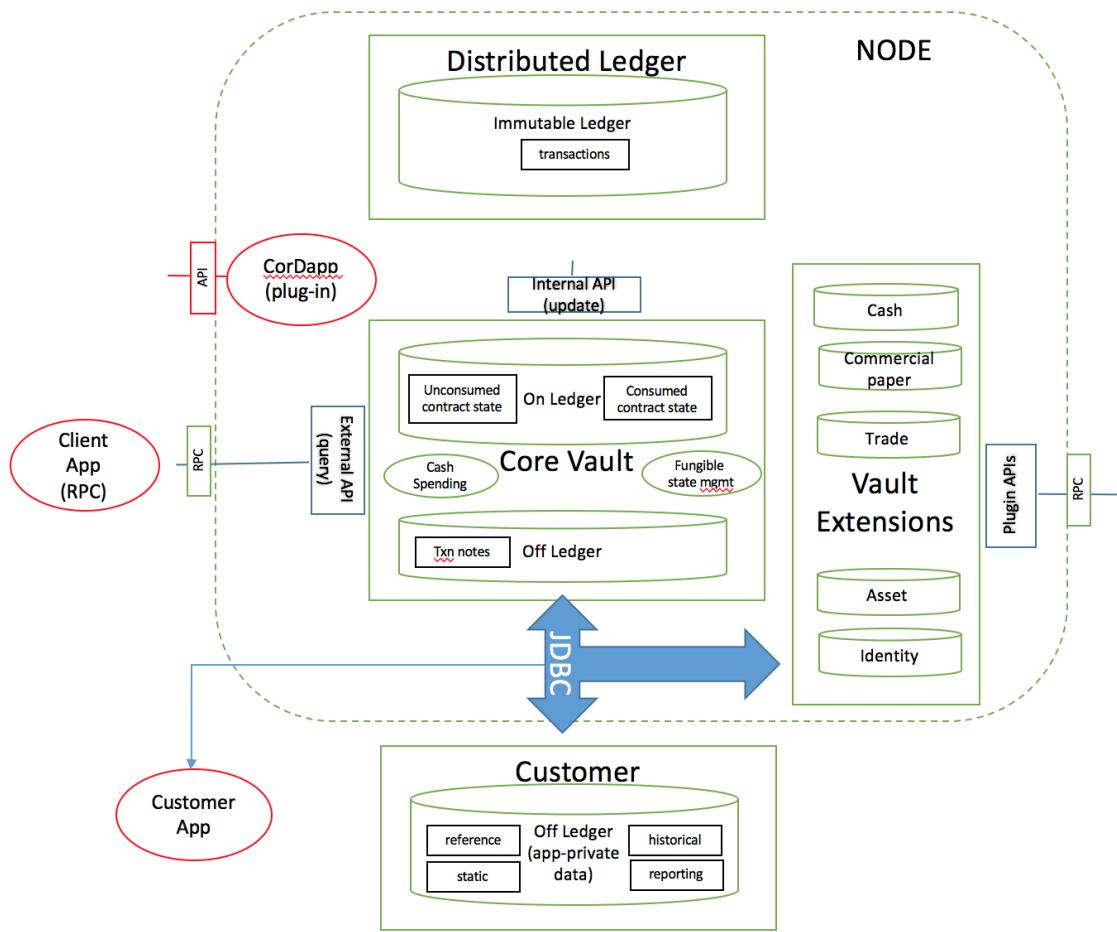
Basic 'coin selection' is currently implemented. Future work includes fungible state optimisation (splitting and merging of states in the background), and 'state re-issuance' (sending of states back to the issuer for re-issuance, thus pruning long transaction chains and improving privacy).

There is also a facility for attaching descriptive textual notes against any transaction stored in the vault.

The vault supports the management of data in both authoritative (“on-ledger”) form and, where appropriate, shadow (“off-ledger”) form:

- “On-ledger” data refers to distributed ledger state (cash, deals, trades) to which a firm is participant.
- “Off-ledger” data refers to a firm’s internal reference, static and systems data.

The following diagram illustrates the breakdown of the vault into sub-system components:



Note the following:

- the vault “On Ledger” store tracks unconsumed state and is updated internally by the node upon recording of a transaction on the ledger (following successful smart contract verification and signature by all participants)
- the vault “Off Ledger” store refers to additional data added by the node owner subsequent to transaction recording
- the vault performs fungible state spending (and in future, fungible state optimisation management including merging, splitting and re-issuance)

- vault extensions represent additional custom plugin code a developer may write to query specific custom contract state attributes.
- customer “Off Ledger” (private store) represents internal organisational data that may be joined with the vault data to perform additional reporting or processing
- a [Vault Query API](#) is exposed to developers using standard Corda RPC and CorDapp plugin mechanisms
- a vault update API is internally used by transaction recording flows.
- the vault database schemas are directly accessible via JDBC for customer joins and queries

Section 8 of the [Technical white paper](#) describes features of the vault yet to be implemented including private key management, state splitting and merging, asset re-issuance and node event scheduling.

[Next](#) [Previous](#)

- Networking and messaging
 - [View page source](#)
-

Networking and messaging

Corda uses AMQP/1.0 over TLS between nodes which is currently implemented using Apache Artemis, an embeddable message queue broker. Building on established MQ protocols gives us features like persistence to disk, automatic delivery retries with backoff and dead-letter routing, security, large message streaming and so on.

Artemis is hidden behind a thin interface that also has an in-memory only implementation suitable for use in unit tests and visualisation tools.

Note

A future version of Corda will allow the MQ broker to be split out of the main node and run as a separate server. We may also support non-Artemis implementations via JMS, allowing the broker to be swapped out for alternative implementations.

There are multiple ways of interacting with the network. When writing an application you typically won't use the messaging subsystem directly. Instead you will build on top of the [flow framework](#), which adds a layer on top of raw messaging to manage multi-step flows and let you think in terms of identities rather than specific network endpoints.

Network Map Service

Supporting the messaging layer is a network map service, which is responsible for tracking public nodes on the network.

Nodes have an internal component, the network map cache, which contains a copy of the network map (which is just a document). When a node starts up its cache fetches a copy of the full network map, and requests to be notified of changes. The node then registers itself with the network map service, and the service notifies subscribers that a new node has joined the network. Nodes do not automatically deregister themselves, so (for example) nodes going offline briefly for maintenance are retained in the network map, and messages for them will be queued, minimising disruption.

Nodes submit signed changes to the map service, which then forwards update notifications on to nodes which have requested to be notified of changes.

The network map currently supports:

- Looking up nodes by service
- Looking up node for a party
- Suggesting a node providing a specific service, based on suitability for a contract and parties, for example suggesting an appropriate interest rates oracle for an interest rate swap contract. Currently no recommendation logic is in place.

Message queues

The node makes use of various queues for its operation. The more important ones are described below. Others are used for maintenance and other minor purposes.

`p2p.inbound.$identity`:

The node listens for messages sent from other peer nodes on this queue. Only clients who are authenticated to be nodes on the same network are given permission to send. Messages which are routed internally are also sent to this queue (e.g. two flows on the same node communicating with each other).

`internal.peers.$identity`:

These are a set of private queues only available to the node which it uses to route messages destined to other peers. The queue name ends in the base 58 encoding of the peer's identity key. There is at most one queue per peer. The broker

creates a bridge from this queue to the peer's `p2p.inbound.$identity` queue, using the network map service to lookup the peer's network address.

`internal.services.$identity`:

These are private queues the node may use to route messages to services. The queue name ends in the base 58 encoding of the service's owning identity key. There is at most one queue per service identity (but note that any one service may have several identities). The broker creates bridges to all nodes in the network advertising the service in question. When a session is initiated with a service counterparty the handshake is pushed onto this queue, and a corresponding bridge is used to forward the message to an advertising peer's p2p queue. Once a peer is picked the session continues on as normal.

`rpc.requests`:

RPC clients send their requests here, and it's only open for sending by clients authenticated as RPC users.

`clients.$user.rpc.$random`:

RPC clients are given permission to create a temporary queue incorporating their username (`$user`) and sole permission to receive messages from it. RPC requests are required to include a random number (`$random`) from which the node is able to construct the queue the user is listening on and send the response to that. This mechanism prevents other users from being able listen in on the responses.

Security

Clients attempting to connect to the node's broker fall in one of four groups:

1. Anyone connecting with the username `SystemUsers/Node` is treated as the node hosting the broker, or a logical component of the node. The TLS certificate they provide must match the one broker has for the node. If that's the case they are given full access to all valid queues, otherwise they are rejected.

2. Anyone connecting with the username `SystemUsers/Peer` is treated as a peer on the same Corda network as the node. Their TLS root CA must be the same as the node's root CA - the root CA is the doorman of the network and having the same root CA implies we've been let in by the same doorman. If they are part of the same network then they are only given permission to send to our `p2p.inbound.$identity` queue, otherwise they are rejected.
3. Every other username is treated as a RPC user and authenticated against the node's list of valid RPC users. If that is successful then they are only given sufficient permission to perform RPC, otherwise they are rejected.
4. Clients connecting without a username and password are rejected.

Artemis provides a feature of annotating each received message with the validated user. This allows the node's messaging service to provide authenticated messages to the rest of the system. For the first two client types described above the validated user is the X.500 subject of the client TLS certificate. This allows the flow framework to authentically determine the `Party` initiating a new flow. For RPC clients the validated user is the username itself and the RPC framework uses this to determine what permissions the user has.

The broker also does host verification when connecting to another peer. It checks that the TLS certificate subject matches with the advertised X.500 legal name from the network map service.

[Next](#) [Previous](#)

- Component library
 - View page source
-

Component library

- Flow library
- Contract catalogue
- Financial model
- Interest rate swaps

[Next](#) [Previous](#)

- »
 - Flow library
 - [View page source](#)
-

Flow library

There are a number of built-in flows supplied with Corda, which cover some core functionality.

FinalityFlow

The `FinalityFlow` verifies the given transactions, then sends them to the specified notary.

If the notary agrees that the transactions are acceptable then they are from that point onwards committed to the ledger, and will be written through to the vault. Additionally they will be distributed to the parties reflected in the participants list of the states.

The transactions will be topologically sorted before commitment to ensure that dependencies are committed before dependents, so you don't need to do this yourself.

The transactions are expected to have already been resolved: if their dependencies are not available in local storage or within the given set, verification will fail. They must have signatures from all necessary parties other than the notary.

If specified, the extra recipients are sent all the given transactions. The base set of parties to inform of each transaction are calculated on a per transaction basis from the contract-given set of participants.

The flow returns the same transactions, in the same order, with the additional signatures.

CollectSignaturesFlow

The `CollectSignaturesFlow` is used to automate the collection of signatures from the counterparties to a transaction.

You use the `CollectSignaturesFlow` by passing it a `SignedTransaction` which has at least been signed by yourself. The flow will handle the resolution of the counterparty identities and request a signature from each counterparty.

Finally, the flow will verify all the signatures and return a `SignedTransaction` with all the collected signatures.

When using this flow on the responding side you will have to subclass the `AbstractCollectSignaturesFlowResponder` and provide your own implementation of the `checkTransaction` method. This is to add additional verification logic on the responder side. Types of things you will need to check include:

- Ensuring that the transaction you are receiving is the transaction you *EXPECT* to receive. I.e. is has the expected type of inputs and outputs
- Checking that the properties of the outputs are as you would expect, this is in the absence of integrating reference data sources to facilitate this for us
- Checking that the transaction is not incorrectly spending (perhaps maliciously) one of your asset states, as potentially the transaction creator has access to some of your state references

Typically after calling the `CollectSignaturesFlow` you then called the `FinalityFlow`.

SendTransactionFlow/ReceiveTransactionFlow

The `SendTransactionFlow` and `ReceiveTransactionFlow` are used to automate the verification of the transaction by recursively checking the validity of all the dependencies. Once a transaction is received and checked it's inserted into local storage so it can be relayed and won't be checked again.

The `SendTransactionFlow` sends the transaction to the counterparty and listen for data request as the counterparty validating the transaction, extra checks can be implemented to restrict data access by overriding the `verifyDataRequest` method inside `SendTransactionFlow`.

The `ReceiveTransactionFlow` returns a verified `SignedTransaction`.

[Next](#) [Previous](#)

- Contract catalogue
 - [View page source](#)
-

Contract catalogue

There are a number of contracts supplied with Corda, which cover both core functionality (such as cash on ledger) and provide examples of how to model complex contracts (such as interest rate swaps). There is also a `Dummy` contract. However it does not provide any meaningful functionality, and is intended purely for testing purposes.

Cash

The `Cash` contract's state objects represent an amount of some issued currency, owned by some party. Any currency can be issued by any party, and it is up to the recipient to determine whether they trust the issuer. Generally nodes are expected to have criteria (such as a whitelist) that issuers must fulfil for cash they issue to be accepted.

Cash state objects implement the `FungibleAsset` interface, and can be used by the commercial paper and obligation contracts as part of settlement of an outstanding debt. The contracts' verification functions require that cash state objects of the correct value are received by the beneficiary as part of the settlement transaction.

The cash contract supports issuing, moving and exiting (destroying) states. Note, however, that issuance cannot be part of the same transaction as other cash commands, in order to minimise complexity in balance verification.

Cash shares a common superclass, `OnLedgerAsset`, with the Commodity contract. This implements common behaviour of assets which can be issued, moved and exited on chain, with the subclasses handling asset-specific data types and behaviour.

Note

Corda supports a pluggable cash selection algorithm by implementing the `CashSelection` interface. The default implementation uses an H2 specific query that can be overridden for different database providers. Please see `CashSelectionH2Impl` and its associated declaration in `META-INF\services\net.corda.finance.contracts.asset.CashSelection`

Commodity

The `Commodity` contract is an early stage example of a non-currency contract whose states implement the `FungibleAsset` interface. This is used as a proof of concept for non-cash obligations.

Commercial paper

`CommercialPaper` is a very simple obligation to pay an amount of cash at some future point in time (the maturity date), and exists primarily as a simplified contract for use in tutorials. Commercial paper supports issuing, moving and redeeming (settling) states. Unlike the full obligation contract it does not support locking the state so it cannot be settled if the obligor defaults on payment, or netting of state objects. All commands are exclusive of the other commercial paper commands. Use the `Obligation` contract for more advanced functionality.

Interest rate swap

The Interest Rate Swap (IRS) contract is a bilateral contract to implement a vanilla fixed / floating same currency interest rate swap. In general, an IRS allows two counterparties to modify their exposure from changes in the underlying interest rate. They are often used as a hedging instrument, convert a fixed rate loan to a floating rate loan, vice versa etc.

See “Interest rate swaps” for full details on the IRS contract.

Obligation

The obligation contract’s state objects represent an obligation to provide some asset, which would generally be a cash state object, but can be any contract state object fulfilling the `FungibleAsset` interface, including other obligations. The obligation contract uses objects referred to as `Terms` to group commands and

state objects together. Terms are a subset of an obligation state object, including details of what should be paid, when, and to whom.

Obligation state objects can be issued, moved and exited as with any fungible asset. The contract also supports state object netting and lifecycle changes (marking the obligation that a state object represents as having defaulted, or reverting it to the normal state after marking as having defaulted).

The `Net` command cannot be included with any other obligation commands in the same transaction, as it applies to state objects with different beneficiaries, and as such applies across multiple terms.

All other obligation contract commands specify obligation terms (what is to be delivered, by whom and by when) which are used as a grouping key for input/output states and commands. Issuance and lifecycle commands are mutually exclusive of other commands (move/exit) which apply to the same obligation terms, but multiple commands can be present in a single transaction if they apply to different terms. For example, a contract can have two different `Issue` commands as long as they apply to different terms, but could not have an `Issue` and a `Net`, or an `Issue` and `Move` that apply to the same terms.

Netting of obligations supports close-out netting (which can be triggered by either obligor or beneficiary, but is limited to bilateral netting), and payment netting (which requires signatures from all involved parties, but supports multilateral netting).

[Next](#) [Previous](#)

- Financial model
 - [View page source](#)
-

Financial model

Corda provides a large standard library of data types used in financial applications and contract state objects. These provide a common language for states and contracts.

Amount

The `Amount` class is used to represent an amount of some fungible asset. It is a generic class which wraps around a type used to define the underlying product, called the *token*. For instance it can be the standard JDK type `Currency`, or an `Issued` instance, or this can be a more complex type such as an obligation contract issuance definition (which in turn contains a token definition for whatever the obligation is to be settled in). Custom token types should implement `TokenizableAssetInfo` to allow the `Amount` conversion helpers `fromDecimal` and `toDecimal` to calculate the correct `displayTokenSize`.

Note

Fungible is used here to mean that instances of an asset is interchangeable for any other identical instance, and that they can be split/merged. For example a £5 note can reasonably be exchanged for any other £5 note, and a £10 note can be exchanged for two £5 notes, or vice-versa.

Here are some examples:

```
// A quantity of some specific currency like pounds, euros, dollars etc.  
Amount<Currency>  
// A quantity of currency that is issued by a specific issuer, for instance central  
bank vs other bank dollars  
Amount<Issued<Currency>>  
// A quantity of a product governed by specific obligation terms  
Amount<Obligation.Terms<P>>
```

`Amount` represents quantities as integers. You cannot use `Amount` to represent negative quantities, or fractional quantities: if you wish to do this then you must use a different type, typically `BigDecimal`. For currencies the quantity represents pennies, cents, or whatever else is the smallest integer amount for that currency, but for other assets it might mean anything e.g. 1000 tonnes of coal, or kilowatt-hours. The precise conversion ratio to displayable amounts is via the `displayTokenSize` property, which is the `BigDecimal` numeric representation of a single token as it would be written. `Amount` also defines methods to do overflow/underflow checked addition and subtraction (these are operator overloads in Kotlin and can be used as regular methods from Java). More complex calculations should typically be done in `BigDecimal` and converted back to ensure due consideration of rounding and to ensure token conservation.

`Issued` refers to a product (which can be cash, a cash-like thing, assets, or generally anything else that's quantifiable with integer quantities) and an associated `PartyAndReference` that describes the issuer of that contract. An issued product typically follows a lifecycle which includes issuance, movement and

exiting from the ledger (for example, see the `Cash` contract and its associated *state* and *commands*)

To represent movements of `Amount` tokens use the `AmountTransfer` type, which records the quantity and perspective of a transfer. Positive values will indicate a movement of tokens from a `source` e.g. a `Party`, or `CompositeKey` to a `destination`. Negative values can be used to indicate a retrograde motion of tokens from `destination` to `source`. `AmountTransfer` supports addition (as a Kotlin operator, or Java method) to provide netting and aggregation of flows. The `apply` method can be used to process a list of attributed `Amount` objects in a `List<SourceAndAmount>` to carry out the actual transfer.

Financial states

In addition to the common state types, a number of interfaces extend `ContractState` to model financial state such as:

`LinearState`

A state which has a unique identifier beyond its `StateRef` and carries it through state transitions. Such a state cannot be duplicated, merged or split in a transaction: only continued or deleted. A linear state is useful when modelling an indivisible/non-fungible thing like a specific deal, or an asset that can't be split (like a rare piece of art).

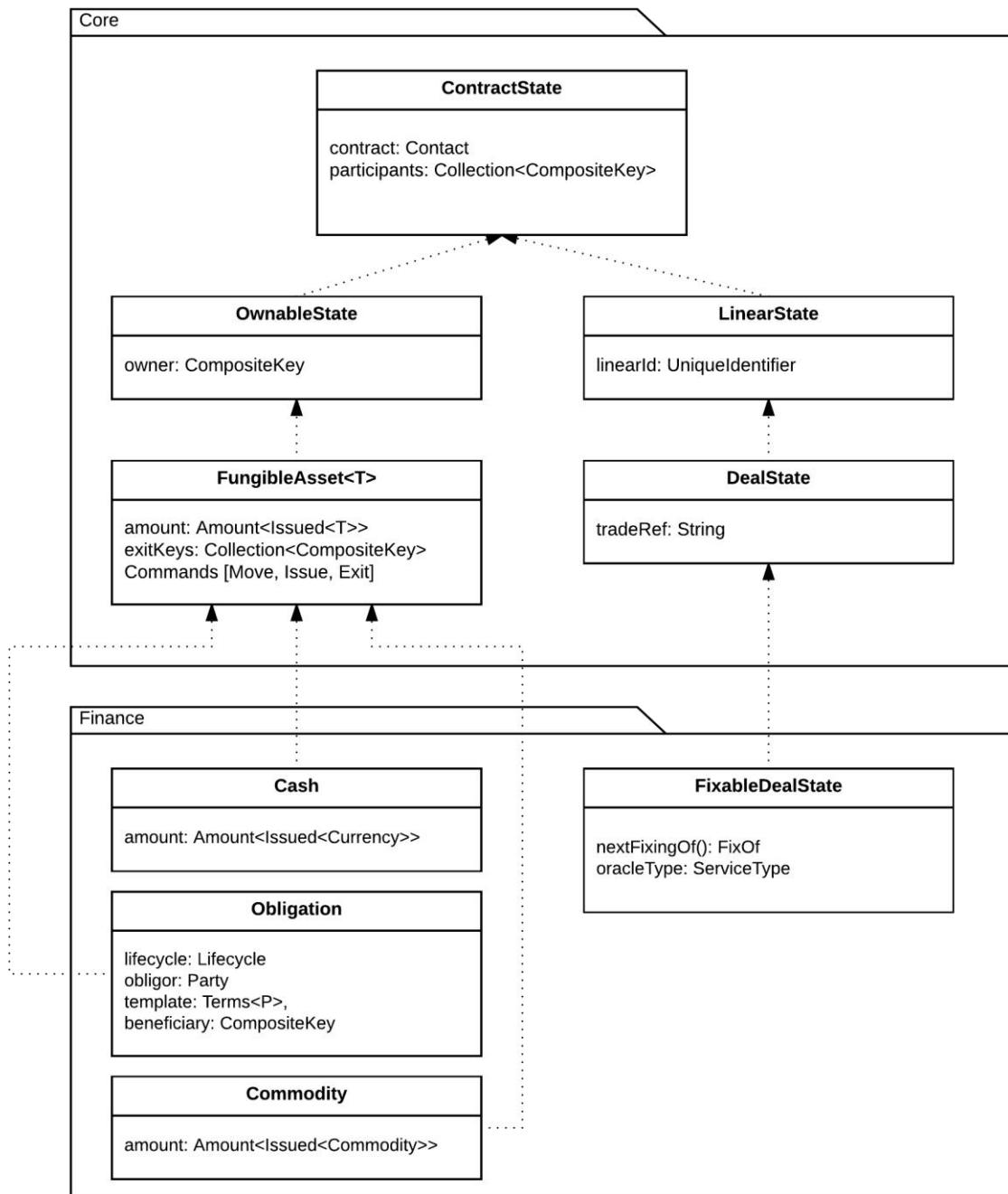
`DealState`

A `LinearState` representing an agreement between two or more parties. Intended to simplify implementing generic protocols that manipulate many agreement types.

`FungibleAsset`

A `FungibleAsset` is intended to be used for contract states representing assets which are fungible, countable and issued by a specific party. States contain assets which are equivalent (such as cash of the same currency), so records of their existence can be merged or split as needed where the issuer is the same. For instance, dollars issued by the Fed are fungible and countable (in cents), barrels of West Texas crude are fungible and countable (oil from two small containers can be poured into one large container), shares of the same class in a specific company are fungible and countable, and so on.

The following diagram illustrates the complete Contract State hierarchy:



Note there are currently two packages, a core library and a finance model specific library. Developers may re-use or extend the Finance types directly or write their own by extending the base types from the Core library.

[Next](#) [Previous](#)

- Interest rate swaps
- [View page source](#)

Interest rate swaps

The Interest Rate Swap (IRS) Contract (source: IRS.kt, IRSUtils.kt, IRSExport.kt) is a bilateral contract to implement a vanilla fixed / floating same currency IRS.

In general, an IRS allows two counterparties to modify their exposure from changes in the underlying interest rate. They are often used as a hedging instrument, convert a fixed rate loan to a floating rate loan, vice versa etc.

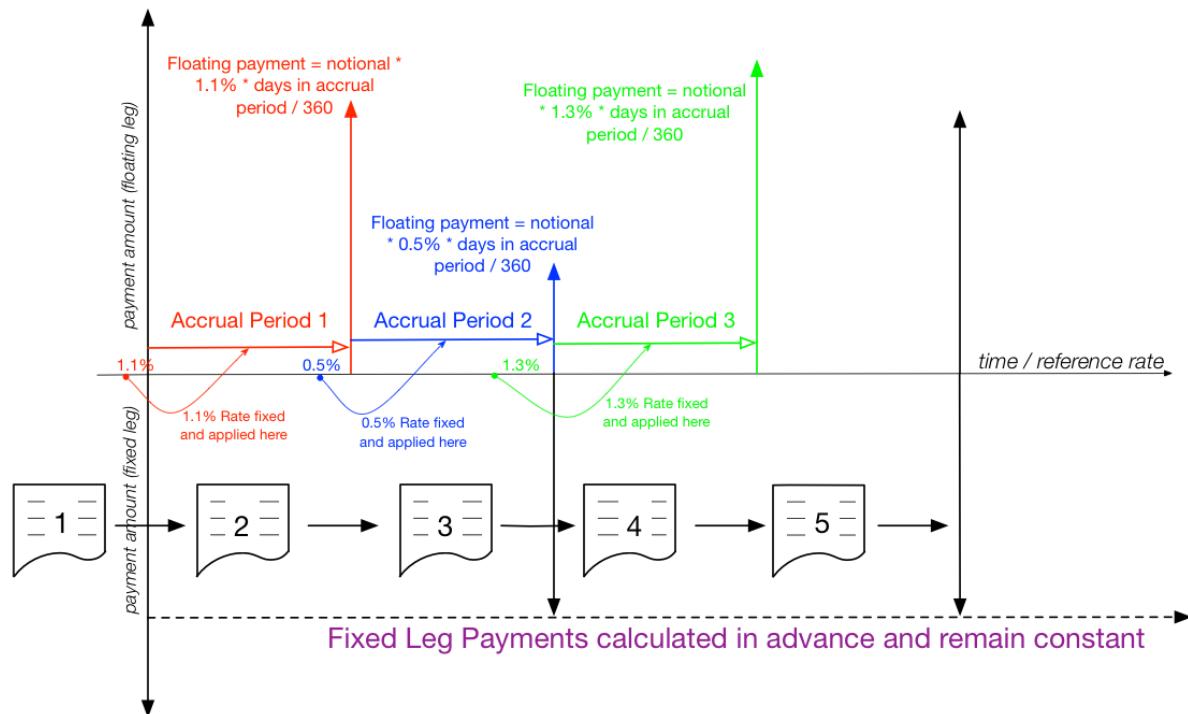
The IRS contract exists over a period of time (normally measurable in years). It starts on its value date (although this is not the agreement date), and is considered to be no longer active after its maturity date. During that time, there is an exchange of cash flows which are calculated by looking at the economics of each leg. These are based upon an amount that is not actually exchanged but notionally used for the calculation (and is hence known as the notional amount), and a rate that is either fixed at the creation of the swap (for the fixed leg), or based upon a reference rate that is retrieved during the swap (for the floating leg). An example reference rate might be something such as 'LIBOR 3M'.

The fixed leg has its rate computed and set in advance, whereas the floating leg has a fixing process whereas the rate for the next period is fixed with relation to a reference rate. Then, a calculation is performed such that the interest due over that period multiplied by the notional is paid (normally at the end of the period). If these two legs have the same payment date, then these flows can be offset against each other (in reality there are normally a number of these swaps that are live between two counterparties, so that further netting is performed at counterparty level).

The fixed leg and floating leg do not have to have the same period frequency. In fact, conventional swaps do not have the same period.

Currently, there is no notion of an actual payment or obligation being performed in the contract code we have written; it merely represents that the payment needs to be made.

Consider the diagram below; the x-axis represents time and the y-axis the size of the leg payments (not to scale), from the view of the floating leg receiver / fixed leg payer. The enumerated documents represent the versions of the IRS as it progresses (note that, the first version exists before the value date), the dots on the "y=0" represent an interest rate value becoming available and then the curved arrow indicates to which period the fixing applies.



Two days (by convention, although this can be modified) before the value date (i.e. at the start of the swap) in the red period, the reference rate is observed from an oracle and fixed - in this instance, at 1.1%. At the end of the accrual period, there is an obligation from the floating leg payer of $1.1\% * \text{notional amount} * \text{days in the accrual period} / 360$. (Also note that the result of "days in the accrual period / 360" is also known as the day count factor, although other conventions are allowed and will be supported). This amount is then paid at a determined time at the end of the accrual period.

Again, two working days before the blue period, the rate is fixed (this time at 0.5% - however in reality, the rates would not be so significantly different), and the same calculation is performed to evaluate the payment that will be due at the end of this period.

This process continues until the swap reaches maturity and the final payments are calculated.

Creating an instance and lifecycle

There are two valid operations on an IRS. The first is to generate via the `Agree` command (signed by both parties) and the second (and repeated operation) is `Fix` to apply a rate fixing. To see the minimum dataset required for the creation of an IRS, refer to `IRSTests.kt` which has two examples in the function `IRSTests.createDummyIRS()`. Implicitly, when the agree function is called, the floating leg and fixed leg payment schedules are created (more details below) and can be queried.

Once an IRS has been agreed, then the only valid operation is to apply a fixing on one of the entries in the `Calculation.floatingLegPaymentSchedule` map. Fixes do not have to be applied in order (although it does make most sense to do them so).

Examples of applying fixings to rates can be seen in `IRSTests.generateIRSandFixSome()` which loops through the next fixing date of an IRS that is created with the above example function and then applies a fixing of 0.052% to each floating event.

Currently, there are no matured, termination or dispute operations.

Technical details

The contract itself comprises of 4 data state classes, `FixedLeg`, `FloatingLeg`, `Common` and `Calculation`. Recall that the platform model is strictly immutable. To further that, between states, the only class that is modified is the `Calculation` class.

The `Common` data class contains all data that is general to the entire swap, e.g. data like trade identifier, valuation date, etc.

The Fixed and Floating leg classes derive from a common base class `CommonLeg`. This is due to the simple reason that they share a lot of common fields.

The `CommonLeg` class contains the notional amount, a payment frequency, the effective date (as well as an adjustment option), a termination date (and optional adjustment), the day count basis for day factor calculation, the payment delay and calendar for the payment as well as the accrual adjustment options.

The `FixedLeg` contains all the details for the `CommonLeg` as well as payer details, the rate the leg is fixed at and the date roll convention (i.e. what to do if the calculated date lands on a bank holiday or weekend).

The `FloatingLeg` contains all the details for the `CommonLeg` and payer details, roll convention, the fixing roll convention, which day of the month the reset is calculated, the frequency period of the fixing, the fixing calendar and the details for the reference index (source and tenor).

The `Calculation` class contains an expression (that can be evaluated via the ledger using variables provided and also any members of the contract) and two schedules - a `floatingLegPaymentSchedule` and a `fixedLegPaymentSchedule`. The fixed leg schedule is obviously pre-ordained, however, during the lifetime of the swap, the floating leg schedule is regenerated upon each fixing being presented.

For this reason, there are two helper functions on the floating leg. `Calculation.getFixing` returns the date of the earliest unset fixing, and `Calculation.applyFixing` returns a new Calculation object with the revised fixing in place. Note that both schedules are, for consistency, indexed by payment dates, but the fixing is (due to the convention of taking place two days previously) not going to be on that date.

Note

Payment events in the `floatingLegPaymentSchedule` that start as a `FloatingRatePaymentEvent` (which is a representation of a payment for a rate that has not yet been finalised) are replaced in their entirety with an equivalent `FixedRatePaymentEvent` (which is the same type that is on the `FixedLeg`).

[Next](#) [Previous](#)

- Contributing
- View page source

Contributing

Corda is an open-source project and contributions are welcome. Our contributing philosophy is described in [CONTRIBUTING.md](#). This guide explains the mechanics of contributing to Corda.

- How to contribute
- Testing Corda
- Code style guide
- Building the documentation

[Next](#) [Previous](#)

- How to contribute
 - [View page source](#)
-

How to contribute

Identifying an area to contribute

There are several ways to identify an area where you can contribute to Corda:

- Browse issues labelled as `good first issue` in the [Corda GitHub Issues](#)
 - Any issue with a `good first issue` label is considered ideal for open-source contributions
 - If there is a feature you would like to add and there isn't a corresponding issue labelled as `good first issue`, that doesn't mean your contribution isn't welcome. Please reach out on the `#design` channel to clarify (see below)
- Ask in the `#design` channel of the [Corda Slack](#)

Making the required changes

1. Create a fork of the master branch of the [Corda repo](#)
2. Clone the fork to your local machine
3. Build Corda by following the instructions here
4. Make the changes, in accordance with the [code style guide](#)

Things to check

Is your error handling up to scratch?

Errors should not leak to the UI. When writing tools intended for end users, like the node or command line tools, remember to add `try` / `catch` blocks. Throw meaningful errors. For example, instead of throwing an `OutOfMemoryError`, use the error message to indicate that a file is missing, a network socket was unreachable, etc. Tools should not dump stack traces to the end user.

Look for API breaks

We have an automated checker tool that runs as part of our continuous integration pipeline and helps a lot, but it can't catch semantic changes where the behavior of an API changes in ways that might violate app developer expectations.

Suppress inevitable compiler warnings

Compiler warnings should have a `@Suppress` annotation on them if they're expected and can't be avoided.

Remove deprecated functionality

When deprecating functionality, make sure you remove the deprecated uses in the codebase.

Avoid making formatting changes as you work

In Kotlin 1.2.20, new style guide rules were implemented. The new Kotlin style guide is significantly more detailed than before and IntelliJ knows how to implement those rules. Re-formatting the codebase creates a lot of diffs that make merging more complicated.

Things to consider when writing CLI apps

- Set exit codes using `exitProcess`. Zero means success. Other numbers mean errors. Setting a unique error code (starting from 1) for each thing that can conceivably break makes your tool shell-scripting friendly
- Do a bit of work to figure out reasonable defaults. Nobody likes having to set a dozen flags before the tool will cooperate

- Your `--help` text or other docs should ideally include examples. Writing examples is also a good way to find out that your program requires a dozen flags to do anything
- Flags should have sensible defaults
- Don't print logging output to the console unless the user requested it via a `-v` flag (conventionally shortened to `-v`) or a `-log-to-console` flag. Logs should be either suppressed or saved to a text file during normal usage, except for errors, which are always OK to print

Testing the changes

Adding tests

Unit tests and integration tests for external API changes must cover Java and Kotlin. For internal API changes these tests can be scaled back to kotlin only.

Running the tests

Your changes must pass the tests described [here](#).

Manual testing

Before sending that code for review, spend time poking and prodding the tool and thinking, "Would the experience of using this feature make my mum proud of me?". Automated tests are not a substitute for dogfooding.

Building against the master branch

You can test your changes against CorDapps defined in other repos by following the instructions [here](#).

Running the API scanner

Your changes must also not break compatibility with existing public API. We have an API scanning tool which runs as part of the build process which can be used to flag up any accidental changes, which is detailed [here](#).

Updating the docs

Any changes to Corda's public API must be documented as follows:

1. Add comments and javadocs/kdocs. API functions must have javadoc/kdoc comments and sentences must be terminated with a full stop. We also start comments with capital letters, even for inline comments. Where Java APIs have synonyms (e.g. `%d` and `%date`), we prefer the longer form for legibility reasons. You can configure your IDE to highlight these in bright yellow
2. Update the relevant [.rst file\(s\)](#)
3. Include the change in the `changelog` if the change is external and therefore visible to CorDapp developers and/or node operators
4. Build the docs locally
5. Check the built .html files (under `docs/build/html`) for the modified pages to ensure they render correctly
6. If relevant, add a sample. Samples are one of the key ways in which users learn about what the platform can do. If you add a new API or feature and don't update the samples, your work will be much less impactful

Merging the changes back into Corda

1. Create a pull request from your fork to the `master` branch of the Corda repo
2. In the PR comments box:
 - Complete the pull-request checklist:
 - [] Have you run the unit, integration and smoke tests as described here? <https://docs.corda.net/head/testing.html>
 - [] If you added/changed public APIs, did you write/update the JavaDocs?
 - [] If the changes are of interest to application developers, have you added them to the changelog, and potentially release notes?
 - [] If you are contributing for the first time, please read the agreement in `CONTRIBUTING.md` now and add to this Pull Request that you agree to it.
 - Add a clear description of the purpose of the PR
 - Add the following statement to confirm that your contribution is your own original work: “I hereby certify that my contribution is in accordance with the Developer Certificate of Origin (<https://github.com/corda/corda/blob/master/CONTRIBUTING.md#developer-certificate-of-origin>).”
4. Request a review from a member of the Corda platform team via the [#design channel](#)

5. The reviewer will either:

- Accept and merge your PR
- Request that you make further changes. Do this by committing and pushing the changes onto the branch you are PRing into Corda. The PR will be updated automatically

[Next](#) [Previous](#)

- Testing Corda
 - [View page source](#)
-

Testing Corda

Automated Tests

Corda has a maintained suite of tests that any contributing developers must maintain and add to if new code has been added.

There are several distinct test suites each with a different purpose;

Unit tests: These are traditional unit tests that should only test a single code unit, typically a method or class.

Integration tests: These tests should test the integration of small numbers of units, preferably with mocked out services.

Smoke tests: These are full end to end tests which start a full set of Corda nodes and verify broader behaviour.

Other: These include tests such as performance tests, stress tests, etc, and may be in an external repo.

These tests are mostly written with JUnit and can be run via `gradle`. On windows run `gradlew test integrationTest smokeTest` and on unix
run `./gradlew test integrationTest smokeTest` or any combination of these three arguments.

Before creating a pull request please make sure these pass.

Manual Testing

Manual testing would ideally be done for every set of changes merged into master, but practically you should manually test anything that would be impacted by your changes. The areas that usually need to be manually tested and when are below;

Node startup - changes in the `node` or `node:capstone` project in both the Kotlin or gradle or the `cordformation` gradle plugin.

Sample project - changes in the `samples` project. eg; changing the IRS demo means you should manually test the IRS demo.

Explorer - changes to the `tools/explorer` project.

Demobench - changes to the `tools/demobench` project.

How to manually test each of these areas differs and is currently not fully specified. For now the best thing to do is ensure the program starts, that you can interact with it, and that no exceptions are generated in normal operation.

TODO: Add instructions on manual testing

[Next](#) [Previous](#)

- Code style guide
 - [View page source](#)
-

Code style guide

This document explains the coding style used in the Corda repository. You will be expected to follow these recommendations when submitting patches for review. Please take the time to read them and internalise them, to save time during code review.

What follows are *recommendations* and not *rules*. They are in places intentionally vague, so use your good judgement when interpreting them.

Note

Parts of the codebase may not follow this style guide yet. If you see a place that doesn't, please fix it!

1. General style

We use the standard Java coding style from Sun, adapted for Kotlin in ways that should be fairly intuitive.

Files no longer have copyright notices at the top, and license is now specified in the global README.md file. We do not mark classes with @author Javadoc annotations.

In Kotlin code, KDoc is used rather than JavaDoc. It's very similar except it uses Markdown for formatting instead of HTML tags.

We target Java 8 and use the latest Java APIs whenever convenient. We use `java.time.Instant` to represent timestamps and `java.nio.file.Path` to represent file paths.

Never apply any design pattern religiously. There are no silver bullets in programming and if something is fashionable, that doesn't mean it's always better. In particular:

- Use functional programming patterns like map, filter, fold only where it's genuinely more convenient. Never be afraid to use a simple imperative construct like a for loop or a mutable counter if that results in more direct, English-like code.
- Use immutability when you don't anticipate very rapid or complex changes to the content. Immutability can help avoid bugs, but over-used it can make code that has to adjust fields of an immutable object (in a clone) hard to read and stress the garbage collector. When such code becomes a widespread pattern it can lead to code that is just generically slow but without hotspots.
- The tradeoffs between various thread safety techniques are complex, subtle, and no technique is always superior to the others. Our code uses a mix of locks, worker threads and messaging depending on the situation.

1.1 Line Length and Spacing

We aim for line widths of no more than 120 characters. That is wide enough to avoid lots of pointless wrapping but narrow enough that with a widescreen monitor and a 12 point fixed width font (like Menlo) you can fit two files next to each other. This is not a rigidly enforced rule and if wrapping a line would be excessively awkward, let it overflow. Overflow of a few characters here and there isn't a big deal: the goal is general convenience.

Where the number of parameters in a function, class, etc. causes an overflow past the end of the first line, they should be structured one parameter per line.

Code is vertically dense, blank lines in methods are used sparingly. This is so more code can fit on screen at once.

We use spaces and not tabs.

1.2 Naming

Naming generally follows Java standard style (pascal case for class names, camel case for methods, properties and variables). Where a class name describes a tuple, “And” should be included in order to clearly indicate the elements are individual parts, for example `PartyAndReference`, not `PartyReference` (which sounds like a reference to a `Party`).

2. Comments

We like them as long as they add detail that is missing from the code. Comments that simply repeat the story already told by the code are best deleted. Comments should:

- Explain what the code is doing at a higher level than is obtainable from just examining the statement and surrounding code.
- Explain why certain choices were made and the tradeoffs considered.
- Explain how things can go wrong, which is a detail often not easily seen just by reading the code.
- Use good grammar with capital letters and full stops. This gets us in the right frame of mind for writing real explanations of things.

When writing code, imagine that you have an intelligent colleague looking over your shoulder asking you questions as you go. Think about what they might ask, and then put your answers in the code.

Don't be afraid of redundancy, many people will start reading your code in the middle with little or no idea of what it's about (e.g. due to a bug or a need to introduce a new feature). It's OK to repeat basic facts or descriptions in different places if that increases the chance developers will see something important.

API docs: all public methods, constants and classes should have doc comments in either JavaDoc or KDoc. API docs should:

- Explain what the method does in words different to how the code describes it.
- Always have some text, annotation-only JavaDocs don't render well. Write "Returns a blah blah blah" rather than "@returns blah blah blah" if that's the only content (or leave it out if you have nothing more to say than the code already says).
- Illustrate with examples when you might want to use the method or class. Point the user at alternatives if this code is not always right.
- Make good use of `{@link}` annotations.

Bad JavaDocs look like this:

```
/** @return the size of the Bloom filter. */
public int getBloomFilterSize() {
    return block;
}
```

Good JavaDocs look like this:

```
/**
 * Returns the size of the current {@Link BloomFilter} in bytes. Larger filters have
 * lower false positive rates for the same number of inserted keys and thus lower
 * privacy,
 * but bandwidth usage is also correspondingly reduced.
 */
public int getBloomFilterSize() { ... }
```

We use C-style (`/** */`) comments for API docs and we use C++ style comments (`//`) for explanations that are only intended to be viewed by people who read the code. When writing multi-line TODO comments, indent the body text past the TODO line, for example:

```
// TODO: Something something
//           More stuff to do
```

// Etc. etc.

3. Threading

Classes that are thread safe should be annotated with the `@ThreadSafe` annotation. The class or method comments should describe how threads are expected to interact with your code, unless it's obvious because the class is (for example) a simple immutable data holder.

Code that supports callbacks or event listeners should always accept an `Executor` argument that defaults to `MoreExecutors.directThreadExecutor()` (i.e. the calling thread) when registering the callback. This makes it easy to integrate the callbacks with whatever threading environment the calling code expects, e.g. serialised onto a single worker thread if necessary, or run directly on the background threads used by the class if the callback is thread safe and doesn't care in what context it's invoked.

In the prototyping code it's OK to use synchronised methods i.e. with an exposed lock when the use of locking is quite trivial. If the synchronisation in your code is getting more complex, consider the following:

1. Is the complexity necessary? At this early stage, don't worry too much about performance or scalability, as we're exploring the design space rather than making an optimal implementation of a design that's already nailed down.
2. Could you simplify it by making the data be owned by a dedicated, encapsulated worker thread? If so, remember to think about flow control and what happens if a work queue fills up: the actor model can often be useful but be aware of the downsides and try to avoid explicitly defining messages, prefer to send closures onto the worker thread instead.
3. If you use an explicit lock and the locking gets complex, and *always* if the class supports callbacks, use the cycle detecting locks from the Guava library.
4. Can you simplify some things by using thread-safe collections like `CopyOnWriteArrayList` or `ConcurrentHashMap`? These data structures are more expensive than their non-thread-safe equivalents but can be worth it if it lets us simplify the code.

Immutable data structures can be very useful for making it easier to reason about multi-threaded code. Kotlin makes it easy to define these via the "data" attribute,

which auto-generates a copy() method. That lets you create clones of an immutable object with arbitrary fields adjusted in the clone. But if you can't use the data attribute for some reason, for instance, you are working in Java or because you need an inheritance hierarchy, then consider that making a class fully immutable may result in very awkward code if there's ever a need to make complex changes to it. If in doubt, ask. Remember, never apply any design pattern religiously.

We have an extension to the `Executor` interface called `AffinityExecutor`. It is useful when the thread safety of a piece of code is based on expecting to be called from a single thread only (or potentially, a single thread pool). `AffinityExecutor` has additional methods that allow for thread assertions. These can be useful to ensure code is not accidentally being used in a multi-threaded way when it didn't expect that.

4. Assertions and errors

We use them liberally and we use them at runtime, in production. That means we avoid the "assert" keyword in Java, and instead prefer to use the `check()` or `require()` functions in Kotlin (for an `IllegalStateException` or `IllegalArgumentException` respectively), or the Guava `Preconditions.check` method from Java.

We define new exception types liberally. We prefer not to provide English language error messages in exceptions at the throw site, instead we define new types with any useful information as fields, with a `toString()` method if really necessary. In other words, don't do this:

```
throw new Exception("The foo broke")
```

instead do this

```
class FooBrokenException extends Exception {}  
throw new FooBrokenException()
```

The latter is easier to catch and handle if later necessary, and the type name should explain what went wrong.

Note that Kotlin does not require exception types to be declared in method prototypes like Java does.

5. Properties

Where we want a public property to have one super-type in public and another sub-type in private (or internal), perhaps to expose additional methods with a greater level of access to the code within the enclosing class, the style should be:

```
class PrivateFoo : PublicFoo

private val _foo = PrivateFoo()
val foo: PublicFoo get() = _foo
```

Notably:

- The public property should have an explicit and more restrictive type, most likely a super class or interface.
- The private, backed property should begin with underscore but otherwise have the same name as the public property. The underscore resolves a potential property name clash, and avoids naming such as “privateFoo”. If the type or use of the private property is different enough that there is no naming collision, prefer the distinct names without an underscore.
- The underscore prefix is not a general pattern for private properties.
- The public property should not have an additional backing field but use “get()” to return an appropriate copy of the private field.
- The public property should optionally wrap the returned value in an immutable wrapper, such as Guava’s immutable collection wrappers, if that is appropriate.
- If the code following “get()” is succinct, prefer a one-liner formatting of the public property as above, otherwise put the “get()” on the line below, indented.

6. Compiler warnings

We do not allow compiler warnings, except in the experimental module where the usual standards do not apply and warnings are suppressed. If a warning exists it should be either fixed or suppressed using `@SuppressWarnings` and if suppressed there must be an accompanying explanation in the code for why the warning is a false positive.

[Next](#) [Previous](#)

- Building the documentation
 - [View page source](#)
-

Building the documentation

The documentation is under the `docs` folder, and is written in reStructuredText format. Documentation in HTML format is pre-generated, as well as code documentation, and this can be done automatically via a provided script.

Requirements

To build the documentation, you will need:

- GNU Make
- Python and pip (tested with Python 2.7.10)
- Sphinx: <http://www.sphinx-doc.org/>
- sphinx_rtd_theme: https://github.com/snide/sphinx_rtd_theme

Note that to install under OS X El Capitan, you will need to tell pip to install under `/usr/local`, which can be done by specifying the installation target on the command line:

```
sudo -H pip install --install-option '--install-data=/usr/local' Sphinx
sudo -H pip install --install-option '--install-data=/usr/local' sphinx_rtd_theme
```

Build

Once the requirements are installed, you can automatically build the HTML format user documentation and the API documentation by running the following script:

```
./gradlew buildDocs
```

Alternatively you can build non-HTML formats from the `docs` folder. Change directory to the folder and then run the following to see a list of all available formats:

```
make
```

For example to produce the documentation in HTML format:

```
make html
```

[Next](#) [Previous](#)

- [Release process](#)
 - [View page source](#)
-

Release process

- [Release notes](#)
- [Changelog](#)

[Next](#) [Previous](#)

- [Release notes](#)
 - [View page source](#)
-

Release notes

Release 3.3

Corda 3.3 brings together many small improvements, fixes, and community contributions to deliver a stable and polished release of Corda. Where both the 3.1 and 3.2 releases delivered a smaller number of critical bug fixes addressing immediate and impactful error conditions, 3.3 addresses a much greater number of issues, both small and large, that have been found and fixed since the release of 3.0 back in March. Rolling up a great many improvements and polish to truly make the Corda experience just that much better.

In addition to work undertaken by the main Corda development team, we've taken the opportunity in 3.3 to bring back many of the contributions made by community members from master onto the currently released stable branch. It has been said many times before, but the community and its members are the real life-blood of Corda and anyone who takes the time to contribute is a star in our eyes. Bringing that code into the current version we hope gives people the opportunity to see their work in action, and to help their fellow community members by having these contributions available in a supported release.

Changes of Note

- **Serialization fixes**

Things “in the lab” always work so much better than they do in the wild, where everything you didn’t think of is thrown at your code and a mockery is made of some dearly held assumptions. A great example of this is the serialization framework which delivers Corda’s wire stability guarantee that was introduced in 3.0 and has subsequently been put to a rigorous test by our users. Corda 3.3 consolidates a great many fixes in that framework, both programmatically in terms of fixing bugs, but also in the documentation, hopefully making things clearer and easier to work with.

- **Certificate Hierarchy**

After consultation, collaboration, and discussion with industry experts, we have decided to alter the default Certificate Hierarchy (PKI) utilized by Corda and the Corda Network. To facilitate this, the nodes have had their certificate path verification logic made much more flexible. All existing certificate hierarchy, certificates, and networks will remain valid. The possibility now exists for nodes to recognize a deeper certificate chain and thus Compatibility Zone operators can deploy and adhere to the PKI standards they expect and are comfortable with.

Practically speaking, the old code assumed a 3-level hierarchy of Root -> Intermediate CA (Doorman) -> Node, and this was hard coded. From 3.3 onward an arbitrary depth of certificate chain is supported. For the Corda Network, this means the introduction of an intermediate layer between the root and the signing certificates (Network Map and Doorman). This has the effect of allowing the root certificate to *always* be kept offline and never retrieved or used. Those new intermediate certificates can be used to generate, if ever needed, new signing certs without risking compromise of the root key.

Special Thanks

The Corda community is a vibrant and exciting ecosystem that spreads far outside the virtual walls of the R3 organisation. Without that community, and the most welcome contributions of its members, the Corda project would be a much poorer place.

We're therefore happy to extend thanks to the following members of that community for their contributions

- [Dan Newton](#) for a fix to cleanup node registration in the test framework. The changes can be found [here](#).
- [Tushar Singh Bora](#) for a number of [documentation tweaks](#). In addition, some updates to the tutorial documentation [here](#).
- [Jiachuan Li](#) for a number of corrections to our documentation. Those contributions can be found [here](#) and [here](#).
- [Yogesh](#) for a documentation tweak that can be see [here](#).
- [Roman Plášil](#) for speeding up node shutdown when connecting to an http network map. This fix can be found [here](#).
- [renlulu](#) for a small [PR](#) to optimize some of the imports.
- [cxyzhang0](#) for making the [IdentitySyncFlow](#) more useful. See [here](#).
- [Venelin Stoykov](#) with updates to the [documentation](#) around the progress tracker.
- [Mohamed Amine Legheraba](#) for updates to the Azure documentation that can be seen [here](#).
- [Stanly Johnson](#) with a [fix](#) to the network bootstrapper.
- [Tittu Varghese](#) for adding a favicon to the docsite. This commit can be found [here](#)

Issues Fixed

- Refactoring [DigitalSignatureWithCertPath](#) for more performant storing of the certificate chain. [[CORDA-1995](#)]
- The serializers class carpenter fails when superclass has double-size primitive field. [[Corda-1945](#)]
- If a second identity is mistakenly created the node will not start. [[CORDA-1811](#)]
- Demobench profile load fails with stack dump. [[CORDA-1948](#)]
- Deletes of NodeInfo can fail to propagate leading to infinite retries. [[CORDA-2029](#)]
- Copy all the certificates from the network-trust-store.jks file to the node's trust store. [[CORDA-2012](#)]
- Add SNI (Server Name Indication) header to TLS connections. [[CORDA-2001](#)]
- Fix duplicate index declaration in the Cash schema. [[CORDA-1952](#)]

- Hello World Tutorial Page mismatch between code sample and explanatory text. [CORDA-1950]
- Java Instructions to Invoke Hello World CorDapp are incorrect. [CORDA-1949]
- Add `VersionInfo` to the `NodeInfo` submission request to the network map element of the Compatibility Zone. [CORDA-1938]
- Rename current INTERMEDIATE_CA certificate role to DOORMAN_CA certificate role. [CORDA-1934]
- Make node-side network map verification agnostic to the certificate hierarchy. [CORDA-1932]
- Corda Shell incorrectly deserializes generic types as raw types. [CORDA-1907]
- The Corda web server does not support asynchronous servlets. [CORDA-1906]
- Amount<T> is deserialized from JSON and YAML as Amount<Currency>, for all values of T. [CORDA-1905]
- `NodeVaultService.loadStates` queries without a `PageSpecification` property set. This leads to issues with large transactions. [CORDA-1895]
- If a node has two flows, where one's name is a longer version of the other's, they cannot be started [CORDA-1892]
- Vault Queries across `LinearStates` and `FungibleState` tables return incorrect results. [CORDA-1888]
- Checking the version of the Corda jar file by executing the jar with the `--version` flag without specifying a valid node configuration file causes an exception to be thrown. [CORDA-1884]
- RPC deadlocks after a node restart. [CORDA-1875]
- Vault query fails to find a state if it extends some class (`ContractState`) and it is that base class that is used as the predicate (`vaultService.queryBy<I>()`). [CORDA-1858]
- Missing unconsumed states from linear id when querying vault caused by a the previous transaction failing with an SQL exception. [CORDA-1847]
- Inconsistency in how a web path is written. [CORDA-1841]
- Cannot use `TestIdentities` with same organization name in `net.corda.testing.driver.Driver`. [CORDA-1837]
- Docs page typos. [CORDA-1834]
- Adding flexibility to the serialization frameworks unit tests support and utility code. [CORDA-1808]

- Cannot use `--initial-registration` with the `networkServices` configuration option in place of the older `compatibilityzone` option within `node.conf`.
[\[CORDA-1789\]](#)
- Document more clearly the supported version of both IntelliJ and the IntelliJ Kotlin Plugins.
[\[CORDA-1727\]](#)
- DemoBench's "Launch Explorer" button is not re-enabled when you close Node Explorer.
[\[CORDA-1686\]](#)
- It is not possible to run `stateMachinesSnapshot` from the shell.
[\[CORDA-1681\]](#)
- Node won't start if CorDapps generate states prior to deletion
[\[CORDA-1663\]](#)
- Serializer Evolution breaks with Java classes adding nullable properties.
[\[CORDA-1662\]](#)
- Add Java examples for the creation of proxy serializers to complement the existing kotlin ones.
[\[CORDA-1641\]](#)
- Proxy serializer documentation isn't clear on how to write a proxy serializer.
[\[CORDA-1640\]](#)
- Node crashes in `--initial-registration` polling mode if doorman returns a transient HTTP error.
[\[CORDA-1638\]](#)
- Nodes started by gradle task are not stopped when the gradle task exits.
[\[CORDA-1634\]](#)
- Notarizations time out if notary doesn't have up-to-date network map.
[\[CORDA-1628\]](#)
- Node explorer: Improve error handling when connection to nodes cannot be established.
[\[CORDA-1617\]](#)
- Validating notary fails to resolve an attachment.
[\[CORDA-1588\]](#)
- Out of process nodes started by the driver do not log to file.
[\[CORDA-1575\]](#)
- Once `--initial-registration` has been passed to a node, further restarts should assume that mode until a cert is collected.
[\[CORDA-1572\]](#)
- An array of primitive byte arrays (an array of arrays) won't deserialize in a virgin factory (i.e. one that didn't build the serializer for serialization).
[\[CORDA-1545\]](#)
- Ctrl-C in the shell fails to aborts the flow.
[\[CORDA-1542\]](#)
- One transaction with two identical cash outputs cannot be save in the vault.
[\[CORDA-1535\]](#)
- The unit tests for the enum evolver functionality cannot be regenerated. This is because verification logic added after their initial creation has a bug that incorrectly identifies a cycle in the graph.
[\[CORDA-1498\]](#)
- Add in a safety check that catches flow checkpoints from older versions.
[\[CORDA-1477\]](#)

- Buggy `CommodityContract` issuance logic. [CORDA-1459]
- Error in the process-id deletion process allows multiple instances of the same node to be run. [CORDA-1455]
- Node crashes if network map returns HTTP 50X error. [CORDA-1414]
- Delegate Property doesn't serialize, throws an erroneous type mismatch error. [CORDA-1403]
- If a vault query throws an exception, the stack trace is swallowed. [CORDA-1397]
- Node can fail to fully start when a port conflict occurs, no useful error message is generated when this occurs. [CORDA-1394]
- Running the `deployNodes` gradle task back to back without a clean doesn't work. [CORDA-1389]
- Stripping issuer from Amount<Issued<T>> does not preserve `displayTokenSize`. [CORDA-1386]
- `CordaServices` are instantiated multiple times per Party when using `NodeDriver`. [CORDA-1385]
- Out of memory errors can be seen when using Demobench + Explorer. [CORDA-1356]
- Reduce the amount of classpath scanning during integration tests execution. [CORDA-1355]
- SIMM demo throws “attachment too big” errors. [CORDA-1346]
- Fix vault query paging example in `ScheduledFlowTests`. [CORDA-1344]
- The shell doesn't print the return value of a started flow. [CORDA-1342]
- Provide access to database transactions for CorDapp developers. [CORDA-1341]
- Error with `VaultQuery` for entity inheriting from `CommonSchemaV1.FungibleState`. [CORDA-1338]
- The `--network-root-truststore` command line option not defaulted. [CORDA-1317]
- Java example in “Upgrading CorDapps” documentation is wrong. [CORDA-1315]
- Remove references to `registerInitiatedFlow` in testing documentation as it is not needed. [CORDA-1304]
- Regression: Recording a duplicate transaction attempts second insert to vault. [CORDA-1303]
- Columns in the Corda database schema should have correct NULL/NOT NULL constraints. [CORDA-1297]

- MockNetwork/Node API needs a way to register `@CordaService` objects. [\[CORDA-1292\]](#)
- Deleting a `NodeInfo` from the additional-node-infos directory should remove it from cache. [\[CORDA-1093\]](#)
- `FailNodeOnNotMigratedAttachmentContractsTableNameTests` is sometimes failing with database constraint “Notary” is null. [\[CORDA-1976\]](#)
- Revert keys for DEV certificates. [\[CORDA-1661\]](#)
- Node Info file watcher should block and load `NodeInfo` when node startup. [\[CORDA-1604\]](#)
- Improved logging of the network parameters update process. [\[CORDA-1405\]](#)
- Ensure all conditions in cash selection query are tested. [\[CORDA-1266\]](#)
- `NodeVaultService` bug. Start node, issue cash, stop node, start node, `getCashBalances()` will not show any cash
- A Corda node doesn't re-select cluster from HA Notary.
- Event Horizon is not wire compatible with older network parameters objects.
- Notary unable to resolve Party after processing a flow from same Party.
- Misleading error message shown when a node is restarted after a flag day event.

Release 3.2

As we see more Corda deployments in production this minor release of the open source platform brings several fixes that make it easier for a node to join Corda networks broader than those used when operating as part of an internal testing deployment. This will ensure Corda nodes will be free to interact with upcoming network offerings from R3 and others who may make broad-access Corda networks available.

- **The Corda Network Builder**

To make it easier to create more dynamic, flexible, networks for testing and deployment, with the 3.2 release of Corda we are shipping a graphical network bootstrapping tool (see [Corda Network Builder](#)) to facilitate the simple creation of more dynamic ad hoc dev-mode environments.

Using a graphical interface you can dynamically create and alter Corda test networks, adding nodes and CorDapps with the click of a button! Additionally, you can leverage its integration with Azure cloud services for remote hosting of Nodes and Docker instances for local testing.

- **Split Compatibility Zone**

Prior to this release Compatibility Zone membership was denoted with a single configuration setting

```
compatibilityZoneURL : "http://<host>(:<port>)"
```

That would indicate both the location of the Doorman service the node should use for registration of its identity as well as the Network Map service where it would publish its signed Node Info and retrieve the Network Map.

Compatibility Zones can now, however, be configured with the two disparate services, Doorman and Network Map, running on different URLs. If the Compatibility Zone your node is connecting to is configured in this manner, the new configuration looks as follows.

```
networkServices {  
    doormanURL: "http://<host>(:<port>)"  
    networkMapURL: "http://<host>(:<port>)"  
}
```

Note

The `compatibilityZoneURL` setting should be considered deprecated in favour of the new `networkServices` settings group.

- **The Blob Inspector**

The blob inspector brings the ability to unpack serialized Corda blobs at the command line, giving a human readable interpretation of the encoded date.

Note

This tool has been shipped as a separate Jar previously. We are now including it as part of an official release.

Documentation on its use can be found here [Blob Inspector](#)

- **The Event Horizon**

One part of joining a node to a Corda network is agreeing to the rules that govern that network as set out by the network operator. A node's membership of a network is communicated to other nodes through the network map, the service to which the node will have published its Node Info, and through which it

receives the set of NodeInfos currently present on the network. Membership of that list is a finite thing determined by the network operator.

Periodically a node will republish its NodeInfo to the Network Map service. The Network Map uses this as a heartbeat to determine the status of nodes registered with it. Those that don't "beep" within the determined interval are removed from the list of registered nodes. The `Event Horizon` network parameter sets the upper limit within which a node must respond or be considered inactive.

Important

This does not mean a node is unregistered from the Doorman, only that its NodeInfo is removed from the Network Map. Should the node come back online it will be re-added to the published set of NodeInfos

Issues Fixed

- Update Jolokia to a more secure version [[CORDA-1744](#)]
- Add the Blob Inspector [[CORDA-1709](#)]
- Add support for the `Event Horizon` Network Parameter [[CORDA-866](#)]
- Add the Network Bootstrapper [[CORDA-1717](#)]
- Fixes for the finance CordApp [[CORDA-1711](#)]
- Allow Doorman and NetworkMap to be configured independently [[CORDA-1510](#)]
- Serialization fix for generics when evolving a class [[CORDA-1530](#)]
- Correct typo in an internal database table name [[CORDA-1499](#)] and [[CORDA-1804](#)]
- Hibernate session not flushed before handing over raw JDBC session to user code [[CORDA-1548](#)]
- Fix Postgres db bloat issue [[CORDA-1812](#)]
- Roll back flow transaction on exception [[CORDA-1790](#)]

Release 3.1

This rapid follow-up to Corda 3.0 corrects an issue discovered by some users of Spring Boot and a number of other smaller issues discovered post release. All users are recommended to upgrade.

Special Thanks

Without passionate and engaged users Corda would be all the poorer. As such, we are extremely grateful to [Bret Lichtenwald](#) for helping nail down a reproducible test case for the Spring Boot issue.

Major Bug Fixes

- **Corda Serialization fails with “Unknown constant pool tag”**

This issue is most often seen when running a CorDapp with a Rest API using / provided by [Spring Boot](#).

The fundamental cause was [Corda 3.0](#) shipping with an out of date dependency for the [fast-classpath-scanner](#) library, where the manifesting bug was already fixed in a released version newer than our dependant one. In response, we've updated our dependent version to one including that bug fix.

- **Corda Versioning**

Those eagle eyed amongst you will have noticed for the 3.0 release we altered the versioning scheme from that used by previous Corda releases (1.0.0, 2.0.0, etc) with the addition of an prepended product name, resulting in [corda-3.0](#). The reason for this was so that developers could clearly distinguish between the base open source platform and any distributions based on on Corda that may be shipped in the future (including from R3). However, we have heard the complaints and feel the pain that's caused by various tools not coping well with this change. As such, from now on the versioning scheme will be inverted, with this release being [3.1-corda](#).

As to those curious as to why we dropped the patch number from the version string, the reason is very simple: there won't be any patches applied to a release of Corda. Either a release will be a collection of bug fixes and non API breaking changes, thus eliciting a minor version bump as with this release, or major functional changes or API additions and warrant a major version bump. Thus, rather than leave a dangling [.0](#) patch version on every release we've just dropped it. In the case where a major security flaw needed addressing, for example, then that would generate a release of a new minor version.

Issues Fixed

- RPC server leaks if a single client submits a lot of requests over time
[\[CORDA-1295\]](#)

- Flaky startup, no db transaction in context, when using postgresql [[CORDA-1276](#)]
- Corda's JPA classes should not be final or have final methods [[CORDA-1267](#)]
- Backport api-scanner changes [[CORDA-1178](#)]
- Misleading error message shown when node is restarted after the flag day
- Hash constraints not working from Corda 3.0 onwards
- Serialisation Error between Corda 3 RC01 and Corda 3
- Nodes don't start when network-map/doorman is down

Release 3.0

Corda 3.0 is here and brings with it a commitment to a wire stable platform, a path for contract and node upgradability, and a host of other exciting features. The aim of which is to enhance the developer and user experience whilst providing for the long term usability of deployed Corda instances. This release will provide functionality to ensure anyone wishing to move to the anticipated release of R3 Corda can do so seamlessly and with the assurance that stateful data persisted to the vault will remain understandable between newer and older nodes.

Special Thanks

As ever, we are grateful to the enthusiastic user and developer community that has grown up to surround Corda. As an open project we are always grateful to take code contributions from individual users where they feel they can add functionality useful to themselves and the wider community.

As such we'd like to extend special thanks to

- Ben Wyeth for providing a mechanism for registering a callback on app shutdown
Ben's contribution can be found on GitHub [here](#)
- Tomas Tauber for adding support for running Corda atop PostgreSQL in place of the in-memory H2 service
Tomas's contribution can be found on GitHub [here](#)

Warning

This is an experimental feature that has not been tested as part of our standard release testing.

- Rose Molina Atienza for correcting our careless spelling slip

Rose's change can be found on GitHub [here](#)

Significant Changes in 3.0

- **Wire Stability:**

Wire stability brings the same promise to developers for their data that API stability did for their code. From this point any state generated by a Corda system will always be retrievable, understandable, and seen as valid by any subsequently released version (versions 3.0 and above).

Systems can thus be deployed safe in the knowledge that valuable and important information will always be accessible through upgrade and change. Practically speaking this means from this point forward upgrading all, or part, of a Corda network will not require the replaying of data; “it will just work”.

This has been facilitated by the switch over from Kryo to Corda’s own AMQP based serialization framework, a framework designed to interoperate with stateful information and allow the evolution of such contract states over time as developers refine and improve their systems written atop the core Corda platform.

- **AMQP Serialization**

AMQP Serialization is now enabled for both peer to peer communication and the writing of states to the vault. This change brings a serialisation format that will allow us to deliver enhanced security and wire stability.

This was a key prerequisite to enabling different Corda node versions to coexist on the same network and to enable easier upgrades.

Details on the AMQP serialization framework can be found [here](#). This provides an introduction and overview of the framework whilst more specific details on object evolution as it relates to serialization can be found in [Default Class Evolution](#) and [Enum Evolution](#) respectively.

Note

This release delivers the bulk of our transition from Kryo serialisation to AMQP serialisation. This means that many of the restrictions that were documented in previous versions of Corda are now enforced.

In particular, you are advised to review the section titled Custom Types. To aid with the transition, we have included support in this release for default construction and instantiation of objects with inaccessible private fields, but it is not guaranteed that this support will continue into future versions; the restrictions documented at the link above are the canonical source.

Whilst this is an important step for Corda, in no way is this the end of the serialisation story. We have many new features and tools planned for future releases, but feel it is more important to deliver the guarantees discussed above as early as possible to allow the community to develop with greater confidence.

Important

Whilst Corda has stabilised its wire protocol and infrastructure for peer to peer communication and persistent storage of states, the RPC framework will, for this release, not be covered by this guarantee. The moving of the client and server contexts away from Kryo to our stable AMQP implementation is planned for the next release of Corda

- **Artemis and Bridges**

Corda has now achieved the long stated goal of using the AMQP 1.0 open protocol standard as its communication protocol between peers. This forms a strong and flexible framework upon which we can deliver future enhancements that will allow for much smoother integrations between Corda and third party brokers, languages, and messaging systems. In addition, this is also an important step towards formally defining the official peer to peer messaging protocol of Corda, something required for more in-depth security audits of the Corda protocol.

- **New Network Map Service:**

This release introduces the new network map architecture. The network map service has been completely redesigned and implemented to enable future increased network scalability and redundancy, reduced runtime operational overhead, support for multiple notaries, and administration of network compatibility zones (CZ).

A Corda Compatibility Zone is defined as a grouping of participants and services (notaries, oracles, doorman, network map server) configured within an operational Corda network to be interoperable and compatible with each other.

We introduce the concept of network parameters to specify precisely the set of constants (or ranges of constants) upon which the nodes within a network need to agree in order to be assured of seamless inter-operation. Additional security controls ensure that all network map data is now signed, thus reducing the power of the network operator to tamper with the map.

There is also support for a group of nodes to operate locally, which is achieved by copying each node's signed info file to the other nodes' directories. We've added a bootstrapping tool to facilitate this use case.

Important

This replaces the Network Map service that was present in Corda 1.0 and Corda 2.0.

Further information can be found in the [Changelog](#), [Network Map](#) and [setting-up-a-corda-network](#) documentation.

- **Contract Upgrade**

Support for the upgrading of contracts has been significantly extended in this release.

Contract states express which attached JARs can define and verify them using `_constraints_`. In older versions the only supported constraint was a hash constraint. This provides similar behaviour as public blockchain systems like Bitcoin and Ethereum, in which code is entirely fixed once deployed and cannot be changed later. In Corda there is an upgrade path that involves the cooperation of all involved parties (as advertised by the states themselves), but this requires explicit transactions to be applied to all states and be signed by all parties.

Tip

This is a fairly heavyweight operation. As such, consideration should be given as to the most opportune time at which it should be performed.

Hash constraints provide for maximum decentralisation and minimum trust, at the cost of flexibility. In Corda 3.0 we add a new constraint, a `_network parameters_` constraint, that allows the list of acceptable contract

JARs to be maintained by the operator of the compatibility zone rather than being hard-coded. This allows for simple upgrades at the cost of the introduction of an element of centralisation.

Zone constraints provide a less restrictive but more centralised control mechanism. This can be useful when you want the ability to upgrade an app and you don't mind the upgrade taking effect "just in time" when a transaction happens to be required for other business reasons. These allow you to specify that the network parameters of a compatibility zone (see [Network Map](#)) is expected to contain a map of class name to hashes of JARs that are allowed to provide that class. The process for upgrading an app then involves asking the zone operator to add the hash of your new JAR to the parameters file, and trigger the network parameters upgrade process. This involves each node operator running a shell command to accept the new parameters file and then restarting the node. Node owners who do not restart their node in time effectively stop being a part of the network.

Note

In prior versions of Corda, states included the hash of their defining application JAR (in the Hash Constraint). In this release, transactions have the JAR containing the contract and states attached to them, so the code will be copied over the network to the recipient if that peer lacks a copy of the app.

Prior to running the verification code of a contract the JAR within which the verification code of the contract resides is tested for compliance to the contract constraints:

- For the `HashConstraint`: the hash of the deployed CorDapp jar must be the same as the hash found in the Transaction.
- For the `ZoneConstraint`: the Transaction must come with a whitelisted attachment for each Contract State.

If this step fails the normal transaction verification failure path is followed.

Corda 3.0 lays the groundwork for future releases, when contract verification will be done against the attached contract JARs rather than requiring a locally deployed CorDapp of the exact version specified by the transaction. The future vision for this feature will entail the dynamic downloading of the

appropriate version of the smart contract and its execution within a sandboxed environment.

Warning

This change means that your app JAR must now fit inside the 10mb attachment size limit. To avoid redundantly copying unneeded code over the network and to simplify upgrades, consider splitting your application into two or more JARs - one that contains states and contracts (which we call the app "kernel"), and another that contains flows, services, web apps etc. For example, our [Cordapp template](#) is structured like that. Only the first will be attached. Also be aware that any dependencies your app kernel has must be bundled into a fat JAR, as JAR dependencies are not supported in Corda 3.0.

Future versions of Corda will add support for signature based constraints, in which any JAR signed by a given identity can be attached to the transaction. This final constraint type provides a balance of all requirements: smooth rolling upgrades can be performed without any additional steps or transactions being signed, at the cost of trusting the app developer more and some additional complexity around managing app signing.

Please see the [Upgrading a CorDapp \(outside of platform version upgrades\)](#) for more information on upgrading contracts.

- **Test API Stability**

A great deal of work has been carried out to refine the APIs provided to test CorDapps, making them simpler, more intuitive, and generally easier to use. In addition, these APIs have been added to the *locked* list of the APIs we guarantee to be stable over time. This should greatly increase productivity when upgrading between versions, as your testing environments will work without alteration.

Please see the [Upgrading a CorDapp to a new platform version](#) for more information on transitioning older tests to the new framework.

Other Functional Improvements

- **Clean Node Shutdown**

We, alongside user feedback, concluded there was a strong need for the ability to have a clean inflection point where a node could be shutdown without any in-flight transactions pending to allow for a clean system for upgrade purposes. As such, a flows draining mode has been added. When

activated, this places the node into a state of quiescence that guarantees no new work will be started and all outstanding work completed prior to shutdown.

A clean shutdown can thus be achieved by:

1. Subscribing to state machine updates
2. Trigger flows draining mode by `rpc.setFlowsDrainingModeEnabled(true)`
3. Wait until the subscription setup as phase 1 lets you know that no more checkpoints are around
4. Shut the node down however you want

Note

Once set, this mode is a persistent property that will be preserved across node restarts. It must be explicitly disabled before a node will accept new RPC flow connections.

- **X.509 certificates**

These now have an extension that specifies the Corda role the certificate is used for, and the role hierarchy is now enforced in the validation code. This only has impact on those developing integrations with external PKI solutions; in most cases it is managed transparently by Corda. A formal specification of the extension can be found at see permissioning-certificate-specification.

- **Configurable authorization and authentication data sources**

Corda can now be configured to load RPC user credentials and permissions from an external database and supports password encryption based on the [Apache Shiro framework](#). See [RPC security management](#) for documentation.

- **SSH Server**

Remote administration of Corda nodes through the CRaSH shell is now available via SSH, please see [Shell](#) for more details.

- **RPC over SSL**

Corda now allows for the configuration of its RPC calls to be made over SSL. See [Node configuration](#) for details how to configure this.

- **Improved Notary configuration**

The configuration of notaries has been simplified into a single `notary` configuration object. See [Node configuration](#) for more details.

Note

`extraAdvertisedServiceIds`, `notaryNodeAddress`, `notaryClusterAddresses` and `bftSMaRt` configs have been removed.

- **Database Tables Naming Scheme**

To align with common conventions across all supported Corda and R3 Corda databases some table names have been changed.

In addition, for existing contract ORM schemas that extend from `CommonSchemaV1.LinearState` or `CommonSchemaV1.FungibleState`, you will need to explicitly map the participants collection to a database table. Previously this mapping was done in the superclass, but that makes it impossible to properly configure the table name. The required change is to add the override var `participants: MutableSet<AbstractParty>? = null` field to your class, and add JPA mappings.

- **Pluggable Custom Serializers**

With the introduction of AMQP we have introduced the requirement that to be seamlessly serializable classes, specifically Java classes (as opposed to Kotlin), must be compiled with the `-parameter` flag. However, we recognise that this isn't always possible, especially dealing with third party libraries in tightly controlled business environments.

To work around this problem as simply as possible CorDapps now support the creation of pluggable proxy serializers for such classes. These should be written such that they create an intermediary representation that Corda can serialise that is mappable directly to and from the unserializable class.

A number of examples are provided by the SIMM Valuation Demo in

`samples/simm-valuation-`
`demo/src/main/kotlin/net/corda/vega/plugin/customserializers`

Documentation can be found in [Pluggable Serializers for CorDapps](#)

Security Auditing

This version of Corda is the first to have had select components subjected to the newly established security review process by R3's internal security team.

Security review will be an on-going process that seeks to provide assurance that the security model of Corda has been implemented to the highest standard, and is in line with industry best practice.

As part of this security review process, an independent external security audit of the HTTP based components of the code was undertaken and its recommendations were acted upon. The security assurance process will develop in parallel to the Corda platform and will combine code review, automated security testing and secure development practices to ensure Corda fulfils its security guarantees.

Security fixes

- Due to a potential privacy leak, there has been a breaking change in the error object returned by the notary service when trying to consume the same state twice: *NotaryError.Conflict* no longer contains the identity of the party that initiated the first spend of the state, and specifies the hash of the consuming transaction id for a state instead of the id itself.

Without this change, knowing the reference of a particular state, an attacker could construct an invalid double-spend transaction, and obtain the information on the transaction and the party that consumed it. It could repeat this process with the newly obtained transaction id by guessing its output indexes to obtain the forward transaction graph with associated identities. When anonymous identities are used, this could also reveal the identity of the owner of an asset.

Minor Changes

- Upgraded gradle to 4.4.1.

Note

To avoid potential incompatibility issues we recommend you also upgrade your CorDapp's gradle plugin to match. Details on how to do this can be found on the official [gradle website](#)

- Cash Spending now allows for sending multiple amounts to multiple parties with a single API call
 - documentation can be found within the JavaDocs on [TwoPartyTradeFlow](#).
- Overall improvements to error handling (RPC, Flows, Network Client).
- TLS authentication now supports mixed RSA and ECDSA keys.
- PrivacySalt computation is faster as it does not depend on the OS's entropy pool directly.
- Numerous bug fixes and documentation tweaks.

- Removed dependency on Jolokia WAR file.

Release 2.0

Following quickly on the heels of the release of Corda 1.0, Corda version 2.0 consolidates a number of security updates for our dependent libraries alongside the reintroduction of the Observer node functionality. This was absent from version 1 but based on user feedback its re-introduction removes the need for complicated “isRelevant()” checks.

In addition the fix for a small bug present in the coin selection code of V1.0 is integrated from master.

- **Version Bump**

Due to the introduction of new APIs, Corda 2.0 has a platform version of 2. This will be advertised in the network map structures and via the versioning APIs.

- **Observer Nodes**

Adds the facility for transparent forwarding of transactions to some third party observer, such as a regulator. By having that entity simply run an Observer node they can simply receive a stream of digitally signed, de-duplicated reports that can be used for reporting.

Release 1.0

Corda 1.0 is finally here!

This critical step in the Corda journey enables the developer community, clients, and partners to build on Corda with confidence. Corda 1.0 is the first released version to provide API stability for Corda application (CorDapp) developers. Corda applications will continue to work against this API with each subsequent release of Corda. The public API for Corda will only evolve to include new features.

As of Corda 1.0, the following modules export public APIs for which we guarantee to maintain backwards compatibility, unless an incompatible change is required for security reasons:

- **core**: Contains the bulk of the APIs to be used for building CorDapps: contracts, transactions, flows, identity, node services, cryptographic libraries, and general utility functions.
- **client-rpc**: An RPC client interface to Corda, for use by both UI facing clients and integration with external systems.
- **client-jackson**: Utilities and serialisers for working with JSON representations of basic types.

Our extensive testing frameworks will continue to evolve alongside future Corda APIs. As part of our commitment to ease of use and modularity we have introduced a new test node driver module to encapsulate all test functionality in support of building standalone node integration tests using our DSL driver.

Please read [Corda API](#) for complete details.

Note

it may be necessary to recompile applications against future versions of the API until we begin offering [ABI \(Application Binary Interface\)](#) stability as well. We plan to do this soon after this release of Corda.

Significant changes implemented in reaching Corda API stability include:

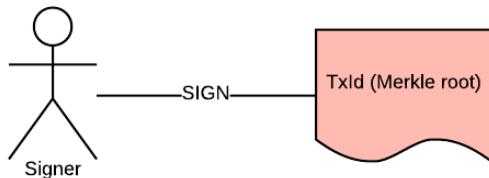
- **Flow framework**: The Flow framework communications API has been redesigned around session based communication with the introduction of a new `FlowSession` to encapsulate the counterparty information associated with a flow. All shipped Corda flows have been upgraded to use the new `FlowSession`. Please read [API: Flows](#) for complete details.
- **Complete API cleanup**: Across the board, all our public interfaces have been thoroughly revised and updated to ensure a productive and intuitive developer experience. Methods and flow naming conventions have been aligned with their semantic use to ease the understanding of CorDapps. In addition, we provide ever more powerful re-usable flows (such as `CollectSignaturesFlow`) to minimize the boiler-plate code developers need to write.
- **Simplified annotation driven scanning**: CorDapp configuration has been made simpler through the removal of explicit configuration items in favour of annotations and classpath scanning. As an example, we have now completely removed the `CordaPluginRegistry` configuration. Contract definitions are no longer required to explicitly define a legal contract reference hash. In their place an optional `LegalProseReference` annotation to specify a URL is used.

- **Java usability:** All code has been updated to enable simple access to static API parameters. Developers no longer need to call getter methods, and can reference static API variables directly.

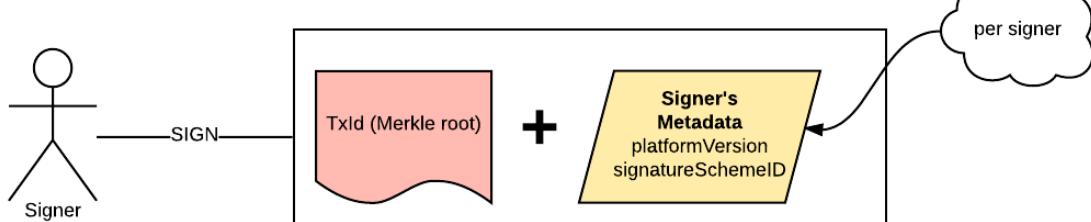
In addition to API stability this release encompasses a number of major functional improvements, including:

- **Contract constraints:** Provides a means with which to enforce a specific implementation of a State's verify method during transaction verification. When loading an attachment via the attachment classloader, constraints of a transaction state are checked against the list of attachment hashes provided, and the attachment is rejected if the constraints are not matched.
- **Signature Metadata support:** Signers now have the ability to add metadata to their digital signatures. Whereas previously a user could only sign the Merkle root of a transaction, it is now possible for extra information to be attached to a signature, such as a platform version and the signature-scheme used.

Previous Model (M14 and before)



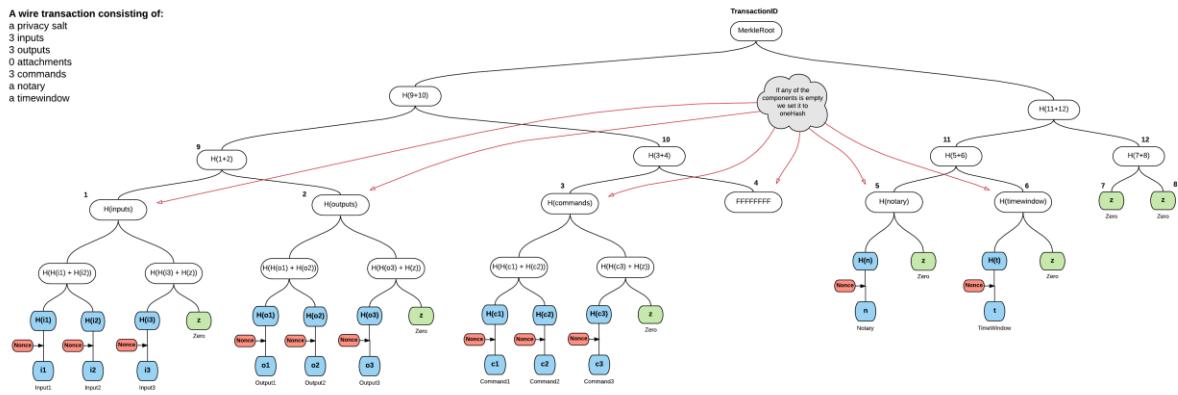
Signature Metadata Support (v1.0)



- **Backwards compatibility and improvements to core transaction data structures:** A new Merkle tree model has been introduced that utilises sub-Merkle trees per component type. Components of the same type, such as inputs or commands, are grouped together and form their own Merkle tree. Then, the roots of each group are used as leaves in the top-level Merkle tree. This model enables backwards compatibility, in the sense that if new

component types are added in the future, old clients will still be able to compute the Merkle root and relay transactions even if they cannot read (deserialise) the new component types. Due to the above, *FilterTransaction* has been made simpler with a structure closer to *WireTransaction*. This has the effect of making the API more user friendly and intuitive for both filtered and unfiltered transactions.

- **Enhanced component privacy:** Corda 1.0 is equipped with a scalable component visibility design based on the above sophisticated sub-tree model and the introduction of nonces per component. Roughly, an initial base-nonce, the “privacy-salt”, is used to deterministically generate nonces based on the path of each component in the tree. Because each component is accompanied by a nonce, we protect against brute force attacks, even against low-entropy components. In addition, a new privacy feature is provided that allows non-validating notaries to ensure they see all inputs and if there was a *TimeWindow* in the original transaction. Due to the above, a malicious user cannot selectively hide one or more input states from the notary that would enable her to bypass the double-spending check. The aforementioned functionality could also be applied to Oracles so as to ensure all of the commands are visible to them.



- **Full support for confidential identities:** This includes rework and improvements to the identity service to handle both *well known* and *confidential* identities. This work ships in an experimental module in Corda 1.0, called *confidential-identities*. API stabilisation of confidential identities will occur as we make the integration of this privacy feature into applications even easier for developers.
- **Re-designed network map service:** The foundations for a completely redesigned network map service have been implemented to enable future

increased network scalability and redundancy, support for multiple notaries, and administration of network compatibility zones and business networks. Finally, please note that the 1.0 release has not yet been security audited.

We have provided a comprehensive [Upgrading a CorDapp to a new platform version](#) to ease the transition of migrating CorDapps to Corda 1.0

Upgrading to this release is strongly recommended, and you will be safe in the knowledge that core APIs will no longer break.

Thank you to all contributors for this release!

Milestone 14

This release continues with the goal to improve API stability and developer friendliness. There have also been more bug fixes and other improvements across the board.

The CorDapp template repository has been replaced with a specific repository for [Java](#) and [Kotlin](#) to improve the experience of starting a new project and to simplify the build system.

It is now possible to specify multiple IP addresses and legal identities for a single node, allowing node operators more flexibility in setting up nodes.

A format has been introduced for CorDapp JARs that standardises the contents of CorDapps across nodes. This new format now requires CorDapps to contain their own external dependencies. This paves the way for significantly improved dependency management for CorDapps with the release of [Jigsaw \(Java Modules\)](#). For those using non-gradle build systems it is important to read [Building a CorDapp](#) to learn more. Those using our [cordformation](#) plugin simply need to update to the latest version ([0.14.0](#)) to get the fixes.

We've now begun the process of demarcating which classes are part of our public API and which ones are internal. Everything found in `net.corda.core.internal` and other packages in the `net.corda` namespace which has `.internal` in it are considered internal and not for public use. In a future

release any CorDapp using these packages will fail to load, and when we migrate to Jigsaw these will not be exported.

The transaction finalisation flow (`FinalityFlow`) has had hooks added for alternative implementations, for example in scenarios where no single participant in a transaction is aware of the well known identities of all parties.

DemoBench has a fix for a rare but inconvenient crash that can occur when sharing your display across multiple devices, e.g. a projector while performing demonstrations in front of an audience.

Guava types are being removed because Guava does not have backwards compatibility across versions, which has serious issues when multiple libraries depend on different versions of the library.

The identity service API has been tweaked, primarily so anonymous identity registration now takes in `AnonymousPartyAndPath` rather than the individual components of the identity, as typically the caller will have an `AnonymousPartyAndPath` instance. See change log for further detail.

Upgrading to this release is strongly recommended in order to keep up with the API changes, removal and additions.

Milestone 13

Following our first public beta in M12, this release continues the work on API stability and user friendliness. Apart from bug fixes and code refactoring, there are also significant improvements in the Vault Query and the Identity Service (for more detailed information about what has changed, see [Changelog](#)). More specifically:

The long awaited new **Vault Query** service makes its debut in this release and

provides advanced vault query capabilities using criteria specifications (see `QueryCriteria`), sorting, and pagination. Criteria specifications enable selective filtering with and/or composition using multiple operator primitives on standard attributes stored in Corda internal vault tables (eg. `vault_states`, `vault_fungible_states`, `vault_linear_states`), and also on custom contract state

schemas defined by CorDapp developers when modelling new contract types. Custom queries are specifiable using a simple but sophisticated builder DSL (see [QueryCriteriaUtils](#)). The new Vault Query service is usable by flows and by RPC clients alike via two simple API functions: `queryBy()` and `trackBy()`. The former provides point-in-time snapshot queries whilst the later supplements the snapshot with dynamic streaming of updates. See [API: Vault Query](#) for full details.

We have written a comprehensive Hello, World! tutorial, showing developers how to build a CorDapp from start to finish. The tutorial shows how the core elements of a CorDapp - states, contracts and flows - fit together to allow your node to handle new business processes. It also explains how you can use our contract and flow testing frameworks to massively reduce CorDapp development time.

Certificate checks have been enabled for much of the identity service. These are part of the confidential (anonymous) identities work, and ensure that parties are actually who they claim to be by checking their certificate path back to the network trust root (certificate authority).

To deal with anonymized keys, we've also implemented a deterministic key

derivation function that combines logic from the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) protocol and the BIP32 hardened parent-private-key -> child-private-key scheme. This function currently supports the following algorithms: ECDSA secp256K1, ECDSA secpR1 (NIST P-256) and EdDSA ed25519. We are now very close to fully supporting anonymous identities so as to increase privacy even against validating notaries.

We have further tightened the set of objects which Corda will attempt to serialise from the stack during flow checkpointing. As flows are arbitrary code in which it is convenient to do many things, we ended up pulling in a lot of objects that didn't make sense to put in a checkpoint, such as `Thread` and `Connection`. To minimize serialization cost and increase security by not allowing certain classes to be serialized, we now support class blacklisting that will return an `IllegalStateException` if such a class is encountered during a checkpoint. Blacklisting supports superclass and superinterface inheritance and always precedes `@CordaSerializable` annotation checking.

We've also started working on improving user experience when searching, by adding a new RPC to support fuzzy matching of X.500 names.

Milestone 12 - First Public Beta

One of our busiest releases, lots of changes that take us closer to API stability (for more detailed information about what has changed, see [Changelog](#)). In this release we focused mainly on making developers' lives easier. Taking into account feedback from numerous training courses and meet-ups, we decided to add `CollectSignaturesFlow` which factors out a lot of code which CorDapp developers needed to write to get their transactions signed. The improvement is up to 150 fewer lines of code in each flow! To have your transaction signed by different parties, you need only now call a subflow which collects the parties' signatures for you.

Additionally we introduced classpath scanning to wire-up flows automatically. Writing CorDapps has been made simpler by removing boiler-plate code that was previously required when registering flows. Writing services such as oracles has also been simplified.

We made substantial RPC performance improvements (please note that this is separate to node performance, we are focusing on that area in future milestones):

- 15-30k requests per second for a single client/server RPC connection. * 1Kb requests, 1Kb responses, server and client on same machine, parallelism 8, measured on a Dell XPS 17(i7-6700HQ, 16Gb RAM)
- The framework is now multithreaded on both client and server side.
- All remaining bottlenecks are in the messaging layer.

Security of the key management service has been improved by removing support for extracting private keys, in order that it can support use of a hardware security module (HSM) for key storage. Instead it exposes functionality for signing data (typically transactions). The service now also supports multiple signature schemes (not just EdDSA).

We've added the beginnings of flow versioning. Nodes now reject flow requests if the initiating side is not using the same flow version. In a future milestone release will add the ability to support backwards compatibility.

As with the previous few releases we have continued work extending identity support. There are major changes to the `Party` class as part of confidential identities, and how parties and keys are stored in transaction state objects. See [Changelog](#) for full details.

Added new Byzantine fault tolerant (BFT) decentralised notary demo, based on the [BFT-SMaRT protocol](#). For how to run the demo see: [notary-demo](#)

We continued to work on tools that enable diagnostics on the node. The newest addition to Corda Shell is `flow watch` command which lets the administrator see all flows currently running with result or error information as well as who is the flow initiator. Here is the view from DemoBench:

The screenshot shows the Corda DemoBench application window. At the top, there are tabs for 'File', 'corda' (selected), 'Notary', 'Bank of Breakfast Tea', and 'Bank of Big Apples'. On the right, there are buttons for 'Add Node', 'View Database', 'Launch Web Server', and 'Launch Explorer'. Below these, the 'Bank of Big Apples' tab is selected, displaying its status: 'States in vault: 0', 'Known transactions: 6', and 'Balance: 0.00 CHF'. To the right of this status are three buttons: 'View Database', 'Launch Web Server', and 'Launch Explorer'. A large table below lists active flows, their initiators, and their statuses. The table has columns for 'Id', 'Flow name', 'Initiator', and 'Status'. Some rows show successful transactions with Tx IDs and return values, while others show errors like 'No return value' or 'Exiting more cash than exists'.

Id	Flow name	Initiator	Status
b4f826fd-e0d9-4d45-98b5-	Issuance Requester	RPC: guest	Tx ID: 6CDE84DAE514A20142E5EFOC94A7325A40E189870C
3d810667-396f-4b3d-9906-	Issuer	Bank of Big Apples	Tx ID: 6CDE84DAE514A20142E5EFOC94A7325A40E189870C
98235684-9a05-4f45-8490-	Fetch Transactions Handler	Bank of Breakfast Tea	No return value
caa1d2a-0e43-46ba-9900-	Issuance Requester	RPC: guest	Tx ID: 4205613B2692D66967A7223200F71081401D9F1
4be425a3-4249-42e9-b5e2-	Issuer	Bank of Big Apples	Tx ID: 4205613B2692D66967A7223200F71081401D9F1
ae9058a5-6c66-42fe-bee9-	Fetch Transactions Handler	Bank of Breakfast Tea	No return value
7472e23b-531f-4487-8a79-	Issuance Requester	RPC: guest	Tx ID: C0E613029415C4C180539C0D6B9AC66EB0C111A33D
de5aa3d7-dae2-4a45-80e2-	Issuer	Bank of Big Apples	Tx ID: C0E613029415C4C180539C0D6B9AC66EB0C111A33D
Bfb6056e-dc30-4da2-b639-	Fetch Transactions Handler	Bank of Breakfast Tea	No return value
04a4ac88-517e-471f-b9f9-	Cash Exit	RPC: guest	Exiting more cash than exists
Waiting for completion or Ctrl-C ...			

We also started work on the strategic wire format (not integrated).

Milestone 11

Special thank you to [Gary Rowe](#) for his contribution to Corda's Contracts DSL in M11.

Work has continued on confidential identities, introducing code to enable the Java standard libraries to work with composite key signatures. This will form the underlying basis of future work to standardise the public key and signature formats to enable interoperability with other systems, as well as enabling the use of composite signatures on X.509 certificates to prove association between transaction keys and identity keys.

The identity work will require changes to existing code and configurations, to replace party names with full X.500 distinguished names (see RFC 1779 for details on the construction of distinguished names). Currently this is not enforced, however it will be in a later milestone.

- “myLegalName” in node configurations will need to be replaced, for example “Bank A” is replaced with “CN=Bank A,O=Bank A,L=London,C=GB”. Obviously organisation, location and country (“O”, “L” and “C” respectively) must be given values which are appropriate to the node, do not just use these example values.
- “networkMap” in node configurations must be updated to match any change to the legal name of the network map.
- If you are using mock parties for testing, try to standardise on the `DUMMY_NOTARY`, `DUMMY_BANK_A`, etc. provided in order to ensure consistency.

We anticipate enforcing the use of distinguished names in node configurations from M12, and across the network from M13.

We have increased the maximum message size that we can send to Corda over RPC from 100 KB to 10 MB.

The Corda node now disables any use of `ObjectInputStream` to prevent Java deserialisation within flows. This is a security fix, and prevents the node from deserialising arbitrary objects.

We've introduced the concept of platform version which is a single integer value which increments by 1 if a release changes any of the public APIs of the entire Corda platform. This includes the node's public APIs, the messaging protocol, serialisation, etc. The node exposes the platform version it's on and we envision CorDapps will use this to be able to run on older versions of the platform to the

one they were compiled against. Platform version borrows heavily from Android's API Level.

We have revamped the DemoBench user interface. DemoBench will now also be installed as "Corda DemoBench" for both Windows and MacOSX. The original version was installed as just "DemoBench", and so will not be overwritten automatically by the new version.

Milestone 10

Special thank you to [Qian Hong](#), [Marek Skocovsky](#), [Karel Hajek](#), and [Jonny Chiu](#) for their contributions to Corda in M10.

A new interactive **Corda Shell** has been added to the node. The shell lets developers and node administrators easily command the node by running flows, RPCs and SQL queries. It also provides a variety of commands to monitor the node. The Corda Shell is based on the popular [CRaSH project](#) and new commands can be easily added to the node by simply dropping Groovy or Java files into the node's `shell-commands` directory. We have many enhancements planned over time including SSH access, more commands and better tab completion.

The new "DemoBench" makes it easy to configure and launch local Corda nodes. It is a standalone desktop app that can be bundled with its own JRE and packaged as either EXE (Windows), DMG (MacOS) or RPM (Linux-based). It has the following features:

1. New nodes can be added at the click of a button. Clicking "Add node" creates a new tab that lets you edit the most important configuration properties of the node before launch, such as its legal name and which CorDapps will be loaded.
2. Each tab contains a terminal emulator, attached to the pseudoterminal of the node. This lets you see console output.
3. You can launch an Corda Explorer instance for each node at the click of a button. Credentials are handed to the Corda Explorer so it starts out logged in already.
4. Some basic statistics are shown about each node, informed via the RPC connection.

5. Another button launches a database viewer in the system browser.
6. The configurations of all running nodes can be saved into a single `.profile` file that can be reloaded later.

Soft Locking is a new feature implemented in the vault to prevent a node constructing transactions that attempt to use the same input(s) simultaneously. Such transactions would result in naturally wasted effort when the notary rejects them as double spend attempts. Soft locks are automatically applied to coin selection (eg. cash spending) to ensure that no two transactions attempt to spend the same fungible states.

The basic Amount API has been upgraded to have support for advanced financial use cases and to better integrate with currency reference data.

We have added optional out-of-process transaction verification. Any number of external verifier processes may be attached to the node which can handle loadbalanced verification requests.

We have also delivered the long waited Kotlin 1.1 upgrade in M10! The new features in Kotlin allow us to write even more clean and easy to manage code, which greatly increases our productivity.

This release contains a large number of improvements, new features, library upgrades and bug fixes. For a full list of changes please see [Changelog](#).

Milestone 9

This release focuses on improvements to resiliency of the core infrastructure, with highlights including a Byzantine fault tolerant (BFT) decentralised notary, based on the BFT-SMaRT protocol and isolating the web server from the Corda node.

With thanks to open source contributor Thomas Schroeter for providing the BFT notary prototype, Corda can now resist malicious attacks by members of a distributed notary service. If your notary service cluster has seven members, two can become hacked or malicious simultaneously and the system continues unaffected! This work is still in development stage, and more features are coming in the next snapshot!

The web server has been split out of the Corda node as part of our ongoing hardening of the node. We now provide a Jetty servlet container pre-configured to contact a Corda node as a backend service out of the box, which means individual webapps can have their REST APIs configured for the specific security environment of that app without affecting the others, and without exposing the sensitive core of the node to malicious Javascript.

We have launched a global training programme, with two days of classes from the R3 team being hosted in London, New York and Singapore. R3 members get 5 free places and seats are going fast, so sign up today.

We've started on support for confidential identities, based on the key randomisation techniques pioneered by the Bitcoin and Ethereum communities. Identities may be either anonymous when a transaction is a part of a chain of custody, or fully legally verified when a transaction is with a counterparty. Type safety is used to ensure the verification level of a party is always clear and avoid mistakes. Future work will add support for generating new identity keys and providing a certificate path to show ownership by the well known identity.

There are even more privacy improvements when a non-validating notary is used; the Merkle tree algorithm is used to hide parts of the transaction that a non-validating notary doesn't need to see, whilst still allowing the decentralised notary service to sign the entire transaction.

The serialisation API has been simplified and improved. Developers now only need to tag types that will be placed in smart contracts or sent between parties with a single annotation... and sometimes even that isn't necessary!

Better permissioning in the cash CorDapp, to allow node users to be granted different permissions depending on whether they manage the issuance, movement or ledger exit of cash tokens.

We've continued to improve error handling in flows, with information about errors being fed through to observing RPC clients.

There have also been dozens of bug fixes, performance improvements and usability tweaks. Upgrading is definitely worthwhile and will only take a few minutes for most apps.

For a full list of changes please see [Changelog](#).

[Next](#) [Previous](#)

- [Changelog](#)
 - [View page source](#)
-

Changelog

Here's a summary of what's changed in each Corda release. For guidance on how to upgrade code from the previous release, see [Upgrading a CorDapp to a new platform version](#).

Version 3.3

- Vault query fix: support query by parent classes of Contract State classes (see <https://github.com/corda/corda/issues/3714>)
- Fixed an issue preventing Shell from returning control to the user when CTRL+C is pressed in the terminal.
- Fixed a problem that sometimes prevented nodes from starting in presence of custom state types in the database without a corresponding type from installed CorDapps.
- Introduced a grace period before the initial node registration fails if the node cannot connect to the Doorman. It retries 10 times with a 1 minute interval in between each try. At the moment this is not configurable.
- Fixed an error thrown by NodeVaultService upon recording a transaction with a number of inputs greater than the default page size.
- Changes to the JSON/YAML serialisation format from [JacksonSupport](#), which also applies to the node shell:
 - [Instant](#) and [Date](#) objects are serialised as ISO-8601 formatted strings rather than timestamps
 - [PublicKey](#) objects are serialised and looked up according to their Base58 encoded string
 - [Party](#) objects can be deserialised by looking up their public key, in addition to their name
 - [NodeInfo](#) objects are serialised as an object and can be looked up using the same mechanism as [Party](#)

- `NetworkHostAndPort` serialised according to its `toString()`
 - `PartyAndCertificate` is serialised as the name
 - `SerializedBytes` is serialised by materialising the bytes into the object it represents, and then serialising that object into YAML/JSON
 - `X509Certificate` is serialised as an object with key fields such as `issuer`, `publicKey`, `serialNumber`, etc. The encoded bytes are also serialised into the `encoded` field. This can be used to deserialise an `X509Certificate` back.
 - `CertPath` objects are serialised as a list of `X509Certificate` objects.
- `fullParties` boolean parameter added to `JacksonSupport.createDefaultMapper` and `createNonRpcMapper`. If `true` then `Party` objects are serialised as JSON objects with the `name` and `owningKey` fields. For `PartyAndCertificate` the `certPath` is serialised.
- Several members of `JacksonSupport` have been deprecated to highlight that they are internal and not to be used
- `ServiceHub` and `CordaRPCOps` can now safely be used from multiple threads without incurring in database transaction problems.
- Fixed an issue preventing out of process nodes started by the `Driver` from logging to file.
- The Vault Criteria API has been extended to take a more precise specification of which class contains a field. This primarily impacts Java users; Kotlin users need take no action. The old methods have been deprecated but still work - the new methods avoid bugs that can occur when JPA schemas inherit from each other.
- Removed `-xmx` VM argument from Explorer's Capsule setup. This helps avoiding out of memory errors.
- Node will now gracefully fail to start if one of the required ports is already in use.
- Fixed incorrect exception handling in `NodeVaultService._query()`.
- Avoided a memory leak deriving from incorrect MappedSchema caching strategy.
- Fix CORDA-1403 where a property of a class that implemented a generic interface could not be deserialised in a factory without a serialiser as the subtype check for the class instance failed. Fix is to compare the raw type.
- Fix CORDA-1229. Setter-based serialization was broken with generic types when the property was stored as the raw type, List for example.

- Table name with a typo changed from `NODE_ATTACHMENT_CONTRACTS` to `NODE_ATTACHMENTS_CONTRACTS`.

Version 3.2

- Doorman and NetworkMap URLs can now be configured individually rather than being assumed to be the same server.
Current `compatibilityZoneURL` configurations remain valid. See both Node configuration and Network permissioning for details.
- Table name with a typo changed from `NODE_ATTACHMENT_CONTRACTS` to `NODE_ATTACHMENTS_CONTRACTS`.

Version 3.1

- Update the fast-classpath-scanner dependent library version from 2.0.21 to 2.12.3

Note

Whilst this is not the latest version of this library, that being 2.18.1 at time of writing, versions later

than 2.12.3 (including 2.12.4) exhibit a different issue.

- Updated the api scanner gradle plugin to work the same way as the version in master. These changes make the api scanner more accurate and fix a couple of bugs, and change the format of the api-current.txt file slightly. Backporting these changes to the v3 branch will make it easier for us to ensure that apis are stable for future versions. These changes are released in gradle plugins version 3.0.10. For more information on the api scanner see the [documentation](#).
- Fixed security vulnerability when using the `HashAttachmentConstraint`. Added strict check that the contract JARs referenced in a transaction were deployed on the node.
- Fixed node's behaviour on startup when there is no connectivity to network map. Node continues to work normally if it has all the needed network data, waiting in the background for network map to become available.

Version 3.0

- Due to a security risk, the `conflict` property has been removed from `NotaryError.Conflict` error object. It has been replaced with `consumedStates` instead. The new property no longer specifies the

original requesting party and transaction id for a consumed state. Instead, only the hash of the transaction id is revealed. For more details why this change had to be made please refer to the release notes.

- Added `NetworkMapCache.getNodesByLegalName` for querying nodes belonging to a distributed service such as a notary cluster where they all share a common identity. `NetworkMapCache.getNodeByLegalName` has been tightened to throw if more than one node with the legal name is found.
- Introduced Flow Draining mode, in which a node continues executing existing flows, but does not start new. This is to support graceful node shutdown/restarts. In particular, when this mode is on, new flows through RPC will be rejected, scheduled flows will be ignored, and initial session messages will not be consumed. This will ensure that the number of checkpoints will strictly diminish with time, allowing for a clean shutdown.
- Removed blacklisted word checks in Corda X.500 name to allow “Server” or “Node” to be used as part of the legal name.
- Separated our pre-existing Artemis broker into an RPC broker and a P2P broker.
- Refactored `NodeConfiguration` to expose `NodeRpcOptions` (using top-level “rpcAddress” property still works with warning).
- Modified `CordaRPCClient` constructor to take a `SSLConfiguration?` additional parameter, defaulted to `null`.
- Introduced `CertificateChainCheckPolicy.UsernameMustMatchCommonName` sub-type, allowing customers to optionally enforce `username == CN` condition on RPC SSL certificates.
- Modified `DriverDSL` and sub-types to allow specifying RPC settings for the Node.
- Modified the `DriverDSL` to start Cordformation nodes allowing automatic generation of “`rpcSettings.adminAddress`” in case “`rpcSettings.useSsl`” is `false` (the default).
- Introduced `UnsafeCertificatesFactory` allowing programmatic generation of X509 certificates for test purposes.
- JPA Mapping annotations for States extending `CommonSchemaV1.LinearState` and `CommonSchemaV1.FungibleState` on the `participants` collection need to be moved to the actual class. This allows to properly specify the unique table name per a collection. See:
`DummyDealStateSchemaV1.PersistentDummyDealState`
- Database schema changes - an H2 database instance of Corda 1.0 and 2.0 cannot be reused for Corda 3.0, listed changes for Vault and Finance module:

- **NODE_TRANSACTIONS** :
 - column "TRANSACTION" renamed to TRANSACTION_VALUE, serialization format of BLOB stored in the column has changed to AMQP
 - **VAULT_STATES** :
 - column CONTRACT_STATE removed
 - **VAULT_FUNGIBLE_STATES** :
 - column ISSUER_REFERENCE renamed to ISSUER_REF and the field size increased
 - "VAULTSCHEMAM1\$VAULTFUNGIBLESTATES_PARTICIPANTS" :
 - table renamed to VAULT_FUNGIBLE_STATES_PARTS,
 - column "VAULTSCHEMAM1\$VAULTFUNGIBLESTATES_OUTPUT_INDEX" renamed to OUTPUT_INDEX,
 - column "VAULTSCHEMAM1\$VAULTFUNGIBLESTATES_TRANSACTION_ID" renamed to TRANSACTION_ID
 - **VAULT_LINEAR_STATES** :
 - type of column "UUID" changed from VARBINARY to VARCHAR(255) - select varbinary column as CAST("UUID" AS UUID) to get UUID in varchar format
 - "VAULTSCHEMAM1\$VAULTLINEARSTATES_PARTICIPANTS" :
 - table renamed to VAULT_LINEAR_STATES_PARTS,
 - column "VAULTSCHEMAM1\$VAULTLINEARSTATES_OUTPUT_INDEX" renamed to OUTPUT_INDEX,
 - column "VAULTSCHEMAM1\$VAULTLINEARSTATES_TRANSACTION_ID" renamed to TRANSACTION_ID
 - **contract_cash_states** :
 - columns storing Base58 representation of the serialised public key (e.g. issuer_key) were changed to store Base58 representation of SHA-256 of public key prefixed with DL
 - **contract_cp_states** :
 - table renamed to cp_states, column changes as for contract_cash_states
- X.509 certificates now have an extension that specifies the Corda role the certificate is used for, and the role hierarchy is now enforced in the validation code. See `net.corda.core.internal.CertRole` for the current implementation until final documentation is prepared. Certificates at NODE_CA, WELL_KNOWN_SERVICE_IDENTITY and above must only ever be issued by network services and therefore issuance constraints are not relevant to end users. The TLS, WELL_KNOWN_LEGAL_IDENTITY roles must be issued by the NODE_CA certificate issued by the Doorman, and CONFIDENTIAL_IDENTITY certificates must be issued from

- a `WELL_KNOWN_LEGAL_IDENTITY` certificate. For a detailed specification of the extension please see [Network permissioning](#).
- The network map service concept has been re-designed. More information can be found in [Network Map](#).
 - The previous design was never intended to be final but was rather a quick implementation in the earliest days of the Corda project to unblock higher priority items. It suffers from numerous disadvantages including lack of scalability, as one node is expected to hold open and manage connections to every node on the network; not reliable; hard to defend against DoS attacks; etc.
 - There is no longer a special network map node for distributing the network map to the other nodes. Instead the network map is now a collection of signed `NodeInfo` files distributed via HTTP.
 - The `certificateSigningService` config has been replaced by `compatibilityZoneURL` which is the base URL for the doorman registration and for downloading the network map. There is also an end-point for the node to publish its node-info object, which the node does each time it changes. `networkMapService` config has been removed.
 - To support local and test deployments, the node polls the `additional-node-infos` directory for these signed `NodeInfo` objects which are stored in its local cache. On startup the node generates its own signed file with the filename format “nodeInfo-*”. This can be copied to every node’s `additional-node-infos` directory that is part of the network.
 - Cordform (which is the `deployNodes` gradle task) does this copying automatically for the demos. The `NetworkMap` parameter is no longer needed.
 - For test deployments we’ve introduced a bootstrapping tool (see [setting-up-a-corda-network](#)).
 - `extraAdvertisedServiceIds`, `notaryNodeAddress`, `notaryClusterAddresses` and `bftSMaRT` configs have been removed. The configuration of notaries has been simplified into a single `notary` config object. See [Node configuration](#) for more details.
 - Introducing the concept of network parameters which are a set of constants which all nodes on a network must agree on to correctly interop. These can be retrieved from `ServiceHub.networkParameters`.
 - One of these parameters, `maxTransactionSize`, limits the size of a transaction, including its attachments, so that all nodes have sufficient memory to validate transactions.

- The set of valid notaries has been moved to the network parameters. Notaries are no longer identified by the CN in their X500 name.
 - Single node notaries no longer have a second separate notary identity. Their main identity *is* their notary identity. Use `NetworkMapCache.notaryIdentities` to get the list of available notaries.
 - Added `NetworkMapCache.getNodesByLegalName` for querying nodes belonging to a distributed service such as a notary cluster where they all share a common identity. `NetworkMapCache.getNodeByLegalName` has been tightened to throw if more than one node with the legal name is found.
 - The common name in the node's X500 legal name is no longer reserved and can be used as part of the node's name.
 - Moved `NodeInfoSchema` to internal package as the node info's database schema is not part of the public API. This was needed to allow changes to the schema.
- Support for external user credentials data source and password encryption [CORDA-827].
 - Exporting additional JMX metrics (artemis, hibernate statistics) and loading Jolokia agent at JVM startup when using DriverDSL and/or cordformation node runner.
 - Removed confusing property database.initDatabase, enabling its guarded behaviour with the dev-mode. In devMode Hibernate will try to create or update database schemas, otherwise it will expect relevant schemas to be present in the database (pre configured via DDL scripts or equivalent), and validate these are correct.
 - `AttachmentStorage` now allows providing metadata on attachments upload - username and filename, currently as plain strings. Those can be then used for querying, utilizing `queryAttachments` method of the same interface.
 - `SSH Server` - The node can now expose shell via SSH server with proper authorization and permissioning built in.
 - `CordaRPCOps` implementation now checks permissions for any function invocation, rather than just when starting flows.
 - `wellKnownPartyFromAnonymous()` now always resolve the key to a `Party`, then the party to the well known party. Previously if it was passed a `Party` it would use its name as-is without verifying the key matched that name.
 - `OpaqueBytes.bytes` now returns a clone of its underlying `ByteArray`, and has been redeclared as `final`. This is a minor change to the public API, but is required to ensure that classes like `SecureHash` are immutable.

- Experimental support for PostgreSQL: CashSelection done using window functions
- `FlowLogic` now exposes a series of function called `receiveAll(...)` allowing to join `receive(...)` instructions.
- Renamed “plugins” directory on nodes to “cordapps”
- The `Cordformation` gradle plugin has been split into `cordformation` and `cordapp`. The former builds and deploys nodes for development and testing, the latter turns a project into a cordapp project that generates JARs in the standard CorDapp format.
- `Cordapp` now has a name field for identifying CorDapps and all CorDapp names are printed to console at startup.
- Enums now respect the whitelist applied to the Serializer factory serializing / deserializing them. If the enum isn’t either annotated with the `@CordaSerializable` annotation or explicitly whitelisted then a `NotSerializableException` is thrown.
- Gradle task `deployNodes` can have an additional parameter `configFile` with the path to a properties file to be appended to node.conf.
- Cordformation node building DSL can have an additional parameter `configFile` with the path to a properties file to be appended to node.conf.
- `FlowLogic` now has a static method called `sleep` which can be used in certain circumstances to help with resolving contention over states in flows. This should be used in place of any other sleep primitive since these are not compatible with flows and their use will be prevented at some point in the future. Pay attention to the warnings and limitations described in the documentation for this method. This helps resolve a bug in `Cash` coin selection. A new static property `currentTopLevel` returns the top most `FlowLogic` instance, or null if not in a flow.
- `CordaService` annotated classes should be upgraded to take a constructor parameter of type `AppServiceHub` which allows services to start flows marked with the `StartableByService` annotation. For backwards compatibility service classes with only `ServiceHub` constructors will still work.
- `TimeWindow` now has a `length` property that returns the length of the time-window as a `java.time.Duration` object, or `null` if the time-window isn’t closed.
- A new `SIGNERS_GROUP` with ordinal 6 has been added to `ComponentGroupEnum` that corresponds to the `Command` signers.
- `PartialMerkleTree` is equipped with a `leafIndex` function that returns the index of a hash (leaf) in the partial Merkle tree structure.

- A new function `checkCommandVisibility(publicKey: PublicKey)` has been added to `FilteredTransaction` to check if every command that a signer should receive (e.g. an Oracle) is indeed visible.
- Changed the AMQP serialiser to use the officially assigned R3 identifier rather than a placeholder.
- The `ReceiveTransactionFlow` can now be told to record the transaction at the same time as receiving it. Using this feature, better support for observer/regulator nodes has been added. See [Observer nodes](#).
- Added an overload of `TransactionWithSignatures.verifySignaturesExcept` which takes in a collection of ```PublicKey```s.
- `DriverDSLEnabledInterface` has been renamed to `DriverDSL` and the `waitForAllNodesToFinish()` method has instead become a parameter on driver creation.
- Values for the `database.transactionIsolationLevel` config now follow the `java.sql.Connection` int constants but without the “TRANSACTION_” prefix, i.e. “NONE”, “READ_UNCOMMITTED”, etc.
- Peer-to-peer communications is now via AMQP 1.0 as default. Although the legacy Artemis CORE bridging can still be used by setting the `useAMQP Bridges` configuration property to false.
- The Artemis topics used for peer-to-peer communication have been changed to be more consistent with future cryptographic agility and to open up the future possibility of sharing brokers between nodes. This is a breaking wire level change as it means that nodes after this change will not be able to communicate correctly with nodes running the previous version. Also, any pending enqueued messages in the Artemis message store will not be delivered correctly to their original target. However, assuming a clean reset of the artemis data and that the nodes are consistent versions, data persisted via the AMQP serializer will be forward compatible.
- The ability for CordaServices to register callbacks so they can be notified of shutdown and clean up resource such as open ports.
- Move to a message based control of peer to peer bridge formation to allow for future out of process bridging components. This removes the legacy Artemis bridges completely, so the `useAMQP Bridges` configuration property has been removed.
- A `CordaInternal` attribute has been added to identify properties that are not intended to form part of the public api and as such are not intended for public use. This is alongside the existing `DoNotImplement` attribute for classes which provide Corda functionality to user applications, but should not be

implemented by consumers, and any classes which are defined in `.internal` packages, which are also not for public use.

- Marked `stateMachine` on `FlowLogic` as `CordaInternal` to make clear that it is not part of the public API and is only for internal use
- Provided experimental support for specifying your own webserver to be used instead of the default development webserver in `Cordform` using the `webserverJar` argument
- Created new `StartedMockNode` and `UnstartedMockNode` classes which are wrappers around our MockNode implementation that expose relevant methods for testing without exposing internals, create these using a `MockNetwork`.
- The test utils in `Expect.kt`, `SerializationTestHelpers.kt`, `TestConstants.kt` and `TestUtils.kt` have moved from the `net.corda.testing` package to the `net.corda.testing.core` package, and `FlowStackSnapshot.kt` has moved to the `net.corda.testing.services` package. Moving existing classes out of the `net.corda.testing.*` package will help make it clearer which parts of the API are stable. Scripts have been provided to smooth the upgrade process for existing projects in the `tools\scripts` directory of the Corda repo.
- `TransactionSignature` includes a new `partialMerkleTree` property, required for future support of signing over multiple transactions at once.
- Updating Jolokia dependency to latest version (includes security fixes)

Release 1.0

- Unification of VaultQuery And VaultService APIs Developers now only need to work with a single Vault Service API for all needs.
- Java 8 lambdas now work properly with Kryo during check-pointing.
- Java 8 serializable lambdas now work properly with Kryo during check-pointing.
- String constants have been marked as `const` type in Kotlin, eliminating cases where functions of the form `get<constant name>()` were created for the Java API. These can now be referenced by their name directly.
- `FlowLogic` communication has been extensively rewritten to use functions on `FlowSession` as the base for communication between nodes.
 - Calls to `send()`, `receive()` and `sendAndReceive()` on `FlowLogic` should be replaced with calls to the function of the same name on `FlowSession`. Note that the replacement functions do not take in a destination parameter, as this is defined in the session.

- Initiated flows now take in a `FlowSession` instead of `Party` in their constructor. If you need to access the counterparty identity, it is in the `counterparty` property of the flow session.
- Added X509EdDSAEngine to intercept and rewrite EdDSA public keys wrapped in X509Key instances. This corrects an issue with verifying certificate paths loaded from a Java Keystore where they contain EdDSA keys.
- Confidential identities are now complete:
 - The identity negotiation flow is now called `SwapIdentitiesFlow`, renamed from `TransactionKeyFlow`.
 - `generateSpend()` now creates a new confidential identity for the change address rather than using the identity of the input state owner.
 - Please see the documentation [Identity and API: Identity](#) for more details.
- Remove the legacy web front end from the SIMM demo.
- `NodeInfo` and `NetworkMapCache` changes:
 - Removed `NodeInfo::legalIdentity` in preparation for handling of multiple identities. We left list of `NodeInfo::legalIdentitiesAndCerts`, the first identity still plays a special role of main node identity.
 - We no longer support advertising services in network map. Removed `NodeInfo::advertisedServices`, `serviceIdentities` and `notaryIdentity`.
 - Removed service methods from `NetworkMapCache`: `partyNodes`, `networkMapNodes`, `notaryNodes`, `regulatorNodes`, `getNodesWithService`, `getPeersWithService`, `getRecommended`, `getNodesByAdvertisedServiceIdentityKey`, `getAnyNotary`, `notaryNode`, `getAnyServiceOfType`. To get all known `NodeInfo`'s call `allNodes`.
 - In preparation for `NetworkMapService` redesign and distributing notaries through `NetworkParameters` we added `NetworkMapCache::notaryIdentities` list to enable to lookup for notary parties known to the network. Related `CordaRPCOps::notaryIdentities` was introduced. Other special nodes parties like Oracles or Regulators need to be specified directly in CorDapp or flow.
 - Moved `ServiceType` and `ServiceInfo` to `net.corda.nodeapi` package as services are only required on node startup.
- Adding enum support to the class carpenter

- `ContractState::contract` has been moved to `TransactionState::contract` and its type has changed to `String` in order to support dynamic classloading of contract and contract constraints.
- CorDapps that contain contracts are now automatically loaded into the attachment storage - for CorDapp developers this now means that contracts should be stored in separate JARs to flows, services and utilities to avoid large JARs being auto imported to the attachment store.
- About half of the code in test-utils has been moved to a new module `node-driver`, and the test scope modules are now located in a `testing` directory.
- `CordaPluginRegistry` has been renamed to `SerializationWhitelist` and moved to the `net.corda.core.serialization` package. The API for whitelisting types that can't be annotated was slightly simplified. This class used to contain many things, but as we switched to annotations and classpath scanning over time it hollowed out until this was the only functionality left. You also need to rename your services resource file to the new class name. An associated property on `MockNode` was renamed from `testPluginRegistries` to `testSerializationWhitelists`.
- Contract Upgrades: deprecated RPC authorisation / deauthorisation API calls in favour of equivalent flows in `ContractUpgradeFlow`. Implemented contract upgrade persistence using JDBC backed persistent map.
- Vault query common attributes (state status and contract state types) are now handled correctly when using composite criteria specifications. State status is overridable. Contract states types are aggregatable.
- Cash selection algorithm is now pluggable (with H2 being the default implementation)
- Removed usage of Requery ORM library (replaced with JPA/Hibernate)
- Vault Query performance improvement (replaced expensive per query SQL statement to obtain concrete state types with single query on start-up followed by dynamic updates using vault state observable))
- Vault Query fix: filter by multiple issuer names in `FungibleAssetQueryCriteria`
- Following deprecated methods have been removed:
 - In `DataFeed`
 - `first` and `current`, replaced by `snapshot`
 - `second` and `future`, replaced by `updates`
 - In `CordaRPCOps`
 - `stateMachinesAndUpdates`, replaced by `stateMachinesFeed`
 - `verifiedTransactions`, replaced by `verifiedTransactionsFeed`

- `stateMachineRecordedTransactionMapping`, replaced by `stateMachineRecordedTransactionMappingFeed`
 - `networkMapUpdates`, replaced by `networkMapFeed`
- Due to security concerns and the need to remove the concept of state relevancy (which isn't needed in Corda), `ResolveTransactionsFlow` has been made internal. Instead merge the receipt of the `SignedTransaction` and the subsequent sub-flow call to `ResolveTransactionsFlow` with a single call to `ReceiveTransactionFlow`. The flow running on the counterparty must use `SendTransactionFlow` at the correct place. There is also `ReceiveStateAndRefFlow` and `SendStateAndRefFlow` for dealing with `StateAndRef`'s.
- Vault query soft locking enhancements and deprecations
 - removed original `VaultService`softLockedStates`` query mechanism.
 - introduced improved `SoftLockingCondition` filterable attribute in `VaultQueryCriteria` to enable specification of different soft locking retrieval behaviours (exclusive of soft locked states, soft locked states only, specified by set of lock ids)
- Trader demo now issues cash and commercial paper directly from the bank node, rather than the seller node self-issuing commercial paper but labelling it as if issued by the bank.
- Merged handling of well known and confidential identities in the identity service. Registration now takes in an identity (either type) plus supporting certificate path, and de-anonymisation simply returns the issuing identity where known. If you specifically need well known identities, use the network map, which is the authoritative source of current well known identities.
- Currency-related API in `net.corda.core.contracts.ContractsDSL` has moved to ``net.corda.finance.CurrencyUtils``.
- Remove `IssuerFlow` as it allowed nodes to request arbitrary amounts of cash to be issued from any remote node. Use `CashIssueFlow` instead.
- Some utility/extension functions (`sumOrThrow`, `sumOrNull`, `sumOrZero` on `Amount` and `Commodity`) have moved to be static methods on the classes themselves. This improves the API for Java users who no longer have to see or know about file-level FooKt style classes generated by the Kotlin compile, but means that IntelliJ no longer auto-suggests these extension functions in completion unless you add import lines for them yourself (this is Kotlin IDE bug KT-15286).
- `:finance` module now acting as a CorDapp with regard to flow registration, schemas and serializable types.

- `WebServerPluginRegistry` now has a `customizeJSONSerialization` which can be overridden to extend the REST JSON serializers. In particular the IRS demos must now register the `BusinessCalendar` serializers.
- Moved `:finance` gradle project files into a `net.corda.finance` package namespace. This may require adjusting imports of Cash flow references and also of `StartFlow` permission in `gradle.build` files.
- Removed the concept of relevancy from `LinearState`. The `ContractState`'s relevancy to the vault can be determined by the flow context, the vault will process any transaction from a flow which is not derived from transaction resolution verification.
- Removed the tolerance attribute from `TimeWindowChecker` and thus, there is no extra tolerance on the notary side anymore.
- The `FungibleAsset` interface has been made simpler. The `Commands` grouping interface that included the `Move`, `Issue` and `Exit` interfaces have all been removed, while the `move` function has been renamed to `withNewOwnerAndAmount` to be consistent with the `withNewOwner` function of the `OwnableState`.
- The `IssueCommand` interface has been removed from `Structures`, because, due to the introduction of nonces per transaction component, the issue command does not need a nonce anymore and it does not require any other attributes.
- As a consequence of the above and the simpler `FungibleAsset` format, fungible assets like `Cash` now use `class Issue : TypeOnlyCommandData()`, because it's only its presence (`Issue`) that matters.
- A new *PrivacySalt* transaction component is introduced, which is now an attribute in `TraversableTransaction` and inherently in `WireTransaction`.
- A new `nonces: List<SecureHash>` feature has been added to `FilteredLeaves`.
- Due to the `nonces` and `PrivacySalt` introduction, new functions have been added to `MerkleTransaction`:

```
: fun <T : Any> serializedHash(x: T, privacySalt: PrivacySalt?
, index: Int): SecureHash
fun <T : Any> serializedHash(x: T, nonce: SecureHash): SecureHash
fun computeNonce(privacySalt: PrivacySalt, index: Int).
```
- A new `SignatureMetadata` data class is introduced with two attributes, `platformVersion: Int` and `schemeNumberID: Int` (the signature scheme used).
- As part of the metadata support in signatures, a new `data class SignableData(val txId: SecureHash, val signatureMetadata: SignatureMetadata)` is introduced, which represents the object actually signed.

- The unused `MetaData` and `SignatureType` in `crypto` package have been removed.
- The `class TransactionSignature(bytes: ByteArray, val by: PublicKey, val signature Metadata: SignatureMetadata): DigitalSignature(bytes)` class is now utilised Vs the old `DigitalSignature.WithKey` for Corda transaction signatures. Practically, it takes the `signatureMetadata` as an extra input, in order to support signing both the transaction and the extra metadata.
- To reflect changes in the signing process, the `Crypto` object is now equipped with the: `fun doSign(keyPair: KeyPair, signableData: SignableData): TransactionSignature` and `fun doVerify(txId: SecureHash, transactionSignature: TransactionSignature): Boolean` functions.
- `SerializationCustomization.addToWhitelist()` now accepts multiple classes via varargs.
- Two functions to easily sign a `FilteredTransaction` have been added to `ServiceHub.createSignature(filteredTransaction: FilteredTransaction, publicKey : PublicKey)` and `createSignature(filteredTransaction: FilteredTransaction)` to sign with the legal identity key.
- A new helper method `buildFilteredTransaction(filtering: Predicate<Any>)` is added to `SignedTransaction` to directly build a `FilteredTransaction` using provided filtering functions, without first accessing the `tx: WireTransaction`.
- Test type `NodeHandle` now has method `stop(): CordaFuture<Unit>` that terminates the referenced node.
- **Fixed some issues in IRS demo:**
 - Fixed leg and floating leg notional amounts were not displayed for created deals neither in single nor in list view.
 - Parties were not displayed for created deals in single view.
 - Non-default notional amounts caused the creation of new deals to fail.

Warning

Renamed configuration property key `basedir` to `baseDirectory`. This will require updating existing configuration files.

- Removed deprecated parts of the API.
- Removed `PluginServiceHub`. Replace with `ServiceHub` for `@CordaService` constructors.
- `X509CertificateHolder` has been removed from the public API, replaced by `java.security.X509Certificate`.

- Moved `CityDatabase` out of `core` and into `finance`
- All of the `serializedHash` and `computeNonce` functions have been removed from `MerkleTransaction`. The `serializedHash(x: T)` and `computeNonce` were moved to `CryptoUtils`.
- Two overloaded methods `componentHash(opaqueBytes: OpaqueBytes, privacySalt: PrivacySalt, componentGroupIndex: Int, internalIndex: Int): SecureHash` and `componentHash(nonce: SecureHash, opaqueBytes: OpaqueBytes): SecureHash` have been added to `CryptoUtils`. Similarly to `computeNonce`, they internally use SHA256d for nonce and leaf hash computations.
- The `verify(node: PartialTree, usedHashes: MutableList<SecureHash>): SecureHash` in `PartialMerkleTree` has been renamed to `rootAndUsedHashes` and is now public, as it is required in the verify function of `FilteredTransaction`.
- `TraversableTransaction` is now an abstract class extending `CoreTransaction`, `WireTransaction` and `FilteredTransaction` now extend `TraversableTransaction`.
- Two classes, `ComponentGroup(open val groupIndex: Int, open val components: List<OpaqueBytes>)` and `FilteredComponentGroup(override val groupIndex: Int, override val components: List<OpaqueBytes>, val nonces: List<SecureHash>, val partialMerkleTree: PartialMerkleTree): ComponentGroup(groupIndex, components)` have been added, which are properties of the `WireTransaction` and `FilteredTransaction`, respectively.
- `checkAllComponentsVisible(componentGroupEnum: ComponentGroupEnum)` is added to `FilteredTransaction`, a new function to check if all components are visible in a specific component-group.
- To allow for backwards compatibility, `WireTransaction` and `FilteredTransaction` have new fields and constructors: `WireTransaction(componentGroups: List<ComponentGroup>, privacySalt: PrivacySalt = PrivacySalt(), FilteredTransaction private constructor(id: SecureHash, filteredComponentGroups: List<FilteredComponentGroup>, groupHashes: List<SecureHash>)`. `FilteredTransaction` is still built via `buildFilteredTransaction(wtx: WireTransaction, filtering: Predicate<Any>)`.
- `FilteredLeaves` class have been removed and as a result we can directly call the components from `FilteredTransaction`, such as `ftx.inputs` Vs the old `ftx.filteredLeaves.inputs`.

- A new `ComponentGroupEnum` is added with the following enum items: `INPUTS_GROUP`, `OUTPUTS_GROUP`, `COMMANDS_GROUP`, `ATTACHMENTS_GROUP`, `NOTARY_GROUP`, `ROUP`, `TIMEWINDOW_GROUP`.
- `ContractUpgradeFlow.Initiator` has been renamed to `ContractUpgradeFlow.Initiate`
- `@RPCSinceVersion`, `RPCException` and `PermissionException` have moved to `net.corda.client.rpc`.
- Current implementation of SSL in `CordaRPCClient` has been removed until we have a better solution which doesn't rely on the node's keystore.

Milestone 14

- Changes in `NodeInfo`:
 - `PhysicalLocation` was renamed to `WorldMapLocation` to emphasise that it doesn't need to map to a truly physical location of the node server.
 - Slots for multiple IP addresses and `legalIdentitiesAndCert`'s were introduced. Addresses are no longer of type ``SingleMessageRecipient`, but of `NetworkHostAndPort`.
- `ServiceHub.storageService` has been removed. `attachments` and `validatedTransactions` are now direct members of `ServiceHub`.
- Mock identity constants used in tests, such as `ALICE`, `BOB`, `DUMMY_NOTARY`, have moved to `net.corda.testing` in the `test-utils` module.
- `DummyContract`, `DummyContractV2`, `DummyLinearContract` and `DummyState` have moved to `net.corda.testing.contracts` in the `test-utils` modules.
- In Java, `QueryCriteriaUtilsKt` has moved to `QueryCriteriaUtils`. Also `and` and `or` are now instance methods of `QueryCriteria`.
- `random63BitValue()` has moved to `CryptoUtils`
- Added additional common Sort attributes (see `Sort.CommandStateAttribute`) for use in Vault Query criteria to include `STATE_REF`, `STATE_REF_TXN_ID`, `STATE_REF_INDEX`
- Moved the core flows previously found in `net.corda.flows` into `net.corda.core.flows`. This is so that all packages in the `core` module begin with `net.corda.core`.
- `FinalityFlow` can now be subclassed, and the `broadcastTransaction` and `lookupParties` function can be overriden in order to handle cases where no single transaction participant is aware of all parties, and therefore the transaction must be relayed between participants rather than sent from a single node.

- `TransactionForContract` has been removed and all usages of this class have been replaced with usage of `LedgerTransaction`. In particular `Contract.verify` and the `Clauses` API have been changed and now take a `LedgerTransaction` as passed in parameter. The principal consequence of this is that the types of the input and output collections on the transaction object have changed, so it may be necessary to `map` down to the `ContractState` sub-properties in existing code.
- Added various query methods to `LedgerTransaction` to simplify querying of states and commands. In the same vein `Command` is now parameterised on the `CommandData` field.
- Kotlin utilities that we deemed useful enough to keep public have been moved out of `net.corda.core.Utils` and into `net.corda.core.utilities.KotlinUtils`. The other utilities have been marked as internal.
- Changes to `Cordformation` / cordapp building:
 - `Cordformation` modifies the JAR task to make cordapps build as semi fat JARs containing all dependencies except other cordapps and Corda core dependencies.
 - `Cordformation` adds a `corda` and `cordaRuntime` configuration to projects which cordapp developers should use to exclude core Corda JARs from being built into Cordapp fat JARs.
- `database` field in `AbstractNode` class has changed the type from `org.jetbrains.exposed.sql.Database` to ‘`net.corda.node.utilities.CordaPersistence`’ - no change is needed for the typical use (i.e. `services.database.transaction { code block }`) however a change is required when Database was explicitly declared
- `DigitalSignature.LegallyIdentifiable`, previously used to identify a signer (e.g. in Oracles), has been removed. One can use the public key to derive the corresponding identity.
- Vault Query improvements and fixes:
 - FIX inconsistent behaviour: Vault Query defaults to UNCONSUMED in all QueryCriteria types
 - FIX serialization error: Vault Query over RPC when using custom attributes using `VaultCustomQueryCriteria`.
 - Aggregate function support: extended `VaultCustomQueryCriteria` and associated DSL to enable specification of

Aggregate Functions (sum, max, min, avg, count) with, optional, group by clauses and sorting (on calculated aggregate)

- Pagination simplification

Pagination continues to be optional, but with following changes:

- If no PageSpecification provided then a maximum of MAX_PAGE_SIZE (200) results will be returned, otherwise we fail-fast with a `VaultQueryException` to alert the API user to the need to specify a PageSpecification. Internally, we no longer need to calculate a results count (thus eliminating an expensive SQL query) unless a PageSpecification is supplied (note: that a value of -1 is returned for total_results in this scenario). Internally, we now use the AggregateFunction capability to perform the count.
- Paging now starts from 1 (was previously 0).
- Additional Sort criteria: by StateRef (or constituents: txId, index)
- Confidential identities API improvements
 - Registering anonymous identities now takes in AnonymousPartyAndPath
 - AnonymousParty.toString() now uses toStringShort() to match other toString() functions
 - Add verifyAnonymousIdentity() function to verify without storing an identity
 - Replace pathForAnonymous() with anonymousFromKey() which matches actual use-cases better
 - Add unit test for fetching the anonymous identity from a key
 - Update verifyAnonymousIdentity() function signature to match registerAnonymousIdentity()
 - Rename AnonymisedIdentity to AnonymousPartyAndPath
 - Remove certificate from AnonymousPartyAndPath as it's not actually used.
 - Rename registerAnonymousIdentity() to verifyAndRegisterAnonymousIdentity()
- Added JPA `AbstractPartyConverter` to ensure identity schema attributes are persisted securely according to type (well known party, resolvable anonymous party, completely anonymous party).

Milestone 13

Special thank you to [Frederic Dalibard](#), for his contribution which adds support for more currencies to the DemoBench and Explorer tools.

- A new Vault Query service:
 - Implemented using JPA and Hibernate, this new service provides the ability to specify advanced queries using criteria specification sets for both vault attributes and custom contract specific attributes. In addition, new queries provide sorting and pagination capabilities. The new API provides two function variants which are exposed for usage within Flows and by RPC clients: - `queryBy()` for point-in-time snapshot queries

(replaces several existing VaultService functions and a number of Kotlin-only extension functions)
 - `trackBy()` for snapshot and streaming updates (replaces the VaultService `track()` function and the RPC `vaultAndUpdates()` function)

Existing VaultService API methods will be maintained as deprecated until the following milestone release.

- The NodeSchema service has been enhanced to automatically generate mapped objects for any ContractState objects that extend FungibleAsset or LinearState, such that common attributes of those parent states are persisted to two new vault tables: `vault_fungible_states` and `vault_linear_states` (and thus queryable using the new Vault Query service API). Similarly, two new common JPA superclass schemas (`CommonSchemaV1.FungibleState` and `CommonSchemaV1.LinearState`) mirror the associated FungibleAsset and LinearState interface states to enable CorDapp developers to create new custom schemas by extension (rather than duplication of common attribute mappings)
- A new configurable field `requiredSchemas` has been added to the CordaPluginRegistry to enable CorDapps to register custom contract state schemas they wish to query using the new Vault Query service API (using the `VaultCustomQueryCriteria`).
- See `vault-query` for full details and code samples of using the new Vault Query service.

- Identity and cryptography related changes:
 - Enable certificate validation in most scenarios (will be enforced in all cases in an upcoming milestone).
 - Added DER encoded format for CompositeKey so they can be used in X.509 certificates.
 - Corrected several tests which made assumptions about counterparty keys, which are invalid when confidential identities are used.
 - A new RPC has been added to support fuzzy matching of X.500 names, for instance, to translate from user input to an unambiguous identity by searching the network map.
 - A function for deterministic key derivation `Crypto.deriveKeyPair(privateKey: PrivateKey, seed: ByteArray)` has been implemented to support deterministic `KeyPair` derivation using an existing private key and a seed as inputs. This operation is based on the HKDF scheme and it's a variant of the hardened parent-private -> child-private key derivation function of the BIP32 protocol, but it doesn't utilize extension chain codes. Currently, this function supports the following schemes: ECDSA secp256r1 (NIST P-256), ECDSA secp256k1 and EdDSA ed25519.
- A new `ClassWhitelist` implementation, `AllButBlacklisted` is used internally to blacklist classes/interfaces, which are not expected to be serialised during checkpoints, such as `Thread`, `Connection` and `HashSet`. This implementation supports inheritance and if a superclass or superinterface of a class is blacklisted, so is the class itself. An `IllegalStateException` informs the user if a class is blacklisted and such an exception is returned before checking for `@CordaSerializable`; thus, blacklisting precedes annotation checking.
- `TimeWindow` has a new 5th factory method `TimeWindow.fromStartAndDuration(fromTime: Instant, duration: Duration)` which takes a start-time and a period-of-validity (after this start-time) as inputs.
- The node driver has moved to `net.corda.testing.driver` in the test-utils module.
- Web API related collections `CordaPluginRegistry.webApis` and `CordaPluginRegistry.staticServeDirs` moved to `net.corda.webserver.services.WebServerPluginRegistry` in `webserver` module. Classes serving Web API should now extend `WebServerPluginRegistry` instead of `CordaPluginRegistry` and they should be registered in `resources/META-INF/services/net.corda.webserver.services.WebServerPluginRegistry`.

- Added a flag to the driver that allows the running of started nodes in-process, allowing easier debugging. To enable use `driver(startNodesInProcess = true)` { .. }, or `startNode(startInSameProcess = true, ..)` to specify for individual nodes.
- **Dependencies changes:**
 - Upgraded Dokka to v0.9.14.
 - Upgraded Gradle Plugins to 0.12.4.
 - Upgraded Apache ActiveMQ Artemis to v2.1.0.
 - Upgraded Netty to v4.1.9.Final.
 - Upgraded BouncyCastle to v1.57.
 - Upgraded Requery to v1.3.1.

Milestone 12 (First Public Beta)

- Quite a few changes have been made to the flow API which should make things simpler when writing CorDapps:
 - `CordaPluginRegistry.requiredFlows` is no longer needed. Instead annotate any flows you wish to start via RPC with `@StartableByRPC` and any scheduled flows with `@SchedulableFlow`.
 - `CordaPluginRegistry.servicePlugins` is also no longer used, along with `PluginServiceHub.registerFlowInitiator`. Instead annotate your initiated flows with `@InitiatedBy`. This annotation takes a single parameter which is the initiating flow. This initiating flow further has to be annotated with `@InitiatingFlow`. For any services you may have, such as oracles, annotate them with `@CordaService`. These annotations will be picked up automatically when the node starts up.
 - Due to these changes, when unit testing flows make sure to use `AbstractNode.registerInitiatedFlow` so that the flows are wired up. Likewise for services use `AbstractNode.installCordaService`.
 - Related to `InitiatingFlow`, the `shareParentSessions` boolean parameter of `FlowLogic.subFlow` has been removed. This was an unfortunate parameter that unnecessarily exposed the inner workings of flow sessions. Now, if your sub-flow can be started outside the context of the parent flow then annotate it with `@InitiatingFlow`. If it's meant to be used as a continuation of the existing parent flow, such as `CollectSignaturesFlow`, then it doesn't need any annotation.

- The `InitiatingFlow` annotation also has an integer `version` property which assigns the initiating flow a version number, defaulting to 1 if it's not specified. This enables versioning of flows with nodes only accepting communication if the version number matches. At some point we will support the ability for a node to have multiple versions of the same flow registered, enabling backwards compatibility of flows.
 - `ContractUpgradeFlow.Instigator` has been renamed to just `ContractUpgradeFlow`.
 - `NotaryChangeFlow.Instigator` has been renamed to just `NotaryChangeFlow`.
 - `FlowLogic.getCounterpartyMarker` is no longer used and been deprecated for removal. If you were using this to manage multiple independent message streams with the same party in the same flow then use sub-flows instead.
- There are major changes to the `Party` class as part of confidential identities:
 - `Party` has moved to the `net.corda.core.identity` package; there is a deprecated class in its place for backwards compatibility, but it will be removed in a future release and developers should move to the new class as soon as possible.
 - There is a new `AbstractParty` superclass to `Party`, which contains just the public key. This now replaces use of `Party` and `PublicKey` in state objects, and allows use of full or anonymised parties depending on use-case.
 - A new `PartyAndCertificate` class has been added which aggregates a Party along with an X.509 certificate and certificate path back to a network trust root. This is used where a Party and its proof of identity are required, for example in identity registration.
 - Names of parties are now stored as a `X500Name` rather than a `String`, to correctly enforce basic structure of the name. As a result all node legal names must now be structured as X.500 distinguished names.
- The identity management service takes an optional network trust root which it will validate certificate paths to, if provided. A later release will make this a required parameter.
- There are major changes to transaction signing in flows:
 - You should use the new `CollectSignaturesFlow` and corresponding `SignTransactionFlow` which handle most of the details of this for you. They may get more complex in future as signing becomes a more featureful operation.

- `ServiceHub.legalIdentityKey` no longer returns a `KeyPair`, it instead returns just the `PublicKey` portion of this pair.

The `ServiceHub.notaryIdentityKey` has changed similarly. The goal of this change is to keep private keys

encapsulated and away from most flow code/Java code, so that the private key material can be stored in HSMs and other key management devices.

- The `KeyManagementService` no longer provides any mechanism to request the node's `PrivateKey` objects directly. Instead signature creation occurs in the `KeyManagementService.sign`, with the `PublicKey` used to indicate which of the node's keypairs to use. This lookup also works for `CompositeKey` scenarios and the service will search for a leaf key hosted on the node.
- The `KeyManagementService.freshKey` method now returns only the `PublicKey` portion of the newly generated `KeyPair` with the `PrivateKey` kept internally to the service.
- Flows which used to acquire a node's `KeyPair`, typically via `ServiceHub.legalIdentityKey`, should instead use the helper methods on `ServiceHub`. In particular to freeze a `TransactionBuilder` and generate an initial partially signed `SignedTransaction` the flow should use `ServiceHub.toSignedTransaction`. Flows generating additional party signatures should use `ServiceHub.createSignature`. Each of these methods is provided with two signatures. One version that signs with the default node key, the other which allows key selection by passing in the `PublicKey` partner of the desired signing key.
- The original `KeyPair` signing methods have been left on the `TransactionBuilder` and `SignedTransaction`, but should only be used as part of unit testing.

- `Timestamp` used for validation/notarization time-range has been renamed to `TimeWindow`.

There are now 4 factory methods `TimeWindow.fromOnly(fromTime: Instant)`, `TimeWindow.untilOnly(untilTime: Instant)`, `TimeWindow.between(fromTime: Instant, untilTime: Instant)` and `TimeWindow.withTolerance(time: Instant, tolerance: Duration)`. Previous constructors `TimeWindow(fromTime: Instant, untilTime: Instant)` and `TimeWindow(time: Instant, tolerance: Duration)` have been removed.

- The Bouncy Castle library `X509CertificateHolder` class is now used in place of `X509Certificate` in order to have a consistent class used internally. Conversions to/from `X509Certificate` are done as required, but should be avoided where possible.
- **The certificate hierarchy has been changed in order to allow corda node to sign keys with proper certificate chain.**
 - The corda node will now be issued a restricted client CA for identity/transaction key signing.
 - TLS certificate are now stored in `ssl/keystore.jks` and identity keys are stored in `nodekeystore.jks`

Warning

The old keystore will need to be removed when upgrading to this version.

Milestone 11.1

- Fix serialisation error when starting a flow.
- Automatically whitelist subclasses of `InputStream` when serialising.
- Fix exception in DemoBench on Windows when loading CorDapps into the Node Explorer.
- Detect when localhost resolution is broken on MacOSX, and provide instructions on how to fix it.

Milestone 11.0

- **API changes:**
 - Added extension function `Database.transaction` to replace `databaseTransaction`, which is now deprecated.
 - Starting a flow no longer enables progress tracking by default. To enable it, you must now invoke your flow using one of the new `CordaRPCOps.startTrackedFlow` functions. `FlowHandle` is now an interface, and its `progress: Observable` field has been moved to the `FlowProgressHandle` child interface. Hence developers no longer need to invoke `notUsed` on their flows' unwanted progress-tracking observables.
 - Moved `generateSpend` and `generateExit` functions into `OnLedgerAsset` from the vault and `AbstractConserveAmount` clauses respectively.

- Added `CompositeSignature` and `CompositeSignatureData` as part of enabling `java.security` classes to work with composite keys and signatures.
- `CompositeKey` now implements `java.security.PublicKey` interface, so that keys can be used on standard classes such as `Certificate`.
 - There is no longer a need to transform single keys into composite - `composite` extension was removed, it is impossible to create `CompositeKey` with only one leaf.
 - Constructor of `CompositeKey` class is now private.
Use `CompositeKey.Builder` to create a composite key. Keys emitted by the builder are normalised so that it's impossible to create a composite key with only one node. (Long chains of single nodes are shortened.)
 - Use extension function `PublicKeys.keys` to access all keys belonging to an instance of `PublicKey`. For a `CompositeKey`, this is equivalent to `CompositeKey.leafKeys`.
 - Introduced `containsAny`, `isFulfilledBy`, `keys` extension functions on `PublicKey` - `CompositeKey` type checking is done there.
- Corda now requires JDK 8u131 or above in order to run. Our Kotlin now also compiles to JDK8 bytecode, and so you'll need to update your CorDapp projects to do the same. E.g. by adding this to `build.gradle`:

```
tasks.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompile).all {
    kotlinOptions {
        languageVersion = "1.1"
        apiVersion = "1.1"
        jvmTarget = "1.8"
    }
}
```

or by adjusting `Settings/Build,Execution,Deployment/Compiler/KotlinCompiler` in IntelliJ:

- Language Version: `1.1`
- API Version: `1.1`
- Target JVM Version: `1.8`
- DemoBench is now installed as `Corda DemoBench` instead of `DemoBench`.
- Rewrote standard test identities to have full X.500 distinguished names. As part of this work we standardised on a smaller set of test identities, to reduce risk of subtle differences (i.e. similar common names varying by whitespace) in naming making it hard to diagnose issues.

Milestone 10.0

Special thank you to [Qian Hong](#), [Marek Skocovsky](#), [Karel Hajek](#), and [Jonny Chiu](#) for their contributions to Corda in M10.

Warning

Due to incompatibility between older version of IntelliJ and gradle 3.4, you will need to upgrade IntelliJ to 2017.1 (with kotlin-plugin v1.1.1) in order to run Corda demos in IntelliJ. You can download the latest IntelliJ from [JetBrains](#).

Warning

The Kapt-generated models are no longer included in our codebase. If you experience `unresolved references` errors when building in IntelliJ, please rebuild the schema model by running `gradlew kaptKotlin` in Windows or `./gradlew kaptKotlin` in other systems. Alternatively, perform a full gradle build or install.

Note

Kapt is used to generate schema model and entity code (from annotations in the codebase) using the Kotlin Annotation processor.

- **Corda DemoBench:**

- DemoBench is a new tool to make it easy to configure and launch local Corda nodes. A very useful tool to demonstrate to your colleagues the fundamentals of Corda in real-time. It has the following features:
 - Clicking “Add node” creates a new tab that lets you edit the most important configuration properties of the node before launch, such as its legal name and which CorDapps will be loaded.
 - Each tab contains a terminal emulator, attached to the pseudoterminal of the node. This lets you see console output.
 - You can launch an Corda Explorer instance for each node via the DemoBench UI. Credentials are handed to the Corda Explorer so it starts out logged in already.
 - Some basic statistics are shown about each node, informed via the RPC connection.
 - Another button launches a database viewer in the system browser.
 - The configurations of all running nodes can be saved into a single `.profile` file that can be reloaded later.
- You can download Corda DemoBench from [here](#)

- **Vault:**
 - Soft Locking is a new feature implemented in the vault which prevent a node constructing transactions that attempt to use the same input(s) simultaneously.
 - Such transactions would result in naturally wasted effort when the notary rejects them as double spend attempts.
 - Soft locks are automatically applied to coin selection (eg. cash spending) to ensure that no two transactions attempt to spend the same fungible states.
- **Corda Shell :**
 - The shell lets developers and node administrators easily command the node by running flows, RPCs and SQL queries.
 - It provides a variety of commands to monitor the node.
 - The Corda Shell is based on the popular [CRaSH project](#) and new commands can be easily added to the node by simply dropping Groovy or Java files into the node's `shell-commands` directory.
 - We have many enhancements planned over time including SSH access, more commands and better tab completion.
- **API changes:**
 - The new Jackson module provides JSON/YAML serialisers for common Corda datatypes. If you have previously been using the JSON support in the standalone web server, please be aware that Amounts are now serialised as strings instead of { quantity, token } pairs as before. The old format is still accepted, but the new JSON will be produced using strings like "1000.00 USD" when writing. You can use any format supported by `Amount.parseCurrency` as input.
 - **We have restructured client package in this milestone.**
 - `CordaClientRPC` is now in the new `:client:rpc` module.
 - The old `:client` module has been split up into `:client:jfx` and `:client:mock`.
 - We also have a new `:node-api` module (package `net.corda.nodeapi`) which contains the shared code between `node` and `client`.
 - The basic Amount API has been upgraded to have support for advanced financial use cases and to better integrate with currency reference data.
- **Configuration:**
 - Replace `artemisPort` with `p2pPort` in Gradle configuration.
 - Replace `artemisAddress` with `p2pAddress` in node configuration.

- Added `rpcAddress` in node configuration for non-ssl RPC connection.
- **Object Serialization:**
 - Pool Kryo instances for efficiency.
- **RPC client changes:**
 - RPC clients can now connect to the node without the need for SSL. This requires a separate port on the Artemis broker, SSL must not be used for RPC connection.
 - CordaRPCClient now needs to connect to `rpcAddress` rather than `p2pAddress`.
- **Dependencies changes:**
 - Upgraded Kotlin to v1.1.1.
 - Upgraded Gradle to v3.4.1.
 - Upgraded requery to v1.2.1.
 - Upgraded H2 to v1.4.194.
 - Replaced `kotlinx-support-jdk8` with `kotlin-stdlib-jre8`.
- **Improvements:**
 - Added `--version` command line flag to print the version of the node.
 - Flows written in Java can now execute a sub-flow inside `UntrustworthyData.unwrap`.
 - Added optional out-of-process transaction verification. Any number of external verifier processes may be attached to the node which can handle loadbalanced verification requests.
- **Bug fixes:**
 - `--logging-level` command line flag was previously broken, now correctly sets the logging level.
 - Fixed bug whereby Cash Exit was not taking into account the issuer reference.

Milestone 9.1

- Correct web server ports for IRS demo.
- Correct which corda-webserver JAR is published to Maven.

Milestone 9

- With thanks to [Thomas Schroeter](#) for the Byzantine fault tolerant (BFT) notary prototype.
- Web server is a separate JAR. This is a breaking change. The new webserver JAR (`corda-webserver.jar`) must be invoked separately to node startup, using

the command ``java -jar corda-webserver.jar`` in the same directory as the `node.conf`. Further changes are anticipated in upcoming milestone releases.

- API:

- Pseudonymous `AnonymousParty` class added as a superclass of `Party`.
- Split `CashFlow` into individual `CashIssueFlow`, `CashPaymentFlow` and `CashExitFlow` flows, so that fine grained permissions can be applied. Added `CashFlowCommand` for use-cases where cash flow triggers need to be captured in an object that can be passed around.
- `CordaPluginRegistry` method `registerRPCKryoTypes` is renamed `customizeSerialization` and the argument types now hide the presence of Kryo.
- New extension functions for encoding/decoding to base58, base64, etc. See `core/src/main/kotlin/net/corda/core/crypto/EncodingUtils.kt`
- Add `openAttachment` function to Corda RPC operations, for downloading an attachment from a node's data storage.
- Add `getCashBalances` function to Corda RPC operations, for getting cash balances from a node's vault.

- Configuration:

- `extraAdvertisedServiceIds` config is now a list of strings, rather than a comma separated string. For example `["corda.interest_rates"]` instead of `"corda.interest_rates"`.

- Flows:

- Split `CashFlow` into separate `CashIssueFlow`, `CashPaymentFlow` and `CashExitFlow` so that permissions can be assigned individually.
- Split single example user into separate “bankUser” and “bigCorpUser” so that permissions for the users make sense rather than being a combination of both roles.
- `ProgressTracker` emits exception thrown by the flow, allowing the ANSI renderer to correctly stop and print the error

- Object Serialization:

- Consolidated Kryo implementations across RPC and P2P messaging with whitelisting of classes via plugins or with `@CordaSerializable` for added node security.

- Privacy:

- Non-validating notary service now takes in a `FilteredTransaction` so that no potentially sensitive transaction details are unnecessarily revealed to the notary
- **General:**
 - Add vault service persistence using Requery
 - Certificate signing utility output is now more verbose

Milestone 8

- Node memory usage and performance improvements, demo nodes now only require 200 MB heap space to run.
- The Corda node no longer runs an internal web server, it's now run in a separate process. Driver and Cordformation have been updated to reflect this change. Existing CorDapps should be updated with additional calls to the new `startWebserver()` interface in their Driver logic (if they use the driver e.g. in integration tests). See the IRS demo for an example.
- Data model: `Party` equality is now based on the owning key, rather than the owning key and name. This is important for party anonymisation to work, as each key must identify exactly one party.
- Contracts: created new composite clauses called `AllOf`, `AnyOf` and `FirstOf` to replace `AllComposition`, `AnyComposition` and `FirstComposition`, as this is significantly clearer in intent. `AnyOf` also enforces that at least one subclause must match, whereas `AnyComposition` would accept no matches.
- Explorer: the user can now configure certificate path and keystore/truststore password on the login screen.
- Documentation:
 - Key Concepts section revamped with new structure and content.
 - Added more details to [Getting set up](#) page.
- Flow framework: improved exception handling with the introduction of `FlowException`. If this or a subtype is thrown inside a flow it will propagate to all counterparty flows and subsequently be thrown by them as well. Existing flows such as `NotaryFlow.Client/Service` and others have been modified to throw a `FlowException` (in this particular case a `NotaryException`) instead of sending back error responses.
- Notary flow: provide complete details of underlying error when contract validation fails.

Milestone 7

- With thanks to [Thomas Schroeter](#) [NotaryFlow](#) is now idempotent.
- Explorer:
 - The GUI for the explorer now shows other nodes on the network map and the transactions between them.
 - Map resolution increased and allows zooming and panning.
 - [Video demonstration](#) of the Node Explorer.
- The CorDapp template now has a Java example that parallels the Kotlin one for developers more comfortable with Java. ORM support added to the Kotlin example.
- Demos:
 - Added the Bank of Corda demo - a demo showing a node (Bank of Corda) acting as an issuer of Cash, and a client driver providing both Web and RPC access to request issuance of cash.
 - Demos now use RPC to communicate with the node from the webserver. This brings the demos more in line with how interaction with nodes is expected to be. The demos now treat their webservers like clients. This will also allow for the splitting of the webserver from the node for milestone 8.
 - Added a SIMM valuation demo integration test to catch regressions.
- Security:
 - MQ broker of the node now requires authentication which means that third parties cannot connect to and listen to queues on the Node. RPC and P2P between nodes is now authenticated as a result of this change. This also means that nodes or RPC users cannot pretend to be other nodes or RPC users.
 - The node now does host verification of any node that connects to it and prevents man in the middle attacks.
- Improvements:
 - Vault updates now contain full [StateAndRef](#) which allows subscribers to check whether the update contains relevant states.
 - Cash balances are calculated using aggregate values to prevent iterating through all states in the vault, which improves performance.
 - Multi-party services, such as notaries, are now load balanced and represented as a single [Party](#) object.
 - The Notary Change flow now supports encumbrances.

Milestone 6

- Added the [Corda technical white paper](#). Note that its current version is 0.5 to reflect the fact that the Corda design is still evolving. Although we expect only relatively small tweaks at this point, when Corda reaches 1.0 so will the white paper.
- Major documentation restructuring and new content:
 - More details on Corda node internals.
 - New CorDapp tutorial.
 - New tutorial on building transactions.
 - New tutorials on how to run and use a notary service.
- An experimental version of the deterministic JVM sandbox has been added. It is not integrated with the node and will undergo some significant changes in the coming releases before it is integrated, as the code is finished, as bugs are found and fixed, and as the platform subset we choose to expose is finalised. Treat this as an outline of the basic approach rather than something usable for production.
- Developer experience:
 - Samples have been merged back into the main repository. All samples can now be run via command line or IntelliJ.
 - Added a Client RPC python example.
 - Node console output now displays concise startup information, such as startup time or web address. All logging to the console is suppressed apart from errors and flow progress tracker steps. It can be re-enabled by passing `--log-to-console` command line parameter. Note that the log file remains unchanged and will still contain all log entries.
 - The `runnodes` scripts generated by the Gradle plugins now open each node in separate terminal windows or (on macOS) tabs.
 - A much more complete template app.
 - JARs now available on Maven Central.
- Data model: A party is now identified by a composite key (formerly known as a “public key tree”) instead of a single public key. Read more in [composite-keys](#). This allows expressing distributed service identities, e.g. a distributed notary. In the future this will also allow parties to use multiple signing keys for their legal identity.
- Decentralised consensus: A prototype RAFT based notary composed of multiple nodes has been added. This implementation is optimised for high

performance over robustness against malicious cluster members, which may be appropriate for some financial situations. See `notary-demo` to try it out. A BFT notary will be added later.

- Node explorer app:
 - New theme aligned with the Corda branding.
 - The New Transaction screen moved to the Cash View (as it is used solely for cash transactions)
 - Removed state machine/flow information from Transaction table. A new view for this will be created in a future release.
 - Added a new Network View that displays details of all nodes on the network.
 - Users can now configure the reporting currency in settings.
 - Various layout and performance enhancements.
- Client RPC:
 - Added a generic `startFlow` method that enables starting of any flow, given sufficient permissions.
 - Added the ability for plugins to register additional classes or custom serialisers with Kryo for use in RPC.
 - `rpc-users.properties` file has been removed with RPC user settings moved to the config file.
- Configuration changes: It is now possible to specify a custom legal name for any of the node's advertised services.
- Added a load testing framework which allows stress testing of a node cluster, as well as specifying different ways of disrupting the normal operation of nodes. See [Load testing](#).
- Improvements to the experimental contract DSL, by Sofus Mortensen of Nordea Bank (please give Nordea a shoutout too).

API changes:

- The top level package has been renamed from `com.r3corda` to `net.corda`.
- Protocols have been renamed to “flows”.
- `OpaqueBytes` now uses `bytes` as the field name rather than `bits`.

Milestone 5

- A simple RPC access control mechanism. Users, passwords and permissions can be defined in a configuration file. This mechanism will be extended in future to support standard authentication systems like LDAP.

- New features in the explorer app and RPC API for working with cash:
 - Cash can now be sent, issued and exited via RPC.
 - Notes can now be associated with transactions.
 - Hashes are visually represented using identicons.
 - Lots of functional work on the explorer UI. You can try it out by running `gradle tools:explorer:runDemoNodes` to run a local network of nodes that swap cash with each other, and then run `gradle tools:explorer:run` to start the app.
- A new demo showing shared valuation of derivatives portfolios using the ISDA SIMM has been added. Note that this app relies on a proprietary implementation of the ISDA SIMM business logic from OpenGamma. A stub library is provided to ensure it compiles but if you want to use the app for real please contact us.
- Developer experience (we plan to do lots more here in milestone 6):
 - Demos and samples have been split out of the main repository, and the initial developer experience continues to be refined. All necessary JARs can now be installed to Maven Local by simply running `gradle install`.
 - It's now easier to define a set of nodes to run locally using the new "CordFormation" gradle plugin, which defines a simple DSL for creating networks of nodes.
 - The template CorDapp has been upgraded with more documentation and showing more features.
- Privacy: transactions are now structured as Merkle trees, and can have sections "torn off" - presented for verification and signing without revealing the rest of the transaction.
- Lots of bug fixes, tweaks and polish starting the run up to the open source release.

API changes:

- Plugin service classes now take a `PluginServiceHub` rather than a `ServiceHubInternal`.
- `UniqueId` equality has changed to only take into account the underlying UUID.
- The contracts module has been renamed to finance, to better reflect what it is for.

Milestone 4

New features in this release:

- Persistence:
 - States can now be written into a relational database and queried using JDBC. The schemas are defined by the smart contracts and schema versioning is supported. It is reasonable to write an app that stores data in a mix of global ledger transactions and local database tables which are joined on demand, using join key slots that are present in many state definitions. Read more about persistence.
 - The embedded H2 SQL database is now exposed by default to any tool that can speak JDBC. The database URL is printed during node startup and can be used to explore the database, which contains both node internal data and tables generated from ledger states.
 - Protocol checkpoints are now stored in the database as well. Message processing is now atomic with protocol checkpointing and run under the same RDBMS transaction.
 - MQ message deduplication is now handled at the app layer and performed under the RDMS transaction, so ensuring messages are only replayed if the RDMS transaction rolled back.
 - “The wallet” has been renamed to “the vault”.
- Client RPC:
 - New RPCs added to subscribe to snapshots and update streams state of the vault, currently executing protocols and other important node information.
 - New tutorial added that shows how to use the RPC API to draw live transaction graphs on screen.
- Protocol framework:
 - Large simplifications to the API. Session management is now handled automatically. Messages are now routed based on identities rather than node IP addresses.
- Decentralised consensus:
 - A standalone one-node notary backed by a JDBC store has been added.

- A prototype RAFT based notary composed of multiple nodes is available on a branch.
- Data model:
 - Compound keys have been added as preparation for merging a distributed RAFT based notary. Compound keys are trees of public keys in which interior nodes can have validity thresholds attached, thus allowing boolean formulas of keys to be created. This is similar to Bitcoin's multi-sig support and the data model is the same as the InterLedger Crypto-Conditions spec, which should aid interop in future. Read more about key trees in the “[API: Core types](#)” article.
 - A new tutorial has been added showing how to use transaction attachments in more detail.
- Testnet
 - Permissioning infrastructure phase one is built out. The node now has a notion of developer mode vs normal mode. In developer mode it works like M3 and the SSL certificates used by nodes running on your local machine all self-sign using a developer key included in the source tree. When developer mode is not active, the node won't start until it has a signed certificate. Such a certificate can be obtained by simply running an included command line utility which generates a CSR and submits it to a permissioning service, then waits for the signed certificate to be returned. Note that currently there is no public Corda testnet, so we are not currently running a permissioning service.
- Standalone app development:
 - The Corda libraries that app developers need to link against can now be installed into your local Maven repository, where they can then be used like any other JAR. See [Running nodes locally](#).
- User interfaces:
 - Infrastructure work on the node explorer is now complete: it is fully switched to using the MQ based RPC system.
 - A library of additional reactive collections has been added. This API builds on top of Rx and the observable collections API in Java 8 to give “live” data structures in which the state of the node and ledger can be viewed as an ordinary Java `List`, `Map` and `Set`, but which also emit callbacks when these views change, and which can have additional views derived in a functional manner (filtered, mapped, sorted, etc).

Finally, these views can then be bound directly into JavaFX UIs. This makes for a concise and functional way of building application UIs that render data from the node, and the API is available for third party app developers to use as well. We believe this will be highly productive and enjoyable for developers who have the option of building JavaFX apps (vs web apps).

- The visual network simulator tool that was demoed back in April as part of the first Corda live demo has been merged into the main repository.
- Documentation
 - New secure coding guidelines. Corda tries to eliminate as many security mistakes as practical via the type system and other mechanically checkable processes, but there are still things that one must be aware of.
 - New attachments tutorial.
 - New Client RPC tutorial.
 - More tutorials on how to build a standalone CorDapp.
- Testing
 - More integration testing support
 - New micro-DSLs for expressing expected sequences of operations with more or less relaxed ordering constraints.
 - QuickCheck generators to create streams of randomised transactions and other basic types. QuickCheck is a way of writing unit tests that perform randomised fuzz testing of code, originally developed by the Haskell community and now also available in Java.

API changes:

- The transaction types (Signed, Wire, LedgerTransaction) have moved to `net.corda.core.transactions`. You can update your code by just deleting the broken import lines and letting your IDE re-import them from the right location.
- `AbstractStateReplacementProtocol.verifyProposal` has changed its prototype in a minor way.
- The `UntrustworthyData<T>.validate` method has been renamed to `unwrap` - the old name is now deprecated.

- The wallet, wallet service, etc. are now vault, vault service, etc. These better reflect the intent that they are a generic secure data store, rather than something which holds cash.
- The protocol send/receive APIs have changed to no longer require a session id. Please check the current version of the protocol framework tutorial for more details.

Milestone 3

- More work on preparing for the testnet:
 - Corda is now a standalone app server that loads “CorDapps” into itself as plugins. Whilst the existing IRS and trader demos still exist for now, these will soon be removed and there will only be a single Corda node program. Note that the node is a single, standalone jar file that is easier to execute than the demos.
 - Project Vega (shared SIMM modelling for derivative portfolios) has already been converted to be a CorDapp.
 - Significant work done on making the node persist its wallet data to a SQL backend, with more on the way.
 - Upgrades and refactorings of the core transaction types in preparation for the incoming sandboxing work.
- The Clauses API that seeks to make writing smart contracts easier has gone through another design iteration, with the result that clauses are now cleaner and more composable.
- Improvements to the protocol API for finalising transactions (notarising, transmitting and storing).
- Lots of work done on an MQ based client API.
- Improvements to the developer site:
 - The developer site has been re-read from start to finish and refreshed for M3 so there should be no obsolete texts or references anywhere.
 - The Corda non-technical white paper is now a part of the developer site and git repository. The LaTeX source is also provided so if you spot any issues with it, you can send us patches.
 - There is a new section on how to write CorDapps.
- Further R&D work by Sofus Mortensen in the experimental module on a new ‘universal’ contract language.
- SSL for the REST API and webapp server can now be configured.

Milestone 2

- Big improvements to the interest rate swap app:
 - A new web app demonstrating the IRS contract has been added. This can be used as an example for how to interact with the Corda API from the web.
 - Simplifications to the way the demo is used from the command line.
 - Detailed documentation on how the contract works and can be used has been written.
 - Better integration testing of the app.
- Smart contracts have been redesigned around reusable components, referred to as “clauses”. The cash, commercial paper and obligation contracts now share a common issue clause.
- New code in the experimental module (note that this module is a place for work-in-progress code which has not yet gone through code review and which may, in general, not even function correctly):
 - Thanks to the prolific Sofus Mortensen @ Nordea Bank, an experimental generic contract DSL that is based on the famous 2001 “Composing contracts” paper has been added. We thank Sofus for this great and promising research, which is so relevant in the wake of the DAO hack.
 - The contract code from the recent trade finance demos is now in experimental. This code comes thanks to a collaboration of the members; all credit to:
 - Mustafa Ozturk @ Natixis
 - David Nee @ US Bank
 - Johannes Albertsen @ Dankse Bank
 - Rui Hu @ Nordea
 - Daniele Barreca @ Unicredit
 - Sukrit Handa @ Scotiabank
 - Giuseppe Cardone @ Banco Intesa
 - Robert Santiago @ BBVA
- The usability of the command line demo programs has been improved.
- All example code and existing contracts have been ported to use the new Java/Kotlin unit testing domain-specific languages (DSLs) which make it easy to construct chains of transactions and verify them together. This cleans up and unifies the previous ad-hoc set of similar DSLs. A tutorial on how to use it

has been added to the documentation. We believe this largely completes our testing story for now around smart contracts. Feedback from bank developers during the Trade Finance project has indicated that the next thing to tackle is docs and usability improvements in the protocols API.

- Significant work done towards defining the “CorDapp” concept in code, with dynamic loading of API services and more to come.
- Inter-node communication now uses SSL/TLS and AMQP/1.0, albeit without all nodes self-signing at the moment. A real PKI for the p2p network will come later.
- Logging is now saved to files with log rotation provided by Log4J.

API changes:

- Some utility methods and extension functions that are specific to certain contract types have moved packages: just delete the import lines that no longer work and let IntelliJ replace them with the correct package paths.
- The `arg` method in the test DSL is now called `command` to be consistent with the rest of the data model.
- The messaging APIs have changed somewhat to now use a new `TopicSession` object. These APIs will continue to change in the upcoming releases.
- Clauses now have default values provided for `ifMatched`, `ifNotMatched` and `requiredCommands`.

New documentation:

- Contract catalogue
- Interest rate swaps
- Writing a contract test

Milestone 1

Highlights of this release:

- Event scheduling. States in the ledger can now request protocols to be invoked at particular times, for states considered relevant by the wallet.
- Upgrades to the notary/consensus service support:
 - There is now a way to change the notary controlling a state.

- You can pick between validating and non-validating notaries, these let you select your privacy/robustness tradeoff.
- A new obligation contract that supports bilateral and multilateral netting of obligations, default tracking and more.
- Improvements to the financial type system, with core classes and contracts made more generic.
- Switch to a better digital signature algorithm: ed25519 instead of the previous JDK default of secp256r1.
- A new integration test suite.
- A new Java unit testing DSL for contracts, similar in spirit to the one already developed for Kotlin users (which depended on Kotlin specific features).
- An experimental module, where developers who want to work with the latest Corda code can check in contracts/cordapp code before it's been fully reviewed. Code in this module has compiler warnings suppressed but we will still make sure it compiles across refactorings.
- Persistence improvements: transaction data is now stored to disk and automatic protocol resume is now implemented.
- Many smaller bug fixes, cleanups and improvements.

We have new documentation on:

- Event scheduling
- core-types
- Consensus

Summary of API changes (not exhaustive):

- Notary/consensus service:
 - `NotaryService` is now extensible.
 - Every `ContractState` now has to specify a *participants* field, which is a list of parties that are able to consume this state in a valid transaction. This is used for e.g. making sure all relevant parties obtain the updated state when changing a notary.
 - Introduced `TransactionState`, which wraps `ContractState`, and is used when defining a transaction output. The notary field is moved from `ContractState` into `TransactionState`.
 - Every transaction now has a *type* field, which specifies custom build & validation rules for that transaction type. Currently two types are supported: General (runs the default build and validation logic) and

NotaryChange (contract code is not run during validation, checks that the notary field is the only difference between the inputs and outputs). `TransactionBuilder()` is now abstract, you should use `TransactionType.General.Builder()` for building transactions.

- The cash contract has moved from `net.corda.contracts` to `net.corda.contracts.cash`
- `Amount` class is now generic, to support non-currency types such as physical assets. Where you previously had just `Amount`, you should now use `Amount<Currency>`.
- Refactored the Cash contract to have a new FungibleAsset superclass, to model all countable assets that can be merged and split (currency, barrels of oil, etc.)
- Messaging:
 - `addMessageHandler` now has a different signature as part of error handling changes.
 - If you want to return nothing to a protocol, use `Ack` instead of `Unit` from now on.
- In the IRS contract, dateOffset is now an integer instead of an enum.
- In contracts, you now use `tx.getInputs` and `tx.getOutputs` instead of `getInStates` and `getOutStates`. This is just a renaming.
- A new `NonEmptySet` type has been added for cases where you wish to express that you have a collection of unique objects which cannot be empty.
- Please use the global `newSecureRandom()` function rather than instantiating your own SecureRandom's from now on, as the custom function forces the use of non-blocking random drivers on Linux.

Milestone 0

This is the first release, which includes:

- Some initial smart contracts: cash, commercial paper, interest rate swaps
- An interest rate oracle
- The first version of the protocol/orchestration framework
- Some initial support for pluggable consensus mechanisms
- Tutorials and documentation explaining how it works
- Much more ...

[Next](#) [Previous](#)

Troubleshooting

Please report any issues on our StackOverflow page: <https://stackoverflow.com/questions/tagged/corda>.

[Next](#) [Previous](#)

- Other
 - View page source
-

Other

- Corda repo layout
- Building the documentation
- JSON

[Next](#) [Previous](#)

- Corda repo layout
 - View page source
-

Corda repo layout

The Corda repository comprises the following folders:

- **buildSrc** contains necessary gradle plugins to build Corda
- **client** contains libraries for connecting to a node, working with it remotely and binding server-side data to JavaFX UI
- **confidential-identities** contains experimental support for confidential identities on the ledger
- **config** contains logging configurations and the default node configuration file
- **core** containing the core Corda libraries such as crypto functions, types for Corda's building blocks: states, contracts, transactions, attachments, etc. and some interfaces for nodes and protocols
- **docs** contains the Corda docsite in restructured text format
- **experimental** contains platform improvements that are still in the experimental stage

- **finance** defines a range of elementary contracts (and associated schemas) and protocols, such as abstract fungible assets, cash, obligation and commercial paper
- **gradle** contains the gradle wrapper which you'll use to execute gradle commands
- **gradle-plugins** contains some additional plugins which we use to deploy Corda nodes
- **lib** contains some dependencies
- **node** contains the core code of the Corda node (eg: node driver, node services, messaging, persistence)
- **node-api** contains data structures shared between the node and the client module, e.g. types sent via RPC
- **samples** contains all our Corda demos and code samples
- **testing** contains some utilities for unit testing contracts (the contracts testing DSL) and flows (the mock network) implementation
- **tools** contains the explorer which is a GUI front-end for Corda, and also the DemoBench which is a GUI tool that allows you to run Corda nodes locally for demonstrations
- **verifier** allows out-of-node transaction verification, allowing verification to scale horizontally
- **webserver** is a servlet container for CorDapps that export HTTP endpoints. This server is an RPC client of the node

[Next](#) [Previous](#)

- [JSON](#)
 - [View page source](#)
-

JSON

Corda provides a module that extends the popular Jackson serialisation engine. Jackson is often used to serialise to and from JSON, but also supports other formats such as YAML and XML. Jackson is itself very modular and has a variety of plugins that extend its functionality. You can learn more at the [Jackson home page](#).

To gain support for JSON serialisation of common Corda data types, include a dependency on `net.corda:jackson:xxx` in your Gradle or Maven build file, where

XXX is of course the Corda version you are targeting (0.9 for M9, for instance). Then you can obtain a Jackson `ObjectMapper` instance configured for use using the `JacksonSupport.createNonRpcMapper()` method. There are variants of this method for obtaining Jackson's configured in other ways: if you have an RPC connection to the node (see “Client RPC”) then your JSON mapper can resolve identities found in objects.

The API is described in detail here:

- [Kotlin API docs](#)
- [JavaDoc](#)

Glossary

AMQP

The serialisation mechanism used within Corda for everything except flow checkpoints and RPC.

Artemis

The message queuing middleware used within Corda

Attachment

An attachment is a piece of data that can be referred to within a transaction but is never marked as used, i.e. can be referred to multiple times.

Command

Used for directing a transaction, sometimes containing command data. For example, a Cash contract may include an Issue command, which signals that one of the purposes of the transaction is to issue cash on to the ledger (i.e. by creating one or more Cash outputs, without any corresponding inputs.)

Composite Key

A tree data structure containing regular cryptographic public keys. It allows expressing threshold signature requirements, e.g. “either Alice or Bob” needs to sign.

Contract

A contract is code that specifies how states are to be created and used within Corda.

Corda

A Distributed Ledger for recording and managing financial agreements

CorDapp

A Corda Distributed Application. A shared ledger application on Corda consisting of components from: State objects (data), Contract Code (allowable operations),

Flows (aka Transaction Flows, the business logic choreography), any necessary APIs, wallet plugins, and UI components.

Cordformation

A gradle plugin that can be configured via your gradle buildscripts to locally deploy a set of Corda nodes

Counterparty

The other party in a financial or contract transaction

DSL Domain Specific Language - a language specifically designed for a particular domain. Kotlin allows the definition of DSLs and they are used within the Corda framework.

Flow

The set of instructions which determines how nodes communicate over Corda with the goal of arriving at consensus.

Fungible

An item that can be exchanged or interchanged for another identical item, e.g. Cash (as a \$10 note can be exchanged for two \$5 notes), but not diamonds (as they tend to have very identifying characteristics).

Gradle

Industry standard build and deployment tool. Used extensively within Corda.

Kotlin

The language used to code Corda. Fully compatible with any JVM language, including (obviously) Java.

Kryo

The serialisation mechanism used within Corda for flow checkpoints and RPC.

Input

In Corda terms, an input state is one that is used and consumed within a transaction. Once consumed, it cannot be re-used.

JVM

The Java Virtual Machine. The “computing machine” that Corda is executed within.

Lizard People

I would put their true identity in here but I fear the overlords may banish me.

Merkle Tree

A tree where each non leaf node is tagged with a hash of the data within that node and also the nodes beneath it. This ensures that the data in any node cannot be modified without causing hash verification failures in the parent node, and therefore all subsequent parents.

Network Map Service

A network service that maintains a map of node names and their network locations. Used by nodes such that they can communicate with other parties directly (after locating).

Node

A communication point on the Corda network and also the provider of the virtual machine in which Corda runs.

Notary Service

A network service that guarantees that it will only add its signature to transactions if all input states have not been consumed

Oracle

An oracle is a well known service that signs transactions if they state a fact and that fact is considered to be true. They may also optionally also provide the facts.

Output

In the Corda model, an output is a state generated from a transaction (note that multiple outputs can be generated from one transaction). They are then used as inputs for subsequent transactions.

Protocol

The old name for a Corda “Flow”

Quasar

A library that provides performant lightweight threads that can be suspended and restored extremely quickly.

SIMM

Standard Initial Margin Model. A way of determining a counterparty's margin payment to another counterparty based on a collection of trades such that, in the event of default, the receiving counterparty has limited exposure.

Serialization

Object serialization is the process of converting objects into a stream of bytes and, deserialization, the reverse process.

Service Hub

A hub in each Corda node that manages the services upon which other components of the node depend. Services may include facilities for identity management, storage management, network map management etc.

Signed Transaction

A signed transaction is a transaction that has been agreed by all parties relevant to that transaction as well as optionally a notary if relevant.

State

An element of data that is output from one transaction and then used / consumed in another transaction. States can only be consumed once and this confirmation is performed by the Notary service.

A transaction is the means by which states are both created and consumed. They can be designed to accept between zero and any number of input states, and then generate between zero and any number of output states.

UTXO

Unspent Transaction Output. First introduced by the bitcoin model, an unspent transaction is data that has been output from a transaction but not yet used in another transaction.

To confirm that the transaction is valid by ensuring the the outputs are correctly derived from the inputs combined with the command of the transaction.

To indicate that a class is intended to be passed between nodes or between a node and an RPC client, it is added to a whitelist. This prevents the node presenting a large surface area of all classes in all dependencies of the node as containing possible vulnerabilities.

[Previous](#)